UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

DESIGN, IMPLEMENTATION AND EVALUATION OF AN IN-HOUSE CONTROLLER FOR SOFTWARE DEFINED NETWORKING WITH APPLICATIONS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By

YIMING XU
Norman, Oklahoma
2017

Downloaded from SHAREOK. Persistent Handle URL: https://hdl.handle.net/11244/52389

DESIGN, IMPLEMENTATION AND EVALUATION OF AN IN-HOUSE CONTROLLER FOR SOFTWARE DEFINED NETWORKING WITH APPLICATIONS

A DISSERTATION APPROVED FOR THE SCHOOL OF COMPUTER SCIENCE

BY

Dr. Sridhar Radhakrishnan, Chair

Dr. Charles D. Nicholson

Dr. Mohammed Atiquzzaman

Dr. Qi Cheng

Dr. Sivaramakrishnan Lakshmivarahan

© Copyright by YIMING XU 2017
All Rights Reserved.

To Mi and my parents

Acknowledgements

Over the last five years, I have received support and encouragement from several individuals. I would like to express my deepest gratitude to my advisor, Dr. Sridhar Radhakrishnan, for his guidance and support for my Ph.D. study. Being an excellent teacher and mentor, He has not only taught me numerous skills and inspired me ideas in research, but also be a guide and supporter in this rewarding journey.

I would like to thank my dissertation committee of Dr. Mohammed Atiquzzaman, Dr. Charles Nicholson, Dr. Cheng Qi and Dr. Lakshmivarahan for their valuable time, and invaluable feedback. I would like to extend my thanks to Dr. Mahendran Veeramani for being a great tutor and partner. His passion for research is always encouraging me to go further. I also would like to thank Wei Guo, Asif Adnan, Michael Nelson, Dr. Chandrika Satyavolu and Dr. Amlan Chatterjee for listening to and validating my ideas.

In addition, I would like to thank all the colleagues and managers in New England Vascular Plant (NEVP) project and Grand River and Dam Authority (GRDA) for their excellent work and funding support.

Finally, I'd like to extend my gratitude and thanks to my mother, my wife Mi, and the rest of my family, without whom I would not be the person I am, and be here doing this work I love.

Table of Contents

List of Tables **vii**

List of Figures **ix**

Abstract **x**

1 Introduction **1**
1.1 Software-Defined Networking, Concepts, Implementation and Challenges 1
1.2 OpenFlow Implementations and Datapath Performance 5
1.3 Internet of Things and Edge Computing, Challenges in Data Flow Aggregation and Processing 7
1.4 Delay Tolerant Network (DTN) implementation in SDN approach . . . 9
1.5 Organization of the Dissertation 12

2 Design, Implementation and Evaluation of SDN In-House Controller 14
2.1 Introduction of Controllers in OpenFlow SDN 14
2.1.1 OpenFlow Protocol and Switch Specification 14
2.1.2 SDN Controller Design and Performance Evaluation 16
2.2 Distributed Controllers Solution in SDN 17
2.2.1 Flat Structure in Distributed Controllers in SDN 18
2.2.2 Hierarchical Structure of Distributed Controller in SDN 20
2.2.3 OpenFlow Data-Plane Implementation 21
2.2.4 Data-Plane Based Controller Implementation 22
2.3 SDN In-house Controller Design 23
2.3.1 Features Analysis in SDN Controller 23
2.3.2 Contributions In This Work 24
2.3.3 In-house Controller Architecture 25
2.4 Network Global Information Synchronization in Distributed Controller . 27
2.4.1 Learning Switch Based Routing Mechanism in In-House Controller 28
2.5 In-House Controller Performance Study 29
2.5.1 Roundtrip Performance for Learning Switch Application 30
2.5.2 Performance Study with Video-based Rerouting Application . . 33
2.6 Summary 35

3 A Networking Application Docking/Un-Docking Framework 37
3.1 Contributions In This Work 38
3.2 Inside In-House Controller Application Docking/Un-Docking Scheme . . 39
3.2.1 Inside SDN Docker Framework 40
3.2.2 Docker Application Processing Framework 43
3.2.3 Application Binary Organization 43
3.2.4 Performance Study of Inside SDN Docker 47
3.3 Outside In-House Controller Application Docking/Un-Docking Scheme [71] 49
3.3.1 Virtualized Connection in Linux 50
3.3.2 Outside SDN Docker Framework 51
3.3.3 Application Docking Procedure 54
3.3.4 Application UnDocking Procedure 56

3.3.5 Switch In-House Controller Module . . . 57
3.3.6 Testbed Implementation And Performance Study . . . 58
3.4 Summary . . . 62

4 Flow Aggregation in Internet of Things 63
4.1 MQTT Flow Aggregation in SDN Docker . . . 63
4.1.1 Contributions in This Work . . . 65
4.2 Short Flow Aggregation in MQTT Protocol [70] . . . 65
4.2.1 System Design and Implementation of MQTT Short Flow Aggregation . . . 66
4.2.2 Testbed Setup of MQTT Short Flow Aggregation . . . 68
4.2.3 Delivery Throughput Analysis of Fog Node . . . 71
4.2.4 Conclusion . . . 74
4.3 Long Flow Aggregation of MQTT Protocol[73] . . . 75
4.3.1 System Model of Long Flow Aggregation in SDN Docker . . . 76
4.3.2 Framework Development of MQTT Long Flow Aggregation . . . 78
4.4 Performance Evaluation of MQTT Long Flow Aggregation . . . 80
4.4.1 Evaluation Study of Traditional Fog Network . . . 80
4.4.2 Proposed Framework and Performance Evaluation Study . . . 82
4.5 Summary . . . 85

5 SDN Based Opportunistic Networking in Internet of Things [72] 86
5.1 Introduction to Delay Tolerant Network (DTN) Implementation . . . 86
5.1.1 DTN2 Implementation . . . 87
5.1.2 IBR-DTN Implementation . . . 88
5.2 Flexible Packet Forwarding Scheme For DTN . . . 88
5.3 Contributions in This Work . . . 90
5.4 Software-Defined DTN Infrastructure Offloading Framework . . . 91
5.5 A Novel Software-Define Flexible DTN Forwarding Architecture . . . 95
5.5.1 L2 Forwarding Scheme for DTN Bundle Forwarding . . . 95
5.5.2 Flexible On-the-fly Routing and Transport Services for Crowd-Friendly Environments . . . 98
5.6 Framework Performance Study . . . 101
5.6.1 A Novel Performance Evaluation Experiment . . . 102
5.6.2 Advanced Emulation with DTN nodes on RasperryPI . . . 106
5.7 Internet of Hybrid Opportunistic Things [72] . . . 108
5.7.1 Introduction to IoT and DTN Interconnecting . . . 108
5.7.2 A Novel Framework for IoT and DTN Interconnection . . . 109
5.7.3 System Evaluation and Conclusion . . . 112

6 Conclusion 113
6.1 Future Work . . . 114

References 117

List of Tables

2.1 Positioning of In-House Controller 24

List of Figures

1.1 SDN Architecture 2
1.2 SDN Data-plane Architecture 6
1.3 DTN Components and Event Scheduler [14] 11

2.1 Main components of a flow entry in a flow table [2] 14
2.2 Fields from packets used to match against flow entries 15
2.3 Open vSwitch Architecture 22
2.4 Positioning In-House Controller In OpenFlow SDN Architecture 25
2.5 In-House Controller Internal Architecture 27
2.6 CDF of roundtrip time for learning switch application in Mininet emulation environment 30
2.7 CDF of roundtrip time for learning switch application in real testbed . . 30
2.8 CDF of roundtrip time for learning switch application in Mininet with 500 nodes 31
2.9 Network topology used for video rerouting application testbed experiment 32
2.10 CDF for rerouting application on a testbed implementation 34
2.11 CDF for rerouting application on a Mininet implementation 34

3.1 Inside SDN Docker Architecture 40
3.2 Inside SDN Docker Framework and Components 41
3.3 Inside SDN Docker Binary Image Organization 44
3.4 Performance results of Reactive implementation of Centralized framework and our proposed Switch-inhouse controller framework 48
3.5 Performance results of Proactive implementation of Centralized framework and our proposed Switch-inhouse controller framework 48
3.6 Virtualized Network Application Routing to A Peer Node 51
3.7 An SDN docker framework implementation connected to a local PC server. Blue ports indicate virtual ports and Green ports indicate physical ports 53
3.8 Testbed implementation of the proposed framework. Blue ports indicate virtual ports and Green ports indicate physical ports 58
3.9 Time taken for individual steps involved in application docking 61
3.10 Time taken for individual steps involved in application undocking . . . 61

4.1 MQTT Publisher-Broker Architecture 66
4.2 Proxy Broker Architecture in Aggregated Node 67
4.3 MQTT Network with Fog Node Testbed Setup. *Each element inside 'Host PC' is run as virtual machine. 'MQTT-Broker' and 'OvS' represent the same architecture as shown in Fig. 4.2, but running on Ubuntu OS.* 69
4.4 Throughput performance for respective UDP and TCP clients, with Fog node computing 71
4.5 Congestion window size instantaneous vs average 74
4.6 Average throughput for different loss probability 75
4.7 Fog nodes internal architecture in long flow aggregation 77
4.8 Experiment network structure 79
4.9 Traditional DTN-based WLAN offloading scenario with four infrastructure nodes, and one mobile node. 81

4.10 Proposed SDN-based DTN offloading framework with four infrastructure nodes, and one mobile node. 81
4.11 Fast retransmission throughput of the traditional fog network. 82
4.12 Fairness indices of traditional network vs proposed framework. 84
4.13 Total number of received IoT (MQTT) messages in the traditional network vs proposed framework. 84

5.1 Traditional DTN-based WLAN offloading scenario with four infrastructure nodes, and one mobile node. 92
5.2 Proposed SDN-based DTN offloading framework with four infrastructure nodes, and one mobile node. 92
5.3 A frame with DTN bundle application payload. The in-built SDN controllers will perform deep-packet inspection on DTN primary blocks 93
5.4 Three vehicles platoon scenario: Traditional crowdsourced P2P DTN forwarding 95
5.5 Proposed SDN-based DTN Architecture. 96
5.6 Vehicle platoon scenario: Proposed SDN-based Layer-2 (L2) forwarding, and parallel multicast forwarding to intermediate nodes storage. 97
5.7 Offloading Application: Performance comparison of number of bundle copies in traditional DTN offloading and proposed SDN-based DTN offloading frameworks, in a four DTN node infrastructure network. . . . 101
5.8 Vehicular Platooning Application: Improved delay performance of proposed SDN-based DTN layer-2 forwarding, against the traditional DTN forwarding. 101
5.9 The testbed snapshot used in the implementation and performance evaluation study. 103
5.10 Heterogeneous DTN forwarding application: CDF Performance comparison of traditional IBR-DTN and proposed SDN-DTN offloading showing the number of messages received by fraction of nodes in the network . . 105
5.11 Heterogeneous DTN forwarding architecture: CDF Performance comparison of traditional IBR-DTN and proposed SDN-DTN offloading showing the number of messages received by fraction of nodes in the network, in a 10-nodes RaspberryPI experiment testbed 107
5.12 Proposed IoT-cum-DTN framework. 109
5.13 Proposed IoT-cum-DTN Gateway Node Architecture. The extended modules are shown in blue shaded boxes. 110
5.14 Throughput comparison between MQTT-over-DTN and MQTT-cum-DTN 112
5.15 Logic topology of the testbed. 112

Abstract

Over the past several decades, there has been a dramatic improvement in networking technologies. Network devices and protocols are becoming more powerful and complex. The vertical structure of the network protocol layers also leads to a coupled control plane and data plane in data frames. To solve this issue from a structural level, researchers introduced a new architecture of networking, the Software Defined Networking (SDN). By decoupling the control plane and data plane from a frame level and aggregating the protocols into software run in a centralized controller dynamically, engineers obtained a new way to build and control a network dynamically in real time.

Meanwhile, with the development of Internet of Things (IoT), data volume from mobile devices and low power terminals are dramatically increasing. However, the traditional cloud computing is still in a relatively centralized architecture, which causes huge traffic volume of IoT applications in the network. To this end, researchers proposed the concept of Edge Computing, which utilizes the capacity of the edge nodes in the network to process data and aggregate data from terminals.

This research introduces In-House Controller of SDN which has a distributed characteristic and deployed within SDN nodes to minimize the costs in control plane communication. The In-House controller also enables data processing and aggregation capacity in access points which host these functionalities as SDN applications. To research the system performance of the In-House controller in

different application scenarios, in this work, following applications were studied:

- Data flow aggregation of Message Queue Telemetry Transport (MQTT) protocol in Internet of Things, an MQTT proxy in edge switch which is aggregating short MQTT flows from multiple clients into a long MQTT flow to reduce the control plane traffic overhead in TCP.

- A novel delay tolerant network architecture and a new convergence layer over MQTT protocol in opportunistic networking. Using in-house controller as host and event scheduler for Delay Tolerant Network (DTN) [47] modules and convergence layers which run as applications guest applications in the controller.

With the study of applications, this research also proposed a generalized framework named as SDN Docker which support dynamically docking and un-docking applications in network devices with the help of the In-House controller.

Chapter 1

Introduction

Traditional IP network is growing fast in both scales of the network infrastructure and traffic volume and variety. On the other hand, the vertical structure of network protocols is making the network more complex and difficult to configure and manage. To solve these issues, researchers proposed a new networking architecture named as Software Defined Networking (SDN).

1.1 Software-Defined Networking, Concepts, Implementation and Challenges

SDN architecture was built in 2011. A typical SDN architecture including the following key characters[1]

- Decoupled control plane and data plane. SDN extract the control plane functionalities (packet forwarding and management) into a logically centralized controller which host network applications to define packet forwarding rules in the network.

- Instead of using destination addresses, in SDN, forwarding rules are defined by flows. A flow consists of two parts: matching rules and actions. Matching rules defined by a customized combination of packet header fields and corresponding values. Actions define the packet processing and forwarding operations.

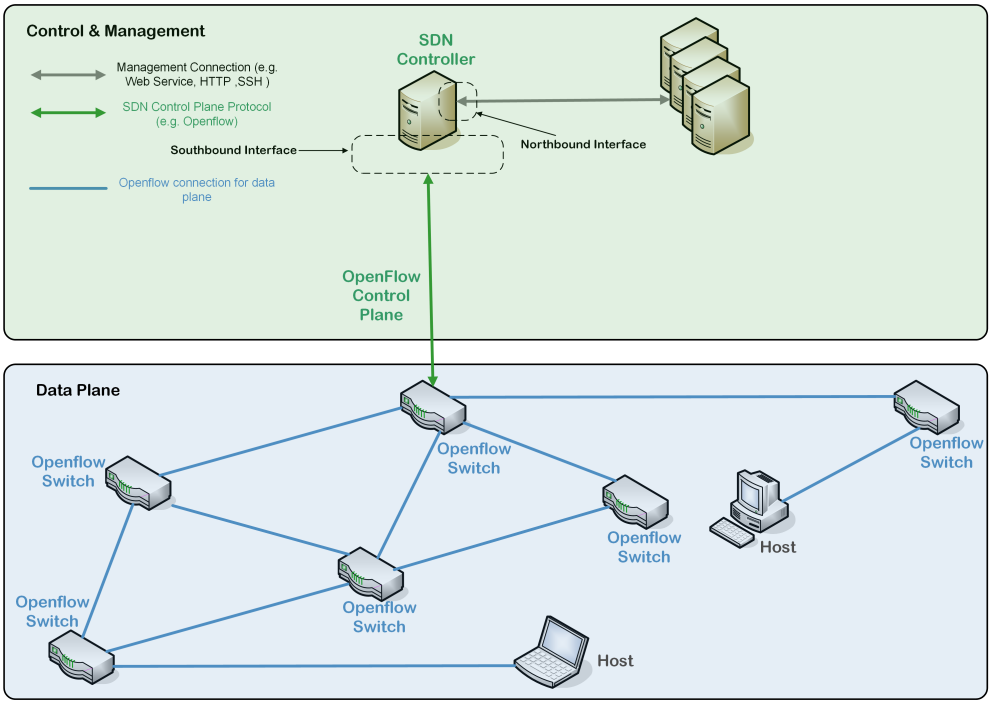


Figure 1.1: SDN Architecture

- Control logic is defined in an external entity in the network, also known as SDN controller. The controller is a software platform which is able to host network applications and provide them an abstracted, global view of the network.

- By providing a programmable interface, applications are able to operate packets and flows in the network to implement their functionalities.

Following these principles, multiple implementations of SDN has been proposed and developed by researchers and industry fields. As one of the most widely used standard, OpenFlow [2] protocol has been developed and deployed in both network devices and controller software. OpenFlow implement SDN features from the following perspectives:

Ports: Defined the functionalities ports on an OpenFlow switch. For physical ports in an OpenFlow switch, corresponding operations to ingress and egress packets are described. Meanwhile, logical ports forwarding packets to other components (like controller, pipeline or broadcast port) in OpenFlow specification are also described.

Flow Tables: Following the principle of SDN architecture, OpenFlow defined the two parts of entries in a flow table: Matching rules and actions. Besides these features, OpenFlow also defined the flow table pipeline, group and priority to make multiple flow entry sets could coordinate with each other.

OpenFlow protocol: The protocol runs over TCP with an OpenFlow header which includes the message type and associated actions for the packets interact between controller and switches. This part of the specification is also considered as the southbound API of SDN controllers.

Base on the OpenFlow specification, software for both switch side and controller side are implemented to build an OpenFlow network in the real world. On the switch side, OpenFlow for OpenWRT [4] was developed as an user space implementation in Linux based systems like Ubuntu and OpenWRT. This implementation included most of the switch side features from OpenFlow specification v1.0 to v1.3. Open vSwitch[36], implement virtualized switch datapath in kernel space of Linux based systems which is also following the OpenFlow specification. These implementations provide the possibilities to turn a regular physical network device to an OpenFlow device. Moreover, they are open source software which is able to modify for research purpose.

Comparing to switch side, controller side has more open source implementations even in different languages, such as NOX [25], Floodlight [24], Ryu [26] and OpenDaylight [27]. All these controllers implemented the southbound API to OpenFlow switches and the northbound API to network applications. Performance comparison of these controllers is also studied by researchers.

A centralized controller is one of the primary concern of SDN in the performance of scalability. In a large scale network with thousands of switches, the centralized controller will be the bottleneck of SDN network because of the capacity of controller software (number of requests served per seconds) and the control plane traffic volume exceeding the maximum throughput of the OpenFlow connection. To solve this issue, researchers have been working on solutions from the different perspectives: Flat structure distributed controller, hierarchy structure controller deployment and hybrid distributed controller solution[3]. The idea of these solutions is deploying distributed controllers in the network to reduce the scale of the sub-network each controller taking charge of, and build a proper communication mechanism between those controllers to maintain the global view of the entire network.

To some extend the solutions above solved the scalability issue of the centralized controller. However, even for a distributed controller which is close to the network devices, the communication overhead of OpenFlow protocol is inevitable. A latency between the flow management request and response could make the request meaningless because of the changing of network flows in this very short period of time. In this research, we introduce an In-House controller, which is

directly deployed in the OpenFlow components on the switch side. To reduce the latency and packet loss between controller and switch, the In-house controller is deployed in the switch and talking to the OpenFlow implementation through API without using any network connections and OpenFlow protocols. Meanwhile, it retains the southbound API to regular SDN controllers over OpenFlow, which could be used to communicate with other In-House controllers in a hierarchy flavor and be compatible to original OpenFlow setup. The In-House controller not only makes the SDN switch could be controlled locally but also provides possibilities for the switch to host application level functionalities in the device, which is far beyond the functionality of a network switch.

1.2 OpenFlow Implementations and Datapath Performance

In practice, OpenFlow is most often added as a feature to an existing Ethernet switch, IPv4 router or wireless access point in their operating system. Thanks to OpenWRT, an open-source Linux distribution for embedded systems, which support the OpenFlow implementations introduced above, the study to the lower level implementation of OpenFlow component which provides the foundation to In-House controller implementation. The structure of an OpenFlow data-plane implementation on switch side is demonstrated as Fig. 1.2.

According to OpenFlow specification, the main functionalities of data plane software has three parts: **Data-path:** The data plane component which is taking charge of packet matching, counting, forwarding and modification in kernel

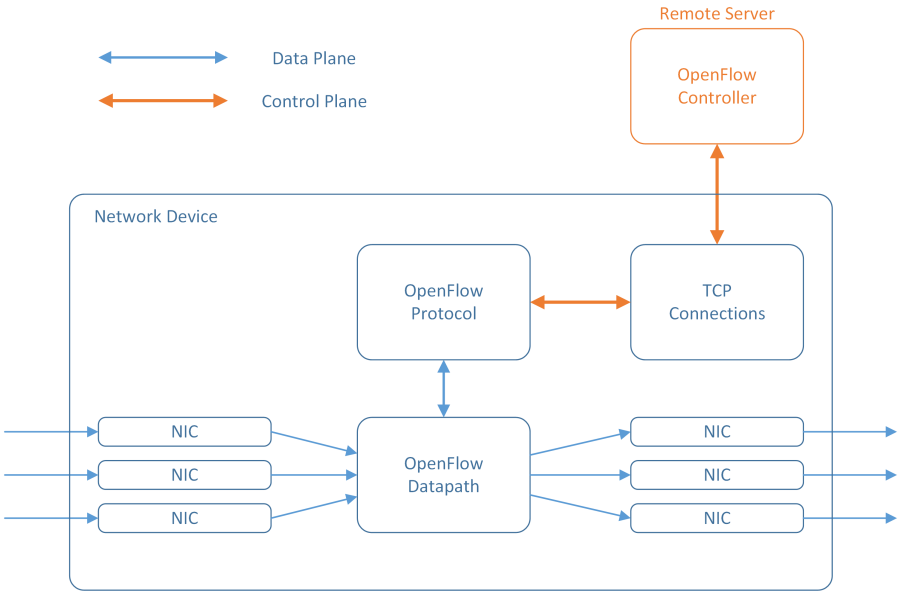


Figure 1.2: SDN Data-plane Architecture

or user space. It is the implementation of flow entries in OpenFlow. **Control-path:** This component manages TCP connections to remote controller. **OF-Protocol:** Protocol component for OpenFlow. This component implemented OpenFlow protocol and pack/unpack OpenFlow packets send to or receive from controller. The implementation of datapath decide the throughput of flow entries in OpenFlow switch. In OpenFlow for OpenWRT implementation, the datapath runs over raw IP stack with network devices API in Linux user space. Compare to previous implementation, Open vSwitch use kernel space code to keep traffic in flow entries in Linux kernel space to achieve a much higher throughput for traffics in a flow.

As an application development and deployment tool, the well-known Docker use containers to host application and its runtime environment. The Docker engine builds and runs the docker image instance locally while the Docker hub is playing

as a remote server which is keeping the docker images. For network applications, there are more obstacles than desktop or service applications in development and deployment. Normally, network applications are hosted in network devices which have limited computing and storage capacity. They also need extra network configuration which is tightly combined with the global network configuration which is hard to instantiate.

Learning from the Docker[37] framework, this work proposed an SDN Docker framework which is used to instantiate network applications and host them in network devices with dedicated configuration independent to each other. With the help of Linux Container (LXC) [38] and virtual Ethernet device (which are supported on OpenWRT and most of Linux distribution), network application could be hosted in a container and occupy a dedicated virtualized connection to the world outside through physical ports. The In-House controller will be the Docker engine in this framework to manage the runtime of these applications. A remote server which stores application images as a Docker hub is also enabled in this framework.

1.3 Internet of Things and Edge Computing, Challenges in Data Flow Aggregation and Processing

- One Delivery (At Most): Messages are delivered according to the best effort of the network; an acknowledgement is not required. (Least level of QoS)
- One Delivery (At Least): Message sends at least once, some duplicate

message may exist, and an acknowledgement message is required.

- On Delivering (Exactly): Requires an additional protocol to ensure that the message is delivered once and only once. (Highest level of QoS)

With these features, MQTT provide a lightweight reliable message transmission mechanism over transport layer protocol as an IoT application.

Because of the characteristic of IoT traffic and its applications, we introduce the concept of Edge Computing or Fog Computing in this work. Backbone network structure in normal cloud computing environment cannot accommodate the demands of IoT application due to flows competition on the backhaul links and the long latency. By providing elastic resources and services, like storage, computing and networking services, to end users at the edge of the network, Fog computing could provide similar functionalities as cloud computing with advanced features like low latency, geographical distribution, supported mobility management and online data processing [6].

Because of the lightweight characteristic of MQTT, In this work, with the help of SDN Docker framework, we propose a virtualized MQTT broker proxy in edge nodes to provide an effective and reliable data delivery in Internet of Things scenario. The proxy aggregate traffic flows from the clients which connect to the host access point to reduce bandwidth competition and traffic volume in control plane.

1.4 Delay Tolerant Network (DTN) implementation in SDN approach

Delay Tolerant Network is oriented to the heterogeneous network that may lack continuous connectivity. Packets are delivered in DTN in Bundle format[48] with a primary block, which contains the source and destination entity ID and application ID in DTN, and several data blocks which carry on payload data. To achieve reliable packet delivery in this scenario, an architecture with the following components have been proposed by researchers [13] [14] **Convergence Layers:**

To ensure packet delivery in a heterogeneous network, protocols in or above transport layer are generalized into a concept of convergence layer. A convergence layer could be a transport layer implementation, like UDP and TCP socket, or an upper network application like FTP or Email. One node could have multiple convergence layers in the same runtime. Two nodes have one or more same convergence layers could talk to each other. **Neighbour Management:**

In a wireless opportunistic networking, mobile nodes communicate with each other by chance of getting close. To claim the existence of node itself, a beacon message is broadcasting over a wireless channel. This beacon message contains convergence layers supported by the node and its configurations, like TCP port number or email addresses etc.. By detecting this message, the node could find neighbors getting close to it. Once a node with same convergence layer is detected, the communication will begin. **Storage:**

As a pre-assumption of DTN scenario, limits of wireless radio range and

sparsity of mobile nodes decide nodes have very limited chance to meet another node. To increase the probability of success in message delivery, in this scenario, DTN nodes are following the principle of store and forward in message delivery. DTN nodes carry messages by storage and deliver to neighbors by chance until the message arrives its destination node. **Routing modules:**

Another pre-assumption of DTN nodes is limitations in energy consumption. In the original scenario of DTN, communication in aerospace and deep space, nodes have strictly constrained in energy consumption while transmitting a message over wireless channels could one of the main consumers. To prevent unnecessary packet transfer between nodes, researchers proposed many routing algorithms in DTN. Several typical routing algorithms are listed as follows:

- Flooding routing: Always attempt to deliver all the messages stored on the local node to neighbors.
- Static routing: Messages will not be delivered unless the next hop node defined in routing table appears.
- Epidemic routing: Attempt to exchange messages which are not existing on the peer node.

Beside of these routing algorithms, researchers also studied the pattern of meeting probability of data mules (nodes) in the real world, which is not a completely random opportunity. Algorithms derived from this idea like (PROPHET[15], MaxProp[16] and RAPID[17] has been proposed and evaluated in a simulated environment. **Event scheduler:**

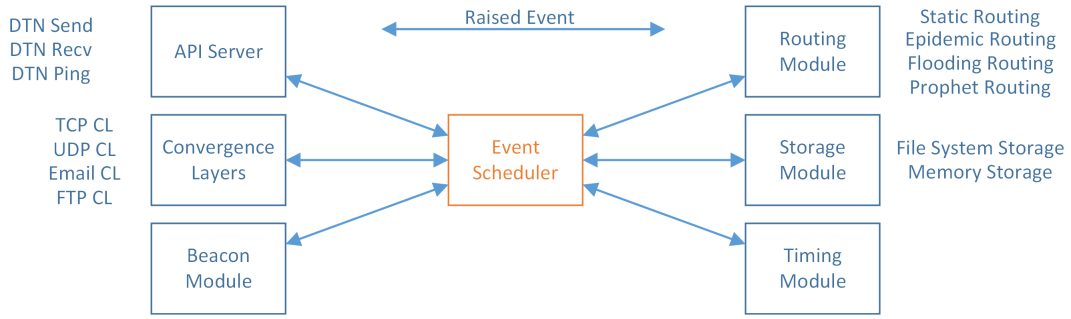


Figure 1.3: DTN Components and Event Scheduler [14]

In DTN node implementation, event scheduler is a hub of all the modules above. Events are raised while specific actions are finished in a specific module, like new neighbor discovery or packet receiving from neighbors, to notify other modules to do their work. The following graph shows its role in the whole architecture.

Coincidentally, this architecture exactly matches the structure of SDN Docker framework, which is hosting network applications in threads and processes in a single network node with multiple interfaces. With this inspiration, in this work, we proposed an enhanced architecture of DTN node in SDN scenario. In this architecture, modules and event scheduler are applications in SDN Docker. The event scheduler hosted as an application thread in SDN In-House controller, other modules are applications hosted in SDN Docker. Modules raise events based on the configuration of the node. These events will be packed with internal messages which go through the event scheduler. Event scheduler cooks these internal messages with an In-House controller to maximize the probability of message delivery and minimize the transmission time and a number of storage copies in DTN node.

1.5 Organization of the Dissertation

Chapter 2 first reviews distributed controller implementations in SDN architecture and introduces the OpenFlow implementation details in two open source software, OpenFlow for OpenWRT and Open vSwitch. A performance comparison between the In-House controller and other distributed controller solution will be made. Base on the controller analysis, we propose our In-House Controller architecture. To evaluate In-House Controller performance and make a comparison to commonly distributed controllers and centralized controllers, we select several applications in experiments.

Chapter 3 introduce our SDN Application docking/undocking framework, the SDN Docker. Based on the In-House controller, we implement the SDN Docker in two approaches, inside data plane, and outside data plane, to host different types of application. A docking/undocking performance evaluation will be provided in this chapter.

Chapter 4 focus on IoT traffic aggregation in edge nodes as a use case in SDN docker. In this chapter, demonstrate two frameworks deployed in edge nodes by SDN Docker to aggregate MQTT short and long flows. By analyzing MQTT short flows aggregation over TCP and UDP connections, we show our performance improvement in throughput. For long flows aggregation, we evaluate the fairness and packet processing throughput of the system. A comparison between proposed long flow aggregation framework and original framework also introduced in this section.

Chapter 5 focus on the usage of SDN docker in DTN network. The implementation and architecture will be introduced, as well as an enhancement in event scheduler. Several use cases of performance enhancement in DTN network, including infrastructure data offloading, adjustable beacon message and convergence layer and their performance comparison will be demonstrated. A novel DTN over IoT setup which introduces MQTT as one of the convergence layers will also be introduced in this chapter.

The last chapter will extend the content of this dissertation to future work in SDN based IoT, Mobile IP, and Opportunistic Networking.

Chapter 2

Design, Implementation and Evaluation of SDN In-House Controller

As the new and crucial component of SDN, controllers has been proposed along with the Openflow protocols. such as ,NOXMT [21], Maestro [22], Beacon [23], and Floodlight [24]. These controllers follow the SDN architecture in a centralized style over Openflow protocol as one of SDN implementation.

2.1 Introduction of Controllers in OpenFlow SDN

2.1.1 OpenFlow Protocol and Switch Specification

OpenFlow specification defines features from the switching perspective. It covers the components and the basic functions of the switch. An OpenFlow Switch consists of one or more flow tables, one group table and an OpenFlow channel to the controller.

A flow table consists of flow entries. Each flow table entry contains[2]:

- **match fields:** to match against packets. These consist of the ingress port and packet headers, and optionally metadata specified by a previous table.
- **counters:**to update for matching packets.



Figure 2.1: Main components of a flow entry in a flow table [2]

Ingress Port	VLAN ID	IP Source
Metadata	VLAN Properity	IP Protocol / ARP opcode
Ethernet Destination	MPLS Label	IPv4 Type of Service
Ethernet Source	MPLS Traffic Class	TCP/UDP/SCTP Source Port
Ethernet Type	IP Destination	TCP/UDP/SCTP Destination Port

Figure 2.2: Fields from packets used to match against flow entries

- **instructions:** to modify the action set or pipeline processing.

Groups represent sets of actions for flooding, as well as more complex forwarding semantics (e.g. multipath, fast reroute, and link aggregation). As a general layer of indirection, groups also enable multiple flows to forward to a single identifier (e.g. IP forwarding to a common next hop). This abstraction allows common output actions across flows to be changed efficiently. The group table contains group entries; each group entry contains a list of action buckets with specific semantics dependent on group type. The actions in one or more action buckets are applied to packets sent to the group. The group could also assign another group as one of its actions for next step packet processing.

Each flow entry contains a set of instructions that are executed when a packet matches the entry. These instructions result in changes to the packet, action set and/or pipeline processing. Supported instructions include:

- **Apply-Action(s)** qpply the specific action(s) immediately, without any change to the Action Set. This instruction may be used to modify the packet between two tables or to execute multiple actions of the same type. The actions are specified as an action list.

- **Clear-Action(s)** clear all the actions in the action set immediately.
- **Write-Action(s)** merges the specified action(s) into the current action set. If an action of the given type exists in the current set, overwrite it, otherwise, add it.
- **Write-Metadata** writes the masked metadata value into the metadata field. The mask specifies which bits of the metadata register should be modified.
- **Goto-Table next-table-id** Indicates the next table in the processing pipeline. The table-id must be greater than the current table-id. The flows of the last table of the pipeline can not include this instruction.

2.1.2 SDN Controller Design and Performance Evaluation

Controller design is addressed in many types of research. To handle a high volume of flow management request from the network, most centralized controllers focus on multithreaded designs and the parallelism of multicore computer architectures to improve the flow throughput. They have been designed as highly concurrent systems, to achieve the throughput required by enterprise class networks and data centers. As an example, by extending the NOX controller to a multithreaded version, NOXMT shows significant improvement in flow throughput and this improvement has near-linear scalability with the number of threads (cores). Meanwhile, the communication overhead between controller was noticed in studies. [28] summarized a flow request into 4 steps:

I. Packet arrives switch with no matching roles.

II. Packet is encapsulated into OpenFlow header and sent to controller.

III. Controller build flow entries for switch(es) and send back to network.

IV. Switch receive the packet, decapsulate the OpenFlow packet from controller, install flow entries and execute actions within.

Compare to the step 1 and 4 which are related to performance of switch devices, the delay caused by step 2 and 3, which is determined by the controllers resources along with the control programs capacity, bring more impact to the performance of the whole procedure.

To solve these performance issues, researchers lay eyes on distributed controller solutions. By controller distribution, 1) controller could be deployed closer to network devices so the communication delay will be reduced; 2) number of switches managed by one single controller could be limited to a reasonable size in large scale SDN network.

2.2 Distributed Controllers Solution in SDN

The logically centralized controller in SDN brings benefits in networking programmability, easier management, and faster innovation because it enables flow-level control over Ethernet switching and provides global visibility of the flows in the network.[29]. However, control plane communication between the controller and network devices, especially while the controller is deployed in a remote site,

will lead to overheads in control plane traffic which will cause performance issue in scalability and time efficiency in the network. Fine-grained flow operation events which are common in network operation make the situation even worse.

Meanwhile, because of the overhead, researchers have observed that the delay in the arrival of a flow's first packet and the controller's installation of new flow-table entries can create many out-of-order packets, leading to a collapse of the flow's initial throughput [30].

To address the issue, researchers proposed distributed controller solutions, use multiple controllers in one network to provide rapid response to flow request and offload traffic volume in the centralized controller. Meanwhile, to keep the consistency, how to synchronize the global status of the networking is the major concern in distributed solutions. The synchronization mechanism divides the solutions into two group:

Hierarchical model: One or some (but not all) SDN controllers in the cluster have the global network state.

Flat model: All of the SDN controllers in the cluster have the global network state. In the following part, we will introduce the two models respectively.

2.2.1 Flat Structure in Distributed Controllers in SDN

Flat model controller distribution requires controllers to handle flow request locally and share the updated network information by exchanging the update via East/Westbound API. The exchanging mechanism could be classified in two ways:[1]

- **Polling:** The controller periodically requests updates from all the other controllers in the same domain. This mechanism mainly has two issues: 1) The controller can not update network status in real-time. 2) Controller repeatedly obtains same updates while there is no changing in the network.

- **Publish/Subscribe:** one controller subscribes updates from other controllers in the domain. Each controller in the domain publishes updates to all its subscribers while network status changed on itself. This mechanism is more efficient than polling because for one update in the network only one copy will be transferred to each controller.

HyperFlow[31] is a typical implementation of flat model controller distribution. Controllers in HyperFlow manage different areas of the network which have no overlap with each other. When a flow path needs to be setup among network areas managed by different controllers, controllers along the path pass the serialized OpenFlow message one by one to exact the flow information and apply the flows in network devices under its control.

Other implementation like ElastiCon[32] focus on a dynamic assignment of switches to controllers. With the master/slave controller setup in OpenFlow, depends on controller load, a switch could seamlessly migrate between different master controllers on the fly.

2.2.2 Hierarchical Structure of Distributed Controller in SDN

In a hierarchical model, researchers define local controllers and a root controllers in the same network. Local controller and root controller communicate with each other by East/Westbound interface. The local controller is deployed close to network devices, while root controller is centrally deployed and connected to local controllers. The root controller holds the global view of the network and takes charge of synchronization among local controllers. Global OpenFlow requests are also processed by the root controller. Local controllers are handling local request from network devices which do not need to involve a global operation. If any global operation needed in the network, local controllers will initiate the request to root controller.

Several typical implementations of hierarchical controller model have been accomplished by researchers. Distributed-SDN [20] designed Main Controller (MC) as root controller in an ISP space and Secondary Controller (SC) as a local controller in a home appliance usage scenario. An SC-MC communication mechanism between these two components are designed as a customized protocol incorporates security concern as an integral part of the framework. Kandoo[33] implemented the hierarchical structure by identifying mouse flow and elephant flow. If a flow is confirmed as an elephant flow, it will be handled by root controller. Otherwise, for mouse flows, which happens much more frequently in traffic, will be handled by applications in local controller.

IRIS[34] introduced a recursively deployed controller solution. Controllers are

deployed in layers and higher level controllers consider the lower level network as a black box to communicate. Flow request will be forward up until the root controller if the current level controller does not have sufficient information to handle the request. After the request is processed, the response controller will notify all the lower level controllers in the same subset recursively.

2.2.3 OpenFlow Data-Plane Implementation

OpenFlow data-plane reference is a minimum implementation built by Stanford University originally in 2011. Based on this implementation, there have been several OpenFlow data-plane implementations on switch devices.

Pantou[35], the OpenFlow component for OpenWRT, implement OpenFlow data-plane in user space. To obtain frame level packet processing and forwarding ability, It sends and receives L2 frames over Linux device socket. This implementation can be deployed in most of Linux based OS, especially suitable for low power devices which have limitation in storage and memory size.

Open vSwitch [36] (Fig. 2.3) is another open source implementation of OpenFlow data plane which can be deployed in OpenWRT. To obtain a higher performance, Open vSwtich implement data-path in kernel space which provides an faster packet processing and forwarding capacity. Components in user space (OpenFlow protocol and control path) communicate with data-plane via Linux UpCall[40] which make a kernel space program to execute a function in user space.

Besides these two implementations, some other implementations have been posted by researchers and industry entities, like Pica8[41], Indigo[42] and Click[43]

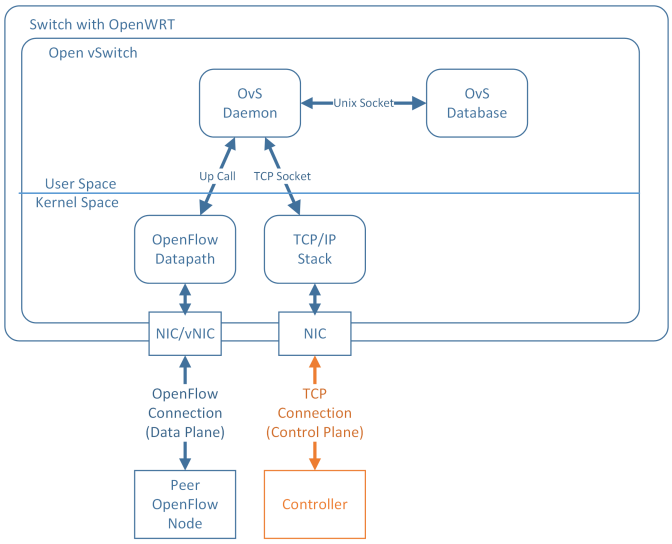


Figure 2.3: Open vSwitch Architecture

etc..

2.2.4 Data-Plane Based Controller Implementation

With the SDN and OpenFlow specification, the controller software run in an entity and communicate with OpenFlow devices through a TCP based connection. This creates overhead in flow requests which make the flow management lost its real-time efficiency even when the controller is deployed close enough to OpenFlow switches. To address the issue, several researches lay eyes on the implementation of OpenFlow components to embed flow management functionalities within data-plane. DIFANE[44] propose authority switches that offloads flow management functionalities into data-plane. A group of switches (namely, authority switches) handles data packets by having a set of pre-installed rules distributed by a central controller. However, unlike our framework, this work does not consider a full-fledged application-thread dynamically installable and runnable inside a switch.

DevoFlow[30] introduced the controller functionality offloading integrated into switches (namely, HP Procurve 5460zl) based on wildcard matching rules and flow clone mechanism in OpenFlow to create sub-flows in OpenFlow data-plane with local actions. However, this work creates flows in fixed granularity which could only be chosen between wildcard flow or packet specific flows. Flow actions in this work also need to be localized to switch side. For any unsupported actions, it still needs help from the remote controller.

2.3 SDN In-house Controller Design

To eliminate the communication overhead and scalability issue in centralized controller, and inherit the concurrent packet processing ability in traditional centralized controller, in this work, based on Pantou and OvS, we propose our In-House controller, an implementation of OpenFlow controller functionality in data-plane.

2.3.1 Features Analysis in SDN Controller

Features in SDN controller could be summarized to following perspectives:

- Whole frame packet inspection, including packet payload parsing and processing.
- Adding/Deleting flow entries and actions in switches by OpenFlow protocol.
- Flow monitoring by querying counters in existing flow entries.

- Parallel hosting applications in multiple threads in controller. Create pipeline among different applications for one single traffic flow.

For any OpenFlow controller implementation, these features should be considered and implemented within. Even in some single thread controller, applications could be switched in different runtime.

2.3.2 Contributions In This Work

Table 2.1: Positioning of In-House Controller

Literature	Centralized	Distributed	Local In-Switch Application	Application Docking
DIFANE	Semi-distributed Central controller		Partial	
DevoFlow		✓	✓	
Kandoo	Centralized Long-flows, Distributed short-flows		Conceptual-(www.kandoo.org)	
This Work	Distributed and local In-House controller,for all flows		✓	✓

Table 2.1 positions our work with respect to the related research works on SDN that focus on reducing the stress on the control-plane at the controller. The work by authors in [44] proposes authority switches that offloads certain functionalities of the controller. A group of switches (namely, authority switches) handles data packets by having a set of pre-installed rules distributed by a central controller. However, unlike our framework, this work does not consider a full-fledged application-thread dynamically installable and runnable inside a switch. The authors in [30], offloads the controller functionality integrated into switches (namely, HP Procurve 5460zl) based on OpenFlow [29]. However, the work does not consider dynamic application docking capability. The Kandoo [33] [61] proposes a hierarchically distributed controller framework, that proposes a hybrid way that has benefits of centralized and distributed system. The small

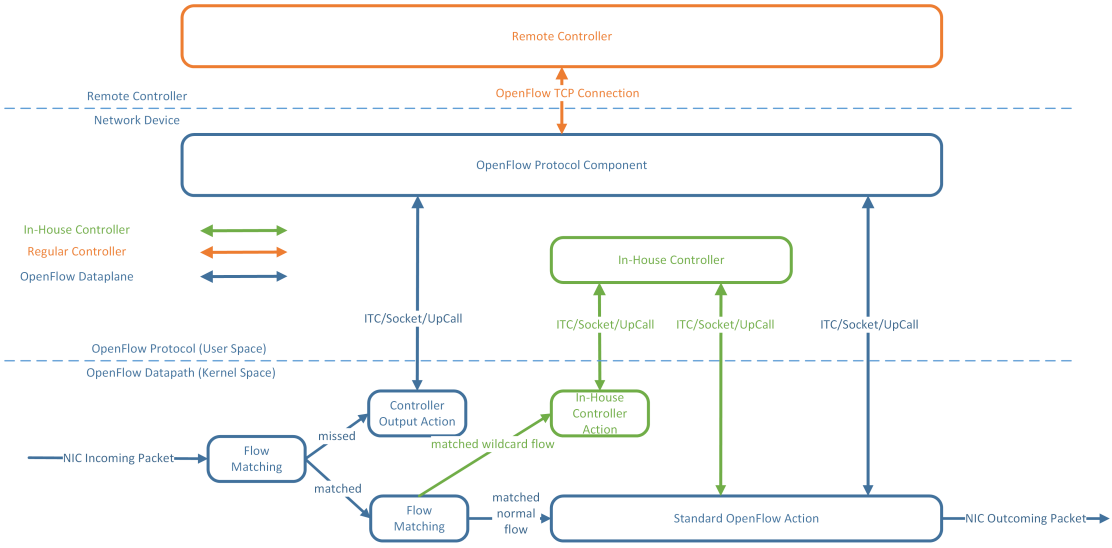


Figure 2.4: Positioning In-House Controller In OpenFlow SDN Architecture

flows will be handled by local (external) controllers and long flows (namely, the elephant flows) are handled by the central controller. The Kandoo framework is based on the Beehive distributed architecture framework. Though the authors have mentioned the possibility of Kandoo being integrated within switches; to the best of knowledge; this is still in conceptual stage, as a resource on such an implementation available in public. For our comparative study, we study our experiments by comparing the performance with respect to the centralized SDN implementation and the Kandoo-based distributed implementation (wherein, one physical controller is assigned to each network-switch).

2.3.3 In-house Controller Architecture

Regular OpenFlow controllers communicate with data-plane via OpenFlow messages(PacketIn, PacketOut, FlowMod etc.). In-House controller intercept packet frames between OpenFlow protocol and OpenFlow data-plane. Packet frames

will not be packed into OpenFlow messages but directly forwarded to the In-House controller for further processing (Create/Modify/Delete flows, create other actions). After the processing, the packet will be sent back to datapath and forwarded. To implement this feature, we need a mechanism to filter frames which failed to match any flow entries in the switch to the In-House controller. Also, to be compatible with original OpenFlow controller, this mechanism also need the ability to decide if a packet should be processed by the In-House controller or regular controller. We implement this mechanism with wildcard flow entries with generalized matching roles, lowest priority and an action which forwarding packet frames to In-House controller API. Packets matched regular flow entries will not be impacted because of their higher priority. Packets did not match any flow entries, including the wildcard entry, still will be forwarded to a regular OpenFlow controller.

The Fig 2.5 illustrated the internal structure of the In-House controller. In-House controller host each application in a separated thread. Each thread contains a message queue so the application can cooperate as a pipeline. Packet scheduler is a thread which receiving packet frame from the In-House controller actions. It decides which application should serve the packet. The application also can send processed packet back to this thread for another application's processing. Applications in the In-House controller can generate flows and actions by OpenFlow interface to communicate to datapath. To this end, the features of an SDN controller have been satisfied in an In-House controller.

The OpenFlow interfaces implemented in different ways in different OpenFlow

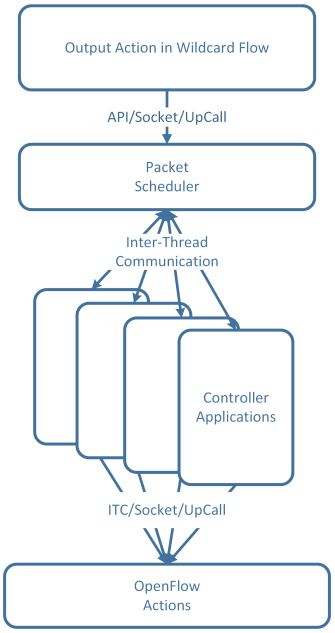


Figure 2.5: In-House Controller Internal Architecture

data plane implementations. In Pantou implementation, datapath and OpenFlow protocol are in the same space, so the OpenFlow interface in it is an API based interface which can be directly used by In-House controller. As we introduced, OvS deploys datapath in kernel space, so the universal interface is an UpCall based communication mechanism between kernel space and user space. In-House controller can use this mechanism to communicate to datapath component.

2.4 Network Global Information Synchronization in Distributed Controller

Global view of the network helps applications in SDN controller to know the topology and make obtaining the current output like routing results. As we introduced, to maintain the global view to the network, distributed controller

solutions developed different ways to implement this feature.

2.4.1 Learning Switch Based Routing Mechanism in In-House Controller

As an application in In-House controller, learning switch is an efficient mechanism to do optimized routing in a static network [18]. It also keeps optimized routing decision for all clients in each node. Learning switch routing including the following steps:

1: Source client send one packet to destination client with unique source ID and destination ID (usually we use source and destination MAC address, so called L2 Learning).

2: When packet arrives any node in the network, if both source and destination address have never seen on this node, record the mapping between the ingress port and source ID, then broadcast to all the other ports on that node except the ingress port.

3: If the mapping of source ID exists in the node, consider the packet is another copy broadcasted from other node and discard.

4: If the mapping of destination ID exists in the node, record the mapping for source ID if the mapping does not exist, build a routing rule (flow entry in OpenFlow context) for the source and destination ID and their mapped port.

In step 3, only the first packet for a specific client recorded in a mapping table. This feature ensured the shortest path was chosen in this setup. However, in a mobile network, clients could appear on different access points from time to time.

In this case, to identify if the client is moved, timestamps and timer in mapping record are necessary. If the timestamp in a record has not expired in a timer, the record is considered as a valid one which will not be replaced. Otherwise, the record will be replaced with new timestamp and the timer will be reset.

In the following part, this application will help us to evaluate the performance of controllers in different implement methods.

2.5 In-House Controller Performance Study

Having described the In-House controller framework, we now perform extensive performance evaluation study to demonstrate the functional capability of the core In-House controller module with respect to the centralized and distributed SDN controller frameworks. Without loss of generality, this test enables a fair comparison of our In-House controller framework with respect to the traditional centralized and distributed SDN frameworks. For our study, in additional to evaluating throughput performance across different frameworks; we also study the performance by running two different full-fledged applications (namely, learning switch application and video rerouting application) in our framework. The comparative centralized and distributed frameworks will run these applications on their external controllers.

For experiments, we run multiple (distributed) controllers on Mininet[19] simulator hosts and use a virtual network to build the control plane connections. The data-plane network is also deployed in Mininet on a different physical

computer; which will be connected to the control plane network via a physical port. This physical port could be considered as a port on the virtual switch in the control plane network. In our testbed, we have configured two VirtualBox machines and built a physical connection between them with no limit on the bandwidth. The physical connection bandwidth of up to 1.45GBits/s is shared by the switch-controller pairs.

2.5.1 Roundtrip Performance for Learning Switch Application

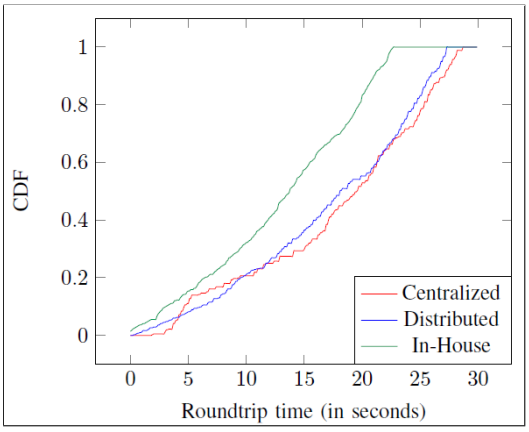


Figure 2.6: CDF of roundtrip time for learning switch application in Mininet emulation environment

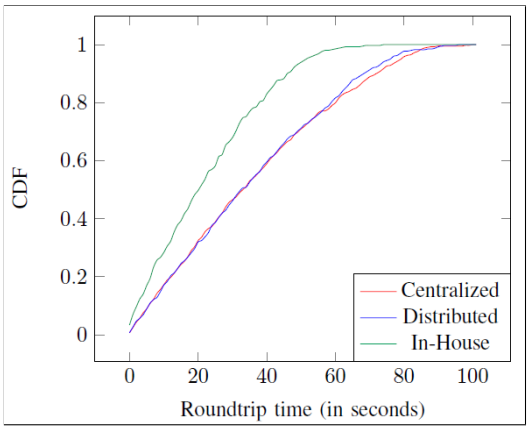


Figure 2.7: CDF of roundtrip time for learning switch application in real testbed

In this section, we study the round trip delay performance of a learning-switch application [18]. Traditionally, learning switch application runs on external SDN controllers that enable paths in the network. This application maps different MAC addresses to ports and subsequently installs the flow entries on the network elements (such as switches).

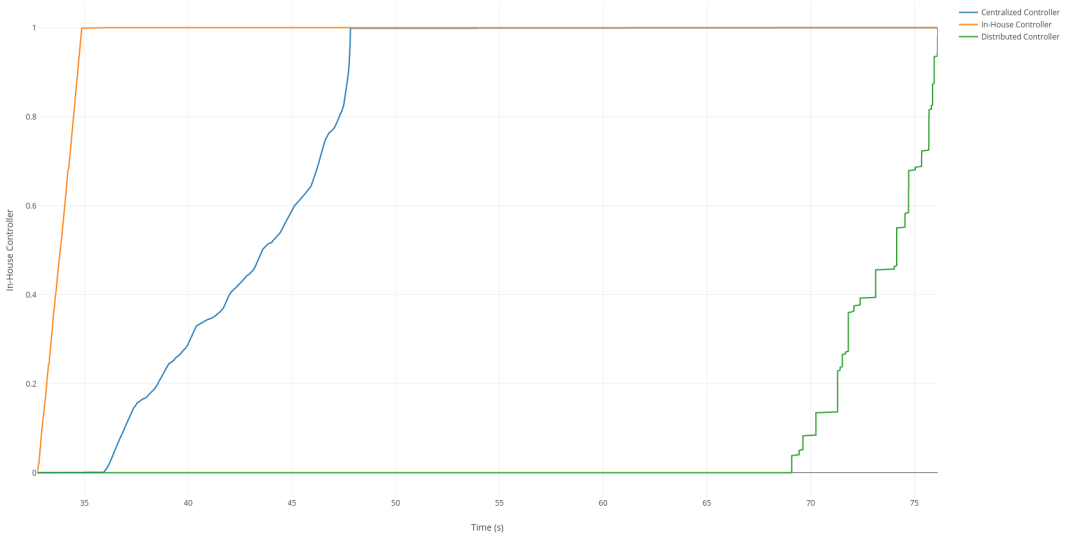


Figure 2.8: CDF of roundtrip time for learning switch application in Mininet with 500 nodes

Fig. 2.6 and Fig. 2.7 show the CDF of round trip delay for the respective Mininet and testbed experiments. As shown in Fig. 2.6, our proposed In-House controller framework outperforms both centralized and the distributed controller implementations. Around 60% of flows take about 16 seconds of round-trip time in In-House controller framework, as opposed to about 21 seconds of the centralized and distributed controller frameworks; which is 23% decrease. Moreover, in the In-House framework, all the flows take less than 23 seconds; as opposed to 26 and 27 seconds for the distributed and centralized controller implementations, respectively. As evident from Fig. 2.7, a similar trend of improvement is observed in the testbed results, as well. In the testbed experiments, all the flows require less than 60 seconds of round-trip time for our In-House controller; whereas the centralized and distributed controller frameworks, this delay is up to 90 seconds.

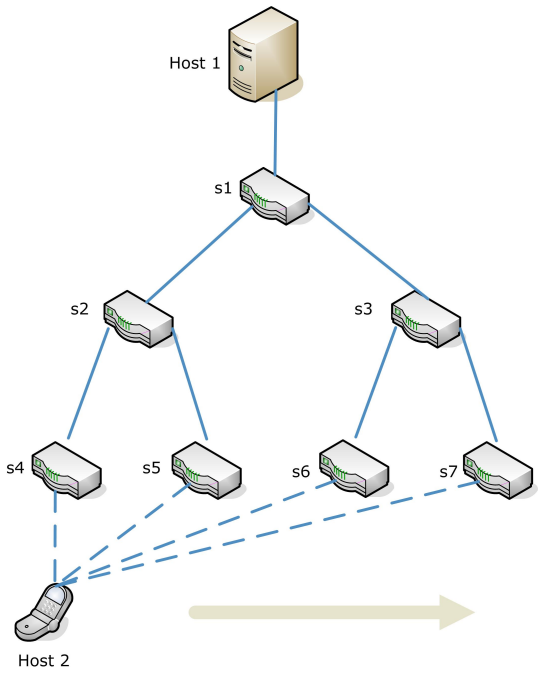


Figure 2.9: Network topology used for video rerouting application testbed experiment

To extend the round trip experiment to a large scale, we create chain topology with Mininet testbed to evaluate the round trip delay with 500 hops. As shown in Fig. 2.8, as the experiment scale increasing, the result gap between different solutions also increases. For in-house controller, all the ping packets finish their round trip in 32 to 35 seconds while the centralized controller needs 37 to 48 seconds to receive all the packets. The distributed controller solution takes more time to finish the round trip than these two solutions. The analysed reason of this result is the controllers in Mininet environments are sharing memory and local socket resource which lead to packets and retransmission attempts.

2.5.2 Performance Study with Video-based Rerouting Application

In typical mobile wireless networks, a moving host may perform handoffs between different access points that serve the video traffic used by the host. During hand-off events, the rerouting of network traffic happens in the network. The rerouting can be a full-rerouting or partial rerouting. In the case of full rerouting; the complete end-to-end routes will be torn down and a new end-to-end path is formed as per the new hand-off position. In the partial rerouting, on the other hand, only a portion of the path is changed. The testbed network topology is shown in Fig. 2.9. The server is the Host1, a static machine serving video traffic to a mobile user (namely, Host2), via an infrastructure composed of network of Linksys (WRT54GL) switches, running OpenFlow-based SDN. The server provides the video stream at the rate of 146KB=s (including video and audio). The detail of events happen during a hand-off is given below:

1. The mobile host is physically disconnected from the original switch and reconnects to a new neighboring switch.

2. The mobile host sends an ARP packet to the network and subsequently receives a response from the network.

3. The controller (centralized, or distributed, or In-House) does the network rerouting to enable connection between the server and the mobile host.

In a centralized controller framework, when the ARP request is received by the new connected switch, by default (due to the absence of flow-rule), this request

s forwarded to the centralized controller. The controller responds with an ARP reply to the mobile host. In our In-House controller this logic is implemented as follows:

1. The In-House controller module generates a control message and send it to the upper layer switch in the tree topology.

2. An ARP reply is also generated and sent to the mobile host.

3. Subsequently, new flow-entries are created based on the incoming port of the ARP request.

4. Upon receiving the control message; the upper layer network switch do the similar process of creating new flow-entries. For obvious reasons, in partial rerouting experiment; when the control message comes from the same port as per the old route the flow entries will not be updated.

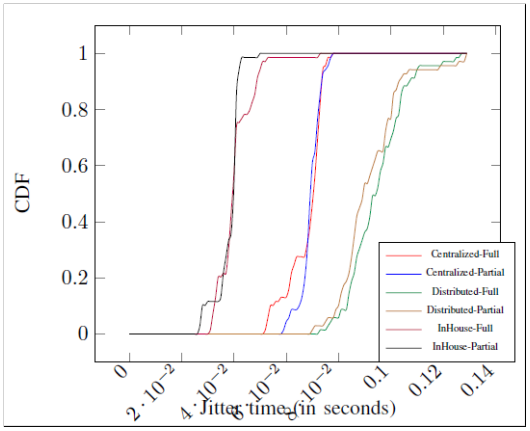


Figure 2.10: CDF for rerouting application on a testbed implementation

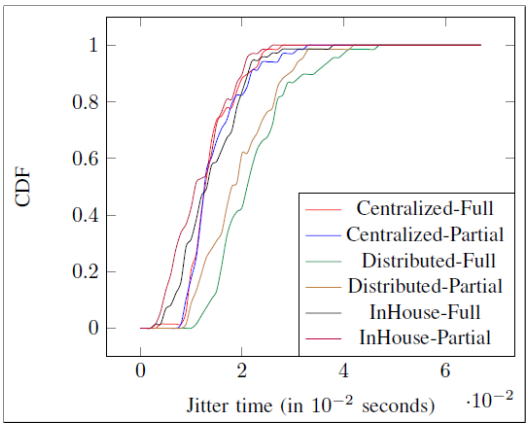


Figure 2.11: CDF for rerouting application on a Mininet implementation

For video applications, the primary characteristic of a network is to deliver video traffic with minimal jitter time. The reduced jitter time improves the Quality-of-Experience (QoE). Fig. 2.10 and Fig. 2.11 show the CDF of jitter time for two rerouting mechanisms across three different respective controller frameworks.

Fig. 2.10 shows the CDF of jitter-time for three different frameworks across to rerouting techniques. In a similar way, Fig. 2.11 shows the jitter-time CDF performance for the respective Mininet emulation experiment. It is clear from Fig. 2.10 and Fig. 2.11 that the In-House controller based SDN docker switch significantly improves the jitter time. The partial rerouting and full rerouting setups for the respective frameworks show similar performance. While the distributed controller performs the least; our In-House framework significantly outperforms the centralized and distributed implementations.

2.6 Summary

In this work, we have proposed design and implementation of In-House Controller under OpenFlow specification. With three different applications, namely, learning switch application, video rerouting application, and infrastructure offloading application; we have demonstrated the full functionality of the proposed framework. The prowess of our implementation is extensively studied by comparing our the performance with the traditional centralized SDN framework and the distributed controllers framework. We believe this work provides novel research direction to

the SDN community that looks for a scalable and flexible solution closer to the datapath.

Chapter 3

A Networking Application Docking/Un-Docking Framework

Docker framework obtained tremendous success in recent years. Its context container based implementation provide an exclusive runtime environment for applications, which make the applications portable and easy to deploy. However, in our SDN context, in the In-House controller, or even most of other regular SDN controllers, applications are static configured, applications can not be deployed in controllers runtime. Containers are also not available in controllers to make applications maintain its own context. Therefore, in this work, we propose the SDN Docker. We have the following design goals for our framework implementation:

I. Reduce Controller Overhead: To provide a framework to support controller applications runnable inside network switch.

II. Application Docking Capability: Without restarting the switch, the framework should support installable platform for new applications in the run-time.

III. Packet On-Demand Application: The incoming packet (of a flow) should decide on the kind of application that needs to be installed and serviced.

IV. Concurrent Application Support: The framework should support multiple applications to run concurrently in the form of threads inside the switch. Therefore, different applications need to be installed, uninstalled, and managed during the system runtime.

In OpenFlow specification, packets match no flow entries will be sent to a controller for flow management operation. In actual implementation, controller use applications to serve received packets. Network applications in SDN Controller follow a certain procedure to process incoming OpenFlow messages. Controller application listening OpenFlow channel to receive messages. Once a message arrives, it parses the packet from header to payload to extract information from it. With this information, the program insert does controller operations (flow management in switches, packet forwarding etc.). Other listeners receiving events in the network like nodes join or leave the network, link status, and statistics in the network.

3.1 Contributions In This Work

In this section, we introduce two implementations of SDN application Docking/Undocking mechanism in the context of OpenFlow its data-plane implementation we introduced above, the Pantou and Open vSwitch. We also will discuss its architectural implementation in detail. We developed an SDN-based auto-docker framework (built on switch embedded controller paradigm) that automatically identifies, and docks/undocks applications without end-user intervention. We believe such an implementation that manages an applications ecosystem and also effectively handles storage, computing, and networking resources of the switch would greatly benefit the research community. The proposed framework uses a remote common-pool for storing applications; and the required switches would

contact the remote docker-manager (an entity that maintains the common pool applications repository) for the specific-version of binary image related to the target switch hardware, for installation. In this manner, our framework would enable network engineers to autonomously manage applications and its future revisions. The contributions of this work are as follows:

- A novel SDN auto-docker framework is proposed and implemented. The working prototype is developed using off-the-shelf network switches.
- An extensive evaluation study is performed to investigate the time taken by individual stages involved in docking and undocking of two applications, namely MQTT-based IoT application, and DTN application.

3.2 Inside In-House Controller Application Docking/Un-Docking Scheme

Inside In-House controller application docking/un-docking framework (Inside SDn Docker Framework) defined as an extension of the In-House controller. It provides an application docking mechanism during the runtime of In-House controller. One application management module is deployed in In-House controller to manage the life cycle of SDN applications within. This solution prevent the restart of controller software when a new application deployed in the framework, especially when this restart is costly in SDN data plane.

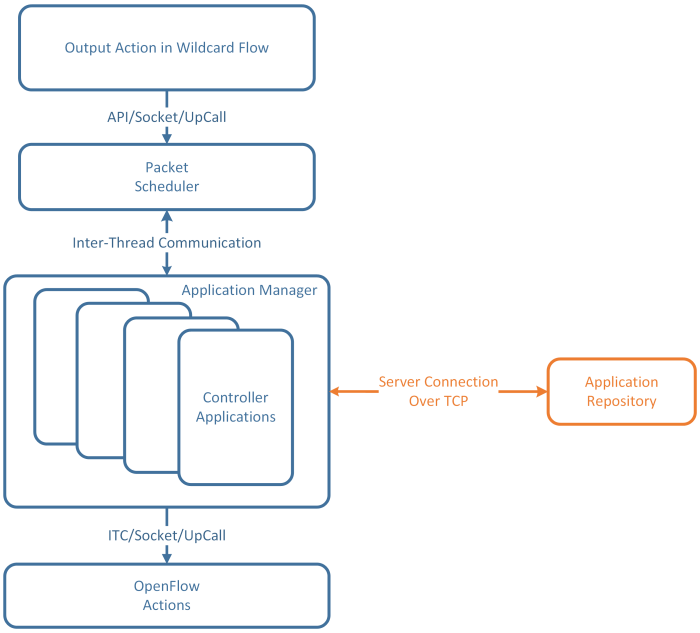


Figure 3.1: Inside SDN Docker Architecture

3.2.1 Inside SDN Docker Framework

Fig. 3.1 shows the conceptual architecture of an SDN docker switch integrated to the Internet cloud. Subject to the limited resources and the embedded network switch hardware (running OpenWRT); we exploit the networking capability of the switch to be connected an external resource for one-time application installation. Unlike the forwarding rules entry made by the typical SDN controller; this framework installs a full fledged application into the switch hardware. Therefore, the switch runs the application in a standalone fashion processing packets within the data-plane.

The docker framework stacks on the regular SDN OpenFlow implementation; creating a platform for multiple applications docking capabilities. The docker framework accesses the end-users packets and allows incoming packets application

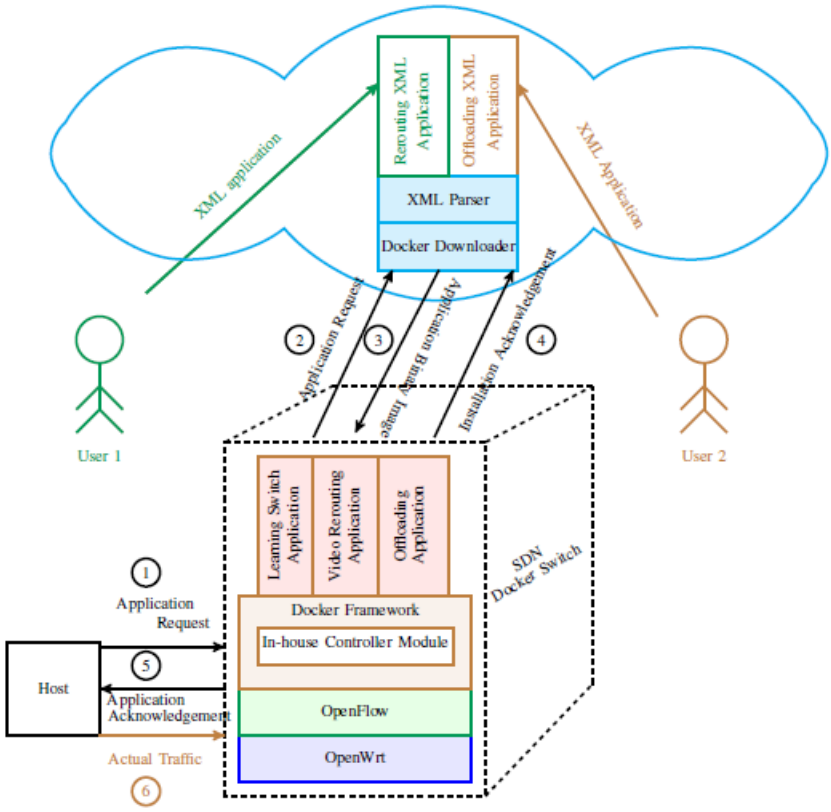


Figure 3.2: Inside SDN Docker Framework and Components

requests to the Internet cloud. The remote server parses the packets content for its application requirement. Subsequently, based on the network switch hardware configuration along the packets path to the destination; the corresponding application images are installed on the respective switches. This novel paradigm provides more flexibility and manageability to the SDN framework while remaining close to the data-path. Inspired the conceptual framework; we provide our actual testbed implementation architecture in Fig. 3.2 The sequence of events of the implemented framework is given below:

End-User Application Request Processing: The end-user (a host PC) sends a packet to the demanded application encoded in the packet-payload to the

SDN docker switch. The docker framework forwards this request to the remote server (a local PC). We believe that in future, this framework without loss of generality can be incrementally extended with Internet connectivity to a server in the Internet cloud.

Pool of Applications: The server maintains a collection of applications in the XML format. This is a user maintained repository of applications in XML. The server processes the necessary application using the XML parser and creates a binary image appropriately configured and installable on the specific network switch.

Binary Image Installation: The docker framework accesses the incoming applications binary image and installs as a runnable thread in the switch. In this work, we have considered three different applications (namely, learning switch, video rerouting, and infrastructure offloading).

Acknowledge the End-User: Upon successful installation, the docker framework informs both the remote application provider and the end-user with appropriate acknowledgments.

Flow Initiation: Upon receipt of the successful application installation, the end-user initiates the traffic flow, that is processed by the switches within their data plane. Thanks to the docker framework, the application is integrated within the network switches.

3.2.2 Docker Application Processing Framework

In this section, we describe the application processing scheme implemented in the framework. To dynamically docking/undocking application in different network devices with different operating systems, we need to convert the application logic to a binary image which could be executed by the framework.

3.2.3 Application Binary Organization

The binary image send to switch has three parts: message, variable and command sequence.

- **Variables** defined reserved memory spaces in application images. Each variable has an ID which will be applied in messages and command entries for identification. Each variable also has a length to indicate the space taken by the variable. The variable could be a number of a certain proto data type, like unsigned int, whose length could be obtained by "sizeof" function. It also could be a message with a certain data structure, the length of the variable is the total length of the message. The variable value could assign to a certain field in a message or other variables. Variables are compiled separately from the scripts.

- **Messages** are generated from all the .mes XML files. The .mes files are defined according to protocol definitions. Each protocol has its own header structure, so different structure may apply to different protocol message. The user also could define payload structure to parsing the content of the

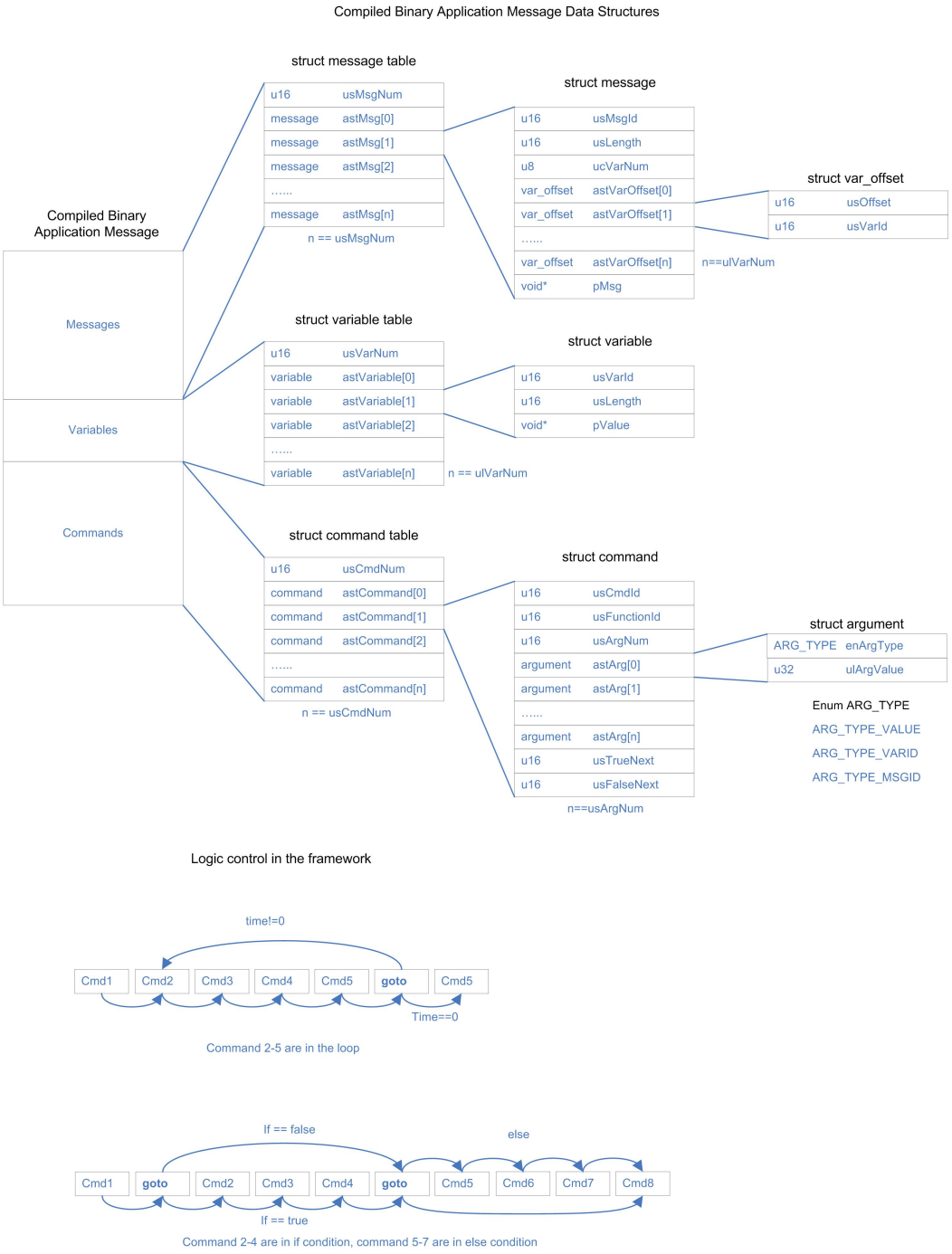


Figure 3.3: Inside SDN Docker Binary Image Organization

packet. Message contents may have two sources, user defined message and received from a framework. In user defined message, the message is defined in a mes file with structure and values. It will be compiled into message part in the binary file. Content received from framework are obtained from physical packets received by the In-House controller and dispatched by the event scheduler. Each message may have multiple variables. These variables could be located by an offset and length in a message space. When a message received by the framework, the receive function automatically fills the variable with a certain content part in the message which located by offset and length. The whole message also could be stored as a variable. In the mes document, any field marked with a variable name instead of the data value is a variable field. When a message is sent out, the message fields defined as variables will be automatically filled with current values in these variables. The message is also compiled separately from the scripts.

- **Atomic operations** define all the operation could be done by the framework. It works as bricks of applications so each application in the framework is composed of atomic operations. Atomic operations can NOT dynamically insert by compiled binary images. Add new atomic operation or extend the functionality of a exist atomic operation need to re-compile the switch application image and flash the switch to update in an offline way. All atomic operations in the framework are defined in a functional style with a uniform format (arguments and return values). each function mapped

to a unique ID statically. These IDs will be used in script XML parsing to indicate which operation will be used. So the remote PC who is taking charge of parsing XML document may have the same mapping of function names and IDs. There are one special atomic operations called "goto" in the framework. This operation does not have to map on remote server, but could be parsed from logical control operations like "if" "else" and "for".

Atomic operations have uniformed argument style. Each atomic operation may have different numbers of arguments but each of the argument has exactly same structure, a type indicator, and a value field. For each argument, it could be one type out of three: value, variable and message. When a message is a value type, the content of the argument is just the value data. When it is variable, the content is the variable ID in the variable table. When it is a message type, the content is the message ID in message table.

- **Command sequences** generated by parsing the script file on the remote machine. It defines the logic flow of the application by link atomic operations in a certain sequence. Each atomic operation has two pointers, true pointer, and false pointer, to indicate which is the next atomic operation. For regular atomic operations, it always uses the true pointer to point at next operation. Unless errors or exceptions occur in atomic operation, the operation will return with the false pointer to exit the program. For the "goto" operation, it may use the true pointer or false pointer. Take the loop logic, for example, we use "for" as a keyword in an XML document and attribute "time" to

indicate the loop times. When it compiled to a "goto" operation, the operation will be attached at the end of the code segment of the loop. The time number in the argument will decrease 1 every time when each loop is finished and "goto" operation was accessed. If the time argument is not zero, it goes back to the beginning of the loop code segment with false pointer, otherwise it goto the next operation out of the loop. "if/else" logic works in a similar way. "goto" operations will attached at the beginning of "if", "else", or "else if" code segment. When one of these "goto" operation matches the judgement condition, it will use true pointer which points the next operation, otherwise, it uses false pointer to jump to next "goto" operation or the end of "else" code segment to jump out. (please find the demonstration of loop and if/else logic in attached graph).

3.2.4 Performance Study of Inside SDN Docker

We performed a series of tests to evaluate the flow management rate (throughput) for different types of controller implementations. We have evaluated the throughput test for two settings: (i) proactive management, and (ii) reactive management. We performed the experiments on both testbed and Mininet emulation environment. For the testbed, we have considered one LINKSYS switch, connected to a Host PC and a remote controller PC. For the Mininet environment, we have used one local controller and Open vSwitch on the same physical PC, and the connection between the controller and switch is enabled through localhost. Since we considered a single switch network; we did not consider the distributed

controllers framework in our comparative study; due to its equivalence to a typical centralized system.

Fig. 3.4 shows the flow-rate performance for a reactive management setting. In this reactive scheme, the controller inserts flow tables in a reactive manner. In other words, the controller reacts to every new packet received by inserting a flow-rule according to the header information of the packet. After inserting the flow entry, the controller immediately sends a reply to the source host of the incoming packet. The time interval of these two packets is recorded as the flow-entry management time. Due to the receiving buffer limitation on the physical switch; packets may be lost; as a consequence, we have also measured the processing rate in this experiment.

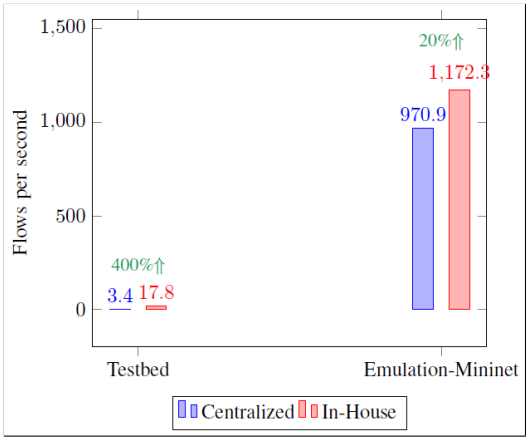


Figure 3.4: Performance results of Reactive implementation of Centralized framework and our proposed Switch-inhouse controller framework

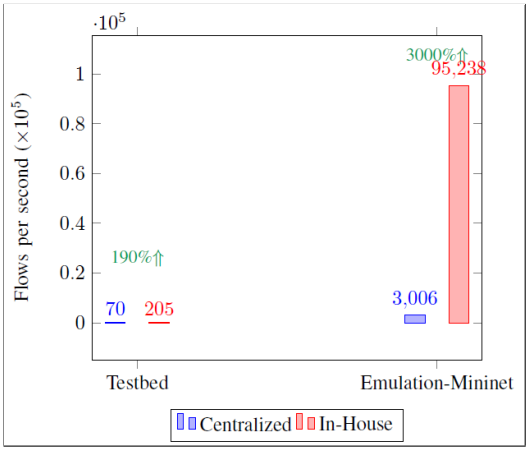


Figure 3.5: Performance results of Proactive implementation of Centralized framework and our proposed Switch-inhouse controller framework

As shown in Fig. 3.4, our proposed In-House framework outperforms the centralized framework by 400%. The emulation experiments on Mininet environments also show significant improvement of 20%. From Table. 2.1, it is evident that our In-House controller framework processes significantly more flows.

Fig. 3.5 shows the similar evaluation for the respective proactive configuration. In the proactive scheme, the controller inserts flow tables in a proactive manner. The controller triggered by an incoming dummy packet. After the receipt of the packet, the controller (in both centralized and In-House frameworks) inserts flow-rules in a continuous manner. In our experiments, we have to insert 400 flow-entries each time. After each flow insertion, the controller replies with another dummy packet to the source host (of the incoming packet). The time interval between these two packets are recorded as the total time consumption of the 400 flow management operations.

As shown in Fig. 5, our proposed In-House framework, on the real testbed, outperforms the centralized implementation with 190% improvement. Subsequently, the performance on emulation shows a significant 3000% improvement for our In-House framework.

3.3 Outside In-House Controller Application Docking/Un-Docking Scheme [71]

Inside SDN Docker works as a compiler which converts network application script logic into a binary format to load and execute in data-plane. Although it obtains

the speed in packet processing, the limitations are obvious. Atomic actions limited operations the frameworks can do. Any operation not on the list of atomic actions is not available for users. Real network applications, like application layer protocols, are much more complicated than the script Inside SDN Docker could express. Practically, for any SDN controller, host an upper layer protocol or any network application (like an HTTP server) as an SDN controller application is difficult. To solve these issue, we propose the Outside SDN Docker, which still host application in network devices, but out of the data plane of OpenFlow components.

3.3.1 Virtualized Connection in Linux

Thanks to the virtualization technologies in Linux based devices, we have the possibility to host network applications in a virtualized container and its associated context. Virtual Ethernet (veth) [45] is a kernel module supported most Linux based OS including OpenWRT. It creates a pair of Ethernet devices which are interconnected by a virtualized Ethernet connection. Virtual bridge created by Open vSwitch is another key component in this setup. The OvS virtual bridge contains SDN In-House controller, where we host controller functionalities for the Outside SDN Docker. Another role of the virtual bridge is bridging the virtual Ethernet connection to the real world, the physical Ethernet NIC on devices.

Fig. 3.6 demonstrate how a network application is routed to peer nodes outside of the switch. The application is hosted in a Linux container (LXC) which is also a virtualized environment. LXC can keep the context of the application while

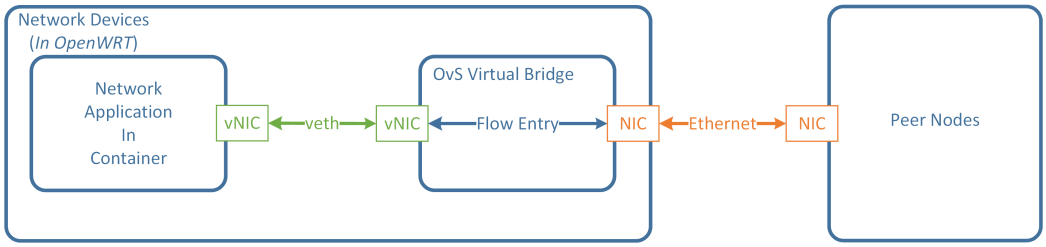


Figure 3.6: Virtualized Network Application Routing to A Peer Node

migration happens in the network.

3.3.2 Outside SDN Docker Framework

With the virtualization technologies we introduced above, we can propose the Outside SDN Docker Framework. End-User Application Request Processing: The end-user (a host PC) sends an application packet of application which can be identified by SDN docker switch. Identification rules defined by SDN Docker configuration file created by the network administrator. Once the packet is identified, the docker framework forwards a request to the remote application repository server (a local PC). The request contains the applications name and current network node's architecture information like hardware platform and operating system version.

- **Pool of Applications:** As the switch platform is built on OpenWrt, the (remote) server maintains a collection of applications built in OpenWrt for different target machines, such as Atheros AR7xxx/AR9xxx. To save time, applications are compiled apriori (as binary images) for the switches to download. Switches are pre-configured to connect to the remote server, via

a regular Internet connection.

- **Binary Image Installation:** In OpenWrt, applications are packaged in ipk packages, which could be installed and removed by using opkg tools. Each installed package can run as independent instances (also known as processes) with different configurations. These processes life cycles are managed by the Docker manager. For instance, in this work, we consider MQTT broker and DTN as the two sample applications.

- **Acknowledge the End-User:** The docker manager will provide an acknowledgment to remote file server after the file is successfully downloaded and installed.

- **Flow Initiation:** Different from the Inside SDN Docker, for each application, the docker manager creates a dedicated virtual Ethernet connection to connect the application with the virtual switch. After successful virtual connection setup and subsequent notification from the docker-manager, the In-House controller will manage packet forwarding between applications and physical ports by matching related fields in packet headers. However, initially, when no flow entries are configured, the packets by default are forwarded to the docker manager.

- **Flow Tear-down:** When the In-House controller receives a tear-down message either in a certain form like TCP FIN packet or application-specific messages like MQTT DISCONNECT (in IoT); the controller forwards a copy of this packet to the docker-manager. Subsequent to the receipt of

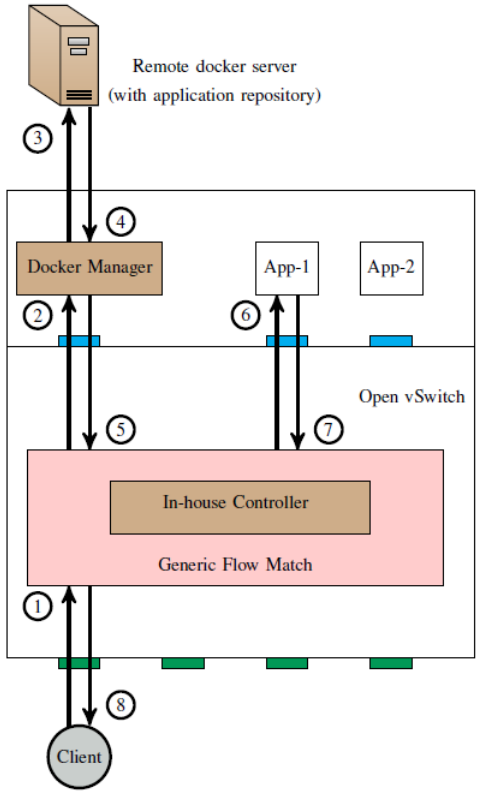


Figure 3.7: An SDN docker framework implementation connected to a local PC server. Blue ports indicate virtual ports and Green ports indicate physical ports

the packet, the docker-manager will parse the packet, identifies the related application, locates the respective virtual Ethernet connection for that application, and subsequently removes the connection and ports from Open vSwitch. Forwarding entries in the In-House controller will also be removed after the session is closed.

3.3.3 Application Docking Procedure

In this section, we discuss in detail the functioning of the proposed docker application framework. As shown in Fig. 3.7, the steps given below describe the sequence of procedure from clients request until its first response:

- **1.** The client (end-host) application sends its connection request to the network. The packet will be matched by pre-configured generic flow matches in the virtual switch, and subsequently received by the In-House controller.

- **2.** The In-House controller looks-up its own managed flow entries, and if there is a matching entry for the received packet, the packet will be forwarded to the virtual port associated with the application instance. Otherwise, the packet will be forwarded to the docker-manager. The In-House controller will keep a record of original packet header fields and ingress port numbers that are used at a later point in time.

- **3.** The docker-manager upon receiving the packet will parse the respective header and payload fields. With the help of configuration file of the docker-manager, it will recognize the application to which the packet belongs. If the specific application is not installed in the switch, the remote server is contacted, by passing application-name and the target platform of the switch.

- **4.** As a response, the remote docker server sends the corresponding application (binary image) installation packet to the switch. Subsequently, the

docker-manager embedded in switch would install, run, and connect the application with a created virtual port.

- **5.** The docker-manager sets original packets IP-TTL field in the IP header to egress virtual port number and sends it back to the In-House controller in the virtual switch.

- **6.** The In-House controller reads the IP-TTL field of the received packet as egress port number and sends it to the corresponding egress port with the recovered IP-TTL field. Meanwhile, the In-House controller builds flow entries within itself by packet header fields, which will forward the packet between the newly created virtual Ethernet connection, and the original ingress port.

- **7.** The application bound to the newly created virtual Ethernet the connection will receive the request packet, processes it, and sends an appropriate response packet back to the In-House controller.

- **8.** The response packet will have reverse header addresses to request packet and will be matched by the reverse flow entry created in Step 2. The response will be forwarded to the physical port which is connected to the end-host. Then the subsequent packets of the session will use flow entries in the In-House controller to forward between application in switch and end-host.

3.3.4 Application UnDocking Procedure

In this section, we present the procedure for undocking application without the user intervention. For each running application, the undocking-part of the framework requires a specific packet to notify that the current session is closing. For our experiments, we have used the TCPs last FIN ACK packet (if the application uses TCP transport) as an identification of the termination signal. Alternatively, the application-level packets can be used identify a tear-down session, such as MQTTs DISCONNECT packet. Upon identification of such packets, the In-House controller initiates the tear-down process as given below:

- **1.** The In-House controller will remove all the related flow entries for the specific session. After which, the packet will be forwarded to the docker-manager to finish the teardown procedure (TCP tear-down in our case). If there is no session for that specific application, the In-House the controller will also send a copy of the packet to the docker-manager.

- **2.** The docker-manager use the information from the headers of the packet looks up its record, and locates the virtual Ethernet ports created for this application. Upon locating, the virtual port will be removed from Open vSwitch and the virtual connection is teared-down.

- **3.** Subsequently, the docker-manager will also terminate the application instance which was running on the virtual port. To effectively utilize the storage space in the switch, if the program has no instances currently running after tear-down, the program will be uninstalled from the system.

3.3.5 Switch In-House Controller Module

The core functionality of the platform is built using two essential components namely, the Open vSwitch and the docker manager application. The Open vSwitch helps create virtual bridges for individual applications, and also comprises of an In-House controller to enable packet forwarding decision making and implementation. The docker manager application is implemented in the user-space of the OpenWrt platform which is also connected to the Open vSwitch through virtual Ethernet connection.

The In-House controller, embedded inside the Open vSwitch supports the typical SDN controller features such as packet creation, modification, and forwarding, header and payload parsing, in-band control message creation and sending among neighbor nodes. The end-to-end path selection is arbitrated by the In-House controller with the help of learning switch application. Unlike traditional SDN controller that modifies flow-tables in the switches, the modified controller in the proposed framework helps users to create and load different applications dynamically to Open vSwitch bridges. Multiple program instances can also be run simultaneously without interfering each other.

For packets that have no matching flow-entries in the In-House the controller will be forwarded by default to the docker manager process. As mentioned in the configuration file, the docker manager will process the packet and performs certain configurations settings for the corresponding application that the packet belongs. If the application is not installed in the switch, the remote server is

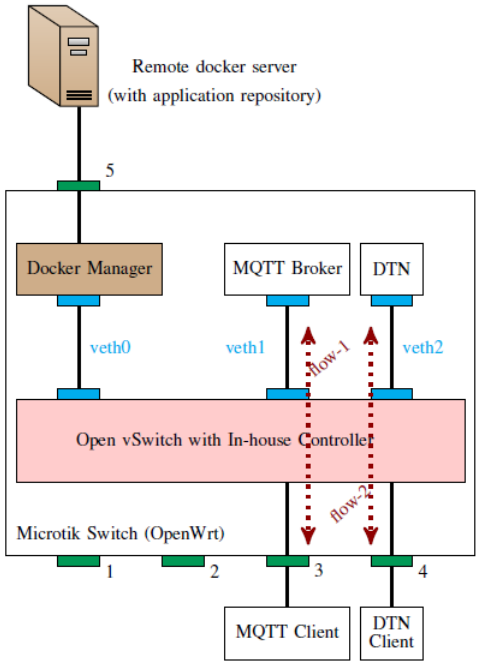


Figure 3.8: Testbed implementation of the proposed framework. Blue ports indicate virtual ports and Green ports indicate physical ports

contacted to get the corresponding application (in the form of compiled binary image), install the application, and run it by binding the newly created virtual Ethernet interface.

It is worthwhile to note that, the aforementioned two components do not impact the original SDN/Openflow architecture. Hence all the Openflow features in Open vSwitch bridge can co-exist with the proposed framework.

3.3.6 Testbed Implementation And Performance Study

For our implementation, we use Microtik Switch running on OpenWrt operating system. Two applications considered are Internet of Things' MQTT protocol and DTN. Upon docking, both these applications are embedded within the switch, as

per the proposed framework. Subsequently, undocked (uninstalled) from the switch as per the need. The following steps provide the functioning of the framework from the context of MQTT application. The DTN application and its clients function in a similar manner. Fig. 3.8 shows the block diagram of our testbed implementation.

- As the MQTT application typically functions on top of TCP transport, upon the MQTT client connected with the switch, the MQTT TCP SYN packet is the first packet received at the physical port-3.

- By default forwarding configuration, the packet is sent to the docker manager via the virtual port-0 (i:e:; veth0).

- The docker manager can parse the header and the payload of the received packet to identify the application it needs for processing. In our implementation, we parse the header and match the port number dedicated for MQTT in order to identify the MQTT application. The docker manager upon identifying the application will request the remote server for downloading the MQTT package for installation.

- The docker manager creates veth-1 with 2 virtual ports. One of the virtual ports is bound to the Open vSwitch. Subsequently, configures another virtual port by reading the destination field of the received packets IP header.

- The docker manager sends the packet back to Open vSwitch with the newly

created veth virtual port number (in the Open Vswitch) being embedded in the ip-ttl field.

- Open vSwitch receives the packet, forwards it to the appropriate egress port with the recovered ip-ttl field. Prior to this event, the In-House controller will create entries for bi-directional packet forwarding between recorded ingress port and fetched egress port from the feedback packets ip-ttl field.

Fig. , shows time taken to compute different steps during docking respective MQTT and DTN applications. Virtual configuration phase includes the virtual switch and virtual Ethernet configuration including the work done to perform the following tasks:

- Packet forwarding to the docker manager.
- Parsing packet header, and extract address fields such as IP and MAC.
- reate virtual Ethernet connections and virtual ports on the virtual switch.
- Configure virtual ports (such as IP address, and ARP).
- Set-up new flow-entries in the In-House controller.

The application download is performed on a connection with the Server, with an average RTT of 0.978ms. Application installation phase includes both installation and running. From Fig. 3.9, it is obvious that DTN consumed substantially more time than MQTT, considering the light-weight nature of the later IoT application. On the other hand, since DTN deals with physical node mobility in the order of

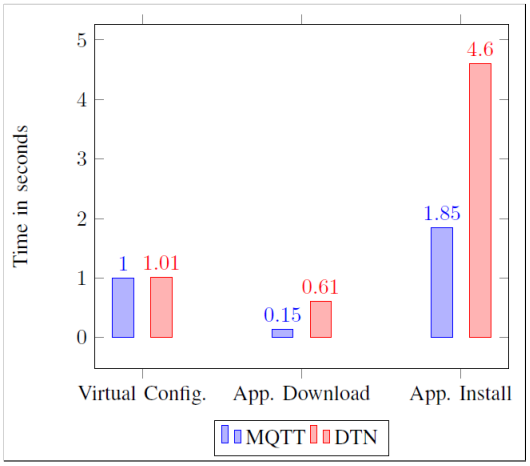


Figure 3.9: Time taken for individual steps involved in application docking

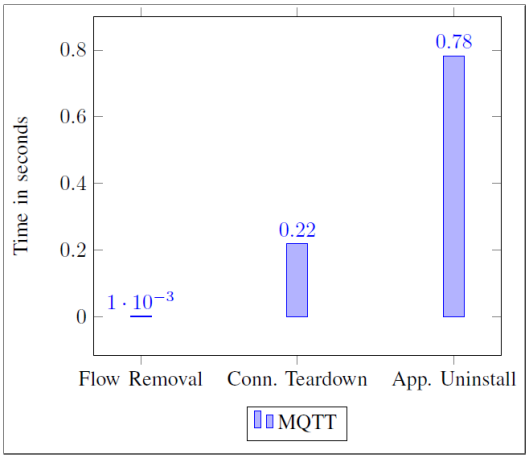


Figure 3.10: Time taken for individual steps involved in application undocking

seconds, typical contact time of a DTN node is expected to persist in the units of a few seconds, and such scenarios can greatly benefit from our proposed docking framework. It also worth to note that the experiment results are obtained from off-the-shelf low computing power switches, we expect an improved performance in a high-end switching hardware. Subsequent to application docking, both MQTT and DTN clients in future contacts had a fast response within 0:002s.

Fig. 3.10 shows the time consumption at different stages of application undocking procedure for the MQTT protocol. For the DTN nodes, by their design, they are tolerant to intermittent connectivity, therefore, they can be torn down at any instant of time. Therefore, we did not consider the tear-down times for DTN applications.

3.4 Summary

In this chapter, we have proposed the novel SDN docker Inside OpenFlow Dataplane and Outside of OpenFlow Dataplane framework implementations and presented the architectural design. We verify the improvement of throughput in moving the controller logic into OpenFlow data plane in the Inside SDN Docker solution. We also obtained the performance of application docking in Outside SDN Docker and docking/undocking time consumption.

In future, we plan to extend this framework to support docker capability using the Internet cloud. In this manner, a versatile SDN docking eco-system can be created for efficient application management and improved performance. We also use virtual Linux-based containers such as LXC to manage each application; therefore the state of the running application can be managed and possibly porting between physical switches can be investigated.

Chapter 4

Flow Aggregation in Internet of Things

Internet-of-Things (IoT) find practical use in a broad spectrum of applications. With a potential increase in the number of IoT devices, the demand for big-data analytics becomes a practical necessity; however, the performance of such analytics depends on the pace of delivery offered by the underlying network transport. Moreover, certain critical analytics such as detecting hazardous events require a quick response. Critical analytics near the data source enables timely actions to performed in controlling the hazard. The right place to utilize computational resources for performing analytics would be at the edge switches. Utilizing the edge switches resources and services for the end-users is called edge computing or Fog computing [49]. In a metamorphic perspective, Fog is closer than the Cloud. Hence the concept of computing at the edge switches (closer to the source of data generation) is termed as Fog Computing.

4.1 MQTT Flow Aggregation in SDN Docker

In this work, we propose and implement Fog Computing architecture at the edge switches using SDN. SDN enables programmability to the network switches, and also provides a centralized control-plane controller to enable routing decisions by appropriately utilizing available network resources. While integration of SDN and IoT has been an active field of research in the recent past [50], to the best of our knowledge, exploiting SDN to perform Fog computing has not been explored

yet. Being an application docking/undocking framework, as we introduced, SDN docker is an ideal platform to host MQTT applications in an edge node. In this manner, edge nodes behave as a discrete functional device that extends services and resources to the end-users. However, without loss of generality, the SDN controller can be used external to the switch (as in typical SDN environment) or can be hosted in SDN docker; based on individual needs.

We chose Message Queuing Telemetry Transport (MQTT) [5] as the candidate IoT protocol for our implementation. As we introduce, MQTT broker is the main component for mediating messages between publishers and subscribers. In other words, both subscribers and publishers exchange messages through the broker node. IoT applications envisioned to connect small battery-cell powered devices to the Internet, are typical of a low-form factor that generates/publish data in a sporadic manner. However, potential large-scale deployments of such IoTs create a huge aggregated traffic to the Broker nodes that causes congestion and thereby reduced throughput (messages per second) in the network.

SDN Docker enables programmability in the network stacks thereby offering flexibility and manageability to the network designers. To adapt MQTT-IoT applications to large-scale operations, we host an MQTT proxy broker in SDN Docker to play the role of aggregating independent MQTT clients traffic for effective transport in the edge nodes.

As an IoT application protocol, MQTT runs over TCP to ensure a reliable delivery. Depends on the size of data segments to delivery, the traffic pattern in MQTT could be classified into long flows and short flows. Long flows can be found

in large size data segment delivery which needs a continuous TCP connection with saturated traffic flow. Short flows happen in small pieces data delivery which only has one or two TCP frame in a single connection. In this cases, control plane messages like TCP handshake consume even more network resources than data plane packets.

4.1.1 Contributions in This Work

In this work, we will analyze these two types of traffic flows aggregation in edge node separately. For short flows, we develop and implement an SDN-based proxy broker to play the role of aggregating independent MQTT clients traffic for effective transport in the network. We also mathematically investigate the improved performance and study the throughputs deviation from mean [51]; with the help of large-deviations theory. For long flows, By highlighting the fact that the key network delay is caused by the unfairness in the delivery throughput of IoT clients, we propose an augmented transport-layer framework to achieve fairness in the fog network system. By achieving fairness in the system, we improve the delivery performance of the proposed network.

4.2 Short Flow Aggregation in MQTT Protocol [70]

The connections between publishers, broker, subscribers are enabled using standard transport protocols such as TCP and UDP. As the IoT devices (MQTT-publishers) are highly resource constrained devices, they establish a connection (say, using TCP) with the broker whenever a new data needs to be published. On the other

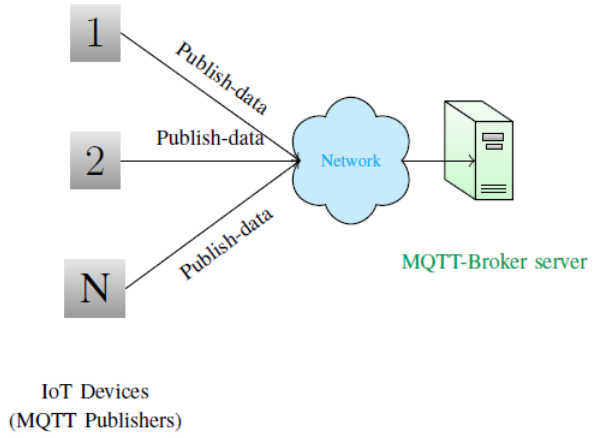


Figure 4.1: MQTT Publisher-Broker Architecture

hand, the MQTT subscribers typically high computing devices (such as PCs) will maintain connections with the broker all the time to receive topic-based publish-messages. The ideal place to perform analytics is the MQTT-broker, which is a central repository of all the published data. We henceforth focus our attention on the MQTT-publishers and MQTT-brokers throughput in this work, as shown in Fig. 4.1.

4.2.1 System Design and Implementation of MQTT Short Flow Aggregation

For the MQTT network considered in Fig. 4.1, we enable the functionality of the broker node in the edge switch (i:e:; the first-mile switch connecting the MQTT publishers). This edge-switch with the broker functionality is henceforth called the Fog node. The Fog node architecture is shown in Fig. 4.2. In order to enable the Fog node to behave as an independent computing node, we use SDN Docker

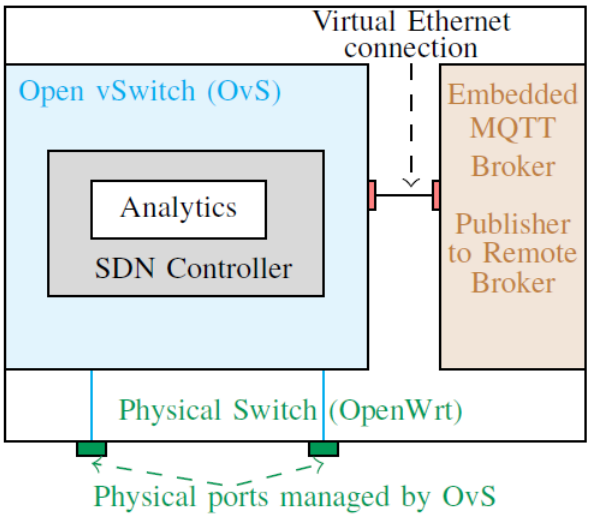


Figure 4.2: Proxy Broker Architecture in Aggregated Node

to deploy proxy broker within the switch hardware. The Fog node serves the following purposes:

- Behaves as actual broker for MQTT clients.
- Serves as a platform for performing analytics at the Fog node.
- Needs to communicate with the end-host broker server for storage and exhaustive deferred analysis. It should be noted, that an external communication from a Fog node serves multiple purposes such as connecting with another Fog node; in a distributed brokers environment.

The MQTT publish-messages arrive at the physical port of the Fog (switch) node. The switch integrated controller acts as a single proxy-publisher will maintain a TCP connection with the remote end-host broker, and publishes all of the received messages from the real MQTT publisher clients. To respond to the MQTT publishers the Fog node runs the entire MQTT broker integrated

within the switch; which is communicated to the SDN controller through virtual-ports (as shown in Fig. 4.2). Before sending the publish-messages, the MQTT publisher establishes application-layer connections with the Broker. To this end, the publisher will send MQTT Connect message to the broker (Fog node) and upon receiving the Connect-ACK message the actual publish-message is sent.

The Open vSwitch (OvS) communicates to external hosts through physical ports of the switch, and internally with the MQTT broker through a virtual port. The SDN controller forwards the MQTT messages between (external) MQTT clients and switch-integrated MQTT broker. In order to send the received MQTT messages from Fog-broker node to the remote broker, we created a separate thread of MQTT publisher running inside the embedded broker that maintains TCP connection with the remote end-host broker.

The SDN controller can serve as a platform for performing analytics by parsing the MQTT payload contents to retrieve topic and associated data value. For instance, a threshold-based analytics can be performed as follows: The data on the topic temperature can be detected for beyond safe-limit values. Other analytics include statistical analysis of the received data. We plan to incorporate such features in our future implementations.

4.2.2 Testbed Setup of MQTT Short Flow Aggregation

Our testbed environment consists of 2 PCs, and 4 switches (namely, the Mikrotik RB2011IAUS). The PCs run on Ubuntu OS 12.04, the switches run on Open-Wrt 15.05. The Fog node; containing integrated SDN controller is run within a

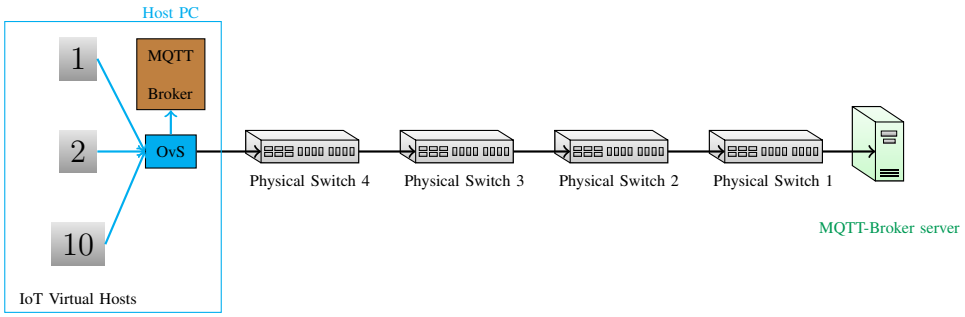


Figure 4.3: MQTT Network with Fog Node Testbed Setup. *Each element inside 'Host PC' is run as virtual machine. 'MQTT-Broker' and 'OvS' represent the same architecture as shown in Fig. 4.2, but running on Ubuntu OS.*

virtual switch environment, namely Open vSwitch (OvS) v10.0.

For scalability reasons, and from high computational resource perspective, we ran IoT devices (MQTT publishers) as a set of virtual hosts, and the Fog node as virtual devices both deployed in a single physical PC. The testbed environment is shown in Fig. 4.3. In our experiments, we ran 10 Mininet virtual hosts as MQTT publishers, 1 MQTT broker (Mosquitto - an open-source broker implementation [68]), and 1 virtual switch using OvS. However, without loss of generality, our implementation can run on physical switches and real devices.

Our network consists of 4 physical switches in a line topology with a physical MQTT broker run by an end-host PC. Each of the physical switches 1 to 4 run a respective instance of OvS that manages two physical ports, namely the input and output ports. We considered the third switch, namely '*Physical Switch 3*' as the bottle neck by controlling delay, bandwidth, and packet loss probability; through emulation using Network Emulator (NETEM). The internal Fog node Broker

receives and processes all of the publisher's request- and data messages; and send appropriate feedback. The integrated controller apart from forwarding messages can also be used to perform In-House analytics (by providing features for parsing MQTT message's payload). For analytical tractability, we used Bernoulli loss model (with loss probability p) in our experiments. Therefore the loss probability function $\mathbf{Pr}(\mathbf{S})$ is given by

$$\mathbf{Pr}(S) = 1 - (1 - p)^S. \tag{4.1}$$

Fig. 4.4, shows the throughput results of respective UDP and TCP MQTT clients. For both the cases, the throughput follows the trend roughly close to the analytical throughput. In contrast, the throughput for the same experiments without Fog node is zero; because the input MQTT traffic is significantly higher as it could not compete to establish connections with a physical end-host broker. Therefore, it is clear that for large scale IoTs the Fog node is essential for transportation. For the traditional setup (without Fog node), with native MQTT clients connected to the remote end-host broker, for similar input traffic configurations, the connections were not stable. For initial few moments of time, the throughput reached up to 250 messages per second (which is lower than the Fog nodes throughput performance. Subsequently, the throughput was close to 0, due to the failure of (large number of) TCP handshaking processes attempting to establish connections with the broker.

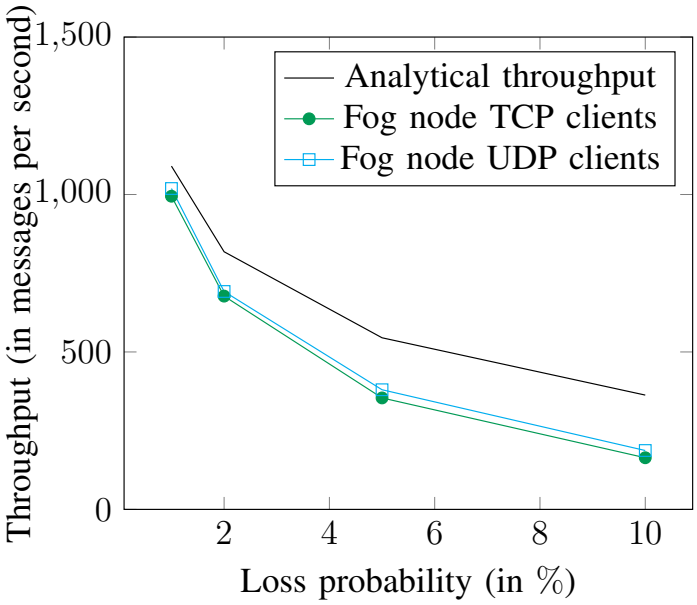


Figure 4.4: Throughput performance for respective UDP and TCP clients, with Fog node computing

4.2.3 Delivery Throughput Analysis of Fog Node

The Fog node maintains a TCP connection with the end-host broker and transports the IoT (MQTT-clients') published messages that are stored for future exhaustive analytics. We are now interested in throughput of the published messages sent by the Fog node in the considered system. Our system model comprises of 10 end-hosts trying to establish 250 TCP sessions over wired connections to the edge switch (*i.e.*, Fog node). Subsequently, the Fog node uses TCP Reno connection to connect with the end-host broker for transferring published messages. Each MQTT publish message is considered to use the entire Maximum Segment Size (MSS) of the underlying TCP transport. Therefore, the TCP throughput computation is sufficient to get the throughput of MQTT messages. Without loss of generality,

the MQTT throughput is a function of TCP segments' throughput.

IoTs are naturally deployed in large scale and the data generated is considered to potentially fall in the regime of 'Big data'. As a result of continuous data being transported in the network, the TCP connection (at the Fog node) is long-lived and therefore it is sufficient to analyze the throughput of this TCP in its congestion avoidance phase. A long-lived TCP Reno's congestion window (in packets) can be modeled by Markov chain [75].

Let the congestion window S denotes a total number of S publish-messages being transmitted by the Fog node. The possible congestion window values are finite, and are given as $\mathrm{E} = \{1, s, S_{\max}\}$; where $S_{\max}$ is the maximum congestion window at the sender (Fog) node. The transition matrix $\mathbf{T}$ of the Markov chain representing the congestion window size (on each RTT instant) is denoted by [75]:

$$\mathbf{T}_{S,S'} = \begin{cases} 1 - \mathbf{Pr}(S), & \text{if } S' = \min(S+1, S_{\max}) \\ \mathbf{Pr}(S), & \text{if } S' = \max(\lfloor \frac{S}{2} \rfloor, 1) \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

where $\mathbf{Pr}(S)$ represents the loss-probability at the congestion window is of size S. It is reasonable to assume that the Markov chain is irreducible and aperiodic. A classical result on Large-Deviations Principle (LDP) states that an irreducible and aperiodic Markov chain, with finite state space, holds a large deviations spectrum as given below [75]:

$$\lim_{\epsilon \to 0} \lim_{N \to \infty} \frac{1}{N} \log \mathbf{Pr}(\overline{S}^{(N)} \in [\alpha - \epsilon, \alpha + \epsilon]) = f(\alpha) \tag{4.3}$$

where, $\overline{S}^{(N)} = \frac{1}{N} \sum_{i=1}^{N} S_i$ is the sample mean $\overline{S}$ congestion window size scale N,

and $f(\alpha)$ is the large deviations spectrum as representing the Legendre-Fenchel transform of the logarithmic moment generating function Λ, and is given below:

$$f(\alpha) = \inf_{q \in \mathbb{R}} (\Lambda(q) - \alpha q)) \tag{4.4}$$

For our context, the large-deviations spectrum $f(\alpha)$ can be computed from the Markov chain transition matrix $\mathbf{T}$. The authors in [77] have shown that the large deviations spectrum can be obtained as the spectral radius'(ρ) logarithm of the matrix $\mathbf{R}(q) = \exp(qj)\mathbf{T}_{ij}$. Hence,

$$f(\alpha) = \inf_{q \in \mathbb{R}} (\log \rho(\mathbf{R}(q)) - \alpha q) \tag{4.5}$$

$$\lim_{\epsilon \to 0} \lim_{n \to \infty} \frac{1}{n} \log \frac{\#\{i \in \{1, \cdots, x_n\} : \overline{S}_i^{(n)} \in [\alpha - \epsilon, \alpha + \epsilon]\}}{x_n} = f(\alpha) \tag{4.6}$$

Eq. 4.6 can be equivalently represented as below:

$$\frac{\#\{i \in \{1, \cdots, x_n\} : \overline{S}_i^{(n)} \in [\alpha - \epsilon, \alpha + \epsilon]\}}{x_n} \underset{1 \ll n \ll N}{\sim} e^{nf(\alpha)} \tag{4.7}$$

The probability function in Eq. 4.3 refers to fractions observed over large number of *independent* realizations (*i.e.*, multiple independent TCP flows). In our study, we are dealing with a single TCP flow between the Fog node and the end host.

Interestingly, the authors in [78] have shown an ergodic form of LDP to hold on almost every realization. Considering a single realization of finite-size $(S_i)_{i \in \{1,s,N\}}$, and its mean at scale n. In other words, the single realization of size N is considered as parts of x_n consecutive intervals of size n where $x_n = \lfloor \frac{N}{n} \rfloor$. The mean over i^{th} interval is given by, $\overline{S}_i^{(n)} = \frac{1}{n} \sum_{y=(i-1)n+1}^{i} S_y$. The result in

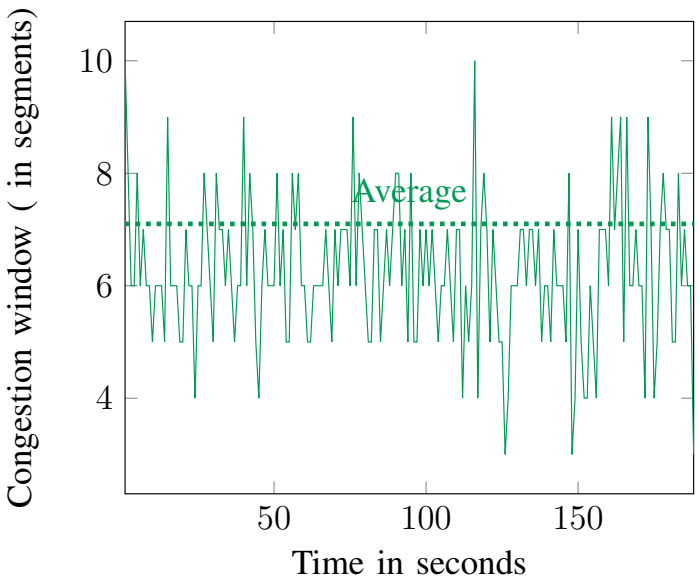


Figure 4.5: Congestion window size instantaneous vs average

Eq. 4.6 holds for almost every (single) realization [75], for a large value of x_n. An essential property of large-deviations spectrum is that it is independent of scale n, and satisfies the following:

$$\begin{cases} \text{if } \alpha = \overline{S}^{(\infty)} & \text{then } f(\alpha) = 0, \text{and} \\ \text{else if } \alpha \neq \overline{S}^{(\infty)} & \text{then } f(\alpha) < 0. \end{cases} \tag{4.8}$$

Therefore it is clear from Eq. 4.7 that the throughput remains close to mean $\overline{S}^{(\infty)}$ and the probability of being in all other values would exponentially degrade. Figure 4.6 validates the throughput at Fog node obtained from the testbed, against the analytical throughput obtained satisfying $f(\alpha) = 0$ in Eq. 4.8.

4.2.4 Conclusion

In this work, we have proposed an SDN-based Fog computing architecture and developed its working prototype. Subsequently, we have mathematically studied

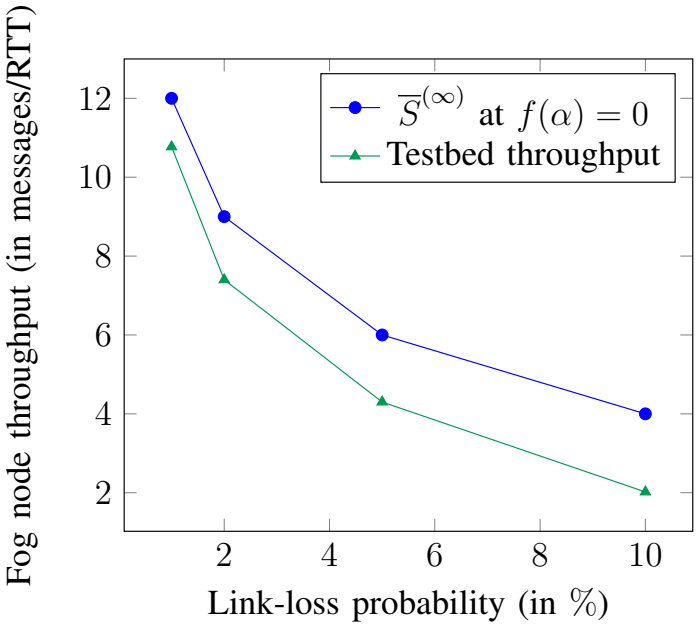


Figure 4.6: Average throughput for different loss probability

the throughput offered by the Fog node, and our experimental results tend to follow roughly in-line with the analysis. It is also demonstrated that the Fog node delivers at a significantly higher throughput, as compared with the respective traditional client and end-host setup.

4.3 Long Flow Aggregation of MQTT Protocol[73]

Though fog networks offer better delivery performance than a traditional cloud-based network, from the IoT analysis perspective, we show that a naive approach of using IoT analytics atop a fog network is not sufficient enough to offer best delivery results. Therefore, in this work, we focus on a critical issue of identifying and eliminating the heterogeneous delays of different IoT flows, in the native fog network setup. In our study, we found that the underlying network transport-layer

typically designed for connecting remote systems (as used in cloud-based network models) would counteract in a fog network scenario, and cause unfairness among different IoT clients. This causes a significant delay in performing IoT analytics, at the fog node.

Having identified the key components that contribute to the delay for fog-based network elements, we further propose an augmented transport layer based framework that provides fairness among IoT clients. To the best of our knowledge, for the first time, we address the fairness issues in the fog networks and propose network solutions to enable the IoT clients to deliver improved and fair throughput performance.

4.3.1 System Model of Long Flow Aggregation in SDN Docker

For our study, we use similar testbed setup to short flow aggregation study. We use MQTT in long flow TCP connection with QoS 0 configuration to prevent extra acknowledge transmission. Moreover, as the MQTT application functions over TCP transport in the network, the reliability is ensured. The application-user that needs the final computed data can participate (as a client) by running as MQTT subscriber connected to the fog node (i:e:; MQTT broker). N number of IoT clients connect to the fog node (at the edge of the network), as well as to the cloud server over the network. The different IoT client flows are aggregated in the fog node, before being transported to the remote cloud server. Without loss of generality, we have included this cloud node (as subscriber node) to get the results analyzed by the fog node. Though this is not an essential part of the network,

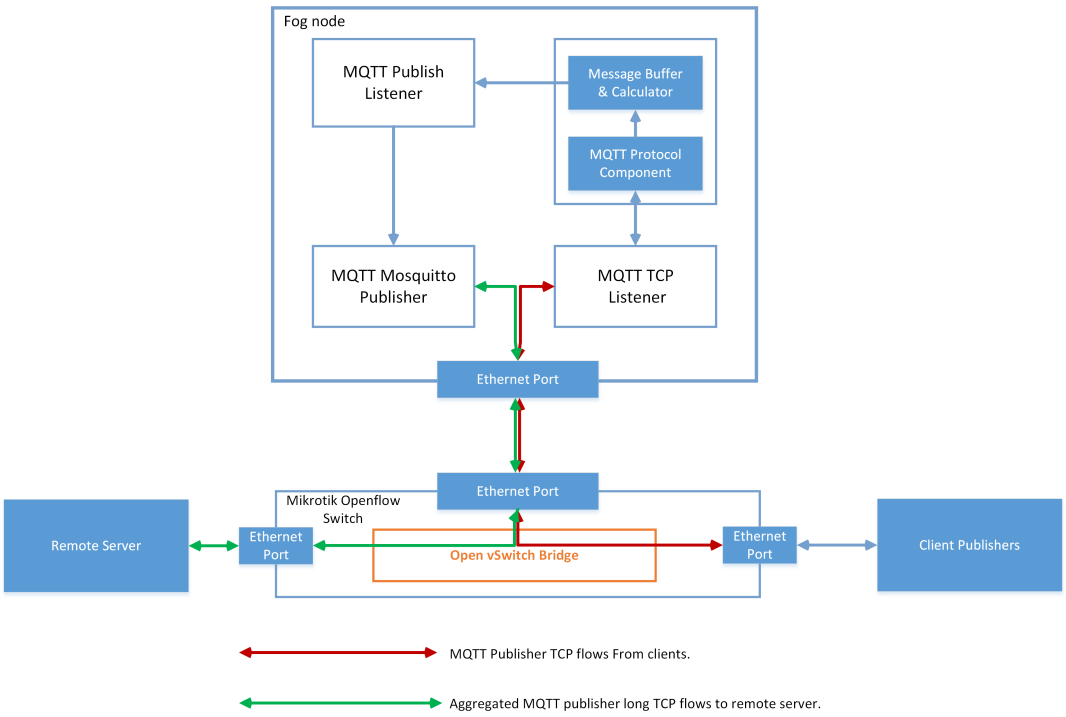


Figure 4.7: Fog nodes internal architecture in long flow aggregation

for the sake of completeness, we have used a generic network architecture that also encourages the users to connect to the fog node through the networks such as Internet.

Every MQTT publisher device that generates data will send it to the MQTT broker device. Each transmitted message will be associated with a special MQTT field known as topic. For instance, if the publisher publishes temperature data to the broker, the possible topic say temperature can be associated with the published data.

4.3.2 Framework Development of MQTT Long Flow Aggregation

Figure 4.7 shows the internal architecture of the fog node that we have used for our study. We have used the IBMs Really Small Message Broker (RSMB), as the light-weight MQTT server for the fog node. The IoT clients (MQTT publishers) are wire connected to the fog nodes RSMB broker through an edge switch. The Message Buffer and Calculator module will buffer the clients publish messages and compute the analytics necessary for the application. Without loss of generality, we have used averaging of received packets as the computing function. Each MQTT published message from the clients is time indexed with a sequence number. We do this by having a 4-bytes of MQTT application payload representing a sequence number, and another 4- bytes containing the actual message. The fog node performs the average computation by using these sequence numbers associated with the messages.

Fig. 4.8 shows the experiment network structure used for evaluation. N = 9 hosts (i:e; IoT clients) are connected through bridge switches to the fog node which is connected via an edge OpenFlow switch. The link to the edge switch has the bottleneck bandwidth of 100Mbps. To have a seamless functionality, the hosts are unaware of the internal fog nodes IP address, they typically assume a traditional cloud based architecture and are aware of the remote brokers IP address. Through SDN-based flow steering, we internally modify the incoming flows destination IP address to the fog nodes IP address. In this manner, we achieve a transparent fog node analytics in our network architecture.

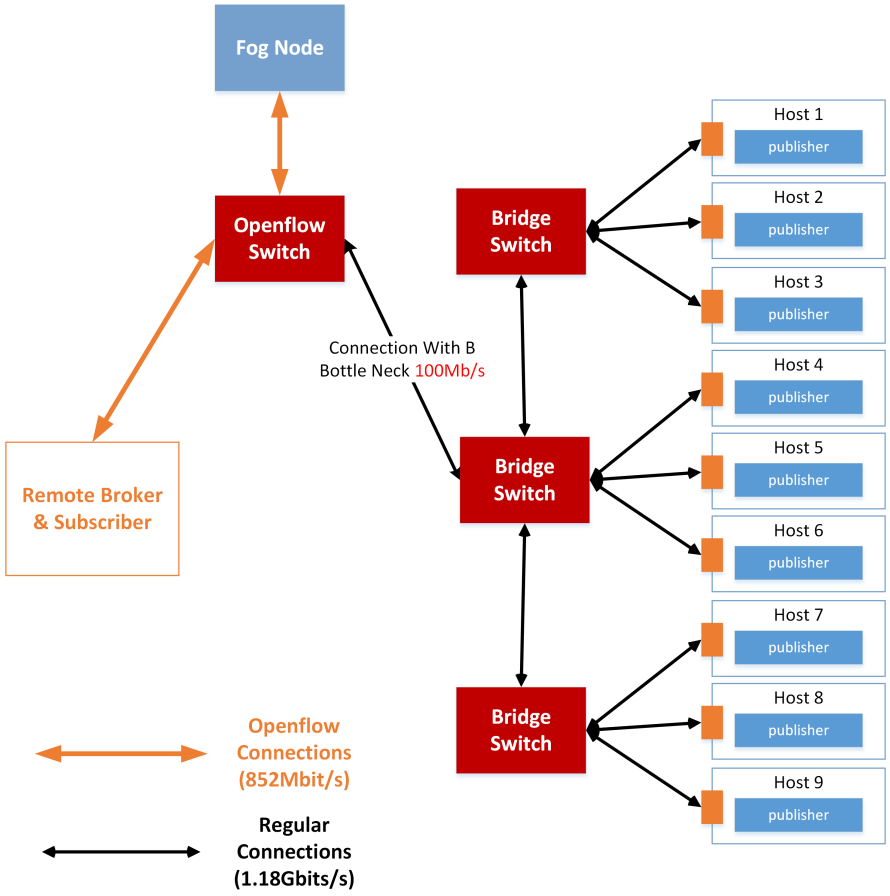


Figure 4.8: Experiment network structure

4.4 Performance Evaluation of MQTT Long Flow Aggregation

According to the introduction of MQTT architecture, we can notice that the data transmission from aggregated gateway to broker server, which is over long flow TCP connection in transport layer, is also important to the network performance. In this section, we extend the framework to TCP long flow use case. By introducing the metric of fairness index, we reveal the relation between the fairness among clients and the congestion window size in TCP connection. Meanwhile, by improving the fairness, we will show an improvement in aggregated data flow with an data processing procedure in aggregate node.

4.4.1 Evaluation Study of Traditional Fog Network

Study of Traditional Fog Network Using the custom-made network architecture described in the previous two sections, we study the performance of a setup that uses a naive approach of using MQTT-based communication in fog network, we henceforth call this naive setup as traditional fog network. The throughput performance of the traditional fog network is shown in Fig. 4.9. The bottom portion of the figure shows the individual IoT clients throughput, and the top red line-plot shows the total received throughput in the network. It is clearly observed that the individual instantaneous throughput of each client is not fair on the network. As a consequence, the Calculator Module which performs analytics needs to wait for an extended time until all the clients send their generated messages

(with the same time-index). This depends on the delivery delay of the most unfairly treated IoT client in the network, at that instant of time. Therefore, it is paramount to ensure fairness to all of the clients connected to the fog node, which reflects on the improved delivery performance, and faster analytics computation time.

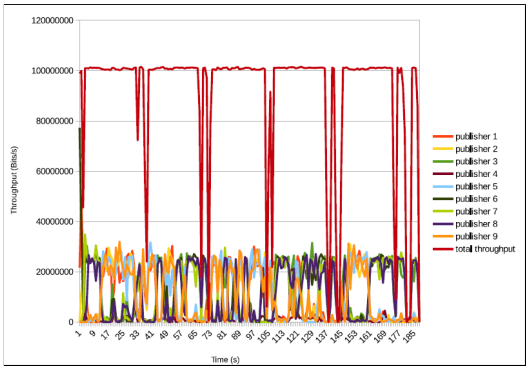


Figure 4.9: Traditional DTN-based WLAN offloading scenario with four infrastructure nodes, and one mobile node.

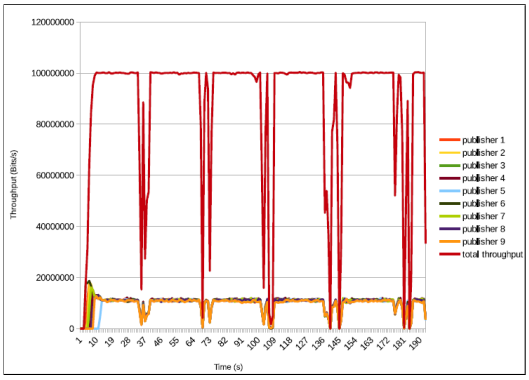


Figure 4.10: Proposed SDN-based DTN offloading framework with four infrastructure nodes, and one mobile node.

A deeper inspection of our experimental results revealed a huge number of retransmissions being transmitted at the transport layer of the network. The result is retransmission attempt is shown in Fig. 5. As evident from the figure, a substantial amount of communication resource has been wasted in retransmission. Even though we have used wired links typically used in static IoT network settings, the queue loss at the bridge switches and edge switch caused significant packet drops that triggered retransmissions at the transport layer of the network.

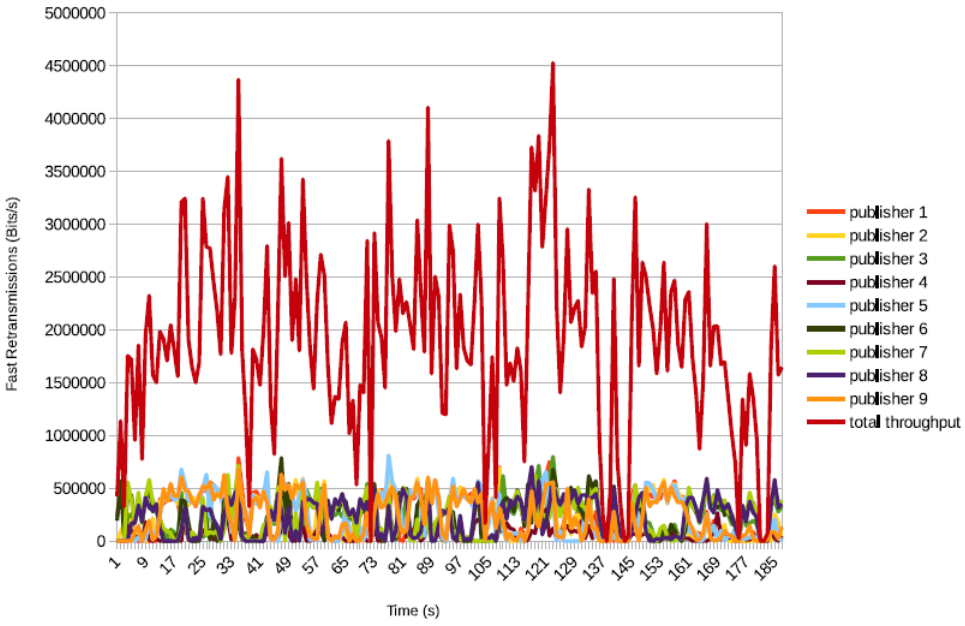


Figure 4.11: Fast retransmission throughput of the traditional fog network.

4.4.2 Proposed Framework and Performance Evaluation Study

From Fig. 4.11 it is evident that the native congestion control mechanism of the TCP has created the negative impact on the fog communication. With the idea of fog network architecture, that enables communication of IoT clients a proximate fog server node that is very few hops away can be given a preferred stop and wait fashion of transmission at the transport layer than with regular congestion control mechanism.

We, therefore, in our framework, we virtually made the transport layer insensitive to the congestion control algorithm, and instead we enabled the transport layer to work in a stop-and-go fashion of sending each segment (or equivalently MQTT messages) only after positively receiving ACKs for the previously sent segment. This simple mechanism enabled to have a strict fairness in the system,

and our proposed framework in the fog network setting showed almost perfect fairness, while still utilizing the same total throughput as used by the traditional network. Fig. 4.10, shows the throughput performance of our proposed framework in the same fog network setting. As evident from the bottom portion of the figure, we observe an almost perfect fairness for every individual IoT client in the network. The Calculator Module in the fog node in our proposed communication framework received the same timeindexed MQTT messages almost instantly. To systematically quantify the fairness achieved by the respective traditional and proposed network frameworks, we used the popular Jains Fairness Index [52] as the fairness performance measure. The Jains Fairness Index [52] is computed as follows:

$$fairnessindex = \frac{(\sum_{i=1}^{n} E[Ti])^2}{n * \sum_{i=1}^{n} E[Ti]^2} \tag{4.9}$$

where E[Ti] is the throughput of TCP flow i and n are the total number of flows in the network. The fairness indices of 9 IoT clients for the respective traditional fog network and the proposed framework is shown in Fig. 4.12. As evident from this figure, it is clear that our proposed framework achieved perfect fairness 100% for most of the time. On the other hand, the traditional fog network severely suffered from fairness issues. To further understand the negative impact caused by the unfairness in the traditional fog network, we show the number of actually received MQTT messages at the fog node over the duration of the experiment, in Fig. 4.13. It is clear that the number of received messages in our framework is by several orders of magnitude higher than the traditional fog network that spent

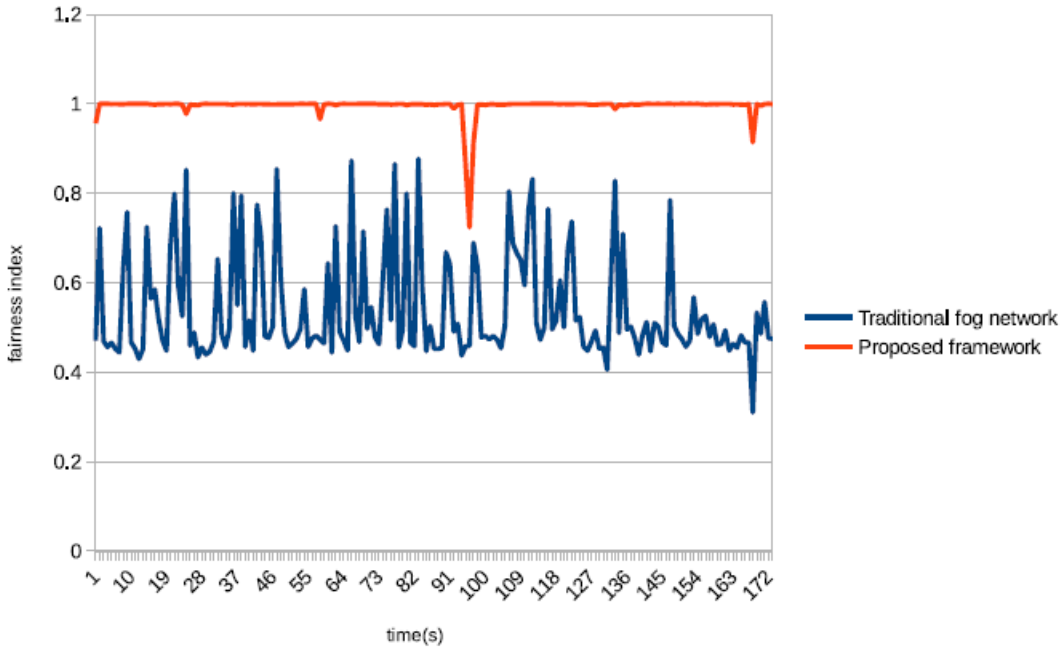


Figure 4.12: Fairness indices of traditional network vs proposed framework.

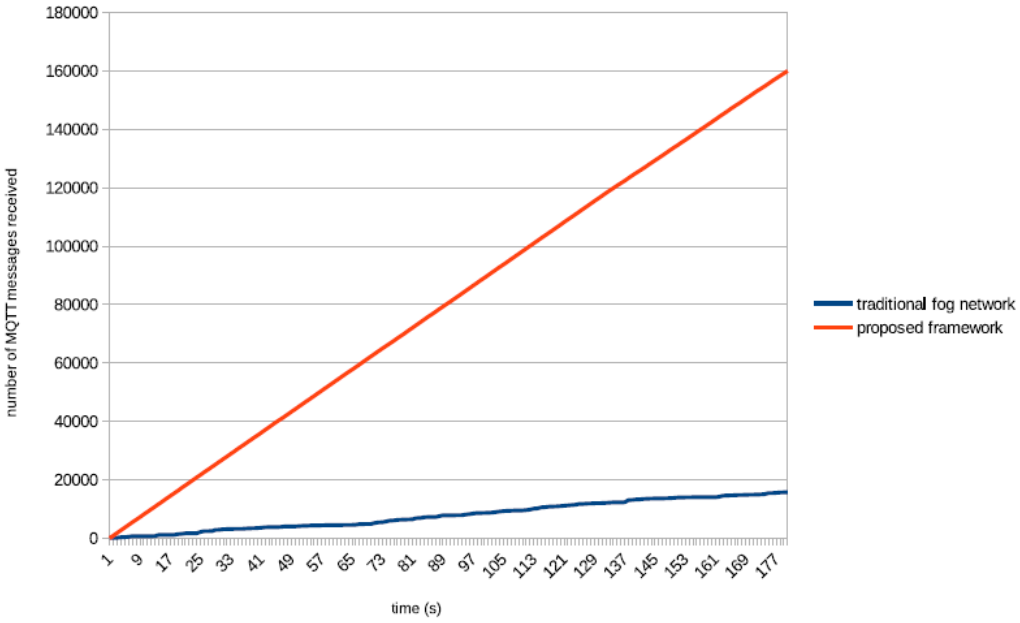


Figure 4.13: Total number of received IoT (MQTT) messages in the traditional network vs proposed framework.

substantial amount time and communication resources in retransmissions. We have performed all of these experiments on real testbed prototype. In the testbed, nine Raspberry Pi 3 [60] devices have been used as MQTT-based IoT clients that run on Ubuntu Mate 16.04 platform, with a modified version of Mosquitto 1.4.9 MQTT publisher. The fog node is a Ubuntu-based PC with Intel Core i7 2.4GHz, 8GB main memory, that runs an RSMB MQTT broker application.

4.5 Summary

In this work, we have proposed a Fog computing architecture hosted in SDN Docker and developed its working prototype. Subsequently, we have mathematically studied the throughput offered by the Fog node, and our experimental results tend to follow roughly in-line with the analysis. It is also demonstrated that the Fog node delivers at a significantly higher throughput, as compared with the respective traditional client and end-host setup.

With the help of real testbed experiments, we also have extensively studied the fog network performance and identified the unfairness among the IoT clients as the key bottleneck that significantly degrades the performance of the network. To this end, we devised a novel framework to restore fairness among the IoT clients and thereby achieved significant improvement in the delivery throughput performance of different flows to the fog node.

Chapter 5

SDN Based Opportunistic Networking in Internet of Things [72]

Inspired from Interplanetary Internet, Delay-Tolerant Networks (DTNs) have been envisaged to provide opportunistic communications in terrestrial application-scenarios that exhibit intermittent and/or disrupted connectivity [53]. Example application scenarios include extending Internet from connected urban places to remote disconnected rural areas, through DTN data mules. The domain of DTN-based applications is growing to much broader context. Of late, DTNs find potential use in new application sectors such as infrastructure offloading [54]. To this end, cellular offloading is a promising area of research that leverages mobile P2P connectivity (such as LTE machinetype communications) to conserve spectrum resources.

5.1 Introduction to Delay Tolerant Network (DTN) Implementation

As a part of Internet Research Task Force (IRTF), the Delay-Tolerant Networking Research Group (DTNRG) focus on interconnecting highly heterogeneous networks together even if end-to-end connectivity may never be available. Examples of such environments include spacecraft, military/tactical, some forms of disaster response, underwater, and some forms of ad-hoc sensor/actuator networks. It may also include Internet connectivity in places where performance may suffer

such as developing parts of the world [66].

As the reference implementation from DTNRG, DTN2 focus on components architecture and functionalities as a DTN node. DTN2 use Bundle Protocol [48], a general overlay network protocol to encapsulate messages exchanged between nodes. The Bundle Protocol Agent and all its support code are implemented as a user space daemon called "dtnd". The daemon has a configuration and control interface which can be run remotely over a TCP connection when the daemon is running 'daemonized' (i.e., without a control terminal).

Following the implementation in DTN2, IBR-DTN has implemented as an optimized light weight DTN implementation for low power systems. It could run on any Linux based system including the OpenWRT we introduced, which makes the IBR-DTN can be deployed on network devices.

5.1.1 DTN2 Implementation

DTN2 has a fairly comprehensive set of DTN functionality, including the application API, support for custody, initial support for some of the DTN security protocol, return receipts, a number of convergence layers including TCP, Bluetooth and LTP (Licklider Transport Protocol) and an extensive collection of routing protocols including the ones we introduced in chapter 1. Optimization still needed in some components design and implementation. DTN2 uses persistent storage to maintain state when the daemon is stopped so that bundles and other information can be reloaded on restart. Various storage mechanisms can be configured, including a file system, in-memory and multiple database interfaces like Berkeley

DB, MySQL, and SQLite.

5.1.2 IBR-DTN Implementation

IBR-DTN following the reference architecture of DTN2. It introduces an event scheduler as the kernel of the DTN node. All the operations in modules like new neighbor discovery and bundle exchange with neighbors are considered as an event and always operated by the event scheduler first. Modules are threads forks from the daemon program. They communicate with the event scheduler by wait/notify API. When an event raised by a module, event scheduler decides which module the event should be forward to.

Comparing to DTN2, IBR-DTN removed the persistent database API which is not applicable in an low power device. The develop libraries are also relied on plain glibc[76] which is applied in DTN2. However, the common features like Bundle Protocol Agent, Convergence Layers, Beacon Module and Persistent Modules are still following the design of DTN2.

5.2 Flexible Packet Forwarding Scheme For DTN

In the current era of crowdsourced network participants, a communication paradigm must allow IP-agnostic forwarding to encourage any participant to seamlessly join or leave a crowdsourced network system. Thanks to the DTNs epidemic style of forwarding which is inherently crowd-source friendly, by enabling content-based forwarding capabilities to the nodes. However, a deeper understanding reveals underlying challenges and inefficiencies that need to be

addressed:

Not crowd-friendly multi-hop forwarding: While DTNs epidemic forwarding can help to communicate with a mobile node with any IP, multi-hop infrastructure nodes on the other hand still might not be able to communicate with (more than two-hops away) mobile node as they need IP address of the mobile node to build their routing/forwarding table.

Flooded Infrastructure: Crowd-friendly epidemic DTN routing can populate the infrastructure network nodes with an unnecessary redundant exchange of messages leading to network overload.

Recently, mobile crowdsourcing is gaining traction, thanks to the advanced P2P-based wireless technologies such as WiFi Direct, and energy-efficient Bluetooth 4.0. DTNs store, carry, and forward communication paradigm can be well suited to the store, compute, and forward strategy of mobile crowdsourcing. All of these promising new application scenarios, require a revival of the present DTN architecture to seamlessly and effectively support communications with minimal resource utilization.

DTN has been considered as a promising candidate for infrastructure offloading such as cellular offloading [55], [56], [57]. These works encourage the use of DTN to offload traffic from the infrastructure. However, to the best of our knowledge, a unified DTN architecture that supports effective communication in a crowd-source compatible offloading environment is not addressed in the literature.

The research related to SDN and DTN is a relatively unexplored and new. The authors in [58], consider an intermittently-connected vehicular ad hoc network

scenario and propose an SDN-based routing framework for efficient message propagation in the considered network. A central SDN controller gathers information about the vehicular nodes, and thereby enable the controller to compute global routing strategies. While the scenario is based on Vehicle to Infrastructure (V2I) framework, the roadside base stations are connected to a central controller. In this work, the authors consider a lightly-coupled SDN, wherein the SDN control plane is utilized effectively forward DTN bundles. On the other hand, our work deeply-integrates SDN, by forming a core part in the DTN architecture, that enables flexible DTN communication.

5.3 Contributions in This Work

This work highlights the areas of improvement required in the current DTN architecture and motivates the need for a flexible (or dynamic) forwarding paradigm that is well suited for infrastructure offloading, as well as mobile crowd sourcing application scenarios. The naive approach of enabling infrastructure (say, WLAN) offloading through DTN can be possible by enabling edge nodes to have DTN functionality. Such primitive approaches enable the infrastructure nodes to offload data to mobile DTN nodes in a limited P2P fashion i.e., between a mobile node and its proximate infrastructure node. An interior infrastructure node cannot have the opportunity to directly contact such mobile nodes. However, a trivial solution to provide offloading access to interior nodes is to enable multicast-style (many-to one) offloading possible through IP-routing inside the infrastructure,

and by modifying the DTN beacons TTL field to propagate multi-hop inside the infrastructure.

In this work, we address the aforementioned challenges through novel SDN-based flexible forwarding techniques that enable multi-hop infrastructure nodes to communicate the mobile node, even without requiring to know the mobile nodes IP addresses. Thanks to the SDNs programmability feature [29], we can also eliminate unnecessary redundant message exchanges among the infrastructure nodes. With powerful deep-packet inspection capabilities of SDN, we were able to provide a scalable and efficient solution of containing the redundant DTN messages exchanged by the infrastructure nodes. Apart from infrastructure network applications, we also examine ad-hoc mobile vehicular networks and apply SDN based Layer-2 forwarding to vehicular DTN nodes in order to reduce delay in vehicle platooning applications.

5.4 Software-Defined DTN Infrastructure Offloading Framework

In this section, we consider a WLAN infrastructure offloading framework (with DTN nodes) that encourages crowdsourced mobile DTN nodes to participate in carrying infrastructure generated bundles[48] . The schematic diagram of the considered scenario is shown in Fig. 5.1. The mobile DTN node is considered from that crowdsource participant, therefore, the IP address of the node is typically assumed as not known to the infrastructure nodes.

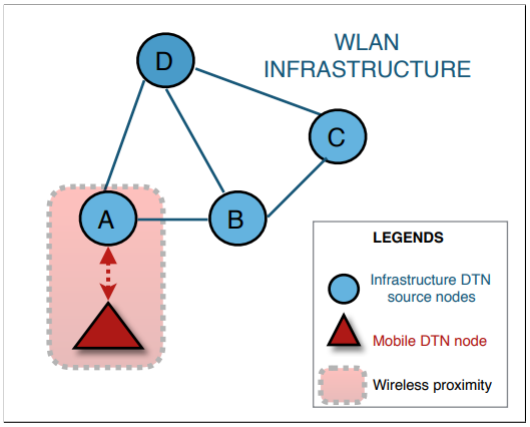


Figure 5.1: Traditional DTN-based WLAN offloading scenario with four infrastructure nodes, and one mobile node.

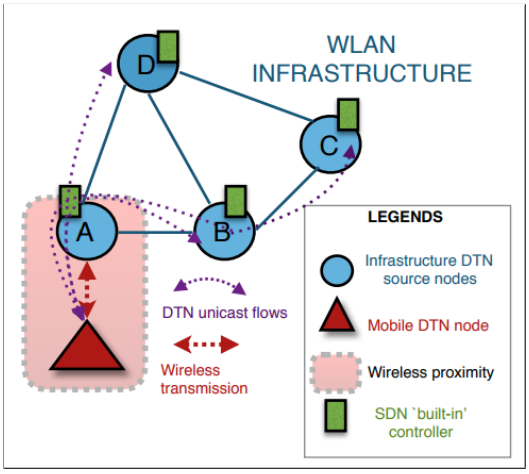


Figure 5.2: Proposed SDN-based DTN offloading framework with four infrastructure nodes, and one mobile node.

As shown in Fig. 5.1, the network consists of four infrastructure DTN source nodes, namely A, B, C, D, and a mobile DTN node. The infrastructure nodes are the DTN source nodes configured with epidemic forwarding. The infrastructure nodes are considered to have configured with IP-layer forwarding to enable communication among them. As a result of epidemic-style based DTN routing, and internal IP-layer connectivity, every source bundle generated in the network is flooded to all other DTN nodes in the infrastructure. This clearly causes unnecessary redundant bundle transmissions inside the network.

To avoid unnecessary internal transmissions, we propose an SDN-based infrastructure offloading framework that enables a unicast style-forwarding to the incoming regular mobile DTN node from a participating crowdsource user. The schematic diagram of the proposed SDN-based DTN offloading infrastructure

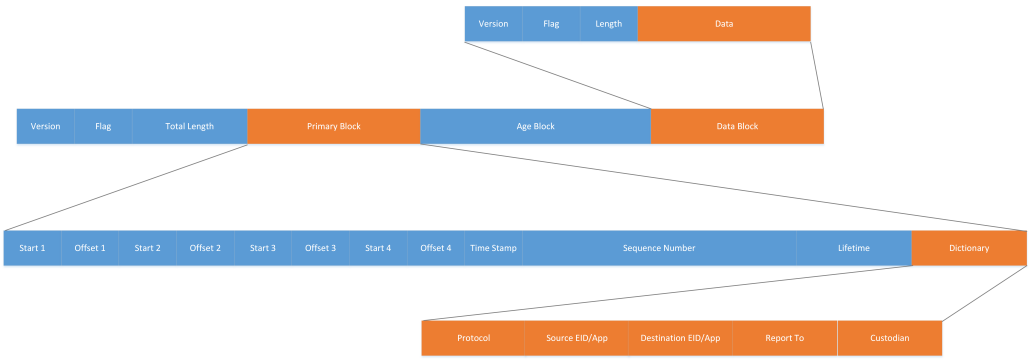


Figure 5.3: A frame with DTN bundle application payload. The in-built SDN controllers will perform deep-packet inspection on DTN primary blocks

framework is shown in Fig. 5.2. Each infrastructure DTN node is built with an integrated SDN controller functionality. Unlike traditional centralized external SDN controller for the nodes, we incorporate independent dedicated controller functionality in each node in the infrastructure. Individual SDN controllers are empowered with the deep-packet inspection.

The deep-packet inspection is performed by the SDN controllers at the Layer-2 of the OSI reference model. In our context, the deep-packet inspection is performed on the DTN bundle header fields of the received MAC frame, as shown in Fig. 5.3. This enables the SDN controllers to parse and differentiate various DTN control and data bundles. For instance, the controllers are equipped to differentiate the summary vector bundles(meta data of bundles used by epidemic routing) and, data bundles. In addition to deep packet inspection, our SDN controllers are also programmed to construct custom summary-vector bundles. When a mobile-host comes to the proximity of the infrastructure node, the inbuilt controller that

receives summary vector from the mobile host. The inbuilt SDN controller of the proximate infrastructure node, would in turn multicast the received summary vector to all other infrastructure nodes. This creates an artificial situation of other infrastructure nodes as if they have a mobile DTN node in their vicinity. When each infrastructure nodes send their summary vectors as a response, the reference infrastructure node with a real mobile node in its vicinity, would construct a compound summary vector comprising of all the individual summary vectors, and send it to the mobile node.

In this manner, a unicast style of communication is performed among the infrastructure nodes without flooding among each other. Therefore, our proposed SDN-based of- floating framework enables to use storage at the infrastructure nodes efficiently. It is worth to note that the summary vector bundle construction is a non-trivial process that involves bloom filter-based bundles list storage.

To also equip the framework to cater mobile crowdsourced nodes with unknown IP, our architecture also supports static routing wherein, the integrated SDN controllers can use fake internal IPs that can be substituted in the place of incoming mobile DTN nodes, which enables effective communication between static-routing configured infrastructure DTN nodes, and any crowdsourced mobile DTN nodes.

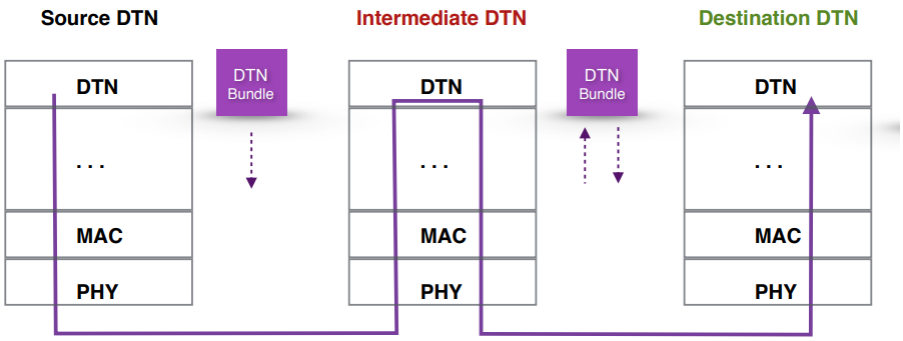


Figure 5.4: Three vehicles platoon scenario: Traditional crowdsourced P2P DTN forwarding

5.5 A Novel Software-Define Flexible DTN Forwarding Architecture

Having studied the efficacy of SDN-based DTN forwarding in DTN-based infrastructure offloading setup, we now turn our focus to regular DTN communication focusing on a P2P style of ad-hoc connectivity. We consider a use case of vehicular-based intermittently connected mobile networks. In particular, we consider a DTN based communication among a platoon of vehicles equipped with DTN nodes. Since this also reflects a crowdsourced fashion of participants, an epidemic or flooding routing scheme is considered as an appropriate routing strategy.

5.5.1 L2 Forwarding Scheme for DTN Bundle Forwarding

The traditional DTN-style of forwarding is schematically shown in Fig. 5.4, with three vehicles forming a tandem arrangement. The source DTN bundle is epidemically forwarded through a crowdsourced intermediate vehicle, to the

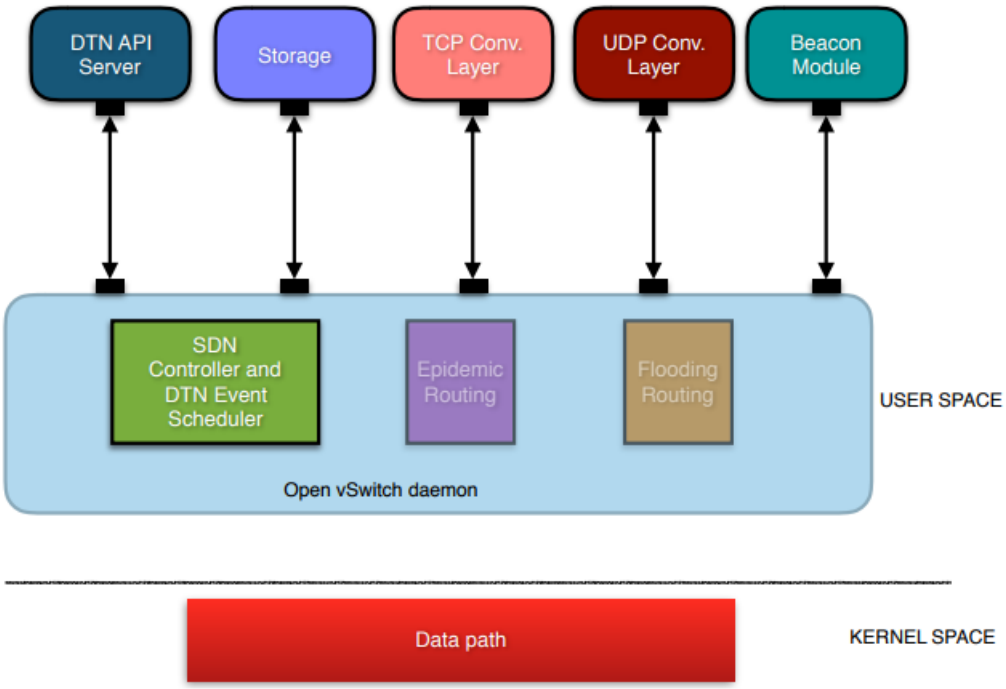


Figure 5.5: Proposed SDN-based DTN Architecture.

destination node in the tail end of the platoon. As the bundle is inspected for destination fields at the DTN application layer, the bundles have to pass through the entire stack of the intermediate nodes which causes additional processing delay in these nodes.

To overcome the DTN application layer processing at the intermediate nodes, we propose a new SDN-based DTN architecture built from the ground-up. The proposed architecture is shown in Fig. 5.5. The core of the architecture is built around the Open vSwitch daemon controlled by SDN controller through OpenFlow protocol. Unlike the traditional setup of external SDN controller, we integrated a built in controller running inside the Open vSwitch daemon in each node. In addition to the controller, the DTN daemon event scheduler, and various

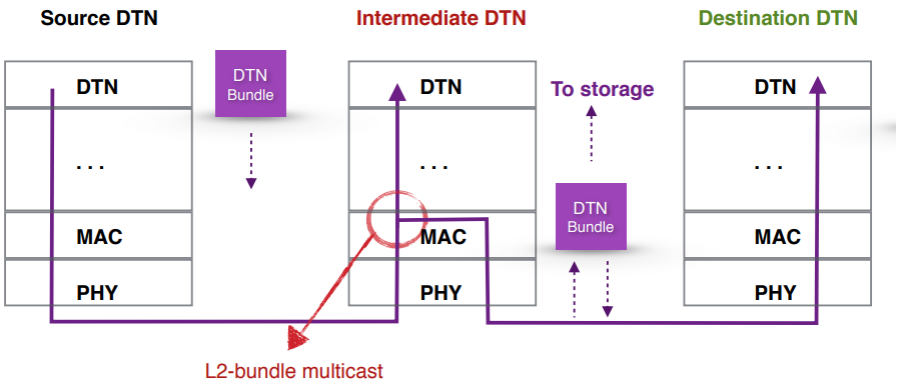


Figure 5.6: Vehicle platoon scenario: Proposed SDN-based Layer-2 (L2) forwarding, and parallel multicast forwarding to intermediate nodes storage.

routing modules are also integrated within Open vSwitch daemon. The other modules such as DTN application API service module, storage module (to store and retrieve bundles from secondary storage), convergence layers, and beacon generation modules are run as independent processes connected to the Open vSwitch through virtual network interfaces.

In this manner, we provide a modular connectivity of individual DTN modules interconnected with a powerful and flexible Open vSwitch based SDN controller at the core. As the Open vSwitch controller operates at the Layer-2 of the OSI network model, unnecessary application layer delays are avoided. Therefore, the intermediate DTN nodes can utilize the deep packet capability of the SDN controller to forward non-destined bundles to the neighbors. This substantially reduces the processing delays at the intermediate nodes. The forwarding style of the proposed SDN-based layer2 bundle transfer is shown in Fig. 5.6. The received beacons indicate the convergence layer choice of the incoming DTN node.

Therefore, with the SDN-controller at the nucleus of our proposed controller, we can dynamically configure routing and convergence layer appropriate to communicate with the neighbor nodes. Therefore, our framework is conducive to the crowdsourced environments wherein the neighbor DTN nodes configurations are not known apriori.

It is also worthwhile to note that, as the beacon messages are sent through out the contact, whenever the nodes in contact want to switch the convergence layer (based on the consuming applications demand), they can intimate the new convergence layer usage in the next beacon message. Our constant deep packet parsing controller, upon receiving this beacon message will dynamically switch to different convergence layer. In this manner, we flexibly operate with per-packet based heterogeneous forwarding.

5.5.2 Flexible On-the-fly Routing and Transport Services for Crowd-Friendly Environments

In addition to the L2 forwarding in vehicular platoon applications, our proposed modular architecture enables DTN node to identify the routing configuration of any proximity nodes through deep packet inspection. For instance, a received summary vector identifies the neighbor nodes epidemic routing feature. The Open vSwitch entity controls the packet forwarding and also manages different components in the DTN node. All the applications connect to the OvS entity via virtual Ethernet connection (veth). To avoid internal message exchange, each of these applications was bound to an IP address that belongs to a unique

subnet. This refrains the applications from talking to each other without passing through OvS entitys routing. The physical ports that connect to outside world (say, neighbor nodes) are also bound to the OvS entity.

The beacon module in the DTN nodes listen to all of the physical ports that bind to the OvS entity. The OvS entity listening on these ports, upon receiving a beacon message from one of the physical port, will multicast to routing modules and convergence layer modules as internal messages (with changed destination IP and port numbers appropriately).

The convergence layer component has two channels, a UDP port for exchanging messages with other components within OvS-DTN boundary, and another convergence layer is for the neighbors. The beacon message received from outside neighbor will indicate the convergence layer information. This allows the recipient DTN node to establish communication with the neighbor node with appropriate convergence layer, in a dynamic manner.

1) Sequence of Actions in SDN-DTN Upon meeting New Neighbor: All DTN nodes periodically emit beacon messages containing the local EID and convergence layer information

- Beacon message exchange: Upon receiving beacon message (through the physical port), the message is multicast to the difference convergence layer modules and routing modules.

- Convergence layer setup: According to the received beacon messages, for newly found neighbors; the convergence layer modules create sockets bound

to the modules local network devices.

- Bundle exchanges: Convergence layer receive bundles from neighbors. If the first bundle is a summary vector, it is forwarded to the internal epidemic/prophet routing modules. On the other hand, if the bundle is a regular DTN data bundle, the convergence layer will send the bundle to flood and static routing modules. In this manner, with the help of SDNs deep-packet inspection, the bundle were going to the appropriate routing modules.

Therefore, our proposed architecture makes dynamic onthe-fly routing decisions which is vital for crowd-sourced environments.

2) Handling Internally Generated Bundles:

- Bundle assembling: Locally received application messages are encapsulated into a DTN bundle with appropriate Destination and Source EIDs.

- Bundle transferring: Applications send the encapsulated bundle to API server module. The API server sends this new bundle to the routing modules for forwarding.

- Bundle Routing: Routing modules upon receiving the bundle, checks for the destination ID. And sends the bundle (according to the routing protocol).

5.6 Framework Performance Study

For our performance study, we developed and implemented our proposed architecture frameworks and tested on Mikrotik routers running OpenWrt 15.05. The end hosts serving as source and destination DTN nodes were usual PC with core i7 processors running on Ubuntu 12.04. The traditional DTN frameworks used for the comparative study were implemented with IBR-DTN [8], an embedded DTN2 open-source implementation architecture.

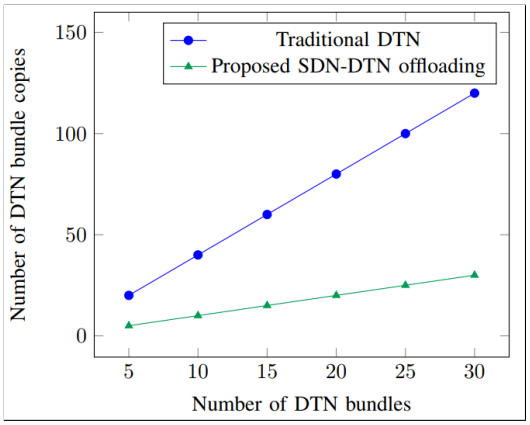


Figure 5.7: Offloading Application: Performance comparison of number of bundle copies in traditional DTN offloading and proposed SDN-based DTN offloading frameworks, in a four DTN node infrastructure network.

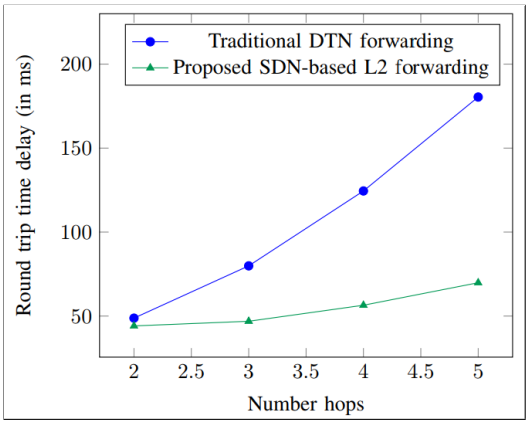


Figure 5.8: Vehicular Platooning Application: Improved delay performance of proposed SDN-based DTN layer-2 forwarding, against the traditional DTN forwarding.

Fig. 5.7 shows the performance plot for the use case 1, demonstrating the SDN-based offloading framework. It is clear that the proposed framework of

SDN-based DTN offloading effectively reduced the redundant bundles. On the other hand, with increasing source bundle generation, the increase in redundant copies in the network occurs in the traditional DTNstyle offloading scenario, in which case the total number of redundant bundles is directly the multiple of total nodes in the network. This clearly shows the fact that the traditional offloading infrastructure is not scalable. Fig. 5.8 shows the improved delivery delay achieved by the proposed SDN-based flexible DTN architecture, in the vehicle platoon application scenario. With powerful deep packet inspection capabilities, we can achieve a significant decrease in delay by bypassing over layer-2 forwarding at the intermediate nodes. Thanks to the SDNs deep packet inspection capabilities and dynamic forwarding support through programmable network flows. The performance improvement scales well with increasing number of hops in the network, which is quite intuitive.

5.6.1 A Novel Performance Evaluation Experiment

Typical existing DTN simulation experiments (such as ONE simulator) while conceptually represent the DTN node functionality, and mainly focus on the intermittent connectivity as the main component of functionality. To study the efficiency of our proposed DTN node architecture against the real-implementation of existing DTN architecture; we have developed a novel experimental design that represents the DTN node implementations and emulates the DTN network connectivity using the public connectivity traces available. In this manner, our experiments are one step closer to reality than the simulation experiments. At the

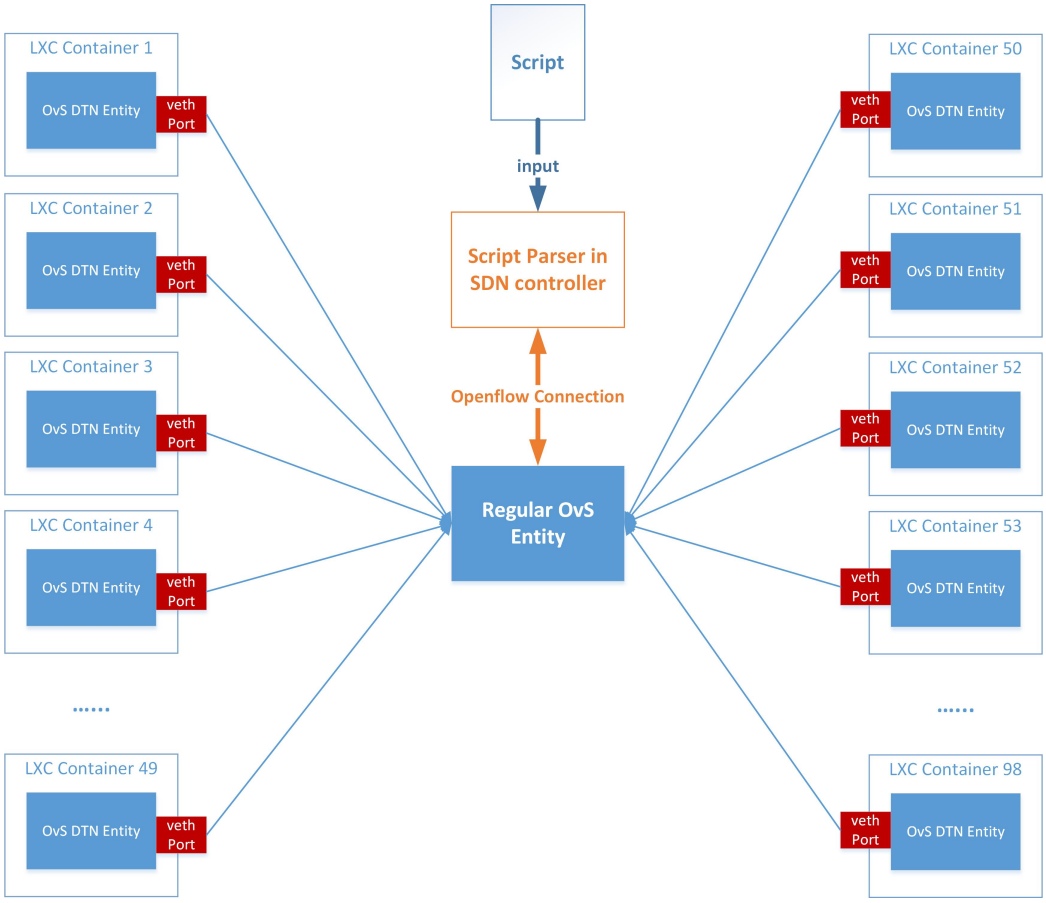


Figure 5.9: The testbed snapshot used in the implementation and performance evaluation study.

core of our emulation testbed, we use SDN controller as the central manager that accepts the mobility traces as the input. At the moment we support any ONE simulator compatible mobility traces in our experiments. However, without loss of generality, any mobility trace with node ID and connection and disconnection time information can be easy to ONE simulator format mobility trace with the help of a simple script.

The SDN controller manages a single Open vSwitch that manages communications between the DTN nodes. To ensure scalability, we implement DTN nodes

in LXC containers. A virtual interface added to the node (LXC container) would act as its network device. The central OvS device acts as a bridge with ports connecting the DTN nodes enclosed in LXC containers. We consider 98 LXC containers equipped with Ubuntu 12.04 LTS 32-bit OS, and the central OvS is based on Open vSwitch 2.3.90 version. The conceptual diagram of the emulation experiment is shown in Fig. 5.9. The entire emulation experiment runs on a single Physical PC with Intel Core i7 2400Ghz CPU, and with 16GB RAM, running Ubuntu 14.04 LTS 64-bit OS. 1) Connectivity Management: The connectivity is managed by the central OvS, wherein each DTN node (contained in LXC) is connected via virtual Ethernet interface (veth). We use SDN flows in the OvS to manage connectivity between DTN nodes. An incoming flow matching at OvS matching a DTN node X would be connected to the appropriate DTN node Y as dictated by the mobility trace script at that time instant. In this manner, the nodes connectivity is managed under the governance to the mobility trace script. 2) Opportunistic Connection Trace Script: Without loss of generality, in this work, we have used the Reality Mining trace-set [59] from the MIT Human Dynamics Labs. The Reality Mining trace in the ONE simulator format consists of rows of connecting and disconnect events represented as tuples. Each tuple consists of four elements namely, node X, node Y, absolute time in the experiment and connection-event (such as CONNECT/DISCONNECT). Hence, each row in the trace denotes the connect or disconnect event of a pair of DTN nodes. The trace consists of 98 nodes connectivity collected for 195 days. We found many instant connections, where the time period of node connections is 0 seconds. In

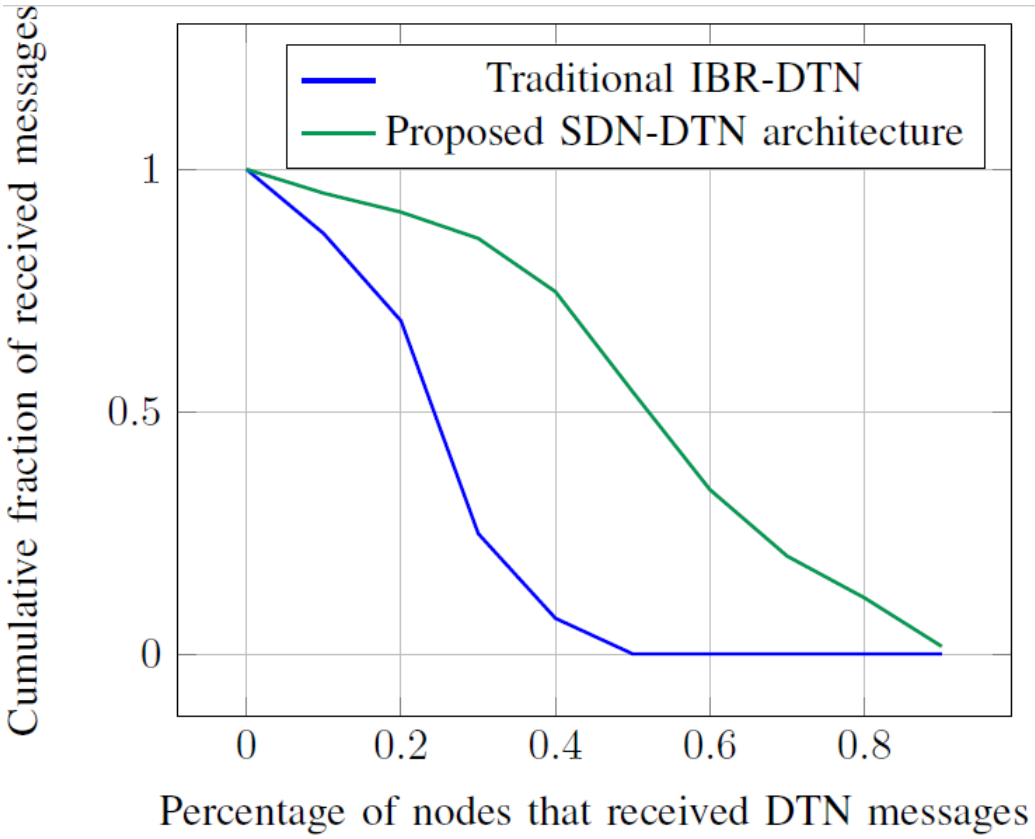


Figure 5.10: Heterogeneous DTN forwarding application: CDF Performance comparison of traditional IBR-DTN and proposed SDN-DTN offloading showing the number of messages received by fraction of nodes in the network

such cases, we set a random connection less than 1 seconds to utilize the instant node connections. The controller will create and send fake beacon messages to both nodes after a flow setup, to initiate connections between nodes. Without loss of generality, for our performance study, we compare the mixed DTN routing with 50% network nodes configured with epidemic routing and the rest with the flooding-based routing protocol. While in traditional DTN implementation (such as IBR-DTN) the heterogeneous nodes do not exchange packets thereby severely limit their precious contact opportunities. On the other hand, our proposed implementation used every possible contact present in the network. As evident in Fig. refFig:dtn11, close to 50% of nodes received more than 50% of the received messages. On the other hand, around 22% of the nodes received close to 50% of messages by the traditional DTN nodes.

5.6.2 Advanced Emulation with DTN nodes on RasperryPI

We took a more practical evaluation approach by using real DTN nodes installed on the RasperryPI devices. Due to limited (10) number of RasperryPI devices, we arbitrarily chose 10 nodes and their contacts from the MIT trace set. The corresponding performance plot Fig. 5.11 also shows the similar trend, thereby proving the efficacy of our proposed SDN-based DTN forwarding framework.

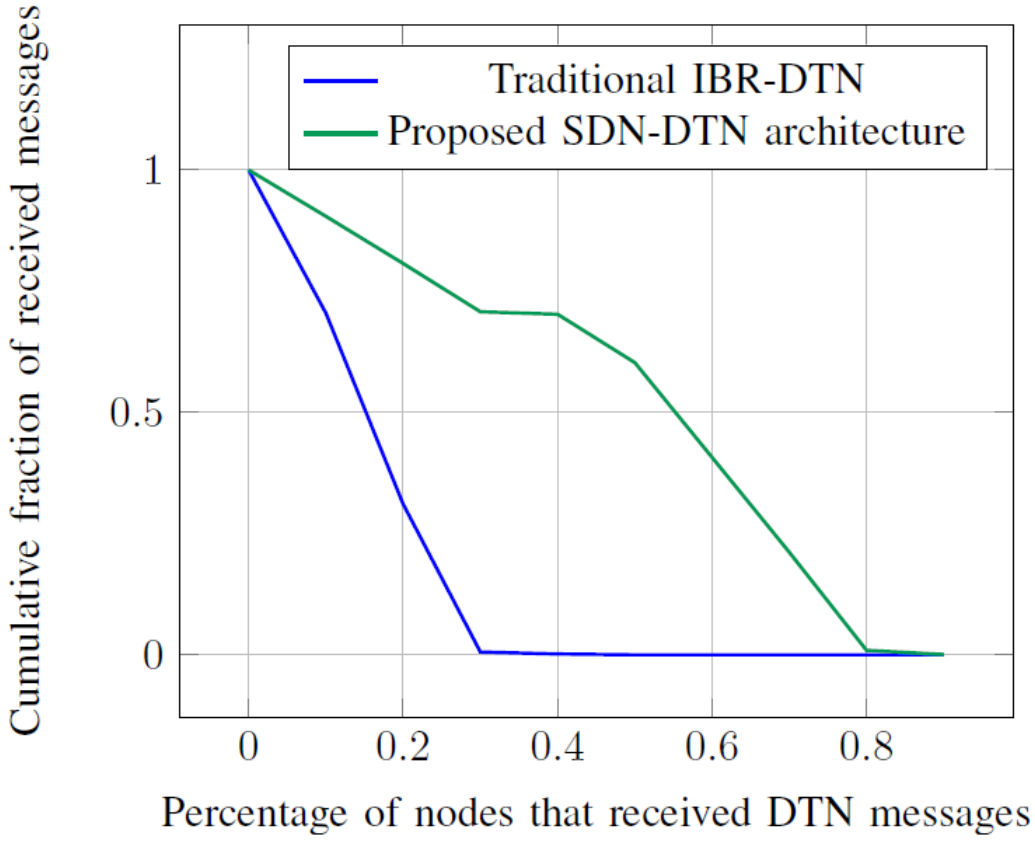


Figure 5.11: Heterogeneous DTN forwarding architecture: CDF Performance comparison of traditional IBR-DTN and proposed SDN-DTN offloading showing the number of messages received by fraction of nodes in the network, in a 10-nodes RaspberryPI experiment testbed

5.7 Internet of Hybrid Opportunistic Things [72]

5.7.1 Introduction to IoT and DTN Interconnecting

To extend connectivity over disrupted environments, the latest research [62], [63], [64] on Internet of Things (IoT) is focused on enabling IoTs to be connected to the Internet with the help networks such as Delay-Tolerant Networks (DTNs). For instance, to enable a delay tolerant IoT, the authors in [62] propose and implement DTN Bundle Protocol (BP) binding for IoTs Constrained Application Protocol (CoAP). This implementation [62] embeds DTN stack into the device, and the IoT application interacts with the integrated-DTN through a custom developed API, thereby creating a paradigm of IoT-over-DTN architecture. The authors in [63] consider a light-weight DTN BP protocol custom tailored for hardware-constrained IoT devices. The authors in [65] extend AllJoyn, a D2D-based communications framework with custom opportunistic communications.

The works such as [62], [63] that rely on Delay-Tolerant transport model as communications substrate would not benefit from IoT communications typical semantics such as the publish-subscribe model of forwarding. As a consequence, delivery performance of the IoT messages depend on the factors related to the DTN transport such as routing, and other (non-IoT) DTN bundles carried by the regular DTN nodes.

As an example application, we consider a practical urban mobile environment encompassing static IoT sensor nodes (publishers) deployed geographically over a region. A crowdsourced DTN mule provides opportunistic connectivity to the

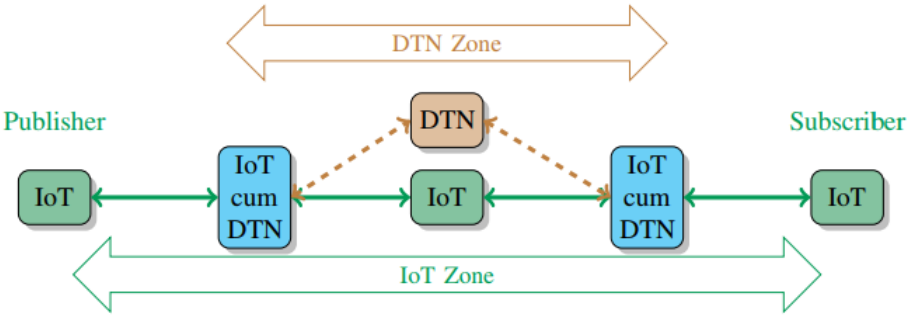


Figure 5.12: Proposed IoT-cum-DTN framework.

deployed IoT publishers by carrying IoT messages in the form of bundles and delivers to a set of remote IoT receiving devices (subscribers) that are connected to the Internet. The DTN mule is crowd-sourced to carry any DTN traffic in the region of interest (including non-IoT traffic). In such scenario, an end-to-end DTN transport approach such as [62] can suffer from reduced delivery throughput of IoT messages, as the deliverer (i.e., DTN mule) is agnostic to the IoT messages encapsulated in the DTN bundle as a payload.

5.7.2 A Novel Framework for IoT and DTN Interconnection

To preserve the semantics of IoT, and also seamlessly utilize the DTN-based communications an IoT-cum-DTN based framework is essential. To the best of our knowledge, such a framework of IoT-cum-DTN is not proposed and implemented. To this end, we propose a novel Internet of Hybrid Opportunistic Things framework based on the aforementioned IoT-cum-DTN paradigm. The proposed IoT-cumDTN architectural framework is shown in Fig. 1. To realize an architecture of the IoT-cum-DTN node proposed in the framework as shown in

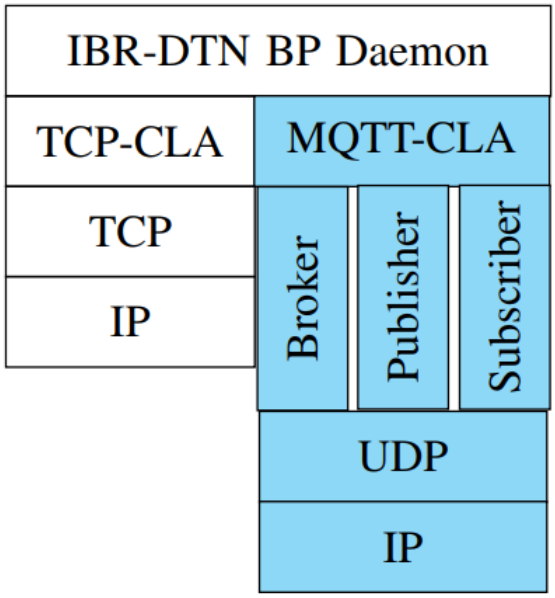


Figure 5.13: Proposed IoT-cum-DTN Gateway Node Architecture. The extended modules are shown in blue shaded boxes.

Fig. 5.12; the following design principles are necessary:

- Transparent DTN connectivity: The IoT-cum-DTN node should communicate as a peer DTN node with regular-DTN nodes.
- Transparent IoT P2P connectivity: The IoT-cum-DTN node should enable P2P communication with the respective IoT publisher and IoT subscriber nodes.
- DTN IoT Interoperability: To fully utilize DTNs persistent storage and robustness from node failures, the IoT-cum-DTN node needs to seamlessly encapsulate IoT messages as DTN bundles. Subsequently, the IoT messages have to be retrieved (from bundles) upon meeting a regular IoT node.

To realize an architecture with the aforementioned design principles, a vertical-plane integration of IoT and DTN stacks is necessary. To this end, we propose IoT as a convergence layer transport to the existing DTN stack and empower convergence-layer adapter with bundle construction and storage access. For our implementation, we have used a standard DTN2 reference implementation for embedded devices, namely IBR-DTN [5]. The IoT protocol considered is MQTT-SN [14], a client-server based publish/subscribe messaging transport protocols for wireless nodes such as Sensor Networks. The native MQTT-SN is based on the client-server model, different neighbor discovery beacons are used by the respective server and clients for identifying their peer nodes. We, therefore, modified MQTT-SN towards using a custom beacon protocol to enable simple P2P neighbor discovery. As the payload-contents of custom neighbor discovery beacon are not used by MQTT-clients, we used the beacons payload to contain information necessary for the regular DTN nodes to recognize them.

The architecture of the IoT-cum-DTN node based MQTTSN and IBR-DTN is shown in Fig. 5.13. In addition to the regular UDP, and TCP convergence layer, we have implemented MQTT components as a new convergence layer. The MQTT Convergence Layer Adapter (CLA) binds the MQTT components, namely broker, publisher, the subscriber to the IBR-DTNs Bundle daemon. Furthermore, we extended MQTT CLA with a cross-layer design, to have direct access to the IBR-DTNs persistent storage. This cross-layer design, enables the MQTT-CLA and underlying MQTT stack to act as an independent IoT process, that enacts the transparent IoT P2P connectivity. The MQTT-CLA also has the feature of DTN

bundle construction from the received MQTT message from an IoT publisher, and de-encapsulate bundle to retrieve actual MQTT message for the IoT subscriber reception.

5.7.3 System Evaluation and Conclusion

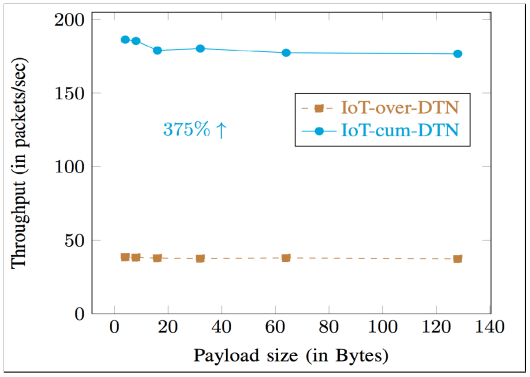


Figure 5.14: Throughput comparison between MQTT-over-DTN and MQTT-cum-DTN

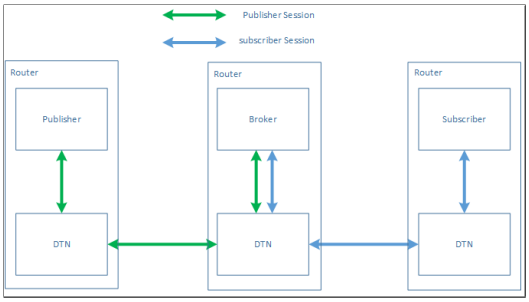


Figure 5.15: Logic topology of the testbed.

We evaluate the system by running MQTT over traditional DTN nodes and over our proposed framework. We define three nodes, respectively publisher, broker and subscriber in the network. Publisher continuously sending packets to the subscriber in the saturated volume of traffic. Fig 5.15 demonstrated the logical connections of the nodes. By evaluating these two frameworks, we obtain the result that proposed MQTT-cum-DTN solution is nearly 4 times better than the MQTT-over-DTN solution (Fig. 5.14).

Chapter 6

Conclusion

In this work, it starts from the analysis of OpenFlow architecture and its controller specification. By identifying the communication overhead between OpenFlow controller and OpenFlow data-plane, we focus on the implementation of OpenFlow data-plane to see if we can reduce this overhead by moving controller features into the data-plane implementation.

By utilizing OpenFlow protocol interface in datapath component of OpenFlow data-plane, we implement necessary features in an SDN controller on data-plane, so called In-House controller. Also, with the help of existing flow entry implementation and its priority feature, we implement the mechanism to filter packets. In this way, we make the In-House controller being compatible with the original setup.

Based on this In-House controller implementation, we design two mechanisms to docking network applications in devices. Inside SDN Docker focus on converting application logic to a readable binary image to dynamically load and run on the device framework. Outside SDN Docker leverage features of the In-House controller, virtual Ethernet, and Linux Containers to dock complicated programs as controller applications. These features brought us possibilities in dynamically deploying applications on network nodes, which is important to our works on MQTT flow aggregation and DTN implementation.

In the topic of MQTT aggregation, we use SDN Docker to dock MQTT

application in edge nodes. by modeling TCP traffic from the edge nodes to a remote server, we reveal the relationship between the congestion window size and the packet loss rate on the nodes on the route. Meanwhile, by aggregating short MQTT flows, we achieve a significant throughput improvement on the scenario of multiple publishers competing for bandwidth on edge node. For long flow aggregation, we proved that the fairness of publishers on the same access points is also related to the congestion window. When we limit the size of the congestion window on the publishers, we will achieve an improved fairness without loss of throughput in total.

By noticing that DTN node owns a similar architecture to the Outside SDN docker, we improve the DTN node in an SDN approach. With the help of In-House controller, we can dynamically change the configuration of convergence layers and routing mechanisms for specific DTN bundles. This approach improved the connectivity of DTN nodes as well as the transmission capacity between peer nodes.

6.1 Future Work

Networking world is keeping evolving with the improvement of hardware capacity of network devices. Data-plane packet processing and forwarding technologies are also improved. These facts bring more possibilities to moving control plane operation into data plane components in the network with separated packets. As a library implementation, Intel DPDK[67] has embedded some control plane

functionalities into data plane. It also provides APIs for programmers to build network applications. By utilizing these APIs, In-House controller and Inside SDN Docker can save development work on atomic operations and compiling issues.

Another issue in distributed controller need to be addressed is global information synchronization among distributed controllers. As application level functionalities, data need to be stored in the distributed controllers for further use. For example, in mobility management, user equipment's latest access nodes need to be recorded for hand-off use. How to retrieve this information from the network is crucial to system performance. Hierarchy structure distributed controllers like Kandoo[33] try to store global information in the root controller. It is easy to keep data consistency in one single controller but the communication overhead from the lowest level node to the root controller is inevitable. Other works like Ubiflow[69] try to solve the problem with distributed storage solution like consistency hash. This solution store global information in different spots in the network. The storage location is decided by a hash mapping between user equipment ID and node ID. This solution provides scalability by preventing storing all data in one node. However, retrieving data from a random node in the network is still difficult. The storage location could be far from the consumer node, while it still needs to do mapping calculation to locate it.

Interned of Things in mobile network scenario has been noticed by researchers by the development of MANET and VANET. In our work, we did an experimental work on combining DTN node with MQTT functionalities, which provide IoT nodes with opportunistic network features. In the future, we can apply this

framework to other IoT applications to improve the connectivity of IoT entities in different scenarios.

References

[1] D. Kreutz, F. M. V. Ramos, P. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking a comprehensive survey,". *Proceedings of the IEEE*, 103(1), 2015.

[2] "OpenFlow switch specification version 1.4.0,". 2013. [Online].Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/OpenFlow/OpenFlow-spec-v1.4.0.pdf

[3] M. Karakus, A. Durresi, "A survey: Control plane scalability issues and approaches in software-defined networking (SDN),". *Computer Networks*, 112, 2017.

[4] "OpenWrt Wiki,". 2017 [Online].Available: https://wiki.openwrt.org/start

[5] "MQTT Specification v3.1.1,". 2014 [Online].Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/csprd02/mqtt-v3.1.1-csprd02.pdf

[6] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, "Fog computing and its role in the internet of things,". *MCC '12 Proceedings of the first edition of the MCC workshop on Mobile cloud computing,* 2012. Pages 13-16

[7] Y. Wang, J. Bi, "A Solution for IP Mobility Support in Software Defined Networks," *ICCCN '14 Proceedings of International Conference on Computer Communication and Networks,* 2014.

[8] "RFC4423, Host Identity Protocol (HIP) Architecture," 2006 [Online].Available: https://tools.ietf.org/html/rfc4423

[9] "RFC5944, IP Mobility Support for IPv4, Revised," 2010 [Online].Available: https://tools.ietf.org/html/rfc5944

[10] "RFC6740, Identifier-Locator Network Protocol (ILNP) Architectural Description," 2012 [Online].Available: https://tools.ietf.org/html/rfc6740

[11] "RFC4830, Problem Statement for Network-Based Localized Mobility Management (NETLMM)," 2007 [Online].Available: https://tools.ietf.org/html/rfc4830

[12] "RFC5050, Bundle Protocol Specification" 2007 [Online].Available: https://tools.ietf.org/html/rfc5050

[13] Delay Tolerant Networking Research Group. "DTN Reference Implementation," 2006. [Online].Available: http://www.dtnrg.org/docs/code/

[14] M. Doering, S. Lahde , J. Morgenroth, L. Wolf, "IBR-DTN: an efficient implementation for embedded systems," *CHANTS '08 Proceedings of the third ACM workshop on Challenged networks,* 2008. Pages 117-120

[15] "RFC6693, Probabilistic Routing Protocol for Intermittently Connected Networks" 2012 [Online].Available: https://tools.ietf.org/html/rfc6693

[16] J. Burgess, B. Gallagher, D. Jensen, B. N. Levine, "MaxProp: Routing for Vehicle-Based Disruption-Tolerant Networks," *INFOCOM '06 Proceedings of the 25th IEEE International Conference on Computer Communications,* 2006.

[17] A. Balasubramanian, B. Levine, A. Venkataramani , "DTN routing as a resource allocation problem," *SIGCOMM '07 Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications* 2007. Pages 373-384

[18] S.A. Shah, J. Faiz, M. Farooq, A. Shafi, S. A. Mehdi "An architectural evaluation of SDN controllers" *ICC '13 Proceedings of the IEEE International Conference on Communications,* 2013.

[19] "Mininet: An instant virtual network on your laptop (or other PC)" 2012 [Online].Available: http://mininet.org

[20] M. A. Santos, B. A. Nunes, K. Obraczka, T. Turletti, B. T. de Oliveira, C. B. Margi, "De-centralizing SDNs control plane" *LCN '14 IEEE 39th Conference on Local Computer Networks,* 2014, pp. 402405.

[21] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, R. Sherwood, "On Controller Performance in Software-Defined Networks" *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, ser. Hot-ICE12.,* Berkeley, CA, USA: USENIX Association, 2012, pp. 1010

[22] Z. Cai, A. L. Cox, and T. S. E. Ng, Maestro: A System for Scalable OpenFlow Control, *Rice University, Tech. Rep.,* 2011.

[23] D. Erickson, The Beacon OpenFlow controller, *in Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking, ser. HotSDN 13. New York, NY, USA: ACM,* 2013, pp. 1318

[24] Floodlight is a Java-based OpenFlow controller" 2012 [Online].Available: http://www.projectfloodlight.org/floodlight/

[25] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, NOX: towards an operating system for networks, *Computer Communication Review.,* 2008.

[26] Nippon Telegraph and Telephone Corporation, Ryu Network Operating System, 2012. [Online]. Available: http://osrg.github.com/ ryu/

[27] OpenDaylight, OpenDaylight: A Linux Foundation Collaborative Project, 2013. [Online]. Available: http://www.opendaylight.org

[28] S. Yeganeh, A. Tootoonchian, and Y. Ganjali, On scalability of software-defined networking, *Communications Magazine, IEEE, vol. 51, no. 2,* pp. 136141, 2013.

[29] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, OpenFlow: enabling innovation in campus networks, *SIGCOMM Computer Communication Review, vol. 38, no. 2,* pp. 6974, Mar. 2008.

[30] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, DevoFlow: scaling flow management for highperformance networks, *Computer Communication Review, vol. 41, no. 4,* pp. 254265, Aug. 2011.

[31] A. Tootoonchian and Y. Ganjali, HyperFlow: a distributed control plane for OpenFlow, *in Proceedings of the 2010 internet network management conference on Research on enterprise networking, ser. INM/WREN10.,* Berkeley, CA, USA: USENIX Association, 2010, pp. 33.

[32] A. A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, Elasticon: An elastic distributed SDN controller, *in Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ser. ANCS 14.,* New York, NY, USA: ACM, 2014, pp. 1728.

[33] S. Hassas Yeganeh and Y. 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, ser. HotSDN 12.,* New York, NY, USA: ACM, 2012, pp. 1924.

[34] S. H. Park, B. Lee, J. Shin, and S. Yang, A high-performance IO engine for SDN controllers, *in Third European Workshop on Software Defined Networks,* 2014, p. 2.

[35] Y. Yiakoumis, J. Schulz-Zander, and J. Zhu, Pantou : OpenFlow 1.0 for OpenWRT, 2011. [Online]. Available: http://www.openflow.org/wk/index.php/OpenFlow 1.0 for OpenWRT

[36] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, Extending networking into the virtualization layer. *in proceeding of Hotnets,* 2009, pp. 16.

[37] Docker, Docker - Build, Ship, and Run Any App, Anywhere, 2017. [Online]. Available: https://www.docker.com/

[38] "LXC Containers", 2017, [Online]. Available: https://linuxcontainers.org/

[39] L. D. Xu, W. He, S Li *IEEE Transactions on Industrial Informatics, VOL. 10, NO. 4,* NOVEMBER 2014

[40] UpCall, "Linux man page", 2017, [Online]. Available: https://linux.die.net/man/8/cifs.upcall

[41] Pica8, Pica8 3920, 2013. [Online]. Available: http://www.pica8.org/documents/pica8-datasheet-64x10gbe-p3780-p3920.pdf

[42] Indigo "Indigo - Open Source OpenFlow Switches," [Online]. Available: http://www.projectfloodlight.org/indigo/

[43] Y. Mundada, R. Sherwood, and N. Feamster, An openflow switch element for click, *in Symposium on Click Modular Router. Citeseer,* 2009, p. 1. [Online]. Available: http://www.cc.gatech.edu/ yogeshm3/click symposium2009.pdf

[44] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, Scalable flow-based networking with difane, *SIGCOMM Computer Communication Review, vol. 41,* no. 4, pp. , Aug. 2010

[45] IP Link, "Linux man page", 2017, [Online]. Available: https://www.systutorials.com/docs/linux/man/8-ip-link/

[46] Mosquitto, "Mosquitto An Open Source MQTT v3.1/v3.1.1 Broker", 2017, [Online]. Available: http://mosquitto.org

[47] "RFC4838, Delay-Tolerant Networking Architecture," 2007 [Online].Available: https://tools.ietf.org/html/rfc4838

[48] "RFC5050, Bundle Protocol Specification," 2008 [Online].Available: https://tools.ietf.org/html/rfc5050

[49] S. Yi, C. Li, Q, Li "A Survey of Fog Computing: Concepts, Applications and Issues," *Mobidata '15 Proceedings of the 2015 Workshop on Mobile Big Data ,* Pages 37-42

[50] K. Sood, S. Yu, and Y. Xiang, Software defined wireless networking opportunities and challenges for Internet of things: A review, *IEEE Internet of Things Journal, vol. PP,* no. 99, pp. 11, 2015.

[51] P. Loiseau, P. Gonalves, J. Barral, and P. V.-B. Primet, Modeling TCP throughput: An elaborated large-deviations-based model and its empirical validation, *Performance Evaluation, vol. 67, no. 11,* pp. 10301043, 2010.

[52] R. Jain, D.-M. Chiu, and W. R. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer system.," Eastern Research Laboratory, Digital Equipment Corporation, 1984.

[53] F. Warthman, Delay-Tolerant Networks (DTNs) - A Tutorial. [Online]. Available: http://www.ipnsig.org/reports/DTN Tutorial11.pdf

[54] Z. Li, Y. Liu, H. Zhu, and L. Sun, Coff: Contact-duration-aware cellular traffic offloading over delay tolerant networks, *IEEE Transactions on Vehicular Technology, vol. 64, no. 11,* pp. 52575268, 2015.

[55] K. Ezirim and S. Jain, Taxi-cab cloud architecture to offload data traffic from cellular networks, in WoWMoM 15: Proceedings of the IEEE International Symposium on World of Wireless, Mobile and Multimedia Networks, 2015, pp. 16.

[56] Y. Li, M. Qian, D. Jin, P. Hui, Z. Wang, and S. Chen, Multiple mobile data offloading through disruption tolerant networks, *IEEE Transactions on Mobile Computing, vol. 13, no. 7,* pp. 15791596, 2014.

[57] X. Zhuo, W. Gao, G. Cao, and S. Hua, An incentive framework for cellular traffic offloading, IEEE Transactions on Mobile Computing, vol. 13, no. 3, pp. 541555, 2014.

[58] M. Zhu, J. Cao, D. Pang, Z. He, and M. Xu, Sdn-based routing for efficient message propagation in vanet. in WASA, ser. Lecture Notes in Computer Science, vol. 9204. Springer, 2015, pp. 788797.

[59] N. Eagle and A. (Sandy) Pentland, Reality mining: Sensing complex social systems, *Personal Ubiquitous Computing., vol. 10, no. 4,* pp. 255 268, Mar. 2006. [Online]. Available: http://dx.doi.org/10.1007/s00779- 005-0046-3

[60] Raspberry Pi "Raspberry Pi - Teach, Learn, and Make with Raspberry Pi" [Online]. Available: https://www.raspberrypi.org/

[61] Kandoo Kandoo: Scale your sdn, [Online]. Available: http://www.kandoo.org.

[62] M. Auzias, Y. Maheo, and F. Raimbault, CoAP over BP for a delaytolerant internet of things, in *FiCloud 15: Proceedings of the International Conference on Future Internet of Things and Cloud,* 2015, pp. 118123.

[63] P. Raveneau and H. Rivano, Tests Scenario on DTN for IOT III Urbanet collaboration, *Inria - Research Centre Grenoble Rhone- Alpes ; INRIA, Technical Report RT-0465,* 2015. [Online]. Available: https://hal.inria.fr/hal-01187114

[64] D. Amendola, F. D. Rango, K. Massri, and A. Vitaletti, Efficient neighbor discovery in RFID based devices over resource-constrained DTN networks, in *ICC 14: Proceedings of the IEEE International Conference on Communications,* 2014, pp. 38423847.

[65] D. A. L. Nuevo, D. R. Valles, E. M. Medina, and R. M. Pallares, OIoT: A platform to manage opportunistic IoT communities, in *IE 15: Proceedings of the International Conference on Intelligent Environments,* 2015, pp. 104111.

[66] DTNRG "Delay-Tolerant Networking Research Group" [Online]. Available: https://sites.google.com/site/dtnresgroup/

[67] DPDK "Data Plane Development Kit" [Online]. Available: http://dpdk.org

[68] Mosquitto "An Open Source MQTT v3.1/v3.1.1 Broker" [Online]. Available: https://mosquitto.org

[69] D. Wu, D. I. Arkhipov, E. Asmare, Z. Qin, and J. A. McCann, UbiFlow: Mobility management in urban-scale software defined IoT, in *2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015, pp. 208216

[70] Y. Xu, V. Mahendran, S. Radhakrishnan, "Towards SDN-based fog computing: MQTT broker virtualization for effective and reliable delivery" in *8th International Conference on Communication Systems and Networks (COMSNETS)*, 2016

[71] Y. Xu, V. Mahendran, S. Radhakrishnan, "SDN docker: Enabling application auto-docking/undocking in edge switch" in *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2016

[72] Y. Xu, V. Mahendran, S. Radhakrishnan, " Internet of Hybrid Opportunistic Things: A novel framework for interconnecting IoTs and DTNs" in *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2016

[73] Y. Xu, V. Mahendran, S. Radhakrishnan, " Fairness in Fog Networks: Achieving Fair Throughput Performance in MQTT-based IoTs" inn *IEEE Consumer Communications and Networking Conference Workshops (CCNC WKSHPS)*, 2017

[74] Y. Xu, V. Mahendran, S. Radhakrishnan, " SoftDTN: A Software-Defined Network Based DTN Routing Architecture", *Submitted to IEEE Communications Letters*, 2017

[75] P. Loiseau, P. Gonalves, J. Barral, and P. V.-B. Primet, Modeling TCP throughput: An elaborated large-deviations-based model and its empirical validation, *Performance Evaluation,* vol. 67, no. 11, pp. 10301043, 2010.

[76] Glibc, "The GNU C Library (glibc)", [Online]. Available: https://www.gnu.org/software/libc/

[77] A. Dembo and O. Zeitouni, *Large Deviations Techniques and Applications*. Jones and Bartlett, Boston, 1993.

[78] J. Barral and P. Loiseau, "Large deviations for the local fluctuations of random walks". *Stochastic Processes and their Applications*, vol. 121, no. 10, pp. 22722302, 2011.