PERFORMANCE EVALUATION OF CONTROLLERS IN LOW-POWER IOT NETWORKS

by

MIHEER KULKARNI

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
December 2020

Table of Contents

List of Tables

List of Figures

Abstract

Software-Defined Networking (SDN) enables network reconfiguration in response to dynamic application requirements. Recently researchers have tried extending SDN in the Internet of Things (IoT) networks, which required domain-specific customization in SDN architecture and protocols. Several architectures and designs have been proposed without adequate performance evaluation except for a couple: μSDN and SDN-WISE. μSDN architecture is built on a standard protocol stack, whereas SDN-WISE proposes a custom one. We first perform an extensive evaluation of these two architectures in terms of energy consumption, latency, and packet delivery ratio (PDR). The results confirm that μSDN can significantly reduce resource utilization while offering high PDR compared to SDN-WISE. Thus, we recommend μSDN as a potential architecture for low-power IoT networks. However, which SDN controller the chosen architecture can use? There is no standard evaluation bench-marking available. Thus, we evaluate the embedded (part of the IoT networks) μSDN and standard (external to the IoT networks) ONOS controllers on their throughput, delay, topology detection, and topology update time. We observe a trade-off between the scalability and performance between the embedded and external controllers. Thus, users can choose a controller based on their application demand, e.g., an embedded one for low traffic rate from a small number of IoT sources.

List of Abbreviations Used

API	Application Programming Interface
CASAS	Centre for Advance Studies in Adaptive System
CONF	Configuration Messages
CP	Control Plane
CSMA/CA	Carrier Sense Multiple Access Collision Avoidance
DAG	Directed Acyclic Graph
DAO	DODAG Destination Advertisement Object
DIO	DODAG Information Object
DIS	DODAG Information Solicitation
DODAG	Destination Oriented Directed Acyclic Graph
DP	Data Plane
DTLS	Datagram Layer Security
EA-SDN/NFV	Energy Aware Network Function Virtualization
FSM	Finite State Machine
FTQ	Flow Table Query
FTS	Flow Table Set
ILP	Integrated Linear Programming
INPP	In Network Packet Processing
IoT	Internet of Things
IP	Internet Protocol
LoWPAN	Low-Power Wireless Personal Area Network
MAC	Medium Access Control
MTU	Maximum Transmission Unit
NFV	Network Function Virtualization
NI	Northbound API
NOS	Network Operating System
NS	Non-Storing
NSU	Node Status Update

ODL	OpenDayLight
ONOS	Open Networking Operating System
OS	Operating System
PDR	Packet Delivery Ratio
RDC	Radio Duty Cycling
RPL	Routing Protocol for Low-power and Lossy Network
RSSI	Received Signal Strength Indicator
RTT	Round-Trip Time
SDN	Software Defined Network
SI	Southbound API
SLIP	Serial In-Line Protocol
SNR	Signal-to-Noise Ratio
TCP	Transmission Control Protocol
TD	Topology Discovery
TI	Texas Instruments
TM	Topology Manager
UDGM	Unit Disk Graph Medium
UDP	User Datagram Protocol
WLAN	Wireless Local Area Network
WSN	Wireless Sensor Network
WSU	Washington State University

Acknowledgements

First of all, I would like to humbly thank my supervisor, Dr. Israat Haque, for guiding and supporting me during the MCS study and research despite her busy work schedule. She taught me the methodology to carry out research and provided constructive feedback on my work time-to-time. I feel really honored to have worked under her supervision and thank her for continuous support during my study at Dalhousie.

I would also like to thank all of my fellow lab members from the Programmable and Intelligent Network (PINet) research lab for their help and opinion regarding my work. I would specially thank Dipon, one of my lab mates, for his feedback on my work and help to complete my research. I am also thankful to Dr. Michael Baddeley from Toshiba, UK, for his guidance and valuable feedback during the evaluation.

Finally, I want to thank my parents for their love, support, and selfless sacrifices that they made to send me to such a prestigious university for my further education. Without their support, I would not have been able to finish my work.

Chapter 1

Introduction

In the recent years, the Internet has gone under tremendous evolution. Furthermore, with the advent of *Internet of Things (IoTs)*, all devices with some processing and network capabilities can communicate with each other and share information [8]. The devices range from small temperature sensor to a big car. IoT has revolutionized the field of smart technologies [25]. This has led to the development of several applications like smart home, smart cars, smart healthcare, etc. to provide seamless delivery of services in order to manage consumer needs. Software Defined Networking (SDN) is one of the emerging technologies in the field of networking [21]. It has gained a lot of ground, since its introduction in campus networks. SDN allows network admins to configure the network without changing logic of each device [21]. Apart from re-configuration, SDN also offers the global knowledge of a network, virtualization, scalability, etc.

Each year billions of IoT devices are being added to networks. These devices have low-power, low-processing abilities and operate in lossy radio medium. Also, the rise in IoT network has introduced a large amount of data traffic and it becomes more difficult to manage by legacy networks [18]. Can we use SDN to manage this dynamic nature of IoT networks? Many SDN-IoT architectures have been proposed to provide solution to increasing IoT data traffic. Some architectures use customized external controller and some use embedded ones, i.e., controller is within the IoT environment. In our thesis work, we investigate whether it is feasible to use embedded controller for a low-power IoT network. For the first part of our work, we compare two prominent SDN-IoT architectures based on our literature survey. We carry out the comparison by extensively evaluating the performance of both architecture. After establishing which architecture is suitable for low-power IoT networks, in the second part of this work, we connect a standard SDN controller to the architecture and compare the performance with an embedded controller. Finally, we propose under

which circumstances we can use embedded controller and how it is more useful than the external one.

For example, we can deploy temperature or humidity sensors in a smart home (bedroom, kitchen, etc.). These sensors can collect data and send it to the embedded controller. If the number of sources is low (e.g., around 25), we can use embedded controller, which can offer better performance than external controller for less number of source nodes. But if there are more than 25 sources, we suggest to switch to external controller for ensuring performance with scalability. Most of smart home applications use less than 20 sensors [32]. Thus, the embedded controller would be useful in that case.

1.1 Motivation

Every year billions of IoT devices are getting added to communication network [25]. The Norton report [27] from USA suggests that by year 2025 there will be 25 billion IoT devices and with the advent of 5G, it is only going to fuel the growth of IoT networks. The same report also estimates that by year 2025 the IoT and its related application will provide business of 11 trillion US dollars [27]. As the IoT networks begin to expand, it will lead to large influx of data and control traffic [18]. In order to manage such large traffic we will need some sort of infrastructure which dynamically program the network or deploy new services. To facilitate such demand we think of SDN as a solution to support such dynamic network. We think SDN in IoT networks can offer a good network framework and withstand future demand.

SDN is an emerging networking paradigm that promises to change current state of traditional networks, by breaking vertical integration, separating the network's control logic from the underlying routers and switches, promoting (logical) centralization of network control, and introducing the ability to program the network [21]. The separation of logic, offers better deployment of network policies and their implementation in switching hardware, and the forwarding of traffic, by breaking the network control problem into tractable pieces. SDN makes it easier to create and introduce new abstractions in networking, simplifying network management and facilitating network evolution. SDN also offers global view which means a centralized controller can have entire knowledge of underlying network [21].

On the other hand, IoT devices include small devices like sensors, actuators, etc. These are small devices which operate on low-power, low bit rate, low processing ability, and have lossy medium. The introduction of SDN in IoT networks given such constraints can lead to additional overhead on these devices [6]. Moreover, to support such large number of devices we need devices to support IPV6 addressing. We need to consider all these challenges and redesign the SDN architecture and protocols. Baddeley *et. al* propose a lightweight SDN framework μSDN [6] for low-power IoT networks. They introduce different optimizations like protocol, memory, architecture, and controller to overcome device overhead challenges. They use base Routing Protocol for Low- power and Lossy Networks (RPL) [35] for communication between controller and node. They offer a good solution but they do not show any extensive evaluation of its performance and comparison with other existing architecture.

Gaullicio *et al* in their work of SDN-WISE [13] offer a stateful solution for WSNs (Wireless Sensor Networks) and use an adapted version of SDN Openflow [23] protocol to suit for WSN domain. They use a customized external controller to program underlying nodes. They do not offer an optimizations and neither show extensive performance evaluation. The same authors introduce to connect Open Network Operating System (ONOS) [34] a standard SDN controller to SDN-WISE architecture. They wanted to show heterogenity by showing communication between sensor nodes and Openflow switches. However, they do not show any evaluations in IoT networks. Thus, almost no architecture shows any extensive evaluation of their performance and its comparison with other existing work.

1.2 Research Objective

In order to overcome challenges of IoT devices, there were several architectures proposed. Some were built using customized protocols and controllers whereas some used extension of SDN OpenFlow version with traditional SDN controller. During our survey we found μSDN and SDN-WISE to be prominent architecture as they have proper implementation in Contiki Operating System (OS) [11] built on Cooja hardware emulator. Our main objective in this research was benchmarking SDN-IoT architecture and controllers for low-power IoT network. We decided to compare their performance by evaluating different performance metrics like Transmission energy consumption,

Controller-Node Round-Trip Time (RTT), Latency between a source-destination pair, and Packet Delivery Ratio (PDR). These metrics are evaluated by varying bit rate and hop distance. In second part of our work, we adapt the existing μSDN architecture to be controlled by ONOS controller. We choose the ONOS controller over OpenDaylight (ODL) based on the evaluation results presented in [9], where ONOS outperforms ODL. We also perform evaluation of μSDN embedded controller and ONOS controller. We used performance metrics like Throughput, Controller response time, Topology update time, and Topology discovery time to compare both controllers. Finally, we propose to use embedded controller for low-power IoT networks if data rate is low. The embedded controller is suitable for low-power networks as.

- The embedded controller shows better performance in low-bit rate in which most IoT devices operate.
- It has almost the same Throughput to that of complex ONOS controller for low-bit rate.
- Responds quickly to flow requests in low-bit range.
- Offers better update time in case of any link failure.

1.3 Contribution

In this thesis, we start with an initial investigation of evaluating performance of two state-of-the-art software-defined architectures μSDN and SDN-WISE for IoT networks. In particular, we use embedded controller for both the architectures for a unbiased comparison. We do that by implementing the controller logic in SDN-WISE sink node. We perform extensive evaluation of both architectures by varying bit rate and hop distance. We evaluate various performance metrics like Energy consumption, Controller-Node RTT, Latency between source-destination pair, and PDR for both grid and random topology. Through our results, we observe that μSDN performs better than SDN-WISE in terms of the mentioned performance metrics. We also provide appropriate reasoning for this performance trend in both the architectures. We implement both architecture in Contiki OS using Cooja network simulator.

After establishing which architecture is better for low power IoT network, we integrate the existing μSDN architecture with standard SDN controller ONOS. ONOS is a well known open-source distributed SDN controller with modular architecture [34]. We adapted the μSDN messages to that of ONOS to ensure compatibility. In particular, we develop an ONOS adapter that converts μSDN messages to ONOS and vice-versa. Similarly, we make some design modification in μSDN architecture to connect it with ONOS controller. We modify existing embedded controller to act as border router to connect to the external controller. We also perform extensive performance evaluation of embedded and ONOS controller, e.g., Throughput, Controller response, Topology discovery time, and Topology update time with a varying flow setup requests and number of nodes. The performance of embedded controller was better than that of ONOS controller for low-bit rate. The summarized contributions are as follows.

- We modify SDN-WISE node architecture to ensure evaluations are unbiased and we compare its performance with μSDN.
- The evaluation results reveal that μSDN has better performance than SDN-WISE in terms of energy, Node-Controller RTT, and PDR.
- We integrate μSDN architecture with ONOS controller.
- We design and develop an adapter for ONOS to convert message format of μSDN.
- We perform extensive evaluation of embedded controller and ONOS on Contiki OS using Cooja network simulator. We found that embedded controller performs better in terms Controller response time, discovery and update time at low-bit rate. The embedded controller has similar performance in terms of Throughput to that of ONOS for low-bit rate. We provide appropriate reasons for this performance trend.
- We propose to use embedded controller for low-bit rate and low-power IoT networks.
- Finally, we provide the portable bench-marking code in our git [22].

1.4 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 presents two sections: Section 2.1 introduces the necessary background to understand this thesis and Section 2 discuss relevant research works. Chapter 3 presents the design and methodology of the thesis. It contains four sections: Section 3.2 define the problem. In Section 3.3, we show design changes in SDN-WISE architecture , while in Section 3.4, we present our μSDN-ONOS architecture, Section 3.5 presents two parts: section (3.5.1 and 3.5.2) presents necessary setup and description of parameters for the evaluation, while the next Section 3.6 presents the results and discussion on performance trends. Chapter 4 presents conclusion 4.1 and future research directions 4.2.

Chapter 2

Background and Related Work

In this chapter, we first discuss necessary background on some topics to better understand this thesis. Then, we review existing literature related to our work.

2.1 Background

2.1.1 Software-Defined Network (SDN)

SDN has brought a paradigm shift in programmable networks. It has provided an opportunity for all network administrators to overcome the challenges of legacy networks [18]. The main feature of SDN is that it breaks vertical integration of networks by separating the control plane and data plane, i.e., the control logic is separated from the device. This separation allows the network switches to act as simple forwarding devices and their control logic can be programmed using the centralized controller. This reduces the effort of network operators to change the control logic by visiting every device present in the network. Some of the advantages of SDN are as follows.

- A logically centralized controller allows to program network remotely, which allows an easy way to update network policies.
- A physically distributed controller in a large network enhances the performance, scalability, and reliability of networks.
- SDN allows the network to adapt to changes dynamically.
- The programmable networks allow the company to design more sophisticated network functions, services, and applications like load balancing, traffic engineering, etc.
- SDN provides a global network view meaning all applications can see the same network information. This allows a more consistent and effective implementation of network policies.

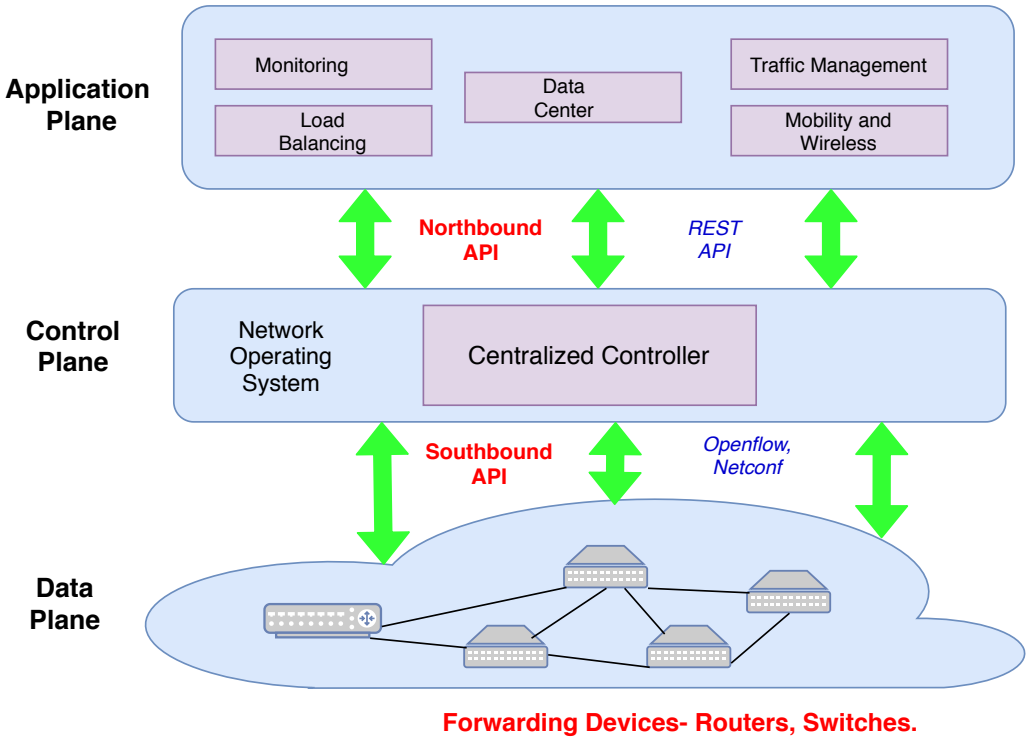


Figure 2.1: Traditional SDN architecture.

A brief overview of SDN architecture is as shown in Fig. 2.1. The architecture has three different planes and two APIs. We will discuss these components one by one.

Data Plane (DP): It includes hardware devices like routers, switches, etc. These devices are either connected via wireless radio medium or wired cables. The forwarding devices have well-defined instructions sets that allow them to perform certain actions such as forward, drop, send it back to the controller when certain packets arrive. The instructions are defined using Southbound API like OpenFlow [23], Netconf, etc.

Southbound API (SI) : The flow table instruction sets of the data plane devices are defined by the Southbound API. The SI also defines the communication protocol between the data plane devices and control plane elements. The well-defined protocols like OpenFlow are used to formalize the communication between control and data plane elements [21].

Control Plane (CP): The control plane is called a network brain [21] because all the control logic of data plane elements can be programmed on this plane. The

Network Operating System (NOS) provides the network programmers with necessary abstraction to program underlying devices.

Northbound API (NI): The NOS can offer an API to application developers to program the underlying data plane devices. The NI act as an abstraction for low-level instructions used by southbound API. The NI is used as a common interface to develop some SDN/NFV applications [21].

Application Plane: The application plane is a collection of several applications where they leverage functions provided by NI to perform some controller related operations like resource allocation, traffic monitoring, load balancing, etc. The application plane allows administrators to define network policies, which are ultimately translated to southbound-specific instructions, which then control the behavior of data plane devices [21].

2.1.2 ONOS architecture

The *Open Network Operating System (ONOS)* is a distributed open source SDN controller, which offers scalability and high performance while avoiding single of failure [34]. ONOS was designed, keeping in mind the needs of network operators who wanted to build high carrier-grade solutions to reduce capital expenditure on traditional Silicon hardware while offering flexibility and ease of deploying new programs with simplified interfaces and code modularity. ONOS supports both configuration and real-time network monitoring, thus eliminating the need to continually re-run routing rules and other communication protocols in device [34]. The ONOS cloud-based controller has led to innovation and allowed end-users to create new applications without worrying about data plane elements.

The ONOS platform includes.

- A set of control applications that offer extensibility, modularity, and distributive SDN controller.
- Simplified management, configuration, and deployment of new software, hardware, and services.
- A scale-out architecture to provide the resiliency and scalability required to meet the rigors of production carrier environments.

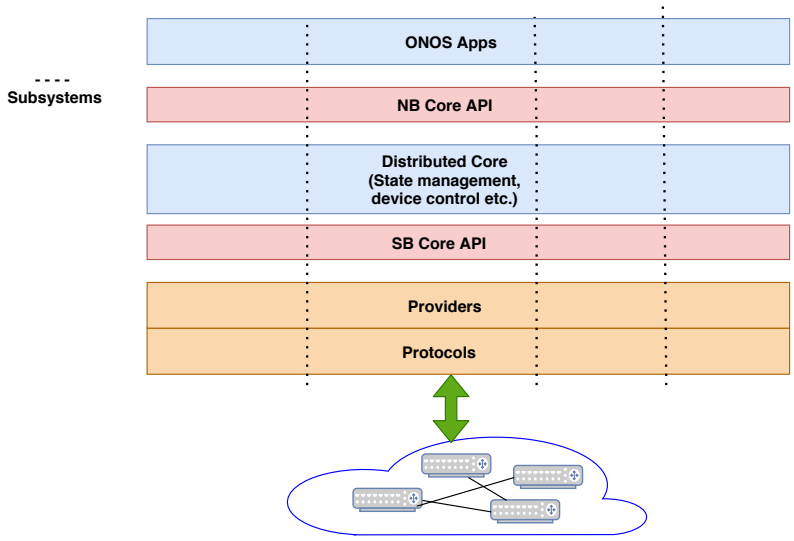


Figure 2.2: Layered ONOS architecture.

ONOS is a multi-layered controller with each layer designated with some functionality. The architecture is as shown in Fig. 2.2 shows four different layers and two APIs, which act as an abstraction for layers above and below them. Each layer can be vertically sliced into different subsystems that we will discuss further [5].

Protocols: It is the lowest layer in ONOS. The components in the Protocols layer are used to handle communication with underlying network elements like OpenFlow switches, routers, etc. This layer consists of components, which act as communication driver implementing the communication protocol as well as handle messaging between devices and drivers [5].

Providers: The Provider layer is responsible for translating the abstractions from higher layers to protocol specific operations and vice-versa. The Provider component receives information from its Southbound part ,i.e., protocol layer and then wraps it into a message format specific to higher layers where the information is stored and processed further. The higher layers use SB core API to trigger events in Provider layer to convert higher layer abstractions to protocol specific formats and sending them to data plane devices. A Provider ID is associated with a Provider layer. The main purpose of Provider ID is that it provides a unique identity to group of components within a Provider layer. This allows devices and other entities to remain associate with their set of components even if other components get uninstalled or deleted [5].

Distributed Core: The component residing inside the Distributed Core layer is referred to as Manager. The Manager receives information from Providers and serves it to applications and other services. It exposes several interfaces like Northbound service, Admin services, Provider registry, and Provider service. These interfaces act as an abstraction for incoming and outgoing packets, flow rules, topology information, etc. These abstractions can provide entire information on network topology and links that can be used for further analysis. Also, the Manager has the task of indexing, persisting, and synchronizing the information received by the Core. This includes ensuring consistency and robustness of information across multiple ONOS instances by directly communicating with stores on other ONOS instances [5].

ONOS Applications: The Application layer is the topmost tier of the ONOS architecture. The components of the Application layer leverage information like network view, node status, etc., which it receives from the NB Core API. It also triggers some critical actions like packet forwarding, which is important for each Subsystem to send the information processed by them back to the device. Typically, ONOS applications are triggered based on certain events from network devices on certain criteria [4] like the Flowrule forwarding event is triggered when it receives Flow request from the device. The corresponding Subsystem generates the flow rules and then those rules are sent back to the device.

Thus, ONOS because of its code modularity and extensibility is widely used in network functions that require scalability and adaptability [34]. In our work, we use the components of these layers to adapt the network element's message format to ONOS format and vice-versa.

2.1.3 ONOS Subsystem and Service

A service is functionality that comprises of several components that are created by vertical slice through different layers of ONOS stack [5]. Each vertical slice includes the service or collection of components is called a Subsystem. Each of the Subsystem components lies in any one of three layers of ONOS and can be identified by one or more Java interfaces that they implement. Some of the primary ONOS subsystems are as follows.

- *Device Subsystem:* It manages the inventory or list of devices present in the

network.

- *Host Subsystem:* It manages an inventory of hosts on end-to-end locations of the network.
- *Link Subsystem:* Manages a list of links or connections between different devices and hosts.
- *Packet Subsystem:* It allows the application to listen to incoming query packets and manages the outgoing packet out messages to set rules on one or more devices.
- *Flowrule Subsystem* It manages the inventory of flow rules, which are installed on devices and provides the flow metrics.
- *Topology Subsystem:* It manages the time-ordered snapshots of network graph views that are visible on the ONOS dashboard.

Thus, ONOS allows users to develop various services across the ONOS stack. The different services can be packaged from various Subsystems to create an ONOS application. These applications can be activated when required. ONOS allows users to program in Java with Maven build configuration [5].

2.1.4 Internet of Things (IoT)

The *Internet of Things (IoT)* is a group of interrelated computing devices which are capable of transferring information without any human interaction. These devices are called unique identifiers that consist of sensors, actuators, microprocessors, etc. These are called unique because each device has its unique ID that it uses while communicating with other devices. These are also called low-power devices as they are operated on low voltage ranges [25]. In the last decade, IoT has created a huge market for itself. Many tech companies are looking to implement IoT platforms. There are a lot of advantages of IoT, like manage a business environment without human intervention, saves time, and it enhances productivity and customer experience. There are a variety of fields where IoT can be used like healthcare, the automobile industry, household appliances, etc. For example, Google Home is an IoT device that can

monitor the usage of your lights, heating, cooling, etc. This device can be remotely controlled by a computer or smartphone [25]. Due to its applications and advantages, it is estimated that IoT business will have a market value of 1 trillion USD by the year 2025 [27].

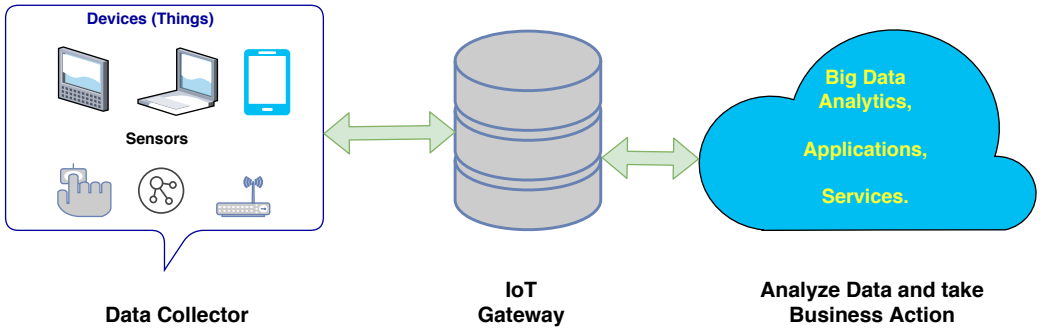


Figure 2.3: An example of IoT ecosystem.

IoT ecosystem has three parts (Fig. 2.3): Data collection via sensors, transfer the data, and analyze it. The fundamental characteristics of IoT are summarized below [25]:

- **Connectivity** It is a basic characteristic that allows the device hardware to communicate with other devices at different network layers.
- **Things** Any object or device, which has the ability to sense data and send it over a network can be part of the IoT network.
- **Data** The data in IoT is the most important thing, which drives intelligence and business action.
- **Intelligence** The aspect of intelligence deals with basically performing big data analytics on gathered data from various IoT devices.
- **Business Action** This can be a manual action, action based upon a discussion regarding improving consumer experience and digital transformation, often the most important piece.

IoT is the future of networking. It connects billions of devices to the internet and involves the use of billions of data points [25]. Along with these advantages, there

are some challenges like device restrictions, security, data management, interference in the communication medium, and cost [25].

2.1.5 Overview of RPL

Routing Protocol for Low-power and lossy network (RPL) is a protocol that is widely used in low power sensor networks [6]. It is based on the IEEE 802.15.4 standard for wireless networks. IETF developed it as a standard routing protocol for this family of network devices. RPL is a distance-vector protocol and uses optimized multi-hop communication to send messages and be used in the one-to-one form. This way, it helps reduce the memory requirement of the nodes [35]. RPL organizes the topology in a tree-like structure as *Directed Acyclic Graph (DAG)*, which means that a single node is partitioned in two paths to two different nodes. Each node is associated with a parent that acts as a gateway to that node. The parents are calculated using *Objective Functions (OF)* decided by the roots of the DAG [2].

RPL creates a routing topology in the form of a *Destination-Oriented Directed Acyclic Graph (DODAG)*, which is a directed graph without cycles, oriented towards a root node, e.g., a border router, sink node, etc. RPL uses three main types of control messages.

- DODAG Information Object (DIO),
- DODAG Information Solicitation (DIS) and
- DODAG Destination Advertisement Object (DAO).

The process of the DODAG construction begins from the DODAG root. First, the DODAG root broadcasts the DODAG information by transmitting a DIO message. The neighbors of the root receive and process the DIO message. The DIO message is processed and transmitted to other nodes one-by-one. Once a neighbor joins the DODAG, it has a route towards the DODAG root, and the root becomes a DODAG parent of the node. Next, the node calculates its rank in the DODAG and replies with a DAO message to its parent to inform its participation. A node that has not received any DIO messages and has not joined any DODAGs can request DODAG information by sending DIS messages periodically to its neighbors. All of the neighbors repeat this process until all of the nodes join the DODAG [2].

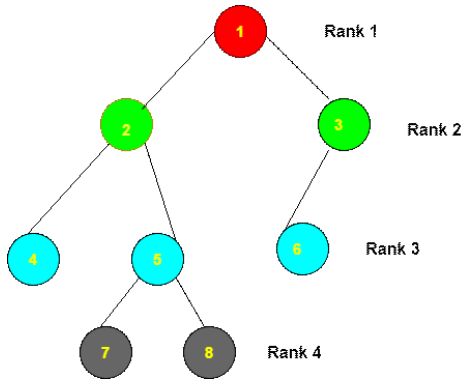


Figure 2.4: An example of RPL DAG.

To understand the working of RPL let us consider a topology as shown in Fig. 2.4. A network administrator configures one or more nodes as root node (node 1 in 2.4). The root node then starts sending multicast DIO messages to nearby nodes to advertise its presence (node 2 and node 3 in 2.4). These nodes process the DIO message mark node 1 as their parent node because they received a DIO message from the lower rank node. Similarly, node 2 and node 3 will transmit DIO messages to their neighboring nodes, this process will continue until all nodes form a DODAG tree. Now a single node may receive multiple DIO messages from lower rank nodes but it chooses the optimized path or route based on the objective function programmed into nodes. The objective function contains different metrics like RSSI value, link quality, etc. So, a node will select its parent based on its objective function. Now, this upward route is established, the root nodes need to know if all nodes have joined the DAG. So, for establishing a downward route all nodes send DAO messages to the root node. In this way, all regular nodes and root nodes come to know about nodes above and below them [2].

RPL is effective in allowing nodes to quickly send information up the tree [35]. However, the RPL graph forces nodes nearer the root to serve messages from nodes further down the tree and exacerbates energy loss in nodes nearer the root. RPL operates in two modes *Storing* and *Non-Storing (NS)* mode. In *Storing* mode nodes maintain a routing table for their Sub-DODAG. In *Non-Storing* mode, nodes only know their parent, and the root keeps a routing table for the whole DODAG and each node use source routing to send packets to the destination. The Non-Storing mode acts as controller-communication in SDN architecture. That's why we use this mode

in our architecture [6].

2.1.6 SDN in IoT Challenges

SDN offers network programmability that allows the network to adapt to any policy changes dynamically. But for low-power IoT networks, SDN proves to be a big overhead because of resource constraints (memory, energy, processing) and energy limitations [6]. The challenges are explained as follows.

Hardware Restrictions: The low power IoT networks are comprised of small devices like sensors, actuators, etc., which operate on battery and have power availability up to a few weeks or months. In addition to this, these devices have low memory and excessive radioactivity, which leads to depletion of energy. This is particularly limiting for traditional SDN, which employs thousands of flows per second, and flow table, which consists of hundreds of flow rules. These devices cannot handle such an enormous amount of information [6].

Packet Fragmentation: The IEEE 802.15.4 standard for low-power IoT network allows *Maximum Transmission Unit (MTU)* of 127B [6]. Even though 6LoWPAN [10] standard has introduced Internet Protocol (IP) capabilities for low-powered devices but by considering the link layer and IP layer header lengths it leads to reduced space in packet. Therefore, a full IPV6 packet will be left with only 53B for application data, which is very insufficient to carry even SDN control information. Thus, to prevent packet fragmentation multiple transmissions per packet are carried out and SDN control messages need to fit in allocated space [6].

Unreliable Links: The low-power devices operate in a lossy radio medium, which is prone to unreliability. But to implement SDN, the network should be stable, including all links. These unreliable links and multi-hop mesh networks add to more problems to implement SDN. Even if the unreliability is ignored, a simple packet sent to the controller for the update can lead to several delays or dropping of the packet [6].

Interference: The low-power devices operating using IEEE 802.15.4 standard are sensitive to communication networks operating at high frequency [6] For example, If a low-power IoT network is operating near an IEEE 802.11 [30] Wireless LAN (WLAN) network on same frequency channel can affect transmission ability of sensors and can lead to packet loss. This also prevents nodes from sending or receiving packets from

the controller, which can lead to failure of the SDN network.

Multi-hop Mesh Topology: The RPL is a distributed protocol that allows maintaining topology reducing the control overhead in the network. The low-power devices have reduced radio range, and a multi-hop packet transmission allows nodes to cover the overall network by a single base station [6]. But introducing multiple hops in an unreliable environment can lead to loss of packets, which leads to more consumption of energy [6].

2.1.7 ONOS Technical Challenges

There were a lot of challenges that we faced while connecting ONOS with Cooja. A majority of those were because of a lack of proper documentation. We will list down some major challenges.

- The major challenge was that ONOS applications are written using Java as the programming language, whereas μSDN is originally written in C language. Since some of the modules (e.g., the controller) need to be written in Java, it was cumbersome to convert all the code.

- ONOS is basically a Java-based controller that used Maven for its build. Recently, ONOS changed its build from Buck to Bazel. Bazel is a more sophisticated way of compiling a Java code developed by Google. Since the technology is pretty recent, there are no discussion forums or proper documentation.

- The new Bazel build comes with the feature of writing Bazel build files for each module. Since very few applications are developed in ONOS using Bazel, there was no reference on how to start writing those files.

- Finally, some important libraries like RPL, Clock Timer are not available in Java. These come in ready with the Contiki package itself. That was a big challenge as timer and RPL are the essences of μSDN architecture.

The solutions to above challenges are given below.

- We had converted some of the control modules of μSDN to Java to ensure that functionalities are the same in the ONOS controller as well. For this, we had to subsequently convert packet structure and flow table modules as well.

- Since Bazel doesn't have discussion groups or forums on the Internet. We had to read the documentation provided by Google on how to import it in IDE like IntelliJ idea, how to create modules, etc.

- The Bazel build file writing is very critical. When you install ONOS for the first time using Bazel, some applications like OpenFlow, Forwarding, etc., which are pre-installed. We referred to OpenFlow build files and saw how they were written in Bazel configuration.

- For the RPL library, we had to create a packet that mimics RPL. For e.g., node while connecting to the sink advertises itself by sending broadcast messages to the sink and this information is sent to the controller via the new packet. For the clock timer, we had to hard code the flow table refresh timers.

2.2 Related Work

The IoT network domain's main challenges are resource constraints like memory, energy, and interference in the radio medium. In addition to that, implementing SDN in this environment adds overhead to devices. To overcome these challenges, several SDN-IoT architectures were proposed. Many of this architecture either used a customized external controller or an embedded controller. In this section, we present existing works related to our proposed framework. We thoroughly review the literature related to our work and point out the gaps in the present solutions.

2.2.1 SDN-IoT based designs

μSDN [6] is a lightweight SDN architecture for IoT networks. The authors propose a modular architecture, where the μSDN stack sits above the IP layer within the IEEE 802.15.4 stack. It has added interoperability with underlying protocol and IPv6. The authors tackle various SDN in IoT challenges by introducing different types of optimizations like *Protocol*, *Architecture*, *Memory* and *Controller*. In protocol optimization, it eliminates the fragmentation by reducing the packet size and reduces the packet transmission frequency. For architecture level optimization, it introduces source routing, which automatically selects the optimum path to reach the final destination. It optimizes the memory of devices by re-using the flow table match-actions.

For instance, it reduces the repeated entries for the same forwarding actions. It does so by using flow refresh timers that retain a specific flow rule for a fixed amount of time. In the controller optimization, it uses an embedded custom controller, which reduces the time for communication between nodes and the controller.

The μSDN stack provides a layered architecture and API to separate core function handling from the specifics of the SDN implementations. The architecture can be split into two processes that are Stack process and the Core process. The Stack process includes the Controller adapter, SDN engine, and Driver. The Controller adapter exposes an abstract controller interface to the SDN layer that allows μSDN protocol to be switched out to implement any other protocol. SDN engine defines how messages to and from the controller are handled. SDN driver provides API for the SDN engine by defining how the flow table is handled. Also, it handles functions like setting up a flow table like creating firewalls, handling routing, or data aggregation path [6]. μSDN uses a lightweight protocol for controller communication. It uses User Datagram Protocol (UDP) over Datagram Layer Security (DTLS) for communication with controller outside mesh. It is highly optimized to ensure that there is no packet fragmentation. The Core process includes Controller Discovery, Controller Join, Configuration and metrics, Flowtable structure, and Overhead reduction [6].

The μSDN core processes handle the complete operation of the IoT network. The three primary operations are *controller discovery and join*, *Node Status Update (NSU)* and *routing*. The first phase of μSDN involves discovering the controller using the RPL control message. When the controller node receives a DAO message from all nodes, it sends a CONF message to configure various operations for the nodes. The next step is that each node periodically sends information regarding its link quality, active nodes, energy, etc. to the controller. These messages are the NSU control message and it allows the controller to gather network information. After the initialization of all nodes, the source node starts sending a data packet to the destination based on the flow rule installed in its flow table. The intermediate nodes use source routing to forward packets because of the RPL configuration [6]. Even though μSDN offers a good solution but it lacks extensive evaluation of its performance. It compares its solution with a standard RPL based network without SDN configuration.

Galluccio *et al.* propose SDN-WISE [13]. The SDN-WISE architecture extends

the OpenFlow approach of SDN. They provide a stateful solution where SDN-WISE sensor nodes are encoded data structures namely WISE state arrays, Accepted ID arrays, and WISE flow table. The nodes act as Finite State Machine (FSM) and actions are based on rules installed in the flow table. Simple controller logic is used to control the nodes. The controller logic allows nodes to be programmed, where they can perform certain actions under one state, like forwarding, dropping packets based on rules in flow tables. It uses a customized flow table namely the WISE flow table, which contains matching rules and associated action [13]. The matching rule is based on the packet header and node state. If rules are matched then the action is taken, else packet is sent back to the controller. The SDN-WISE proposes to be energy-efficient by introducing duty cycling and data aggregation [13].

In SDN-WISE architecture sensor nodes and sink, nodes are distinguished. A sink node is a gateway between the data plane and the control plane. Each sensor node consists of a transceiver and micro control unit (MCU). A forwarding layer runs MCU and handles the packet as specified in the WISE flow table. It has four important layers, which handle the overall operation *forwarding, In-Network Packet Processing (INPP)*, *Topology Discovery (TD)*, *Topology Manager (TM)* and *Adaptation*. The INPP is responsible for functions like data aggregation. The INPP concatenates all similar data packets, which need to be sent the same path. This helps in reducing network overhead. The TD layer gathers local information about nodes and controls them as per controller instructions. It also allows the application layer to access information using APIs. The TM layer is basically used to send information from node to controller like Signal-to-Noise Ratio (SNR), the energy level of nodes, etc. It controls all messages between the controller and the sink node. The adaptation layer is responsible to format messages received from the sink node so that WISE-Visor can process it and vice-versa. WISE-Visor is similar to a network operating system that provides an abstraction to develop network functions or services [13]. SDN-WISE is one of the first stateful solutions that proposes a solution for end-to-end SDN implementation. However, it lacks any form of resource optimization. It has no performance evaluation with other works.

Sensor OpenFlow [31] is one of the earliest proposals for using SDN in IoT networks. Their work highlights the challenges of communication with the control

plane. They propose a low-power custom protocol rather than using OpenFlow directly. They also propose energy efficiency through data aggregation and reduce SDN overhead using Control Message Quenching (CMQ). This reduces additional queries on flow table misses and allows the network sufficient time to respond to the request. This work is one of the earliest work, which proposes a good solution but does not show any practical implementation.

CORAL-SDN [33] introduces a centralized network mechanism to adjust protocol functionalities. It also proposes to introduce better network management and reduce resource utilization. They implement this by adopting SDN for IPv6 based IEEE 802.15.4 RPL networks and try to deal with network overhead by reducing multiple RPL transmissions at the beginning and thus provide more resources to SDN protocols. CORAL-SDN architecture uses a customized external controller and does not include any optimizations. They use their own set of protocols and have customized WishFul NOS, which provides an abstraction to develop applications and services. Even though they introduce some sort of optimization to reduce repeated transmissions, but that is limited to memory utilization. Unlike, μSDN, which introduces optimizations at protocol, controller, and at routing levels as well. This is the reason we preferred to use μSDN over CORAL-SDN. Also, they do not show any extensive performance evaluation or compare their work with previous works.

The authors of SD-NOS [4] facilitate the heterogeneity characteristic of IoT by leveraging the controller Network Operating System (NOS). In their work, an innovative integrated network operating system for the IoT, which is obtained as the evolution of the Open Network Operating System (ONOS) is proposed. This integrated NOS allows the sensor nodes to interact with OpenFlow switches and vice-versa. The prototype is built using SDN-WISE [13] architecture. This prototype helps in satisfying the IoT characteristic of heterogeneity.

They modified the different subsystems of ONOS, like Sensor Node, Device Control, Flow rule, and Packet according to the message format of SDN-WISE. They used the SDN-WISE controller as a medium to connect with the ONOS controller with OpenFlow switches connected to ONOS as well. They also show how messages can be sent from IoT networks to wired OpenFlow devices and vice-versa [4]. This prototype is only for demonstration purposes and no real performance evaluation is

done.

Anglers-Christos *et. al* in their work SD-WISE [3] propose to extend the SDN approach in WSNs. They introduce an SD-WISE controller, which is a software package consisting of a network operating system (NOS), called SD-WISE Operating System (SD-WISE OS). It has several network applications that are similar to that of ONOS. The SD-WISE node architecture has several layers, which host some NFV applications. In their work, they basically show SD-WISE OS can be an alternative to the existing ONOS controller. Their work is built on top of SDN-WISE [13], where they introduce the SD-WISE controller instead simple SDN-WISE controller. In this work, they use a customized external controller and do not show performance comparison with traditional SDN controllers like ONOS. Our work was to compare the performance of SDN-IoT architecture using embedded and standard SDN controllers. Since SD-WISE is an extension of SDN-WISE work, it becomes more logical to compare base SDN-WISE with μSDN architecture as no one has compared the performance of these two SDN-IoT architectures.

Traditional WSN network solutions in [12, 14–16, 19, 20] optimize communication energy by deploying edge-disjoint routes or special spanning topologies. SDSense [17] extends those design strategies in wireless SDN. The authors propose an agile SDN based architecture for WSNs. In this architecture, they separate control tasks into two parts, i.e., a local controller at the sensor level and a global controller. The SDSense controller (global) handles all functions like topology management, resource allocation, congestion control, etc. They design a model and develop an objective function to allocate resources optimally. In their work, they evaluate their architecture with spanning-tree structures and non-SDN based architecture. Their work does not show performance comparison with other existing SDN-IoT architectures like SDN-WISE, μSDN, etc. Moreover, their work focuses on the optimization of the controller tasks and does not talk about optimization at low-power devices. The μSDN authors have implemented optimizations at the controller level as well as sensor node level. This is the reason we chose μSDN over SDSense.

Saha *et. al* in their work design an EA-SDN/NFV (Energy Aware Network Function Virtualization) [29] framework for low-power IoT network. They propose to

Approach	IPv6 & RPL	Type	Placement	Performance
μSDN	✓	Customized	Internal	Yes
SDN-WISE	×	Customized	External	No
SD-NOS	×	Standard	External	No
SD-WISE	×	Customized	External	No
Sensor OpenFlow	×	Customized	External	No
CORAL-SDN	✓	Customized	External	No
SDSense	×	Customized	External	Yes
EA-SDN/NFV	✓	Customized	Internal	Yes

Table 2.1: A comparison of features for SDN based works.

reduce the energy utilization deploying NFVs. They develop an ILP (Integrated Linear Programming) problem to minimize the NFV nodes' activation energy. They develop a heuristic to solve the ILP problem and evaluate it using the Cooja simulator. Their work is built on top of μSDN architecture. Also in their work, they show a comparison with other base architectures. However, they do not show whether a μSDN controller would be feasible for low-power IoT networks.

In summary, none of the existing works shows an extensive evaluation of their architecture nor show any comparison with other existing works. Many works propose using a customized controller for IoT networks, and very few show the use of standard SDN controllers. Does this leave us with a dilemma in which SDN-IoT architecture would be useful for the low-power IoT networks? Whether it is feasible to use a standard SDN controller for such networks? In this work, we address these two questions by providing an extensive evaluation of two architectures i.e. μSDN and SDN-WISE architectures. We will show which architecture is better, and further, we will analyze the performance of the customized internal controller and standard SDN controller.

Chapter 3

Design, Methodology and Evaluation

3.1 Research Methodology

In this section, we present our problem definition. We then discuss modifications in node architecture of SDN-WISE and μSDN while comparing their performance. We also show our evaluation results and discuss their performance trend and point out the reasons behind their behavior.

3.2 Problem Definition

The IoT networks have been on the rise exponentially especially in the last few years [25]. The growth is expected to continue as we move towards smarter applications. With the addition of billions of new devices in networks, it will become very difficult to manage such a huge infrastructure with the current legacy network [18, 25]. SDN can provide a good solution in this case because of its advantages like network reconfiguration, global knowledge, etc [18]. Several architectures were proposed as mentioned in the previous section. The main problem is that there has been no effort to benchmark any of this architecture. We found μSDN and SDN-WISE to be prominent ones as they show proper implementation on Contiki OS using the Cooja network simulator. The question then arises, which architecture would be more suitable for a low-power IoT environment. Thus, we compare the performance of these two architectures using the Cooja simulator.

The μSDN uses an embedded controller, i.e., the controller is within the IoT network. The embedded controller is similar to other nodes except the fact that it has control logic and programs the neighboring nodes with flow rules. On the other hand, SDN-WISE uses an external customized controller. Thus, to ensure fair comparison we decided to include various controller modules of SDN-WISE within the sink node. In the next section, we discuss modifications in SDN-WISE node

architecture.

The second part of our work deals with controller performance in SDN-IoT architecture. After establishing, which SDN-IoT architecture is better in performance metrics mentioned above, we connect ONOS, a standard external controller to μSDN architecture. To integrate the ONOS controller with μSDN ,we have to adapt the ONOS to understand messages from μSDN nodes. Thus, for this, we design an ONOS application that adapts the messages incoming from nodes and converts it to a format that ONOS can understand. In this second part, we ask the research question.

- *RQ 1* is it feasible to use an embedded controller in a low-power and low-bit rate IoT applications?

- *RQ 2* Will a standard external SDN controller outperform an embedded controller considering external controller is equipped with better memory and processing capabilities?

Thus, to answer our questions we compare the performance of both controllers for low and high bit rate environment. We implement this using the Cooja network simulator. In the next section, we will discuss the modified μSDN architecture and various components of ONOS subsystems that we had to develop and extend to allow messages to be understood by ONOS.

3.3 SDN-WISE Node Architecture Modifications

In the first part of our work, we compare two SDN-IoT architectures, i.e., μSDN and SDN-WISE (with embedded control logic). We have seen in the previous section SDN-WISE is one of the earliest architectures to provide an SDN solution for IoT networks. SDN-WISE distinguishes between the sensor node and the sink node. The sink node act as a gateway that connects the IoT network to the external controller. It also has an adaptation layer that converts the node message format to that of the external controller. SDN-WISE uses a customized WISE controller, which programs the underlying nodes. Since in the first part of our work we compare μSDN and SDN-WISE architecture, it is important to understand the difference and similarities between their node architecture. We have also compared SDN-WISE with external

controller to μSDN with the results included in [22]. The results confirmed that μSDN performs better than SDN-WISE.

- The SDN-WISE uses an external controller whereas μSDN uses an embedded controller.
- SDN-WISE uses customized protocols like TD, INPP, etc. for communication. Whereas μSDN uses standard protocols like RPL, UDP, etc. for communicating with the controller.
- SDN-WISE at Medium Access Control (MAC) layer uses Contiki-based Rime protocol whereas μSDN uses standard Carrier Sense Multiple Access Collision Avoidance (CSMA/CA) [26].
- Both SDN-WISE and μSDN use the ContikiMAC [26] protocol for Radio Duty Cycling (RDC).

Thus, based on the above differences, the notable difference is the use of an external controller in SDN-WISE and embedded controller in μSDN architecture. To solve the difference and make sure that our comparison remains unbiased, we decided to modify node architecture. We included three controller modules i.e. *Topology Manager (TM)*, *Openpath* and *Response* modules within the sink node. Fig. 3.1 shows modified SDN-WISE node architecture with control modules within the sink node. We modified only the sink node modules and rest other nodes were kept untouched.

Each module performs specific functionality of controller within a sink node.

- *TM module:* The use of the TM layer is to collect all sensor node information like Signal-to-Noise Ratio (SNR), residual energy, node position, etc. from underlying nodes. This allows the controller to get a consistent view of network topology. Earlier this layer was within WISE-Visor, which required adaptation from sink node to format message. Now since we moved this layer within the sink node, no adaptation is required.
- *OpenPath:* This is basically a packet, which is sent by controller to set up a route between different nodes. The OpenPath packet is also sent by the controller to scan a network if a controller wants to update itself immediately instead of waiting for an update from the nodes.

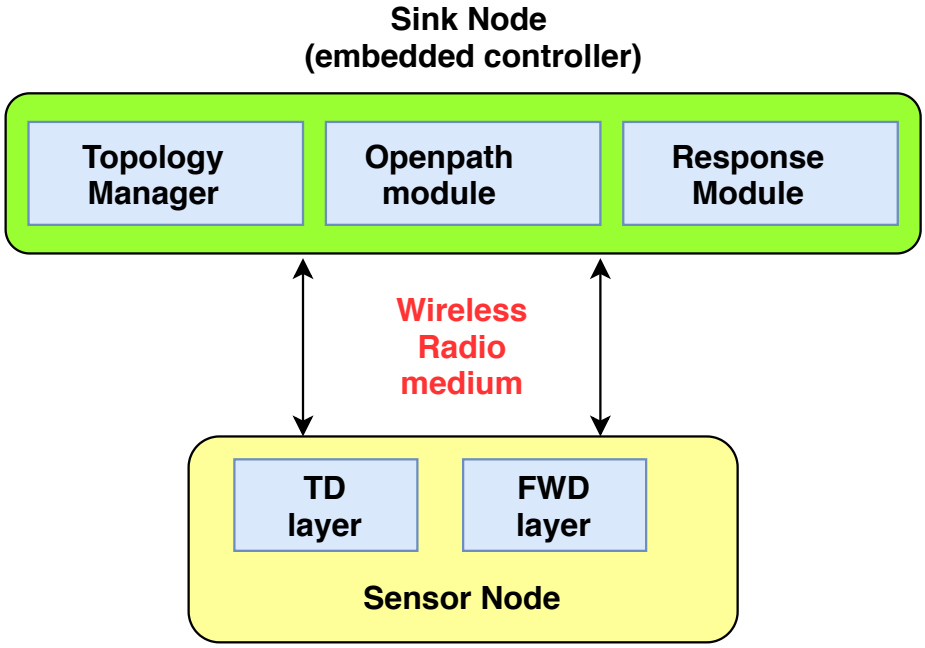


Figure 3.1: SDN-WISE node architecture with embedded controller.

- *Response* These are packets that are generated by the controller in response to flow request packets. The response packets send flow table rules to the corresponding node.

Also, there are *CONFIG* messages, which are sent by the controller to control the behavior of nodes like report timers, rate of generating payload, etc. but it needs to be sent by the controller after all nodes have joined the topology. The μSDN also has a similar message called as CONF, which is used to configure Flow entry lifetime, Node update time, etc. but the controller sends CONF message automatically as soon as node joins the RPL DAG.

Another difference between the μSDN and SDN-WISE architecture is at the MAC layer. The μSDN architecture uses CSMA/CA [26] protocol whereas SDN-WISE uses Rime protocol at their MAC layers, respectively. Thus, we changed the SDN-WISE configuration and added CSMA/CA driver at its MAC layer. This difference is important to fix because the MAC driver decides whether the communication energy consumption will be high or low. Thus, to remain unbiased in our comparison we decided to use CSMA/CA in both the architectures.

The table 3.1 shows the protocol stack that we used for evaluating both the

Layer	μSDN	SDN-WISE
Application	Embedded	SDN-WISE controller
Network	IPV6 + RPL	TD INPP Forwarding
MAC	CSMA/CA	CSMA/CA
RDC	ContikiMAC	ContikiMAC
Physical	IEEE 802.15.4	IEEE 802.15.4

Table 3.1: The Table showing protocol stack of μSDN and SDN-WISE

architecture. The difference lies only at the network layer where μSDN uses RPL protocol whereas SDN-WISE uses TD, INPP, and forwarding protocols. So, the node architecture changes in SDN-WISE enable us to transform the sink node into an embedded controller. The change in MAC layer driver also ensures that node energy consumption comparison of SDN-WISE and μSDN remains the same at the MAC layer and no particular architecture gets the advantage of using a different MAC driver. Thus, these changes enable us to ensure that our performance comparison remains unbiased.

3.4 μSDN-ONOS Architecture

The low-power IoT networks include devices that operate with limited resources like memory, energy, etc. They also operate at a low bit rate so to ensure less communication energy is consumed. We have seen in Chapter 2 there are several architectures proposed to implement SDN in low-power IoT networks. These architectures use either an embedded or external controller. The embedded controller is similar to other nodes except the fact that they are programmed with additional control logic. On the other hand, we have an external controller that run control logic on a server and are equipped with better processing resources. We need to understand which controller behaves how in the context of performance, resource consumption, and scalability. Thus, we compare an embedded controller with a standard SDN controller like ONOS. For this purpose, we choose the μSDN node architecture based on the initial investigation. In this section, we will discuss μSDN-ONOS architecture, where we will discuss various subsystems of ONOS and the necessary changes.

3.4.1 Proposed Architecture

We have used μSDN as our base architecture based on results that we obtained after comparing it with SDN-WISE architecture. We then integrate μSDN with the ONOS controller. We had to develop an ONOS application that acts as an adapter to convert messages received from μSDN node to a format that is compatible with the ONOS controller. Similarly, ONOS after processing those messages sends them back to μSDN node with the format similar to μSDN nodes.

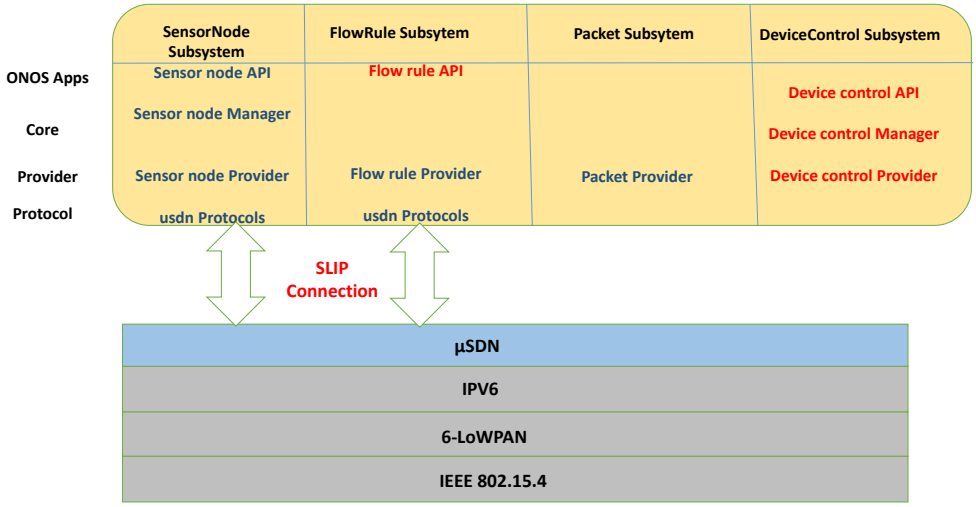


Figure 3.2: The μSDN-ONOS architecture

The Fig. 3.2 shows an architecture of integrating μSDN and ONOS controller. We have used four different subsystems of the ONOS controller. The subsystems are vertically sliced along all the ONOS layers. Each subsystem has different components in different layers marked in blue and red colors. The blue color components are developed from scratch whereas the red color components are extended to support μSDN events. These components are bundled into one single application and associated with a ProviderID. This ProviderID is used to activate the application when required. A forwarding application at Sensornode subsystem is used to send processed messages back to μSDN node.

We have used the Serial In-Line Protocol (SLIP) connection to ensure connectivity between μSDN architecture residing in Cooja and ONOS instance that runs on

the host machine. Since the control logic is now transferred to ONOS, the μSDN embedded controller acts as a sink node. We will describe each ONOS subsystem and its workflow using a flowchart.

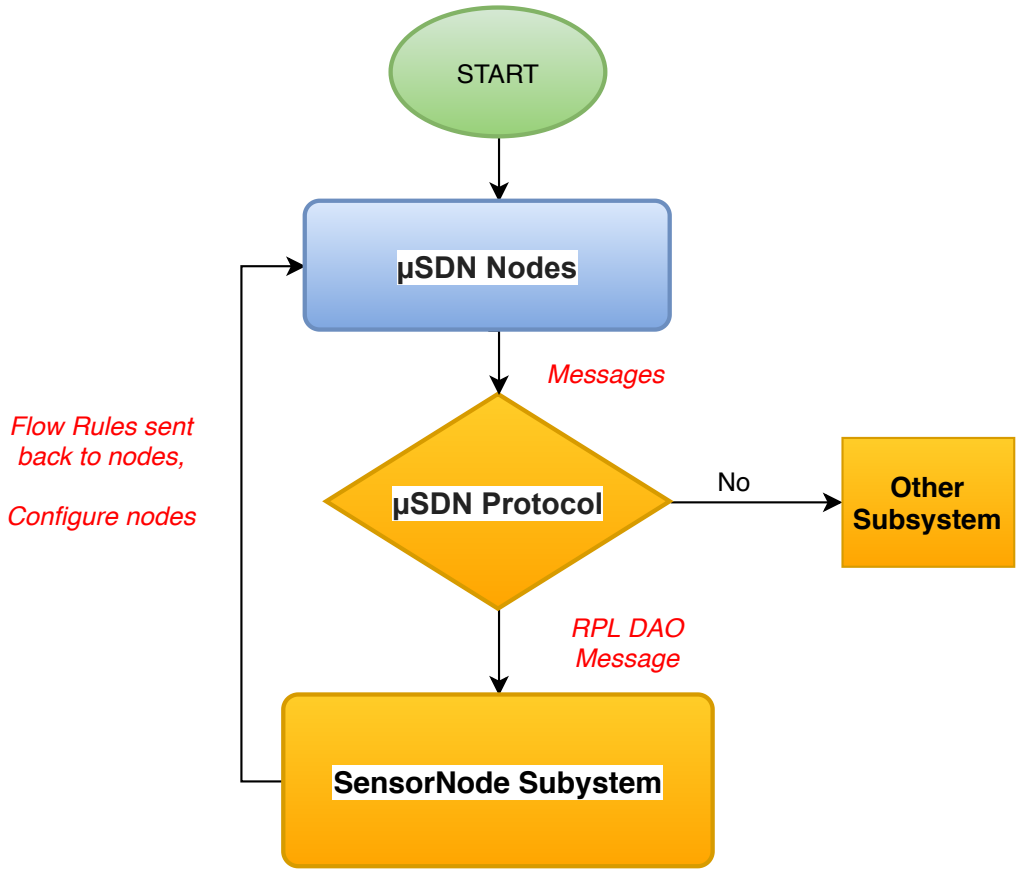


Figure 3.3: The μSDN-ONOS SensorNode Subsystem workflow.

Workflow of Proposed Architecture:

The Fig. 3.3 shows the flow diagram for SensorNode subystem. The SensorNode as the name suggests is used to handle all the information related to sensor nodes present in the Cooja simulator. When a sensor node sends a RPL DAO message, it is received at the Protocol layer. The μSDN protocol component extracts information like source id from the message and raises an event in SensorNode Provider component. The SensorNode Provider actively checks node status from the Protocol layer and wraps this information into ONOS specific format and sends it to SensorNode Manager. At the Core layer, SensorNode Manager stores the node information, which allows ONOS to maintain a consistent view of network topology. This view provides global knowledge to ONOS and helps it to take the right decision like packet

forwarding to the correct node. At the Application layer, the SensorNode API introduces access to sensor node-specific data structures that allow ONOS to convert control messages back to μSDN specific format. We use the SensorNode forwarding application to forward these messages back to μSDN node. The point to be noted is that ONOS receives a message from the sink node but the source can be any node within the network topology.

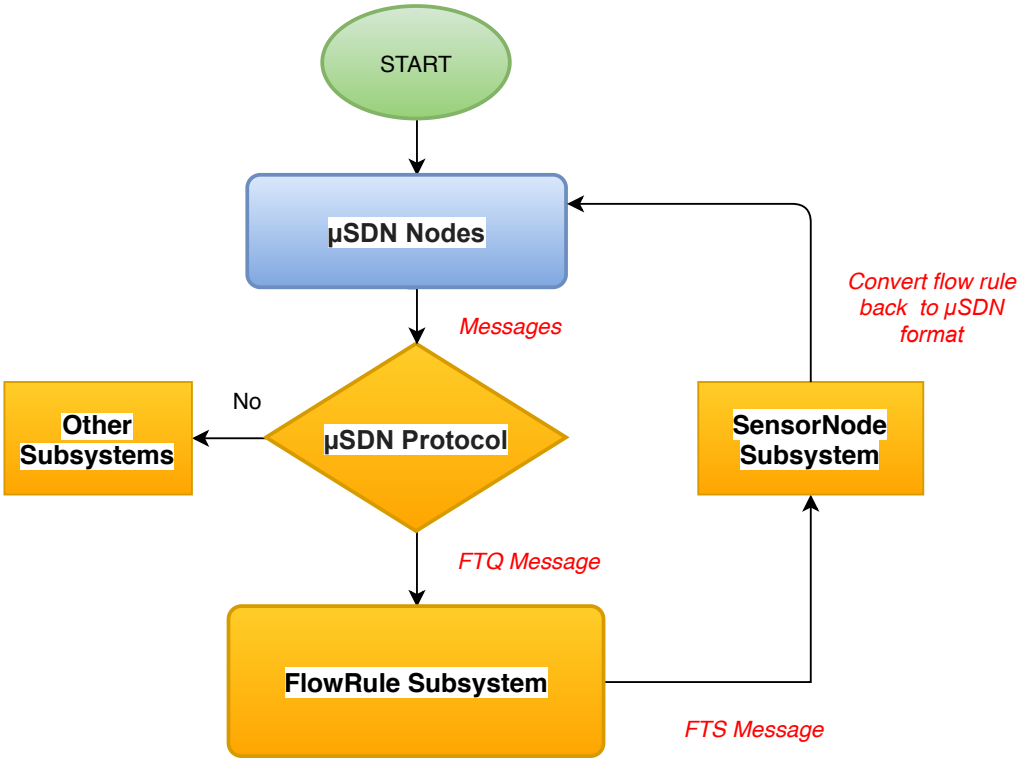


Figure 3.4: The μSDN-ONOS FlowRule Subsystem workflow.

The Fig. 3.4 shows the flow diagram for ONOS FlowRule subsystem. The FlowRule Subsystem is a combination of new as well as extended components. The new components are designed to handle flow table structures of μSDN flow table. The ONOS subsystems are originally designed to handle OpenFlow specific flow table formats. The μSDN protocol extracts the FlowTable Query (FTQ) message and sends it to the FlowRuleProvider. The FlowRule Provider is used to handle μSDN FTQ messages. These messages are wrapped in ONOS specific format and based on the decision made by ONOS the new flow rules, i.e., FlowTable Set (FTS) message need to be sent back to the node. The FlowRule API has been extended to support rules and flow table structures of μSDN. These FTS messages are then forwarded using

SensorNode subsytem and the messages are converted into μSDN format using the flow table structures provided by FlowRule API.

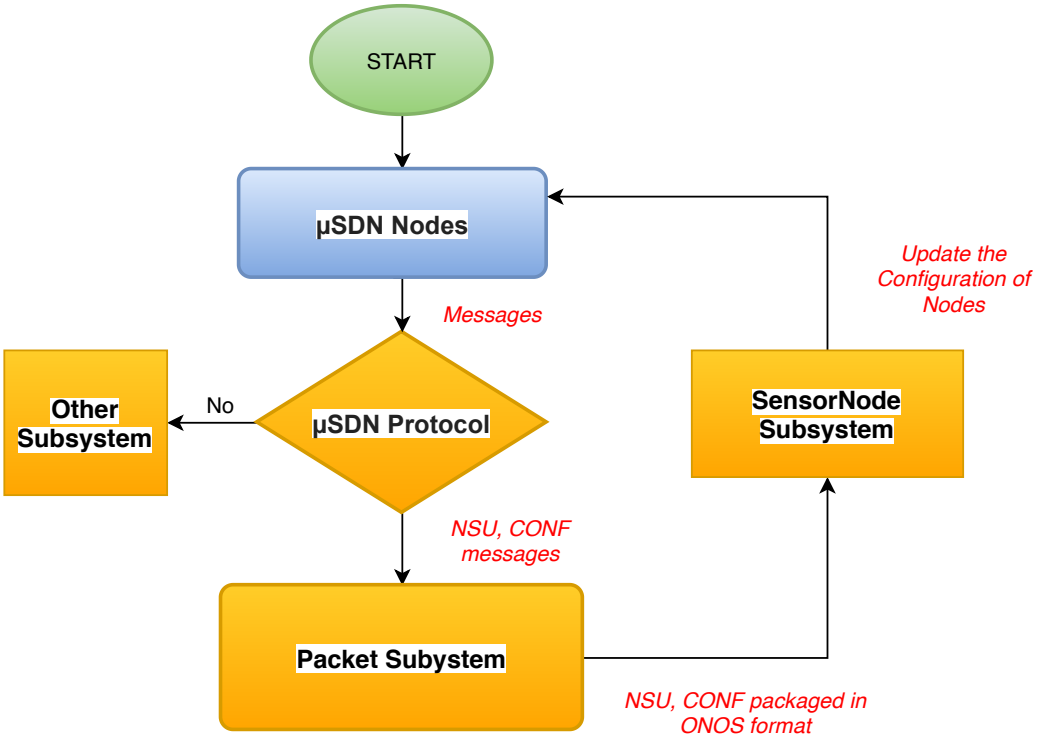


Figure 3.5: The μSDN-ONOS Packet Subsystem workflow.

The Fig. 3.5 describes the flow diagram for Packet subsystem. The Packet Subsystem is designed only for the Provider layer. The main responsibility of the Packet Provider is to handle different types of messages arriving at ONOS. Even though the FTQ messages and RPL DAO packets are handled by FlowRule and SensorNode subsystems, respectively, there are many other messages like Node Status Update (NSU), CONF, etc., which need to be handled. The NSU messages carry important information regarding nodes like energy level, RSSI, neighbor node, etc. This information is important for the ONOS controller for different operations like Topology management, routing, etc. Therefore, the Packet Provider wraps these messages into ONOS specific format and forwards them to SensorNode subsystems which further updates the μSDN nodes.

The Fig. 3.6 illustrates the flow of messages in DeviceControl subsystem. The DeviceControl subsystem extends the current scope of standard ONOS operation. It not only incorporates network flows but also the network device's status. Whereas a

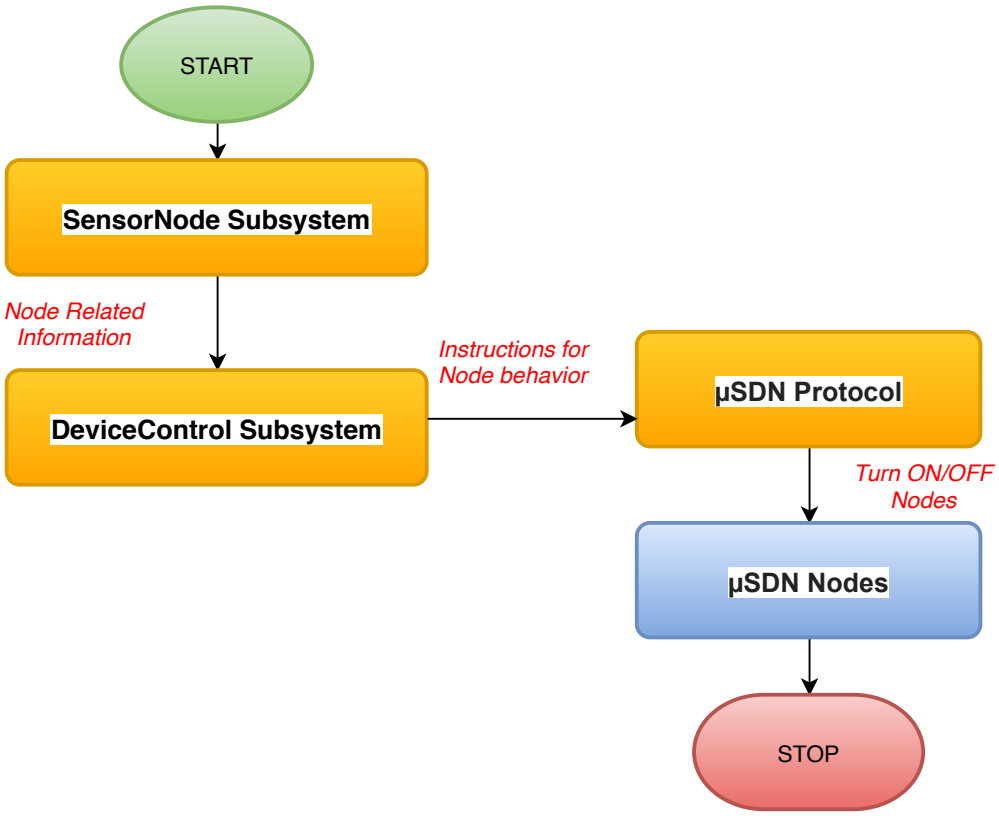
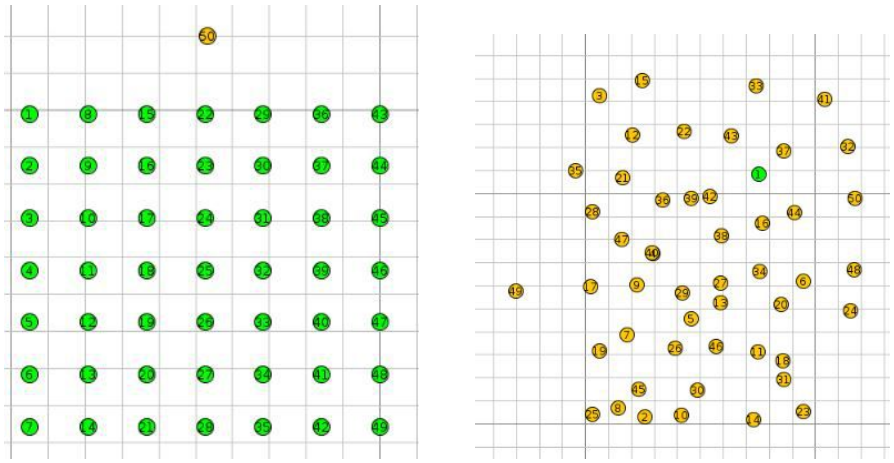


Figure 3.6: The μSDN-ONOS DeviceControl Subsystem workflow

standard ONOS operation for a wired network considers all events like flows, devices, etc. as equal and doesn't serve according to priority. In a low-power IoT network, the actions of ONOS are triggered proactively on the occurrence of any events. Thus, in this case, ONOS needs to decide, which device operations are necessary and trigger actions accordingly. This feature is very useful in low-power environments. The DeviceControl API in northbound provides instructions depending on incoming traffic. These instructions are pretty basic like turn on/off a sensor node. The instructions are based on information received from the SensorNode subsystem like active nodes, newly joined layer, etc. The DeviceControl Manager in the core layer is responsible for keeping track of the instructions and then, forwarding them to the lower layers in order to be sent to the appropriate device. In the Provider layer, the DeviceControl Provider receives the device instructions and converts them into μSDN message format. The messages are then forwarded to sensor nodes via the μSDN protocols. The point to be noted is that here flow rule operation is top-down.

μSDN Node Modification: Now since the control logic is now managed by the



(a) Grid Topology (b) Random Topology

Figure 3.7: Examples of a)Grid and b)Random Topology.

ONOS controller, we transform the embedded controller into a sink node that acts as a border router. The sink node continuously listens to the port number on which the ONOS application is running. The connection between the sink node and the ONOS port is established using a SLIP utility. This allows messages to be sent from μSDN nodes to ONOS and vice-versa. Thus, with all these changes we can integrate ONOS with the μSDN architecture.

3.5 Evaluation

In this section, we present the environment setup that we use to evaluate both μSDN and SDN-WISE architecture that are proposed in the previous sections. In the next section, we will talk about setup for comparing embedded and ONOS controller. In the end, we will discuss results and provide an appropriate reason behind the trends that we observe after the evaluation.

3.5.1 Node Architecture Comparison

First, we compare μSDN and SDN-WISE architecture with both using an embedded controller to ensure unbiased comparison. To compare both the architectures we use the Cooja network simulator based on Contiki OS. We simulate the network for 50 node grid as well as random topology.

Environment: We use the Cooja network simulator based on Contiki OS. The

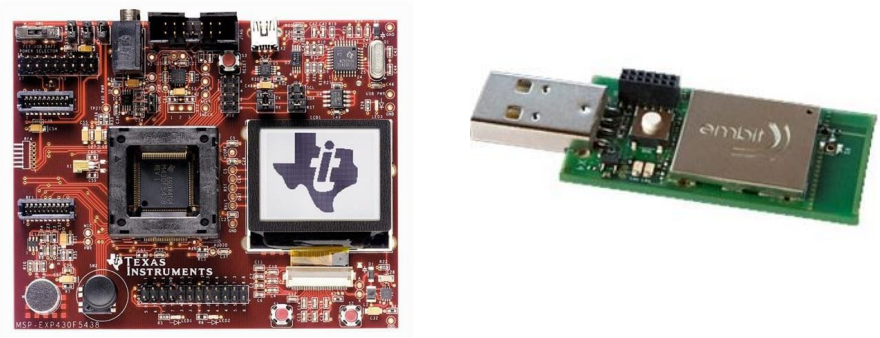
Contiki is an operating system for low-power wireless IoT devices. Contiki is designed to run on devices that have low memory, low energy, and low communication bandwidth. It provides three networking mechanisms namely TCP/IP stack, IPV6, and Rime stack. The IPV6 mechanism supports the RPL protocol for a low-power IoT network. The Contiki system includes a sensor simulator called Cooja, which simulates Contiki nodes. The nodes can be programmed using programming languages like C and Java. We deployed SDN-WISE embedded controller functionality using C language.

Topology and IoT Configuration: For the node architecture performance evaluation we use both Grid (Fig. 3.7a) and Random topology (Fig. 3.7b) of 50 nodes. In each case of topology, we consider 10 source nodes, one controller node, and rest as relay nodes. In both the topologies, each node has a transmission range and interference range of 50m. For generating random nodes, we use different random seeds for each simulation run. We select ten source nodes to avoid overwhelming the nodes, especially for SDN-WISE. If we configure all nodes as sources, it can lead to traffic congestion and impact the performance. Since our primary focus is to compare different performance metrics of both architectures in a fair way, we ensure that both networks are not overwhelmed with any congestion. However, in the case of controller comparison we need to check the controller's performance for a varying load. Thus, we vary the number of sources from low to high (e.g. up to 50). We keep topology size upto 50 nodes because of the limitations in Cooja simulator. It blocks all nodes above 50 and do not show their output in log window [24].

In case of, μSDN we use Texas Instruments (TI) MSP430F5438CPU (Fig. 3.8a)on EXP5438 platform and CC2420 radio. For SDN-WISE we use EMB-Z2530PA (Fig. 3.8b) sensor nodes. For both architectures, we consider the power supply as 3V. The energy consumption rate for the mentioned radio is 17.7mA for transmission and 20.01mA for reception. The Link quality is kept at 90% as it is one of standard setting used during evaluation of different metrics in Cooja simulator [6].

Performance Metric: For node architecture evaluation, we used following performance metric.

- *Transmission Energy Consumption:* the total energy consumed by node for transmission all packets.



(a) (b)

Figure 3.8: Examples of a)MSP430 CPU and b)Zmote sensor.

- *Controller-Node Round-Trip Time:* the total time to send a query packet from a node to the controller and receive back the response at the node.
- *Packet Delivery ratio:* the ratio of number of packets received at destinations to the number of packets sent by sources.
- *Latency between source-destination pair* the time taken by packets to reach the destination node from a source.

Parameters	**Setting**
Duration	5 min
Transmission Range	50m
Interference Range	50m
Max Bit rate	9 bit/sec
Link Quality	90%
Radio Medium	UDGM (distance loss)
RPL mode	Non-storing mode
RPL Route Lifetime	5min
Flowtable Lifetime	5min

Table 3.2: The Table showing parameter settings used in evaluation.

The table 3.2 describes various parameters and their corresponding settings in the Cooja simulator that were used in the evaluation.

Scenario: We evaluated the architectures on two scenarios.

- *Bit rate:* the rate at which a node can transmit any packet. We varied the bit rate and calculated the metrics.

- *Hop distance:* the number of intermediate nodes through, which packet has to travel in order to reach the destination. We evaluated the metrics by varying the hop distance.

Evaluation Setup: We evaluated both μSDN and SDN-WISE architecture based on Transmission energy consumption, Controller-Node RTT, Delay between given source-destination pair, and PDR on both grid and random topology of 50 nodes. We vary the hop distances from the source nodes and the destination node to investigate the effect of network size on the above-mentioned performance metrics. We also investigate the effect of varying bit rates on the performance metric. For each set of evaluations, we run the simulation 50 times. We take the average of the simulations with a 95% confidence interval.

3.5.2 Controller Comparison

The second part of our work was to compare the performance μSDN with an embedded controller and μSDN with an ONOS controller. The major focus of this part was to compare two controllers. Therefore, we used performance metrics that are specific to controllers.

Environment: The environment was similar to the previous experiment. We used the Cooja network simulator and deployed μSDN and ONOS architecture. We connect μSDN to the ONOS controller using a SLIP connection where the sink node acts as an RPL border router to facilitate the connection. The border routers are routers that can be found at the edge of a network. Their function is to connect one network to an external network. The border router receives a prefix, i.e., IP address of external network through the SLIP connection and it communicates with the rest of the nodes present in the network. Once the border router receives the prefix, it sets itself as DAG root after, which all nodes start communicating with the border router using RPL protocol. The SLIP protocol is an encapsulation of internet protocol designed to work on serial ports [28]. The microcontrollers and sensors still use SLIP protocol because of its small overhead.

Topology and Controller Configuration: For controller comparison, we evaluate the performance metrics on-grid as well as a random topology for 50 node topology. In the case of, μSDN ONOS we have 50 nodes including the sink node, which

is connected to the external ONOS controller. We have all nodes as source nodes to maximize the flow requests sent to a controller. We use a similar node configuration that we used for μSDN in the previous experiment. We use the ONOS 2.3 version, which runs on a single machine and can have multiple instances running at the same time. The ONOS is build using Bazel build configuration. We have our adaptation application installed on ONOS, which needs to be activated before we start our simulation.

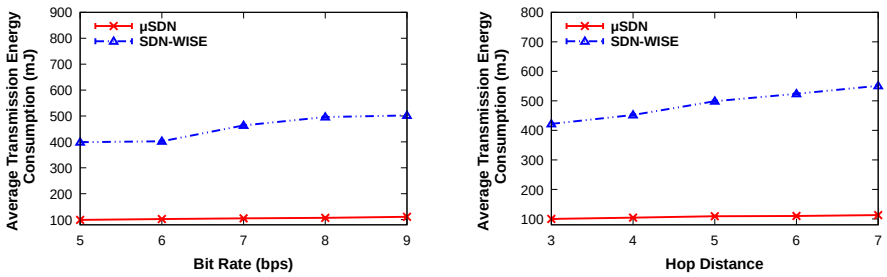
Performance Metrics: We use following metrics to evaluate both the controllers.

- *Throughput:* the ratio of the number of flow setup request that controller responded per second to the number of requests it received per second.
- *Delay in Controller Response:* the total time taken by the controller to respond to a flow setup request.
- *Topology discovery time:* the total time taken by the controller to detect all nodes that are present in the network.
- *Topology update time:* the total time taken by the controller to detect a link failure in a network and update the topology.

The Cooja simulation settings are similar to that of the previous experiment as shown in Table 3.2. The only change is that we have a max flow request rate of 102 flows/sec. ONOS being a wired and resourceful controller can handle a large number of flows ranging in thousands. This would be unfair to compare with the μSDN controller, which lacks resources. Thus, the bit rate chosen for this experiment is between 50 to 100 bit/s as this is the range for a low-power IoT network, which runs using the IEEE 802.15.4 standard [7].

Scenarios: The controller specific performance metrics were evaluated by varying following parameters.

- *Flow Setup Request Rate:* the number of flow setup requests sent by source nodes to controller per second.
- *Number of Nodes:* the total number of nodes present in network topology.



(a) (b)

Figure 3.9: Average Transmission Energy Consumption in Grid Topology.

- *Number of Links Failed:* the total number links that are failed in a network.

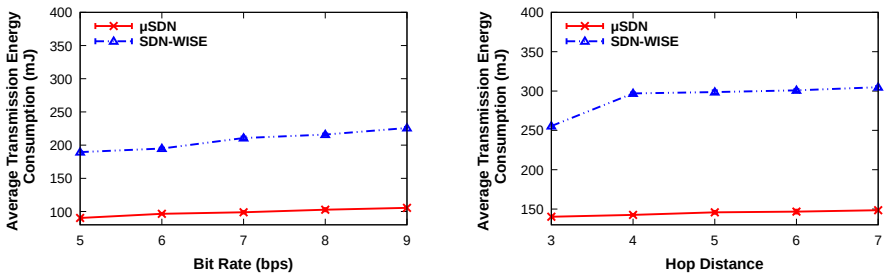
Evaluation Setup: We evaluated both μSDN controller and ONOS controller based on Throughput, Controller Delay in response, Topology discovery time, and Topology Update time on both grid and random topology of 50 nodes. We vary the flow request rate, the number of nodes and links failed to investigate the controller performance. We also investigate the ONOS controller performance by enabling the multi-thread option (not included because flow rate is not too high). For each set of evaluations, we run the simulation 50 times. We take the average of the simulations with a 95% confidence interval.

3.6 Discussions of Results

In this section, we will first discuss the evaluation results of the performance of μSDN and SDN-WISE node architectures with an embedded controller. Then, we discuss the evaluation of μSDN embedded controller and μSDN with an external ONOS controller. We refer to SDN-WISE with an embedded controller as SDN-WISE. Similarly, for μSDN-ONOS architecture, we refer to the border router as a Sink node. We will discuss the reason behind their performance trend in low-power IoT networks.

3.6.1 Node Architecture Evaluation

In this section, we will show the evaluation results of μSDN and SDN-WISE architecture. We compare the different performance metrics as discussed in the previous section 3.5.1.



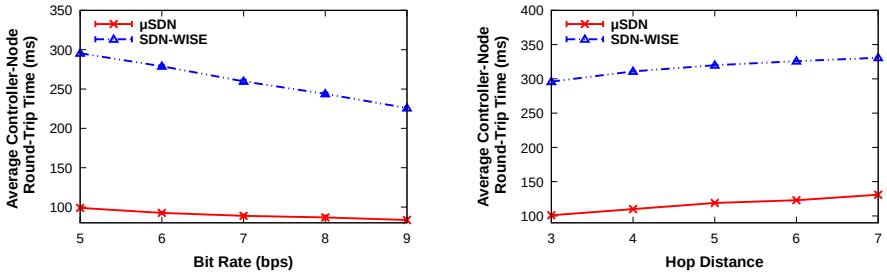
(a) (b)

Figure 3.10: Average Transmission Energy Consumption in Random Topology.

Effect of Bit Rate and Hop Distance on Transmission Energy Consumption: We can see Fig. 3.9 and Fig. 3.10 for the average transmission energy consumption of nodes of μSDN and SDN-WISE for varying bit rate and hop distance. We consider transmission energy for each packet transmission including Data packets, FTQ, NSU, etc. From the figure, it is clear that the energy consumption of μSDN nodes is lower than SDN-WISE. The main reason behind this can be the stateful nature of SDN-WISE nodes, which consumes more energy to maintain its state arrays. Also, it has a micro-control unit that can be responsible for more energy consumption. μSDN, on the other hand, uses stateless nodes that do not have complex arrays and therefore consume less energy. Also, it implements some optimization techniques like protocol optimization that prevents each node from re-transmitting the same packet. The μSDN uses flow table refresh timers that allows nodes to retain a flow entry and allows quicker flow match. This prevents FTQ transmissions to controller. This helps in reducing overall transmission energy consumption.

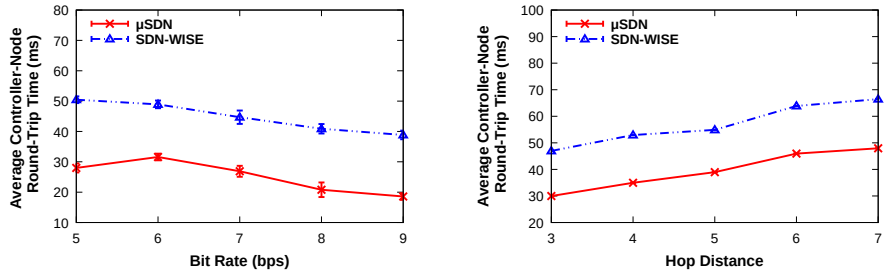
We see some spike in random topology energy consumption for the μSDN architecture, the main reason behind this is because nodes are placed close to each other, which leads to interference and leads to the dropping of packets. These packets need to be re-routed using a different route. Thus, this leads to more energy consumption.

Effect of Bit Rate and Hop Distance on Controller-Node Round-Trip Time: We can clearly see from Fig. 3.11 and Fig. 3.12 that controller-node round-trip time for μSDN is lower than that of SDN-WISE. The main reason behind this performance trend is due to presence of source-routing in μSDN which prevents intermediate nodes to transmit control packets. This reduces number of control packets arriving at controller and allows better processing in quick time. Whereas, for



(a) (b)

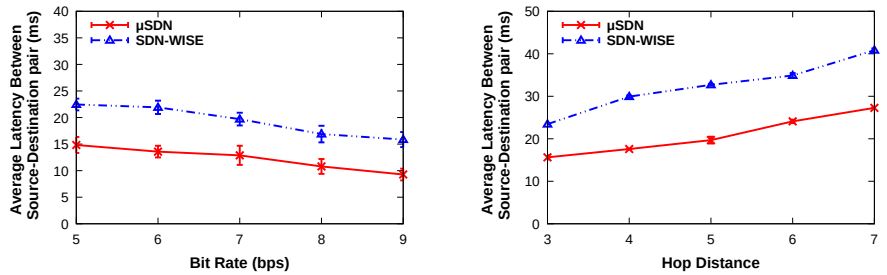
Figure 3.11: Average Controller-Node Round-Trip Time in Grid Topology.



(a) (b)

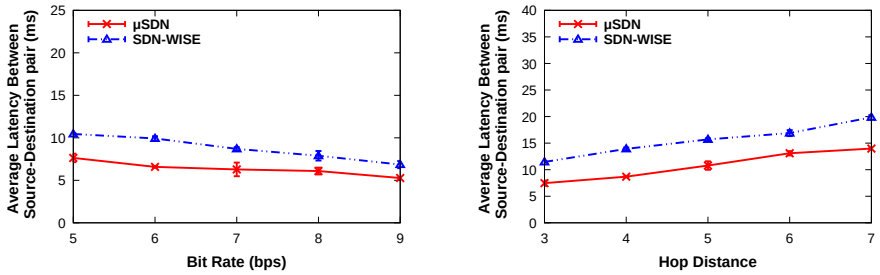
Figure 3.12: Average Controller-Node Round-Trip Time in Random Topology.

SDN-WISE the intermediate nodes also transmit control packets that leads to more processing time at controller, which further affects the overall Round-Trip time. In random topology, we observed that μSDN RTT is almost similar when control packets from source nodes are considered and the time to respond for control packets from intermediate nodes is ignored. Also, we observed a lot of interference is introduced in nodes while sending packets in SDN-WISE for random topology.



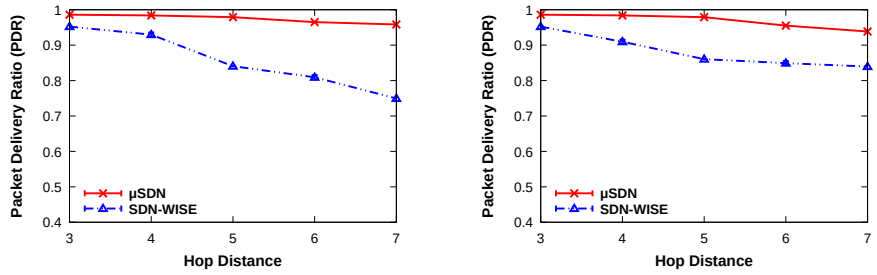
(a) (b)

Figure 3.13: Average Latency between Source-Destination in Grid Topology.



(a) (b)

Figure 3.14: Average Latency between Source-Destination in Random Topology.



(a) (b)

Figure 3.15: Average PDR in a)Grid and b)Random Topology.

Effect of Bit Rate and Hop Distance on Latency between Source Destination pair: The Fig. 3.13 and Fig. 3.14 show the results for latency in both μSDN and SDN-WISE architecture in grid and random topology, respectively. We can see μSDN has less latency compared to SDN-WISE. The point to be noted is that communication happens only at the data plane. The main reason behind this performance is due to difference in flow table structure. The μSDN flowtable is highly optimized and allows flow match on packet's index rather than on specific header fields. On the other hand, SDN-WISE has pretty complex flow table and flow match has to satisfy several header details of a packet. If conditions are not met, the packet is dropped and source has to re-transmit the packet, which can lead to traffic congestion. The μSDN has reduced packet re-transmission frequency that leads to less congestion. The one more feature of μSDN flow table is that controller can configure a *whitelist* that is processed before main flow table. The packets can be matched at *whitelist* and forwarded further to destination. This reduces the latency in packet transmission. These optimizations in μSDN flow table leads to less latency in μSDN than SDN-WISE.

Effect of Hop Distances on Packet Delivery Ratio: From Fig. 3.15 we can clearly see that μSDN has better PDR than SDN-WISE as the hop distance increases. This can be attributed to various μSDN optimization techniques like Architecture and Protocol optimization that they implemented in their architecture. The Architecture optimization implements flow table refresh timers which retains a flow entry for longer time and prevents packet drop. On the other hand, if SDN-WISE flow table doesn't find a match it drops the packet. The Protocol optimization prevents re-transmission of packets by source nodes. While simulating random topology we observed a lot of interference as nodes were placed close to each other, which led to interference. But still, μSDN had better PDR than SDN-WISE. The reason for the PDR trend in random topology is that interference-aware routing in μSDN, which allows μSDN nodes to re-route the packets via another path and helps in improving PDR by a significant margin. The SDN-WISE does not implement interference-based packet re-routing, which leads to the dropping of a lot of packets.

In this section, we saw the performance results of both μSDN and SDN-WISE architecture. We observe that μSDN shows better performance in terms of Transmission Energy consumption, Controller-Node RTT and PDR. The μSDN shows better performance because of its various optimizations like Protocol, Architecture, and Memory. These optimizations help the nodes to consume less energy while transmission. Also, features like source routing and flow entry retention help in improving overall performance . On the other hand, SDN-WISE does not implement any of these optimizations and uses stateful nodes, which leads to packet re-transmission by nodes and more consumption of energy. Thus, based on above evaluation we chose to use μSDN as our base architecture and integrate it with an external ONOS controller.

3.6.2 Controller Evaluation

In this section, we will show evaluation results of μSDN with embedded and ONOS controller as proposed in section 3.4. We compare the performance of both the controllers using the metrics discussed in section 3.5.2.

Effect of Flow Setup Requests on Throughput: The Fig 3.16 shows the controller throughput from low to high flow requests rate for grid and random topologies. We can see from Fig. 3.16 that the Throughput for embedded and ONOS controller

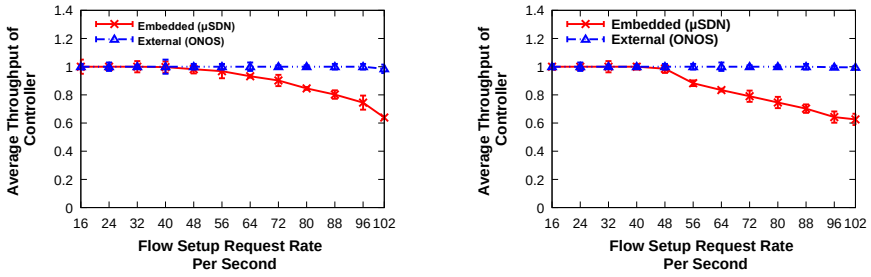


Figure 3.16: Average Throughput of controller in a)Grid and b)Random Topology.

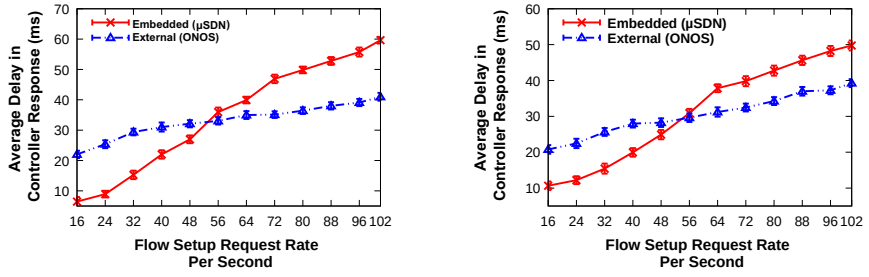
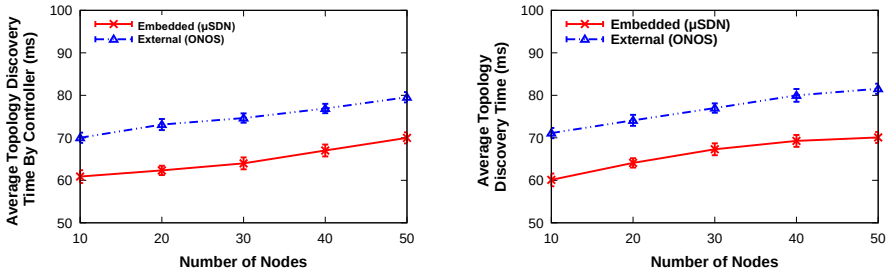


Figure 3.17: Average Controller response time in a)Grid and b)Random Topology.

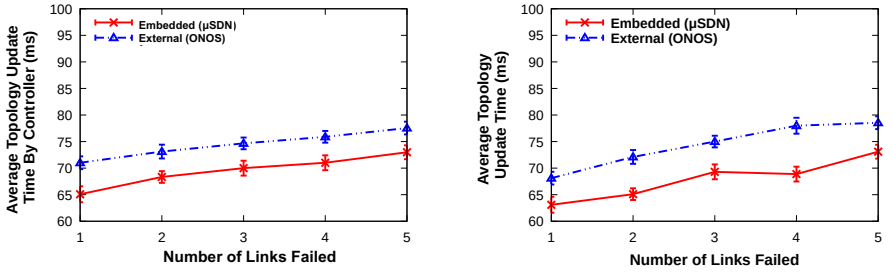
is the same for the low flow request rate. As the flow rate increases the Throughput of embedded controller starts decreasing whereas for ONOS it remains same for even high flow rates. The main reason behind this trend could be because embedded controller is a low-powered device with low memory and less processing capabilities. Thus, its performance starts degrading as the flow rate increases. On the other hand, ONOS is running on a server with better resources and processing capabilities, which allows it to handle a large number of flow requests. Even though we see a decrease in embedded controller's Throughput but point to be underlined is that its performance is similar to that of ONOS at a low flow rate, which means their performance is similar for low bit rate.

Effect of Flow Request Rate on Controller's Response time: Fig. 3.17 shows that the delay in response for embedded controller is lower than the ONOS controller at low flow request rates. The main reason behind this is that ONOS requires a lot of coordination between the subsystems like Sensor node, Flow rule, etc. This takes time for processing a flow request. On the other hand, embedded controller is a lightweight controller within the IoT network. It uses small number



(a) (b)

Figure 3.18: Average Topology Discovery time in a)Grid and b)Random Topology.

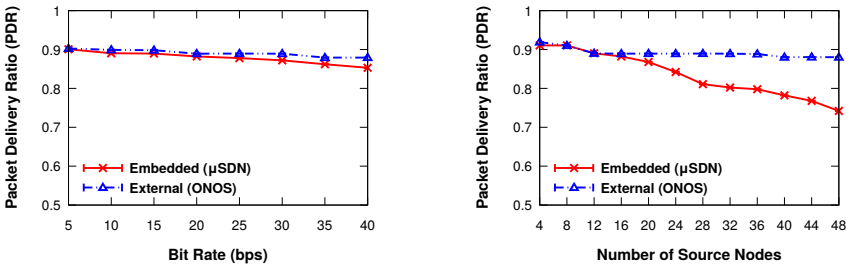


(a) (b)

Figure 3.19: Average Topology Update time in a)Grid and b)Random Topologies.

of functions to process flow requests, which take less time as compared to the ONOS controller. But as the flow request rate increases the embedded controller fails to respond to such a high number of requests and goes under saturation because of its low memory and processing capabilities. The results show that embedded controller's response time is better than ONOS for a low flow rate.

Effect of varying nodes on Topology Discovery Time: The Fig. 3.18 shows the Topology Discovery time for embedded and ONOS controller for a varying number of nodes. From Fig 3.18 it is clear that embedded controller requires less time to detect all nodes present in topology. Whereas, ONOS controller has constant time overhead as compared to the embedded controller while discovering all nodes. The main reason behind this is because ONOS being an external controller, the RPL DAO message needs to travel from the border router onto the backbone network to reach the ONOS controller. This process leads to additional time for packets to reach ONOS and that is the reason why we observe constant time overhead for the ONOS controller. The results of Topology discovery time show that embedded controller being in IoT network can detect all nodes in the topology in a quick time.



(a) (b)

Figure 3.20: Real sensor Data Evaluation for PDR in grid Topology varying a)Bit Rate and b)Source Nodes

Effect of Number of Links Failed on Topology Update Time: We can see from Fig. 3.19 that Topology Update time for embedded controller is less as compared to the ONOS controller. The embedded controller receives update quickly than ONOS as the NSU packet do not have to travel via the backbone network to reach the ONOS controller. This takes place in a non-real time system and leads to additional time overhead. Another reason for this performance trend is that when the NSU packet reaches the Packet subsystem of ONOS it requires coordination with the Sensor node subsystem to register the topology changes, this coordination leads to additional time consumption. Thus, the overall topology update time is increased. Thus, the average topology update time for the embedded controller is better as compared to the ONOS controller.

3.6.3 Real Sensor Data Evaluation

In this section, we have shown evaluation of the embedded controller for real sensor data. This data consists of real time data collected from temperature sensors placed in different floors of an apartment. We obtained the data set from Washington State University (WSU) CASAS databases [1]. We have shown evaluation of the Packet Delivery Ratio by varying the bit rate and number of source nodes. We use a grid topology of 50 nodes. We kept the hop distance as 7 in both cases.

Effect of Bit Rate on Packet Delivery Ratio: The Fig. 3.20a shows us results of the PDR of embedded and ONOS controller using real sensor data. In this evaluation, we varied the bit rate and we see that as the bit rate increases, the PDR of embedded controller starts decreasing but does not dip drastically. The PDR of

embedded controller is similar to that of ONOS controller at low bit rate for the real sensor data.

Effect of Source Nodes on Packet Delivery Ratio: The Fig. 3.20b shows us results of the PDR of embedded and ONOS controller using real sensor data by varying the source nodes. In this evaluation, we observe that as the number of source node increases, the PDR of embedded controller starts decreasing. The reason behind this performance is because as the number of packets arriving at embedded controller starts increasing, the embedded controller cannot handle such large amount of packets due to its low memory and processing capabilities. On the other hand, the ONOS can handle large amount of data packets as it has better resources. One more reason is that ONOS is directly connected to sink node through a tunnel which almost acts as a wired medium which prevents any packet loss. Whereas, embedded controller is in radio medium which can lead to dropping of few packets.

In this section, we discuss the evaluation results of embedded and the ONOS controller based on various performance metrics. We can clearly see that embedded performs better in terms of Delay in Controller response time, Topology Discovery time, and Topology Update time. The Throughput performance of both the controllers is the same for the low flow request rate. The main reason behind this performance is that the embedded controller being embedded in the IoT environment can respond to messages in a quicker time and thus, can detect any topological changes. Also, unlike ONOS it does not require any subsystem co-ordination, which reduces its response time. Many of the applications like Temperature monitoring operate at a low bit rate. Through our results, we show that embedded controller performs better or almost similar to ONOS at a low bit rate in terms of Controller response time and Throughput. We can prefer to use an embedded controller, which has similar hardware to that of other nodes in a low-power IoT network.

Chapter 4

Conclusion and Future Works

4.1 Conclusions

In this work, we proposed to use an embedded controller for low-power IoT networks. First, we considered two prominent SDN-IoT architectures, i.e., μSDN and SDN-WISE. We compared their performance using different performance metrics. To make our comparison unbiased, we introduced the embedded control logic in SDN-WISE node architecture. Then, we modify the existing μSDN architecture to facilitate connection with an external ONOS controller. Since the message format of both μSDN and ONOS is not the same, we developed an adapter application consisting of different components spread across various subsystems. We then compare the performance of embedded controller and external ONOS controller. We also showed the crossover points where the embedded controller goes into saturation.

To evaluate our first part of the work, we compared the performance of μSDN and SDN-WISE approaches by deploying and simulating the architectures using the Cooja simulator, which is based on the Contiki OS. We have found that μSDN solution performs better than SDN-WISE in terms of transmission energy consumption and Controller-Node RTT. The μSDN also has provided better performance in terms of PDR than that of SDN-WISE. We have evaluated the two controllers, i.e., embedded and ONOS. We found that their performance is similar in terms of throughput for a low flow rate. The embedded controller shows better performance in latency in controller response time for a low flow rate. The embedded controller also has better topology discovery and update time. We observe through our evaluation that embedded controller can be preferred over external controller for low bit-rate applications.

In this work, we also evaluated the embedded controller and ONOS controller for real sensor data. We observe that the embedded controller shows similar PDR to that of ONOS. Thus, real sensor data results show that embedded controller has similar performance to that of ONOS at low bit rate.

4.2 Future Works

In this work, we kept the RDC and MAC protocol stack the same for both μSDN and SDN-WISE architecture to ensure unbiased comparison. It would really be interesting to study the effect of different RDC and MAC protocols on the energy consumption of nodes. In addition to that, we would intend to further study the performance of embedded controller to external factors like radio interference, node failure, etc. The main limitation of our work is when certain applications require a high bit rate with high performance, the embedded controller will not provide similar performance for high flow rates. For high flow rates, we have to switch to an external controller like ONOS, which can handle a large number of flows. Thus, for future work, we propose a mechanism that can switch between embedded and ONOS controller depending on flow rates. This will make the network dynamic and would enable the application user to switch between internal and external controllers depending on the application data bit rate. We would also like to extend our work in order to support large number of source nodes like humidity, temperature, and pressure sensors present in applications like smart farm. We propose a hierarchical architecture where we use separate embedded controllers configured to support each subnet of source nodes. This topological information can be synced with one more intermediary node (sink node). We think that by designing hierarchical architecture we can support both performance and scalability using embedded controller.

Appendix A

A.1 Implementation of Node Architecture Comparison

A.1.1 Network Setup Configuration

Setup for μSDN: To evaluate different performance metrics we need to vary the bit rate and hop distance. These parameters are stored as variables in source node program. We can initialize these parameters before every simulation run. The node variables are as shown in Listing A.1. The detailed program is available in *usdn/examples/sdn/node*.

Listing A.1: Node Configuration

```
/* Set Bit Rate value before every run */
#ifndef CONF_APP_BR_MAX
  #define CONF_APP_BR_MAX    32
/* Set the source nodes */
static int app_tx_node[CONF_NUM_APPS] = {1,5,7,16,20,27};
/*Set the destination node */
static int app_rx_node[CONF_NUM_APPS] = {4,2,6,9,12,22};
```

The hop distance can be varied from μSDN configuration file as shown in Listing A.2. The more details about this file can be found in *usdn/core/net/sdn*.

Listing A.2: μSDN Configuration

```
typedef struct sdn_configuration {
/* virtual network id */
uint8_t  sdn_net;
/* hops from destination */
uint8_t  hops;
  }usdn_conf_t;
```

Setup for SDN-WISE: Similarly, to evaluate different performance metrics for SDN-WISE architecture we need to change its configuration file before every run. We can define the bit rate directly by varying the values in file. For hop distance we vary the value from main program. The structure is as shown in Listing A.3. The more details can be found in *sdnwise/node*.

Listing A.3: SDN-WISE Configuration

```
/* Set the Bit Rate Value */
#define RF_SEND_DATA_RATE     32
#define RF_RECEIVE_DATA_RATE     32
typedef struct node_conf_struct {
/*Network ID */
uint8_t my_net;
/* Hops from the Sink*/
uint8_t hops_from_sink;
 } node_conf_t;
```

A.1.2 SDN-WISE Sink Node Changes

We have seen in Chapter 3 section 3.3 the modifications in SDN-WISE node architecture. We added the Topology Manager, Response, and OpenPath modules in SDN-WISE sink node to ensure that we have control logic within the IoT environment. We also changed the MAC driver to ensure that our comparison is unbiased. The nodes initially send a beacon packet which tells the controller about nodes presence. This information is stored in Topology Manager. The implementation is as shown in Listing A.4.

Listing A.4: Topology Manager SDN-WISE

```
/* Using scan network method to save topology*/
void scan_network(uint8_t nodes[], node_t *head)
{
/* The nodes array contains node id and network size */
 int node_id = nodes[3];
 int network_size = nodes[9];
```

```
 /* Store node ID in linked List */
 node_t *new_node;
 new_node = (node_t *) malloc(sizeof(node_t));
 new_node->data = node_id;
 new_node->next= head;
 head = new_node;
 printf("Node ID %d added to the Topology", node_id);
}
```

When a node sends a REQUEST message, the controller should send the flow rules back to node. This is managed by sending a RESPONSE packet. The implementation is as shown in Listing A.5.

Listing A.5: Response in SDN-WISE

```
/* Switch case is applied on packet header type */
Case REQUEST:
  handle_response(p);
void handle_response(packet_t* p)
{
if (is_my_address(&(p->header.dst)))
{
entry_t* e = get_entry_from_array(p->payload,p->header.len);
if (e != NULL)
{
add_entry(e); /*Flow entry is added in flow table*/
}
packet_deallocate(p);
}
else {
match_packet(p);
}
}
```

In some cases, the controller may want update itself with current network status

instead of waiting for regular update from nodes. Therefore, the controller sends OpenPath message to get update from nodes. The detailed implementation of these functionalities can be found in *sdnwise/node*.

A.1.3 Parameter Evaluation

In this work, we have evaluated Transmission energy consumption for both μSDN and SDN-WISE architecture. To evaluate the energy consumption we have used ENERGEST library which is available in the Contiki OS. We display the energy values in our logs at regular intervals. Some of the details of implementation are as shown in Listing A.6. The detailed code is available in *usdn/core/net/sdn*.

Listing A.6: Energy Evaluation in μSDN

```
/*Update all energest times. Should always be called
before energest times are read.*/
 energest_flush();
 all_transmit = energest_type_time(ENERGEST_TYPE_TRANSMIT)
 / (RTIMER_SECOND / 1000);
 double current = (all_transmit * current-tx);
 /* Consider voltage as 3volts */
 double energy = current * 3;
 LOG_STAT_("en: ");
 print_division(energy, time_on);
```

Similarly, energy consumption of SDN-WISE nodes are also calculated. The detailed code is available in *sdnwise/node/stats*.

A.2 Operation of μSDN-ONOS

The μSDN operation starts with initialization of all nodes present in the topology. The sink node acts as border router and serves as the DAG root node for rest of nodes. The sink node receives a prefix of ONOS application through the SLIP connection which allows further communication with other nodes. During the network initialization, all nodes send DAO message back to sink node. The sink node sends these message containing node information to ONOS SensorNode subsystem. The ONOS

stores this information in its format to maintain consistent view of the network. The implementation is shown in Listing A.7, A.8, and A.9. The detailed implementation of sink node with slip connection is available in *usdn/examples/controller/sdn-sink.c* . The SensorNode details are available in *onos/protocols/usdn/protocol/USDNMessage.java*

Listing A.7: RPL Configuration

```
struct rpl_instance {
  /* DAG configuration */
  rpl_metric_container_t mc;
  rpl_of_t *of;
  rpl_dag_t *current_dag;
  /*DAO message configuration */
  uint8_t my_dao_nodeno;
  uint8_t my_dao_distance;
  uint8_t mydao_tag;
} rpl_instance_t;
extern rpl_instance_t rpl;
```

Listing A.8: Implementation in Sink node

```
/* Wait until prefix is received from SLIP */
  while(!prefix_set) {
    etimer_set(&et, CLOCK_SECOND);
    request_prefix();
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
  }
#if RPL_WITH_NON_STORING
  ADD("Links<pre>\n");
  uint16_t buf [] = {rpl.my_dao_nodeno,rpl.my_dao_distance,
  rpl.mydao_tag = 1};
  /* send data via slip connection */
  SEND_STRING(&s->sout, buf);
```

Listing A.9: Implementation in SensorNode Subsystem

```
//static method in Sensor node which extracts
//message from packet
USDNMessage getMessageFromPacket
(USDNnetworkPacket networkPacket) {
USDNMessage usdnMessage = null;
int sensorMessageType = networkPacket.getTyp();
//checks the packet type
switch(sensorMessageType)
{
  case RPL:
    RPLPacket rplpacket = new RPLPacket(networkPacket);
    //message is stored as object.
    usdnMessage = new USDNRPLMessage(rplpacket.getDistance(),
    rplpacket.getNodeID());
    break;
}
```

When a node misses a flow entry, μSDN node sends a FTQ message to ONOS via the sink node. The μSDN protocol component extracts the information and forwards it higher layers of FlowRule subsystem. The FlowRule subsystem creates a FTS message based on current network state and sends the FTS message to SensorNode subsystem. The FTS message implementation is shown in Listing A.10.

Listing A.10: FTS in ONOS

```
static FlowTableSetPacket getResponse()
{
// Based on network state FlowRule provider sets source
// and destination address.
 responsePacket.setTyp(USDNnetworkPacket.FLOWTABLE_SET);
responsePacket.setNet((byte) super.getId());
responsePacket.setDst(new NodeAddress
(super.getDestination().address()));
```

```
responsePacket.setSrc(new NodeAddress
(super.getSource().address()));

FlowtableEntry flowTableEntry = new FlowtableEntry();

flowTableWindows.forEach(flowTableEntry::addWindow);
flowTableActions.forEach(flowTableEntry::addAction);

responsePacket.setRule(flowTableEntry);

return responsePacket;
}
```

The ONOS receives NSU packets from nodes regarding its distance, battery levels, etc. These packets are sent at regular updates. The information must be stored in ONOS format to update the network state. The implementation of NSU packets is as shown in Listing A.11.

Listing A.11: NSU implementation in ONOS

```
public USDNNodeStatusMessage(int distance,
int batteryLevel, int nofNeighbors,
Map<USDNnodeId, Integer> neighborRSSI) {
super();
//The NSU data is stored in ONOS format.
this.distance = distance;
this.batteryLevel = batteryLevel;
this.nofNeighbors = nofNeighbors;
this.neighborRSSI = neighborRSSI;
```

The FTS messages and CONF message must be sent back to nodes. This functionality is handled by Sensornode forwarding application. This application converts these message packets in μSDN format and sends it over backbone network channel. The method is as shown in Listing A.12.

Listing A.12: Send Function

```
private void sendNetworkPacket
(USDNnetworkPacket networkPacket)
{
//Send message in form of array over channel.
ChannelBuffer channelBuffer =
ChannelBuffers.wrappedBuffer(networkPacket.toByteArray());
}
```

In summary, we have tried to provide as much as details of our implementation. Since, it is difficult to cover every aspect of code, we have provided details about our code in git [22]. The research enthusiasts can clone the code and try to reproduce the results.

Bibliography

[1] Washington state university casas datasets. http://casas.wsu.edu/datasets/, September 2015. (Last accessed 17-November-2020).

[2] Hazrat Ali. A performance evaluation of rpl in contiki. volume 1, pages 17–24. SICS, October 2012.

[3] Angelos-Christos G Anadiotis, Laura Galluccio, Sebastiano Milardo, Giacomo Morabito, and Sergio Palazzo. SD-WISE: a software-defined wireless sensor network. *arXiv preprint arXiv:1710.09147*, 2017.

[4] Angelos-Christos G. Anadiotis, Laura Galluccio, Sebastiano Milardoy, Giacomo Morabito, and Sergio Palazzo. Towards a software-defined network operating system for the iot. In *INFOCOM proceedings*, pages 1–6. IEEE, May 2015.

[5] You Wang Ayaka Koshibe. Systems components of onos architecture. https://wiki.onosproject.org/display/ONOS/System+Components, September 2016. (Last accessed 11-October-2020).

[6] Michael Baddeley, Reza Nejabati, George Oikonomou, Mahesh Sooriyabandara, and Dimitra Simeonidou. Evolving SDN for low-power IoT networks. In *IEEE NetSoft*, pages 71–79, 2018.

[7] Larry Burgess. Inexpensive low data rate links for the internet of things. https://www.volersystems.com/wearable-devices/inexpensive-low-data-rate-links-for-the-internet-of-things, November 2018. (Last accessed 22-October-2020).

[8] Matt Burgess. What is the internet of things? wired explains. https://www.wired.co.uk/article/internet-of-things-what-is-explained-iot, Feb 2018. (Last accessed 1-November-2020).

[9] M. Darianian, C. Williamson, and I. Haque. Experimental evaluation of two openflow controllers. In *in the proceeding of IEEE ICNP workshop on PVE-SDN*, Oct 2017.

[10] C. Lakshmi Devasena. Ipv6 low power wireless personal area network (6lowpan) for networking internet of things (iot) – analyzing its suitability for iot. volume 9. IBS University, Hyderabad, India, 2016.

[11] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *IEEE international conference on local computer networks*, pages 455–462, 2004.

[12] T Fevens, I Haque, and L Narayanan. A class of randomized routing algorithms in mobile ad hoc networks. *AlgorithmS for Wireless and mobile Networks (A SWAN 2004), Boston*, 2004.

[13] Laura Galluccio, Sebastiano Milardo, Giacomo Morabito, and Sergio Palazzo. SDN-WISE: Design, prototyping and experimentation of a stateful SDN solution for WIreless SEnsor networks. In *IEEE INFOCOM*, pages 513–521, 2015.

[14] I. Haque. On the overheads of ad hoc routing schemes. *IEEE Systems Journals*, 9(2):605–614, June 2015.

[15] I. Haque and C. Assi. Localized energy efficient routing in mobile ad hoc networks. *The Willey Journal of Wireless and Mobile Computing*, 7(6):781–793, August 2007.

[16] I. Haque, C. Assi, and W. Atwood. Randomized energy-aware routing algorithms in mobile ad hoc networks. In *Proceedings of the 8th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, MSWiM '05, 2005.

[17] I. Haque, M. Nurujjaman, J. Harms, and N. Abu-ghazaleh. SDSense: An agile and flexible SDN-based framework for wireless sensor networks. *The IEEE Transactions on Vehicular Technology*, 68(2):1866 – 1876, February 2019.

[18] I. T. Haque and N. Abu-Ghazaleh. Wireless software defined networking: A survey and taxonomy. *IEEE Communications Surveys Tutorials*, 18(4):2713–2737, 2016.

[19] I. T. Haque and Chadi Assi. OLEAR: Optimal localized energy aware routing in mobile ad hoc networks. In *Proceedings of the 2005 IEEE International Conference on Communications*, ICC '05, 2005.

[20] Israat Haque, Saiful Islam, and Janelle Harms. On selecting a reliable topology in wireless sensor networks. In *Proceedings of the 2015 IEEE International Conference on Communications*, ICC '15, 2015.

[21] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. volume 103, pages 14–76. IEEE, 2014.

[22] Miheer Kulkarni. Controller-compare-thesis. `https://github.com/MiheerKulkarni/Controller-Compare-Thesis`, 2020. (Last accessed 11-December-2020).

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

[24] Mansoor Alam Junghwan Kim Robert Green Nasseri Mona, Hussien Al-Omat and Wei Cheng. Contiki cooja simulation for time bounded localization in wireless sensor network. In *Proceedings of the 18th Symposium on Communications Networking*, page 250. ACM, 2015.

[25] Keyur K Patel, Sunil M Patel, et al. Internet of things-iot: definition, characteristics, architecture, enabling technologies, application & future challenges. *International journal of engineering science and computing*, 6(5), 2016.

[26] Pedro. Mac protocols in contikios. *Autonomous Network Research Group, University of South California*, 1(1):1–4, 2014.

[27] Steve Ranger. What is the iot? everything you need to know about the internet of things right now. `https://www.zdnet.com/article/what-is-the-internet-of-things`, February 2020. (Last accessed 6-October-2020).

[28] J. Romkey. A non-standard for transmission of ip datagrams over serial lines: Slip. pages 1–6. IETF, 1988.

[29] Dipon Saha, Meysam Shojaee, Michael Baddeley, and Israat Haque. An energy-aware sdn/nfv architecture for the internet of things. In *IFIP/IEEE Networking*, pages 1–3. IFIP, June 2020.

[30] S.V. Saliga. An introduction to ieee 802.11 wireless lans. IEEE, 2000.

[31] H. P. Tan T. Luo and T. Q. S. Quek. Sensor openflow: Enabling software-defined wireless sensor networks. In *IEEE Communications Letters*, volume 16, pages 1–5. IEEE, November 2012.

[32] Kulesa Tamara. 8 sensors to help you create a smart home. `ibm.com/blogs/internet-of-things/sensors-smart-home/`, 2016. (Last accessed 11-December-2020).

[33] Tryfon Theodorou and Lefteris Mamatas. Coral-sdn: A software-defined networking solution for the internet of things. In *Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–2. IEEE, May 2017.

[34] Andrea Campanella Brian O'Connor David Bainbridge Ray Milkey Thomas Vachuska, Jordan Halterman and Carmelo Cascone. Open networking foundation: Onos. `https://www.opennetworking.org/onos/`. (Last accessed 12-October-2020).

[35] Tim Winter, Pascal Thubert, Anders Brandt, Jonathan W Hui, Richard Kelsey, Philip Levis, Kris Pister, Rene Struik, Jean-Philippe Vasseur, Roger K Alexander, et al. Rpl: Ipv6 routing protocol for low-power and lossy networks. IETF, March 2012.