UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Study and implementation of a distributed control system for software defined wireless networks

Diploma Thesis

Ippokratis-Vasileios Koukoulis

Supervisor: Athanasios Korakis

Volos 2022

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Study and implementation of a distributed control system for software defined wireless networks

Diploma Thesis

Ippokratis-Vasileios Koukoulis

Supervisor: Athanasios Korakis

Volos 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη και υλοποίηση κατανεμημένου συστήματος ελέγχου ασύρματου δικτύου καθορισμένου από λογισμικό

Διπλωματική Εργασία

Ιπποκράτης Βασίλειος Κουκούλης

Επιβλέπων: Αθανάσιος Κοράκης

Βόλος 2022

Approved by the Examination Committee:

Supervisor **Athanasios Korakis**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Christos Antonopoulos**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Antonios Argyriou**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

First and foremost, I would like to thank Prof. Thanasis Korakis for trusting me and giving me the chance to work on this project. I would also like to extend my deepest gratitude to Kostas Choumas and Ilias Syrigos for their collaboration and continuous support throughout this thesis. They were always available to discuss any issues or questions I had regarding this project and their input was invaluable for the completion of this thesis and furthering my knowledge around the subjects presented in this thesis.

Finally I would like to thank my family for their unconditional love and unwavering support they have provided me throughout these years believing in me. Without them I would have never been able to reach this milestone.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Ippokratis-Vasileios Koukoulis

Abstract

Mobile Ad Hoc Networks and Wireless Mesh Networks are used in a variety of applications where network infrastructure cannot be built in time to meet certain demands, or environmental conditions are extreme. Current routing solutions for these networks cannot make globally optimal decisions due to their decentralized nature. Software defined networking can assist routing in wireless networks by offering centralized control of the network , and making globally optimal routing policies easy to implement on top of a MANET. In this thesis an SDN architecture for wireless networks is proposed and a distributed SDN controller for this architecture is developed. This SDN architecture provides fault tolerance and availability in the face of network partitions by having multiple controllers replicating their state, and assigning control of each network element to a new controller when the old one is partitioned or fails. This architecture is also self-organizing allowing controllers in the network to discover each other and form a cluster. A simple routing policy based around wireless medium contention is implemented to assess the functionality of our controller and the performance benefits of this SDN architecture.

Περίληψη

Τα αδόμητα δίκτυα ασύρματων κινητών συσκευών (MANET) έχουν πληθώρα εφαρμογών σε τοποθεσίες όπου επικρατούν δυσχερείς συνθήκες ή όπου δεν μπορούν να χτιστούν δικτυακές υποδομές σε μικρό χρονικό διάστημα. Οι υπάρχουσες λύσεις για δρομολόγηση κυκλοφορίας (routing) σε αυτά τα δικτύα δεν μπορούν να πάρουν βέλτιστες αποφάσεις λόγω της αποκεντρωμένης φύσης τους. Η αρχιτεκτονική της δικτύωσης καθορισμένης απο λογισμικό (SDN) προσφέρει την δυνατότητα για κεντρικοποιημένο έλεγχο του δικτύου, διευκολύνοντας την υλοποίηση βέλτιστων πολιτικών δρομολόγησης σε MANETs. Σε αυτή την εργασία παρουσιάζεται μια SDN αρχιτεκτονική για ασύρματα δίκτυα και υλοποιείται ένας κατανεμημένος SDN ελεγχτής (controller). Η προτεινόμενη SDN αρχιτεκτονική προσφέρει ανοχή σε σφάλματα και υψηλή διαθεσιμότητα σε περίπτωση διαχωρισμού ενός controller από το υπόλοιπο δίκτυο, χρησιμοποιώντας πολλούς controllers οι οποίοι αντιγράφουν την κάτασταση τους, έτσι ώστε σε περίπτωση που χωριστεί κάποιος controller από το δίκτυο , να ανατεθεί ο έλεγχος του δικτύου σε κάποιον από τους υπόλοιπους διαθέσιμους controllers. Επιπλέον σε αυτή την αρχιτεκτονική ο κάθε controller ανακαλύπτει τους υπόλοιπους μέσα στο δίκτυο και σχηματίζουν μια ομάδα (cluster). Για την αξιολόγηση της λειτουργικότητας αλλά και για να δείξουμε τα οφέλη αυτής της υλοποίησης, ο controller εκτελεί μια απλή πολιτική routing που στοχεύει στην μείωση του ανταγωνισμού για μετάδοση μεταξύ ασύρματων κόμβων.

Table of contents

List of figures

List of Algorithms

Abbreviations

MANET	Mobile Ad Hoc Network
SDN	Software Defined Networking
OLSR	Optimized Link State Routing Protocol
ETX	Expected Transmission Count
BATMAN	Better Approach to Mobile Ad-hoc Networking
RPC	Remote Procedure Call
SSID	Service Set Identifier
GRE	Generic Routing Encapsulation
ARP	Address Resolution Protocol
STP	Spanning Tree Protocol
BFD	Bidirectional Forwarding Detection
LLDP	Link Layer Discovery Protocol
SNR	Signal to Noise Ratio
CCA	Clear Channel Assessment
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
MAC	Medium Access Control
QoS	Quality of Service

Chapter 1

Introduction

1.1 Context

Mobile Ad Hoc Networks and Wireless Mesh Networks have existed for decades, as an infrastructureless and agile architecture to build a network consisting of mobile nodes with wireless capabilities. MANETs are suitable for a variety of applications where normal network infrastructure cannot be built in time, or cannot withstand extreme environmental conditions, like army tactical operations, disaster rescue scenarios, data collection from sensors and more. However the routing protocols running on these networks, may not always achieve critical performance targets due to their decentralized nature and lack of global optimization. Software Defined Networking can be the necessary orchestration tool for MANETs allowing us to implement several globally optimized routing policies, improving the performance of MANETs in various key performance metrics [3] [4].

1.2 Contribution

This thesis introduces a self-organizing distributed SDN architecture which can be deployed in MANETs. Specifically in this thesis we develop an SDN controller optimized for wireless networking conditions, providing fault tolerance using multiple controllers that can be elected as masters to replace failing or partitioned controllers. The controller works as part of a cluster with multiple controllers in the network which share their state and control network devices as a logically centralized controller using the Etcd distributed key value store to reach consensus, providing routing applications with a consistent view of the network and

the ability to operate the network under a centralized point of view. In this approach the controllers do not use any existing routing protocols to connect to each other, instead they they dynamically discover each other by exploiting the connectivity and topology discovery mechanisms already implemented in the OpenFlow protocol. The controller follows a pure in-band approach where the control plane and the data plane use the same physical and virtual network interfaces to deliver messages.

An additional contribution of this thesis is the extension of the Wmediumd simulation used by Mininet-Wifi to support fine grained scheduling of packets according to the signal received by each node. The modifications to wmediumd lead to more realistic throughput and delay results when multiple nodes in a mesh network transmit simultaneously.

1.3 Thesis structure

The rest of the thesis is structured as follows:

- **Chapter 2** introduces the background surrounding this thesis by giving a brief explanation of popular existing routing MANET protocols, the SDN architecture paradigm, and the inner workings of the consensus protocol used by the distributed database that we use as base to build our distributed controller. Furthermore the related work on the integration of SDN on MANETs and the main differences between our implementation is discussed here to make even clearer what is the contribution of this thesis.

- **Chapter 3** presents details about the architecture and implementation of the SDN controller and additionally provides information on the optimizations and modifications that were made to Ryu and Etcd to make the operation of this controller feasible in wireless networks. A simple routing policy is also implemented to operate the SDN controller.

- **Chapter 4** presents the simulation environment used for the experiments and discusses the extension to the contention algorithm.

- **Chapter 5** presents the experimental results for the implemented routing policy using our distributed SDN controller.

- **Chapter 6** provides a conclusion to this thesis and discusses future directions.

Chapter 2

Background

2.1 Routing in MANETs

Various routing protocols have been devised for MANETs each one taking a different to approach on how to distribute network information, do route selection and link quality metrics. Here we present some of these protocols that were used in experiments in the context of this thesis.

2.1.1 OLSR

Optimized Link State Routing Protocol (OLSR) [5] is a proactive link state routing protocol. Link state protocols is a class of protocols where each node shares its view of the network (the network graph including all links between nodes) with its neighbors and each node constructs a global view of the network. Each node uses this information to create a routing table towards all nodes in the network. OLSR specifically uses Topology Control messages to propagate topology information to its neighbors, and Hello messages to make its existence known to its neighbors. Although the original standard for OLSR used a shortest hop path metric to decide routes, the olsrd daemon uses by default the Expected Transmission Count (ETX) [6] metric. The disadvantage to this routing approach is that every node in the network has to exchange a lot of information about the topology, and each node also has to calculate a route using the global network topology which can become computationally expensive in large topologies.

2.1.2 BATMAN

Better Approach to Mobile Ad-hoc Networking (BATMAN) [7] is a protocol which was made as an improvement over OLSR, following an approach where the knowledge about the best end-to-end paths between nodes is divided among all nodes in the network. Each node perceives and maintains only the information about the best next hop towards all other nodes, following an approach similar to distance vector protocols. Specifically in BATMAN each node uses broadcast messages called Originator messages (OGMs) to inform its neighbors for its existence. Then the neighboring nodes use special rules to re-broadcast OGMs to other nodes and this process goes on until all nodes receive the OGM. The quality of links is assessed through OGMs. To decide which neighbor is the most suitable to route packets through a destination, nodes choose neighbors from which they received OGMs of this destination faster and in more quantity.

BATMAN has 2 additional iterations apart from the default one BATMAN III(batmand). BATMAN IV (batman-adv) differentiates from batmand in 2 main ways. Firstly, BATMAN IV operates in Layer 2 of the OSI stack with the use of a Linux kernel module essentially operating as a virtual switch where each node is like a switch port, in contrast to BATMAN III which operates on Layer 3. All traffic is encapsulated with a new header which is used to forward this packet to other nodes, instead of overwriting the destination MAC for the next hop. Secondly, BATMAN IV uses an improved link quality algorithm which considers the impact of asymmetric links. Finally BATMAN V differentiates from its predecessors by employing a throughput based link quality metric which determines the throughput of links through either WiFi driver information or by running periodic throughput tests between nodes.

2.1.3 Babel

Babel[8] is a loop avoiding distance vector routing protocol. It uses the distributed Bellman-Ford algorithm along with some additional conditions to prevent loop formation like the counting to infinity problem, or resolve routing loops that may happen in a timely manner. Each node periodically sends its routing table information to its neighbors, and each node decides the best route based on the distance/routing metric of each path. Each node can also reject the update based on a feasibility condition which assesses if the routing update (based on the routing metric) will lead to a routing loop. Babel by default also uses a variant of the

ETX[6] link quality metric.

2.2 Software Defined Networking

Software Defined Networking (SDN) is a network architecture paradigm that enables the centralized control of network elements. An overview of the SDN architecture paradigm is shown in figure 2.1. The SDN architecture consists of three main components:

- **Datapath**

  Datapath is a network device/element (alternatively called switch or router) that can process and forward traffic according to some specific rules (alternatively called flows). Datapaths expose a southbound interface that allows SDN controllers to set rules for routing/network management on the controllers. The most popular southbound interface is the OpenFlow protocol [9].

- **SDN controller**

  An SDN controller is the centralized entity which controls and sets the rules for packet forwarding and processing to the datapaths.There can be multiple controllers in an SDN architecture, where in this case these controllers communicate through an East/West-Bound interface in order to synchronize their view of the network state and provide a logically centralized view and control of the network. It is important to note here that East/West-Bound interfaces are not usually defined by any open standards, instead many commercial controllers implement their own interfaces to support multiple controller architectures.

- **SDN application**

  SDN application is the component which defines the routing/network management logic that the controller will enforce on the datapaths. Typically an SDN application gathers network status information and issues through the controller flow table updates on the switches to implement routing policies.

In SDN, routing and network management logic is decoupled from the forwarding functions of networking devices into the control plane consisting of the controller and the data plane consisting of packet forwarding devices. SDN offers the ability to programmatically

configure network devices and define custom routing and traffic management policies according to specific requirements in a rapid fashion without the need to know anything about specific proprietary programming interfaces. SDN enables easier development of routing applications since global knowledge of the network state is available at the controller, achieving better performance in target metrics like throughput and delay in certain situations compared to classic distributed routing approaches.

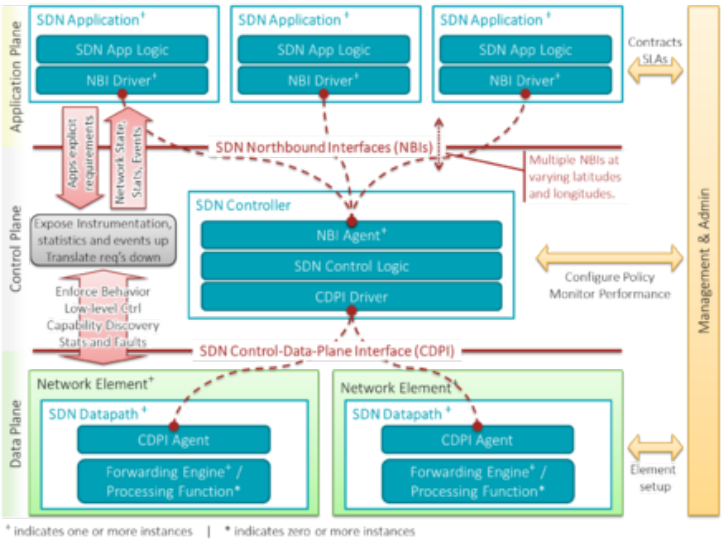

Figure 2.1: SDN architecture [1]

2.3 Etcd and consensus with Raft

To build a distributed controller, we need an East/West Bound interface that the controllers can use to communicate with each other and reach consensus on leadership over switches and other relevant network state information like links and flow tables. For our architecture, communication between controllers is implemented using Etcd. Etcd[10] is a strongly consistent, distributed key-value store that provides a reliable way to store data that needs to be accessed by a distributed system or cluster of nodes.

Etcd uses the Raft[1] algorithm in order to reach consensus on the state of the data store and replicate it to all nodes, even in the face of node failures or network failures between nodes. Raft achieves consensus in the context of replicated state machines as shown in Figure 2.2. Each replicated state machine is implemented as a log, which consists of a series of

[1] https://opennetworking.org/wp-content/uploads/2013/02/SDN-architecture-overview-1.0.pdf

commands that need to be executed in a specific sequence to get the same output from the state machine. Raft is the consensus module which guarantees that all state machines have the same log and will execute the exact same sequence of commands. Raft decomposes the consensus problem into 3 subproblems: leader election, log replication and safety. The next subsections summarize the basic functionality of Raft how it solves these problems.

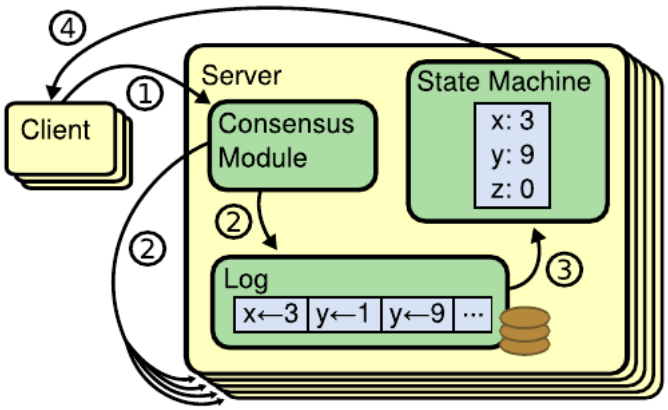

Figure 2.2: Replicated state machine [1]

2.3.1 Raft Basics

Raft works by electing a leader among the server processes in a cluster. For a server to be elected it must gather votes by the majority of the servers in the cluster. A server can be only in one of the 3 following states at any given time: leader, follower or candidate. In the follower state a server only responds to requests from leaders and candidates and in the candidate state a server campaigns for leadership. In Raft time is divided into terms, where each term is numbered with an integer which increases monotonically, serving as a logical clock for the cluster. At the beginning of each term, a new election begins where candidates can become leaders. Figure 2.3 shows the transitions between these states.

2.3.2 Leader election

The leader periodically sends heartbeat messages to all followers to maintain its authority. When a follower does not receive heartbeats (or any communication at all) for a period of time, it assumes that there is no leader and begins a new election becoming a candidate. This period of time is called the election timeout. At the beginning of an election the server increments its term count, votes for itself, and sends RequestVote messages to all other servers.

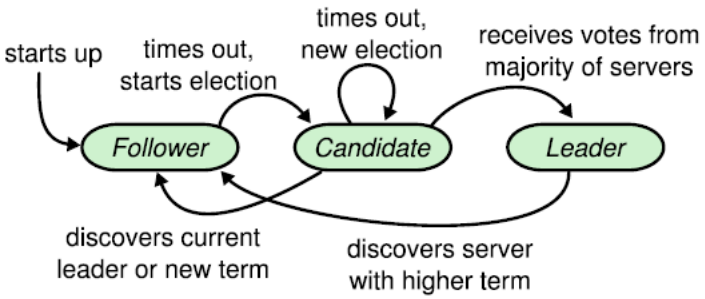

Figure 2.3: Raft server state [1]

The candidate repeats this process until it either wins an election or another server becomes the leader (by sending heartbats), or nobody wins the elections for this term due to split votes. The election timeout is defined by the user, but in order to avoid perpetual split votes Raft uses randomized election timeouts (in the etcd implementation the election timeout is equal to a the user defined election timeout plus a random number from 0 to user defined election timeout).

2.3.3 Log replication

In Raft the leader is responsible for serving requests each containing a command to be executed and replicating log entries to all servers. Each log entry consists of:

- A **command** to be executed by the state machine.
- An **index** number to identify the position of the entry in the log.
- The **term** number.

When the leader receives a request it appends the entry to its log and replicates it in parallel to its followers. This entry becomes committed once it is replicated to the majority of the servers. All preceding entries are also considered committed. After an entry is committed it is applied to the local state machine. The leader attaches the latest committed index to its AppendEntries RPCs to update its followers so they can too apply the entry to their local state machines. Raft maintains coherency between logs by guaranteeing the following properties:

If two entries in different logs have the same index and term then:

- These two entries have the same command
- All preceding entries are identical

To guarantee these properties the index and term of the previous entry in the log are also attached to an AppendEntry RPC by the leader. Each follower can verify an append request and reject it if the previous entry does not match its own. To handle any inconsistencies caused by crashes, the leader also forces follower logs to match its own. When a leader comes to power, subsequent consistency checks are made to find the point which the leader's log agrees with the follower's log (when the consistency check succeeds) and delete any entries that do not match its own from the follower.

2.3.4 Safety

To ensure that a leader will contain all entries committed in previous terms Raft imposes restrictions on which candidate can become leader. Specifically when a candidate requests votes from other servers they will only vote for this candidate if its log is at least as up to date as their own. The index and term of the last log entry is compared, and a server can reject voting for a candidate. Thus only candidates with the most up-to-date log can become leaders. Additionally, an entry is considered committed only if it originates from the current term and has been replicated to the majority.

2.4 Related work

SDN architectures have been implemented and deployed in MANETs and wireless mesh networks to facilitate various needs that arise from the dynamic nature of such networks.

The following works [11] [12] propose and implement a simple SDN architecture consisting of a single controller .The first one explores the applicability of SDN in client mobility scenarios, while the second one implements a simple round robin policy to route flows towards gateways, proposing the use of SDN to deploy custom traffic engineering in wireless mesh networks. The control plane traffic in these works is routed through secondary virtual interfaces separated from the data plane virtual interface using different SSIDs. The controller to switch traffic is handled by OLSR using the kernel routing tables to install forwarding rules in switches. Packet forwarding in data plane is done by overwriting the mac destination of data packets. Alternatively [13] GRE tunnels can also be used to forward packets. This [12] work is further complemented by [14] and [15] where a multiple controller architecture and a master selection scheme is proposed to enhance the fault of tolerance of the SDN architecture

in network partition scenarios, where each switch can poll through a list of controllers and select a master.

Additionally, master selection schemes considering specific metrics like controller capacity assessed by hardware characteristics have been proposed [16] using an in-band control plane and the Babel routing protocol for topology discovery. Master election schemes migrating the state of the master to another controller in case of failure/network partition have also been proposed [17] [18]. Other SDN architectures [19], [20] consider a single controller which communicates directly via a secondary network interface with all switches in the network in an out-of-band manner , using essentially different channels for the control and data plane. Especially in [20] the performance of SDN shortest hop path routing is compared against other MANET routing protocols. Some architectures [21] do not use either MANET routing protocols or out-of-band communication for the control plane, and instead modify the Openflow protocol to automatically configure the connection between the switches and the controller. Completely proprietary solutions that do not use the Openflow protocol have also been suggested [22].

Related work towards the east/west bound communication of multiple controllers in wireless networks has also been done [23] using existing commercial controllers to assess the performance of state synchronization between controllers.

Chapter 3

Architecture

3.1 Controller and cluster architecture

In this SDN architecture each node hosts an application which acts as a switch and as a controller for all switches in the network. This application consists of three components:

- An OpenVswitch[24] virtual switch
- A Ryu [25] controller
- An Etcd[10] instance

As it can be seen in figure 3.1 each node hosts these 3 components where each etcd instance connects to all other etcd instances, and each OVS switch connects to all Ryu controllers in the network. Etcd can be seen as the West/East bound interface where controllers can share information between themselves, since controllers do not have a direct connection to other controllers. Etcd is used to elect a master among all controllers which will have the responsibility of routing data flows for the entire network and collecting statistics. Etcd is also used to share network state information (such as topology state). To communicate with the etcd instance from the Ryu controller the Kragniz [26] python etcd client library was used.

Although in this cluster architecture all controllers have full knowledge of the network topology (since all switches connect to all controllers, and excluding possible host devices connecting to switches), in a hierarchical architecture this may not be true (switches may connect to a specific set of controllers) and thus there has to be a channel for controllers to share information with other controllers, or even assign multiple masters for different parts of the network. Etcd serves this role as a strongly consistent database allowing multiple controllers

to function as one, share an identical view of the network state, and consent for leadership over switches.

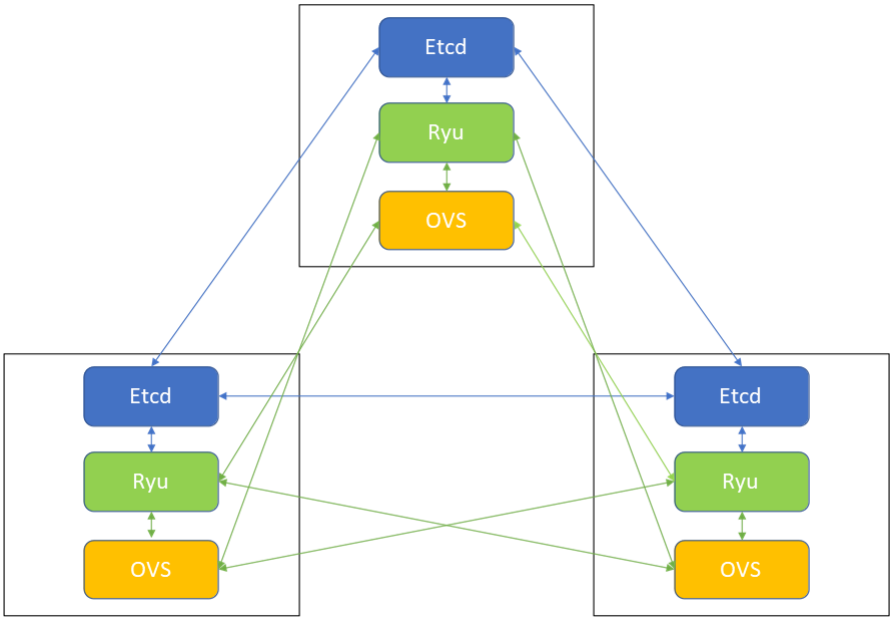

Figure 3.1: Controller-Switch network architecture

3.1.1 Node configuration

The application components are initialized by a configuration script. This script makes the following configurations:

- **Initializing the list of the controllers/etcd peers IPs that the switch/etcd instance will connect to.** This list is generated by the number of nodes in the network.
- **Creating GRE tunnels** necessary for the switch to communicate with its one-hop neighbors. These tunnels route packets from the OpenVswitch bridge (the 192.168.1.x subnet) to the wireless interface which operates on the 10.0.0.x subnet.
- **Setting the switch to standalone mode** so it does not route traffic on its own.
- **Setting arp static entries for the the 192.168.1.x subnet**.
- **Disabling STP** since we want all paths to be available for the controller routing algorithm.

- **Disabling hidden flows**. Hidden flows normally process openflow traffic between the switches and the controller. However these flows output by default to the `OFPP_NORMAL` port which processes packets with normal L2/L3 switching and since STP is disabled and no L3 protocol is running on our nodes, these flows cannot actually process traffic. Openflow traffic is handled by having controllers set flows for control traffic themselves, essentially working like a very simple autonomous routing daemon for openflow traffic. Control traffic consists of Openflow and Etcd traffic using ports 6633 and 2380 respectively.

- **Setting tuning parameters for Etcd**. These parameters include the election timeout and heartbeat interval which the Ryu controller will use to initialize Etcd.

- **Setting up the OpenVswitch process and the Ryu controller**

Figure 3.2 shows the network configuration of a cluster with 3 nodes. Each node configures tunnels to communicate with all other nodes. When a tunnel connection becomes active the node can forward packets through this tunnel directly to a node. Tunnels across non-neighboring (like s1-s3) nodes are inactive. The implementation of tunnel liveness monitoring is discussed in the topology section. Each packet sent by any application (Ryu, Etcd or any other application from user space) to its local switch interface is encapsulated in a GRE header and is sent through the GRE tunnel to the corresponding IP address of the 10.0.0.x subnet using the wireless interface of the node.

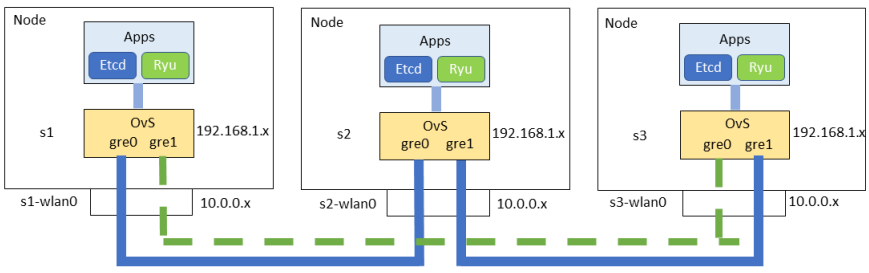

Figure 3.2: Network configuration

3.1.2 Thread/Process architecture

The architecture of the controller consists of 2 processes:

- **The main process of the Ryu controller**
- **The Etcd instance process**, which is forked by the Ryu controller during its initialization.

The Ryu controller process consists of 3 native python threads:

- **The main thread**, which runs the event handling and various functions of the Ryu controller. Ryu makes use of Eventlet [27], a concurrent networking threading library to process events. Eventlet uses the concept of coroutines/green threads in its threading architecture and their main difference between python threads is that each green thread cooperatively yields control to another green thread (via a yield function) instead of being preemptively scheduled by the operating system. Many green threads can exist in a single python/native thread. Figure 3.3 demonstrates how exactly coroutines/green threads co-exist with normal python threads.
- **The producer thread**, a thread responsible for committing data to the etcd.
- **The callback thread**, a thread created by the etcd client library which notifies the controller when a new write/delete is made in etcd by any node in the cluster.

Figure 3.4 shows the internal application architecture of the controller and how various components/threads interact with each other. In the following subsections the utility of each component is described in detail.

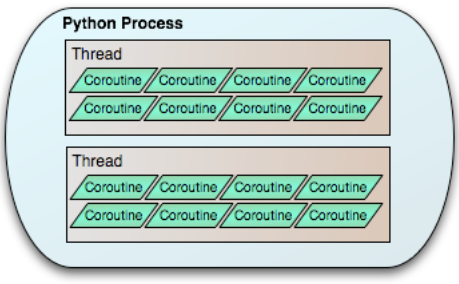

Figure 3.3: Eventlet threading architecture [1]

[1] https://eventlet.net/doc/threading.html

3.1.3 Cluster monitor thread

Additionally to the existing green threads that Ryu uses for event handling, a new green thread was created for Ryu called the cluster monitor thread. Algorithm 1 is the main routine executed by the cluster monitor thread. The cluster monitor thread also performs several functions related to monitoring the status of the Etcd cluster. The main functions of the cluster monitor thread are:

Algorithm 1 Status Monitor algorithm

```
1:  self.master, self.term ← get_cluster_status()
2:  if self.master = self.etcd_url then
3:      role ← MASTER
4:  else if self.master ≠ None then
5:      role ← SLAVE
6:  else
7:      role ← EQUAL
8:  end if
9:  self.master_datapaths ← get_master_datapaths()
10: if self.term ≠ self.prev_term then
11:     init_local_switch(role)
12: end if
13: self.prev_term ← self.term
14: for dpid in self.etcd_terms do
15:     if self.master ≠ None and dpid not in self.master_datapaths then
16:         if self.etcd_terms[dpid]! = 0 then
17:             self.etcd_terms[dpid] ← 0
18:             role_request(dpid, EQUAL, 0)
19:         end if
20:     else if self.term ≠ self.etcd_terms[dpid] then
21:         self.etcd_terms[dpid] ← self.term
22:         role_request(dpid, role, self.term)
23:     end if
24: end for
```

- **Monitoring the status of the local etcd member instance**

  The monitor thread is scheduled to poll periodically the etcd instance for the status of the cluster. The status response contains 2 key values, the **Etcd term** and **the id of the leader**. We compare the id of the leader with our own to determine whether this controller will be the master or if it will be a slave. This is shown in lines 1-9 of Algorithm 1.

- **Requesting flow and port statistics from switches**

  Each controller periodically polls their local switch for flow statistics to monitor the liveness of the installed flows for control traffic. Since these flows can be installed by any controller when all controllers are in equal mode, these flows can become invalid while connection is lost with the controller that installed them, thus we need to monitor whether the connection to the destination of these flows is still live. The master controller additionally periodically polls switches for statistics information.

- **Role requesting to switches**

  Openflow 1.3 supports role assignment to switches using Role request messages. The master/slave controllers send a role request to each switch (except the local switch which is always considered as equal) including a generation id (as referred by Openflow) which is the current etcd term of the etcd cluster, to declare its mastership to the switches for this specific term, as it can be seen lines 14-22. The controller keeps record of the last term that a role request was sent on the self.etcd_term variable. If a switch is not yet connected to the master all other switches in the cluster will request to be equals for this switch.

  When a switch sets a controller to the master role, the controller has read-write access to the flow table of this switch and receives packet-ins from this switch. In the slave role, the controller has only read access to the flow table of the switch and does not receive any packet-ins. In the equal role the same is true as for the master role with the difference between these two roles being that at most only one master can exist for a switch. The previous master is always set to be a slave by the switch upon receiving a master role request with a higher generation/term number than the previous one.

  For the equal role in our implementation, in order to allow for some semi-autonomous routing for control traffic flows, when a master has been elected, all the controllers

set their local switch as equal and modify their openflow session controller_id to their configured datapath id using the OpenVswitch Nicira extensions.

By default when OpenVswitch initiates a connection with a controller it sets the controller_id of this session to 0. Also by default the packet-miss flow which is responsible for sending packet-ins to controllers when a packet's header fails to match any other flow rules, outputs to the `OFPP_NORMAL` port which sends a packet as packet-in to all non-slave controllers with controller_id = 0. Thus by changing the controller id of the session the local controller does not receive packet-ins from its local switch and instead packet-ins are only forwarded to the master, while also letting the controller keep its write access to its collocated switch. This also allows us to forward received LLDP messages from other switches, only towards the local controller and reduce redundant LLDP traffic as it will be explained in the topology section.

- **Monitoring the ids of controllers connected to the etcd cluster.** This is done in order to correctly assign roles to switches and monitor links committed from controllers in the cluster. Controllers request to be slaves to a switch once the node hosting this switch has connected with the master and joined the cluster.

- **Monitoring the liveness of links committed to etcd.** Each node is responsible for committing topology information about its neighborhood to etcd and updating this information when a new link is found or deleted. The neighborhood/domain can be defined depending on the cluster architecture and the hierarchical structure, but in our case it is simply all 1-hop neighboring nodes.

3.1.4 Producer thread

The producer (python) thread is responsible for sending requests to the etcd cluster using the etcd client library. When a green thread/event handler wants to put/delete a key to/from etcd it puts the request in a queue to be sent to the cluster by this thread. If an operation fails the prev_term of the controller is set to None, the queue is cleared and the controller retries to initialize its local switch and commit all known links to etcd.

It is important to note here that the eventlet library offers the option to ”patch” (otherwise called the monkey patch) blocking functions like socket.recv and various libraries and modules like the threading library, the os module etc, to make them behave like green threads

and yield control to other green threads when a green thread blocks. The etcd client library though does not work well with this patch probably because it does not use the default python sockets for its networking operations, thus we opt to not use it. This leads to a few issues.

While the operation could be done from a green thread (inside the link event handler,the monitor thread,the packet-in handler etc.) the green thread has to block until the operation from the etcd client library is finished. Blocking inside an event handler/green thread with a function that is not patched will cause the Ryu controller to be unable to process any incoming messages for the time we are blocked.

We opt to offload the blocking to a different python thread to complete these operations asynchronously in relation to the event handling of the controller, so the controller is free to continue while the producer thread is blocked waiting for etcd client operations to complete.

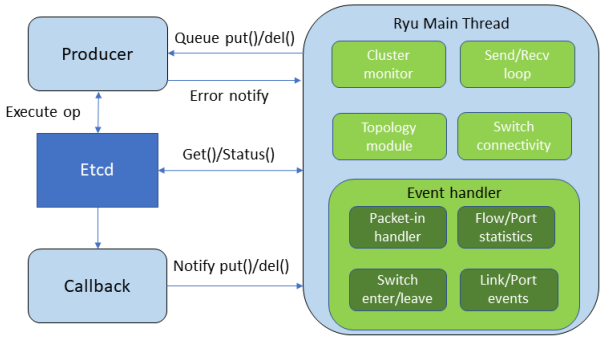

Figure 3.4: Application internal architecture

3.1.5 Callback thread

The callback thread is created by the etcd client library internally, and executes a callback function passed by the user when an event on etcd happens. The callback thread makes use of the etcd Watch API. The Watch API provides an event-based interface for asynchronously monitoring changes to keys. During the initialization of the controller as a slave or master, the controller sends a watch request for the topology keyrange to its local etcd server, to receive

updates on topology events from other cluster members and passes a function to the callback thread, which updates the topology according to these events. After the watch request is sent, we also do a get() operation on the topology keyrange to make sure that updates were not lost during the initialization. Ryu by default also patches the threading library, however since the etcd client library cannot work with the monkey patch we disable patching specifically for the threading library.

3.1.6 Topology discovery

Topology discovery uses three main mechanisms to discover new links and switches in the network:

- **The BFD protocol**

  BFD is a low overhead protocol which is usually used to detect link faults between two switches/routers. BFD establishes a session between two endpoints over a particular link. The session is established with a three-way handshake, and is torn down the same way. In our configuration script when GRE tunnels are initialized for each switch to communicate with other in the network we enable BFD (OpenVswitch implements BFD) for all GRE tunnels which allows the switch to monitor the liveness of the tunnels/ports.

  To detect the liveness of tunnels we make use of OpenFlow Group Actions and specifically we use fast failover groups (defined by Openflow) as the output action set for each control or data flow. Fast failover group actions monitor the liveness of a link through BFD. If the link of the output port is not active the packet is forwarded to a ”failover” destination which in our case is the `OFPP_CONTROLLER` port to send a packet-in to our local controller. For rules that drop packets the failover destination is empty.

- **ARP announcements**

  ARP announcements, which are used by each node to detect the presence of a neighboring node. While the controller itself does not check the ARP table of its host node to detect new incoming links, when the switch attempts to initialize a BFD session it has to know beforehand the MAC address of the other end of the tunnel and thus checks

the ARP table. Traditional ARP request-reply function for the wireless link interface is suspended and instead the arping daemon is used to generate ARP announcements to populate the ARP tables of neighbors. This can be seen as a simple HELLO message which announces the existence of a node to its neighbors so that they can initiate a BFD session with the node.

- **LLDP messages**

  LLDP messages which are generated by the controllers in order to detect links between switches. Ryu has an inbuilt module for topology discovery which we use in our application. This module was modified to send LLDP messages only to switches set to the equal role to avoid redundant LLDP messages to be sent by all controllers. Links can only be discovered towards/from switches that are already connected to a controller.

The process of topology discovery differs during the equal phase and the master phase. During the equal phase each controller connects to its 1-hop neighboring switches when the status of the port connecting the two of them is live as detected by BFD. The 1-hop neighbor controllers of a switch install flows on the switch so it can connect to its 2-hop neighboring switches, and thus each controller progressively connects with each switch in the network. The message exchanges showing this procedure are presented in Figure 3.5.

In the equal phase all controllers are set as equal by all switches. In the master phase all controllers are set to the slave role by all switches except from the elected master controller. The collocated controller of a switch is set to the equal role by this switch.Thus in the master phase each controller monitors only its own local links and reports them to the master through etcd. More specifically in our implementation each controller monitors the incoming links to its local switch, and outgoing links to any neighboring switch that is not part of the cluster (not connected to the master controller yet). Figure 3.6 shows the links contained in the local store (implemented by the Ryu controller) and the etcd store containing the committed links from all nodes.

In the master phase when a controller discovers a link with a neighbor, it commits this link to the etcd topology store, and installs the flow rules necessary in the neighboring switch to communicate with the master. The master installs the necessary flows on the switches along the path to connect itself with the new node, and the new node joins the cluster. After the node is connected to the master, the master installs the necessary flows on all other switches

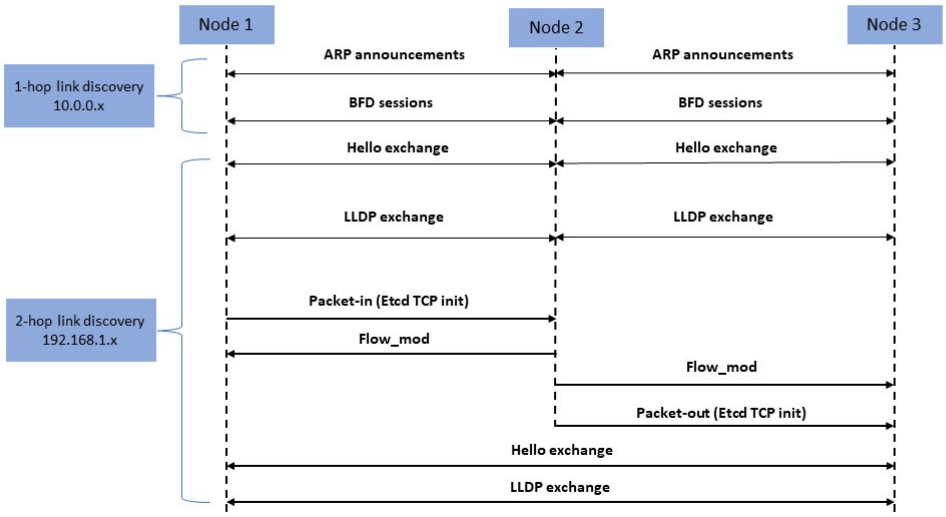

Figure 3.5: Neighborhood discovery

on demand, to connect them to the controller of this node.

3.1.7 Packet-in handler

The packet-in handler is responsible for setting up the control and data flows according to the topology information. When a packet is received from a datapath a controller finds the shortest path using the Djikstra algorithm on the network graph to find the shortest hop path, and installs flows along this path to direct the traffic to the destination of the packet.

All control and data flows are initialized in reaction to an incoming packet. The switch and etcd instance of a node are configured to connect to a set of other controllers/etcd instances in the network. An etcd instance generates packets destined for the configured connections and these packets go through the packet-in handler of each controller connected to this switch, where the handler installs flows in response to these packet-ins. This way the necessary flows for connectivity between etcd instances and controller-switches are set.

Note that openflow traffic is excluded and filtered out via special flow rules during the initialization phase of a switch, as getting packet-ins from the openflow protocol itself leads to the openflow packets getting looped in the node itself. When etcd traffic rules are set we also set openflow rules side by side, essentially piggybacking the openflow traffic flow

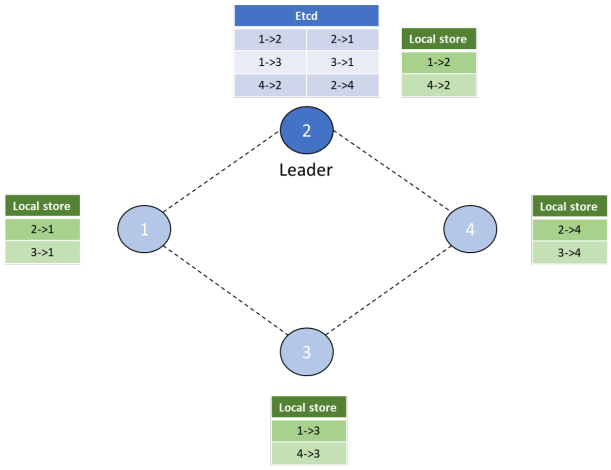

Figure 3.6: Topology stores during master phase

initialization to the etcd traffic flow initialization.

Data traffic is handled in the same way but the weights of the shortest path are calculated differently depending on whether the cluster of controllers has elected a master or not. More about the calculation of weights for the master phase on 3.2.

3.1.8 Local flow monitoring

Each controller sends periodically an `OFPP_PORT_STATUS` message to its local switch to monitor the installed flows. This is only done for control flows. If a control flow has been installed on a switch with destination a datapath that the collocated controller is not yet connected to, the flow is set to expire after 10 seconds when the controller will have to delete it. Since the OpenVswitch maximum backoff time for establishing a connection with a controller is 8 seconds we need a higher expiration time value in order to not prematurely delete flows. If the controller connects to the datapath during this period then the expiration time of the flow is deleted. The liveness of the TCP connection between a controller and a switch is periodically checked by sending `OFPP_ECHO_REQUEST` messages. When a switch disconnects from the controller, the controller erases all flows from its collocated switch with destination to the datapath of this switch.

This handler also optimizes current flows to be up to date with the current shortest paths. When the topology changes instead of erasing flows each controller attempts to change its local flows, to be up to date with the shortest paths. This is only done for control flows, since the shortest path policy for data flows uses different link quality metrics.

3.2 Link Quality Metrics

3.2.1 Link quality metrics in literature

Most routing algorithms for MANETs do not use the shortest hops path as the link quality metric, instead they often measure some characteristics of a link to determine its quality. OLSR for example makes use of ETX (Expected Transmission Count) [6] , which is the number of expected transmissions of a packet necessary for it to be received without error at its destination. This metric generally does a good job on assessing link loss or interference, and is in one way or another used by most MANET routing protocols.

For our controller routing policy though we chose to focus more on the side of traffic engineering, where we gather statistics about existing traffic and make routing decisions based upon them. Thus a simple link quality metric which calculates contention was implemented, which effectively attempts to balance the traffic load between links. Similar attempts have been made in literature to create such link quality metrics incorporating load balancing [28] [29] [30] albeit they were implemented to work with classic MANET routing protocols, and not with centralized architectures like SDN.

3.2.2 A contention metric/routing policy

The link metric used in our routing algorithm attempts to predict the local medium contention of a node by collecting statistics for the amount of traffic that each node. The master is responsible for collecting statistics. When a new data flow needs to be routed, the master finds the least contended path using the Djikstra algorithm (`shortestPath()` function from the NetworkX [31] library), and installs the appropriate flows on the nodes of the path to route the traffic. A data flow has an idle timeout of 3 seconds after which the flow is considered to be inactive, and thus gets deleted by each switch. By having a single entity coordinating the traffic in the network traffic flows can be routed to the least congested path easily, in contrast to the distributed approach where the view of the network in each node must converge for all of them to agree on an optimal path regardless of what metric they may use.

For a node n with S being the set of its neighbors (including n) and $T_x(x)$ being the amount of traffic transmitted by a node $x \in S$, the weight $W_{x \to n}$ of the edge x $\to$ n in the

graph is calculated as:

$$W_{x \to n} = \sum_{x \in S} T_x(x)$$

The value of T_x for a node is calculated by the master, by querying periodically each node with an `OFPPortStats` request which returns the amount of traffic transmitted in bytes to each port corresponding to each GRE tunnel.

3.3 Etcd Traffic Optimizations

Etcd was not designed for use in hosts that are connected by wireless links. Therefore while etcd may not generate a lot of traffic for wired networks or datacenter standards on its own, it does for wireless links. This section describes the optimizations that were made to lower the overhead of etcd messages.

3.3.1 Blocking peer traffic

When a master is elected, according to the Raft[1] protocol each node needs to communicate only with the master to commit entries in the log and the master replicates the entries to all other nodes. Additionally the master has to send heartbeats to all followers tin order to preserve its leadership status otherwise a node can start new elections.

When a cluster is configured in etcd, each node establishes a connection to all other nodes and keeps this connection alive with link heartbeat messages. When a master is elected the connections towards every node except the master are not used and remain idle. Unless we want to use some additional features of etcd which are outside the scope of Raft like leadership transfer, each other operation (get(),put(),status_request()) does not make use of these redundant connections.

Therefore to lower the overhead of etcd idle connections when a node recognizes that a master has been elected it blocks incoming and outgoing etcd traffic to all nodes except the master by installing appropriate flows. The node who recognizes itself as the master keeps all of its connections alive with other etcd instances so it can serve the cluster. When a node returns to being an equal it erases these flows to allow itself to connect with all etcd instances again to elect a new master. When a switch gets disconnected from a controller, the controller deletes from its local switch the flows blocking the outgoing etcd traffic to this switch, since we need etcd traffic to re-establish new flows for the openflow traffic towards this switch.

This approach dramatically reduces the overhead of etcd traffic with minimal tradeoffs. Even in the case of new elections nodes generally quickly establish new connections and elect a master.

3.3.2 Removing diagnostics/statistics

Etcd by default sends special probe messages to all peers to provide debugging diagnostics for connections to the user and measure statistics like latency and clock drift between nodes. These messages were removed since they are not useful to our application in any way, reducing significantly the overhead of etcd traffic.

3.3.3 Merging message streams

Etcd uses 2 kinds of tcp connections to communicate with peers. Streams which are long lived connections between peers and pipelines which are short-lived connections created in demand, or before streams are established. Etcd uses 2 types of streams, one general stream to carry all messages, and an optimized stream which is used by the leader to send MsgApp messages which are the log entries to replicate.

An optimized stream is used, as MsgApp are the main bulk of messages and making the delivery of these messages parallel to the delivery of other control messages improves the performance of writes in etcd and provides lower latency for control messages. However in our use case etcd is not making a significant amount of writes to justify the overhead of preserving this stream with link heartbeat messages, thus we merge the general and the optimized stream into 1 stream.

3.3.4 Adjusting tuning parameters

The election timeout and heartbeat interval that were chosen for etcd are 5 and 1 seconds respectively. Etcd recommends that the election timeout is at least 5 times the heartbeat interval and this was the most acceptable combination to achieve low heartbeat overhead and relatively acceptable recovery time for leader loss. The read and write timeout of etcd TCP connections was also modified from 5 to 10 seconds to reduce the amount of link heartbeat messages sent, which now are sent each 5 seconds (half the read/write timeout) to be inline with the election timeout.

Chapter 4

Simulation

4.1 Wmediumd

To test the SDN controller implementation the Mininet-Wifi [32] network emulator was used. Wmediumd [33] is the simulation tool that Mininet-Wifi uses to simulate frame delivery delay and packet loss. In conjunction with the Mac_hwsim kernel module which simulates radio interfaces for Linux, these two components form a pipeline which simulates packet transmission between virtual interfaces.

Figure 4.1 presents the pipeline that a message from user space follows until it reaches wmediumd and gets delivered to its destined virtual interface.When a packet is received from user space, mac_hwsim notifies and forwards this packet to wmediumd using netlink sockets. When wmediumd receives a message it schedules the delivery time of this packet to a time slot by setting a timer. This time slot is chosen by an algorithm which calculates contention and interference on the channel and finds the appropriate time slot to transmit a packet. When the timer is up the frame is delivered to each destination (if the packet has been marked as successfully delivered by the contention and interference algorithm) and sent to the mac_hwsim module which finally delivers it to the user space receiver interface.

In the following sections we describe the issues encountered with the wmediumd simulation and the changes that were made to make the simulation more realistic.

4.1.1 Contention simulation

Wmediumd makes the simplifying assumption that during the transmission of a packet from any interface inside the simulation , no other interface can freely transmit since it would

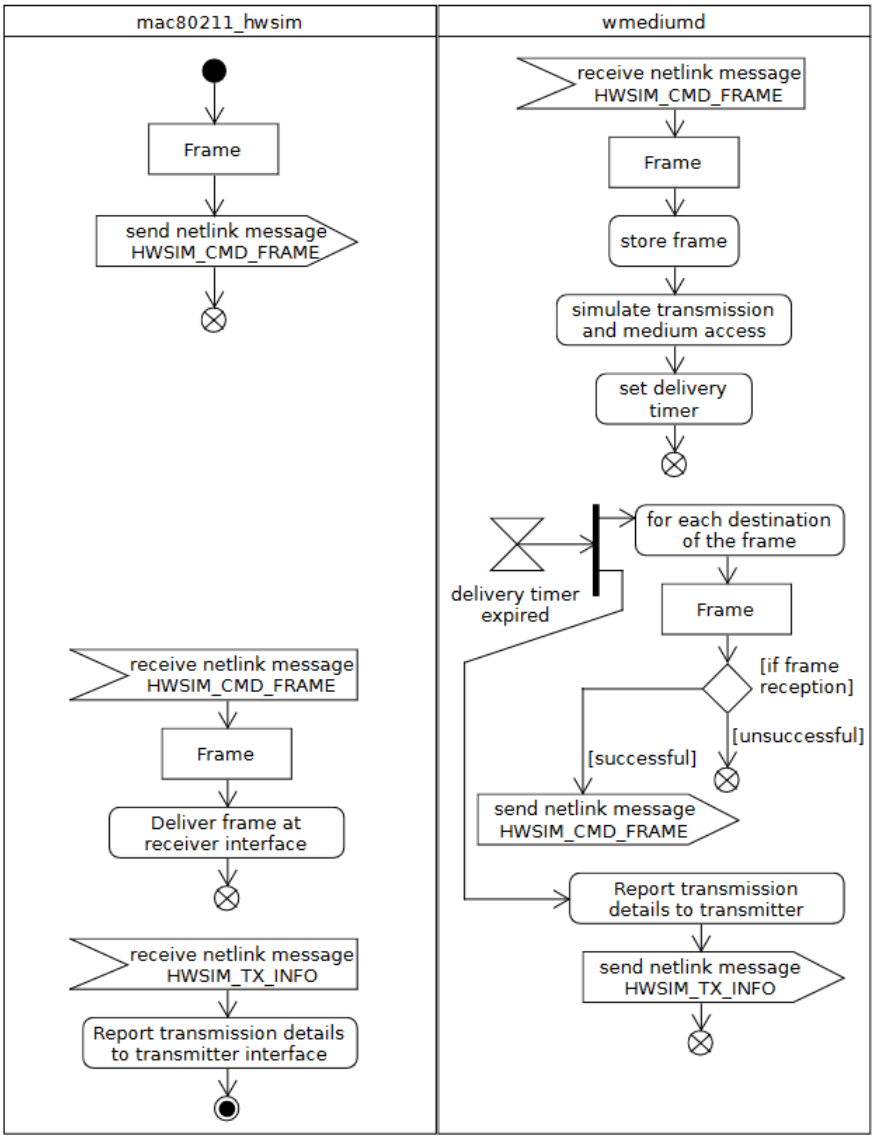

Figure 4.1: The wmediumd and mac_hwsim pipeline [2]

contend and collide with the other transmission. While this simple contention model can work really well for traditional wireless networks where multiple stations associate with a single Access Point, it becomes very limiting and unrealistic once applied to mesh networks.

This became apparent in experiments with 2 pairs of stations, where one station in each pair sent traffic, separated by a long distance (long enough for them to receive virtually no signal from the other pair), could not output the same throughput they did when transmitting on their own and instead their throughput was always halved. To make the simulation more realistic Algorithm 2 was implemented in place of the contention algorithm.

This algorithm calculates the `target` which is the time that the next transmission from a station should begin. The main difference of this algorithm with the previous approach is

Algorithm 2 Contention algorithm

1: $in\_range \leftarrow \{dst \in stations \mid get\_link\_snr(station, dst) > 0\}$
2: $target \leftarrow now$
3: $snr\_station\_dst \leftarrow 0$
4: **if** $deststa$ is unicast **then**
5: $target \leftarrow max(target, station.last\_tx)$
6: $snr\_station\_dst \leftarrow get\_link\_snr(station, deststa)$
7: **if** $snr\_station\_dest > 0$ **then**
8: $target \leftarrow max(target, deststa.last\_tx)$
9: **end if**
10: **for** tmpsta in in_range **do**
11: $target \leftarrow max(target, tmpsta.prev\_exp)$
12: **end for**
13: **else**
14: **for** tmpsta in in_range **do**
15: $target \leftarrow max(target, tmpsta.last\_tx)$
16: **end for**
17: **end if**
18: $target \leftarrow target + send\_time$
19: $station.prev\_exp \leftarrow target$
20: **if** $deststa$ is unicast and $snr\_station\_dst > 0$ **then**
21: $deststa.prev\_exp \leftarrow target$
22: **else if** $deststa$ is broadcast **then**
23: **for** tmpsta in in_range **do**
24: $tmpsta.prev\_exp \leftarrow target$
25: **end for**
26: **end if**
27: **for** tmpsta in in_range **do**
28: $tmpsta.last\_tx \leftarrow max(target, tmpsta.last\_tx)$
29: **end for**

that when a packet is scheduled, we always consider the SNR of the signal that a station receives from a transmission. This is a less restricted approach which allows simultaneous transmissions to occur inside the simulation provided that the stations transmitting do not directly contend with each other. While some stations can still receive a weak signal with $SNR \leq 0$, under normal circumstances they would consider this signal as nothing more than noise as it would be below the CCA threshold and therefore would not back off. Each station has 2 values:

- `last_tx` is the deadline for the last transmission that a station has listened to.
- `prev_exp` is the deadline for the last packet transmission from or to this station.

The difference between these two values is that `last_tx` accounts for any transmission that a station has listened to, even if the message is not destined for this station (or sent by itself), while `prev_exp` only accounts for transmissions destined towards this specific station (or sent by itself). For broadcasts all nodes with receiving $SNR > 0$ are considered as destinations of this transmission. Obviously for these 2 values the following relation always holds: $last\_tx \geq prev\_exp$.

Initially the target is set as the time when a new packet has been received by the simulation. When a station queues a packet transmission , the algorithm first checks the `last_tx` of this station (in line 5), as in order to transmit no other ”neighboring” station must be transmitting during this time and thus target is set as the max value between its current value and `last_tx`. The same procedure also follows for the destination (provided that the SNR received by the destination is greater than zero), where the algorithm checks (in line 8) the `last_tx` of the destination as the destination must not be in the process of sending or listening to any other transmission during our transmission.

Finally we also check the `prev_exp` of all other stations (in line 11) to make sure that our transmission will not interfere with any other neighboring station which is in the process of sending or receiving a packet.

For broadcasts (in lines 13-17) we just check the `last_tx` of all stations that will receive this message (including our `last_tx`), as these stations must not be listening to any other transmission during this time.

The max of all these values is set as the target time to begin transmission for the next packet. `send_time` is added to the target and now target denotes the time that the simulation

will actually deliver the packet the packet to destination interface. This time is set as the `prev_exp` of the transmitter and receiver (or receivers in the case of broadcast) and the `last_tx` of all stations that receive the signal with $SNR > 0$ (including the transmitter).

This algorithm still makes many simplifying assumptions, as it assumes a perfect synchronization among all stations to keep the medium busy at all times. Delivery times and whether a packet will be delivered successfully once computed/decided cannot be changed from later transmissions in real time, although this assumption was also made in the previous algorithm as well. An alternative implementation which improves on many aspects like the handling of events, simulation performance with multiple threads and adds support for multiple medium simulation is proposed in this thesis [2], however we could not use this code for wmediumd since there were changes on the interface with Mininet-Wifi.

4.1.2 Interference simulation

While signals above the CCA threshold are handled by the contention algorithm, stations also receive weak signals. These signals are considered by wmediumd as interference. The interference algorithm of wmediumd accumulates the durations of such signals and assumes (accumulated duration / time slot) is the probability of occurrence of interference. When interference occurs, the model reduces the signal strength.

Fixes were also made to the interference model since it wrongly added the same interference caused at the destination of the packet, to all other stations. The interference caused at each station is different depending on the SNR of the received signal. This did not cause any meaningful changes in any metrics during simulations though.

4.1.3 MAC layer acknowledgements

For successful transmissions the acknowledgement time (for MAC layer acknowledgements) was also added to the total send_time as suggested by [2] , as it was wrongly omitted from the calculation when a packet was successfully delivered, which affects unicast messages and reduced the throughput stations can achieve significantly.

4.1.4 QoS priorities

Finally, the priority of a frame according to its QoS type is not taken into consideration in the contention algorithm. The previous algorithm took QoS priority into consideration however the implementation was problematic, since it allowed for simultaneous transmissions from a station if a higher priority frame arrived for processing after a lower priority frame was already queued for delivery. To implement this we would realistically need to process transmissions in real time rather than precalculating delivery times since frame priorities work with AIFS (Arbritrary Inter Frame Spacing) which is the time period a station will wait until it attempts to transmit a frame giving room for higher priority frames (with lower AIFS) to transmit first. Such an implementation would significantly diverge from the original one and thus QoS priority is not accounted for in the contention algorithm.

Chapter 5

Evaluation and results

5.1 Setup and topology

To test our distributed SDN controller implementation and evaluate the effectiveness of the routing policy enforced by the controller, a simple scenario was devised which showcases the weakness of other MANET protocols to respond to situations where medium contention occurs between nodes. The topology that was used can be seen in Figure 5.1. The experiment that was conducted consisted of two TCP transmissions between s1-s5 and s3-s6 using iperf [34]. The transmissions take place in the following order: at t = 0s s3 initiates a TCP transmission towards s6, and at t = 25s s1 initiates a TCP transmission towards s5. Each transmission lasts for 250 seconds, meaning that from t = 25s through t = 250s s1 and s3 transmit concurrently and from t = 250s through t = 275s s1 transmits on its own. During this scenario s1 has 2 choices: to route traffic through s2 or s4.

The most optimal route for this scenario is the s1-s4-s5 route since the traffic transmitted from s3 towards s6, contends the medium for all nodes that are inside the transmission range of s3 including s2. Therefore if s1 decides to transmit through s2 then its transmission may collide with s3 transmitting towards s6 resulting in packet loss. This is a typical case of the hidden node phenomenon in wireless networks, where a node (s1) transmitting data towards another node (s2), is unaware of the existence of a third node (s3) which also transmits data either to s2 directly or to another neighbor, resulting in interference between s1 and s3. Even if s2 receives the packet, it still has to transmit towards s5, meaning that it will have to backoff until any transmission from s3 is over, and the same is also true for s3 in the case s2 is already transmitting. Transmission through s2 will result in significant packet loss and throughput

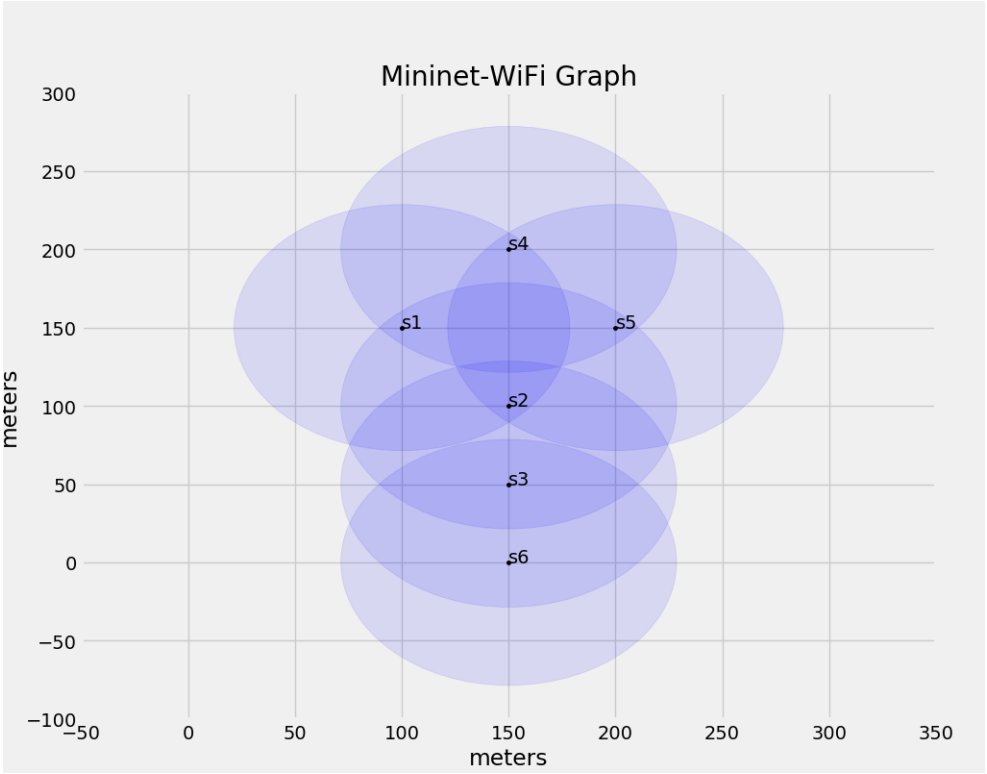

Figure 5.1: Experiment topology

loss for both TCP connections.

Our link quality metrics can predict such scenarios and direct traffic through the least congested path, by measuring the amount of data each node sends and the interference it causes for its neighbors as it is explained in 3.2. The fact that our approach is also centralized in contrast to the distributed nature of other routing protocols, allows us to implement this algorithm in a stable manner, since one node (the master) decides the path of a particular flow based on the metrics it has received until this moment and can keep track of this flow and its impact on the network until the transmission is over. If multiple nodes were to execute this algorithm, they would need to converge to an optimal solution and even then they would have to cooperate to keep the route of this flow stable and not oscillate between different routes (due to each node having a different view of the network state and traffic between paths).

5.2 Results

Figure 5.2 shows the average throughput among 12 measurements in the described scenario, achieved by all conventional routing protocols and by the SDN controller for the s3-s6 and s1-s5 TCP connections. SDN achieves the best performance using the routing policy/link

quality metrics described in 3.2 to detect path congestion. Olsr achieves the lowest throughput because it always chooses the ”wrong” path. This does not change during the transmissions despite the link quality metrics of Olsr being enabled. The reason Olsr chooses this path is due to the ordering of nodes. S2 has the 10.0.0.2 ip address while s4 has the 10.0.0.4 ip address and thus Olsr breaks the ”tie” between the two paths by choosing lowest ip address. The nodes were intentionally placed this way to showcase the inability of Olsr to distinguish between congested paths, if s2 changed place with s4 the results would be completely different, and Olsr would outperform the SDN results due to the lower overhead of Olsr messages.

Batman achieves slightly better results but it still shows reduced throughput, as it is highly unstable in choosing a path thus resulting in much lower throughput due to routing oscillations during the two transmissions.

Batman IV achieves better results from its original version, due to implementing link quality metrics in a much better way, resulting in less path oscillations. Batman V performs the closest to SDN due to its vastly different way of calculating link quality metrics in relation to other protocols, in fact the way batman V does this resembles the way (or at least targets) our solution estimates link quality by estimating the throughput that can be achieved through a link, instead of using a packet loss related metric like the other routing protocols. This allows Batman V to stay more regularly on the right path compared to other protocols.

It is important here to note that for Batman IV and Batman V a lower MSS was used during the TCP transmission of 1436 bytes in order to avoid packet fragmentation and achieve the best performance possible. All routing protocols run with an MTU of 1500 bytes to make the comparison fair.

In general these protocols do not distinguish between congested and not congested paths. Packet loss is correlated to congestion but it does not specifically detect when a specific area of a wireless network has become a hot spot. That is why most protocols oscillate between these two routes as when a path becomes congested it is more likely to experience packet loss, provided that packets are sent at a high enough rate to observe that a path is congested through packet loss. Our approach is more static since once a path is decided for a flow it cannot change unless a link breaks or the flow transmission ends. This limits our options for more dynamic scenarios, where other protocols can change their routes. However even with the 25 second initial transmission all competing MANET protocols failed to detect the congested path more than once when the second transmission from s1 began, while our routing algorithm

running on the SDN controller never failed to make the right choice. The initial choice of the route is done on equal terms for all protocols.

This metric and the routing policy that was used to route flows is not meant to be really optimal but rather to showcase that even with a very simple policy we can get better results in certain scenarios compared to decentralized routing solutions. More sophisticated algorithms for traffic engineering using SDN have been proposed that are fully dynamic and attempt to approximate the optimal routing of flows to maximize network utilization [35]. For SDN in MANETs there have also been proposals for routing policies [36] that attempt to maximize network utilization.

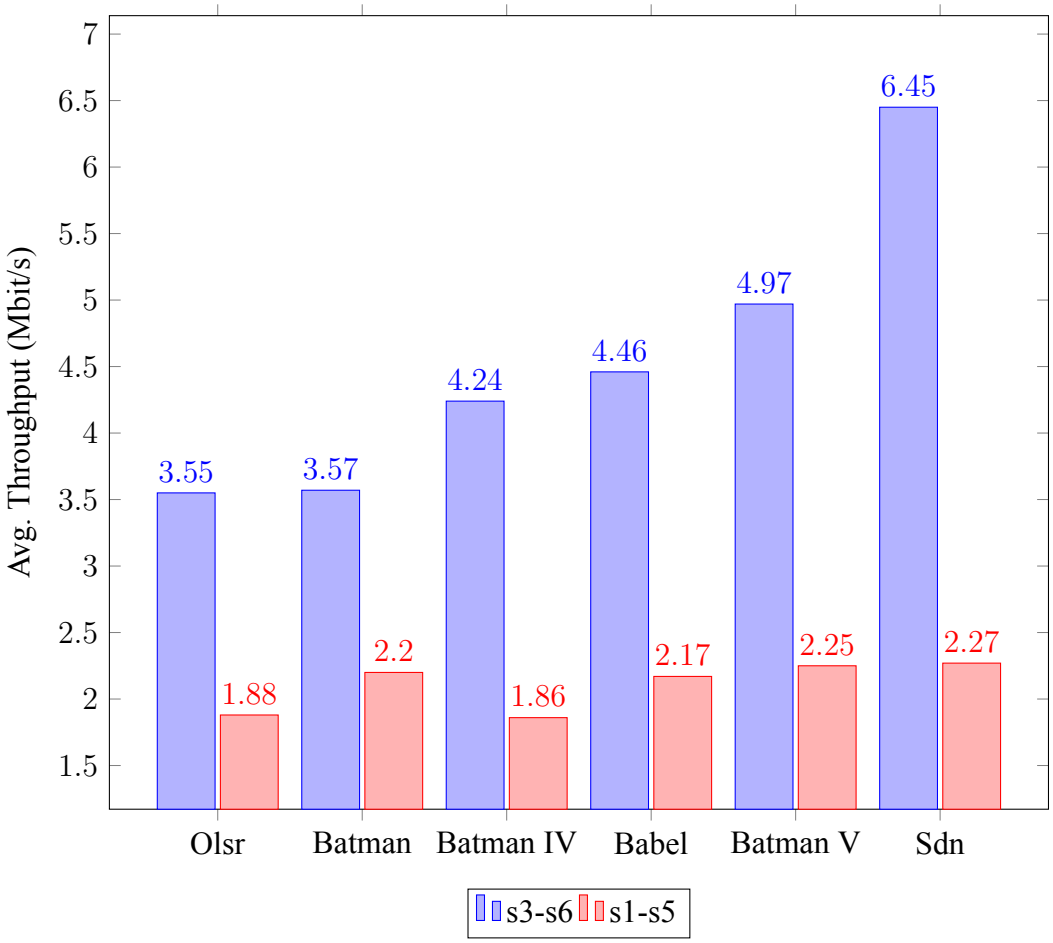

Figure 5.2: Average throughput

Chapter 6

Conclusions and Future Work

In this thesis a distributed SDN controller for wireless networks was presented that enables self-organizing deployment of SDN in MANETs. The controller has been tested and its effectiveness on certain scenarios has been showcased through a simple experiment where global traffic optimization is necessary to avoid inefficient routing paths in wireless networks.

While this work builds a strong basis for the integration of SDN in MANETs there are stil several problems concerning the tight integration of these two technologies. Scalability concerns are among the most serious issues that need to be tackled in order to make SDN architecture viable in MANETs. To this end it is necessary to impose a hierarchical structure on our SDN architecture (similar solutions have been proposed [37]) where each controller with some replicas are responsible for specific domains of the network instead of the entire network in order to reduce controller to switch latency and reduce the overhead of Openflow and etcd messages alike.

Additionally our current implementation can be expanded to synchronize flow table view and operations between controllers using the backend already in place. The election of a master can also be done according to specific metrics instead of letting it happen randomly, using the membership transfer function of etcd.

Finally it is necessary to study and implement more sophisticated traffic engineering algorithms that allow for more dynamic control and fine grained routing of flows taking into account the special nature of wireless transmissions.

Bibliography

[1] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[2] Miguel Jose Meireles Moreira. yawmd: multiple medium support and performance improvements for wmediumd. Master's thesis, 2020.

[3] Dimitrios Kafetzis, Spyridon Vassilaras, Georgios Vardoulias, and Iordanis Koutsopoulos. Software-defined networking meets software-defined radio in mobile ad hoc networks: State of the art and future directions. *IEEE Access*, 10:9989–10014, 2022.

[4] Konstantinos Poularakis, George Iosifidis, and Leandros Tassiulas. Sdn-enabled tactical ad hoc networks: Extending programmable control to the edge. *IEEE Communications Magazine*, 56(7):132–138, 2018.

[5] Olsr. `https://datatracker.ietf.org/doc/html/rfc3626`.

[6] Douglas SJ De Couto, Daniel Aguayo, John Bicket, and Robert Morris. A high-throughput path metric for multi-hop wireless routing. In *Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 134–146, 2003.

[7] Batman documentation. `https://www.open-mesh.org/projects/batman-adv/wiki`.

[8] Babel. `https://datatracker.ietf.org/doc/html/rfc896`.

[9] Openflow switch specification. `https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf`.

[10] Etcd. `https://etcd.io/`.

[11] Peter Dely, Andreas Kassler, and Nico Bayer. Openflow for wireless mesh networks. In *2011 proceedings of 20th international conference on computer communications and networks (ICCCN)*, pages 1–6. IEEE, 2011.

[12] Andrea Detti, Claudio Pisa, Stefano Salsano, and Nicola Blefari-Melazzi. Wireless mesh software defined networks (wmsdn). In *2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 89–95, 2013.

[13] Sachin Sharma, Avishek Nag, Paul Stynes, and Maziar Nekovee. Automatic configuration of openflow in wireless mobile ad hoc networks. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 367–373. IEEE, 2019.

[14] Stefano Salsano, Giuseppe Siracusano, Andrea Detti, Claudio Pisa, Pier Luigi Ventre, and Nicola Blefari-Melazzi. Controller selection in a wireless mesh SDN under network partitioning and merging scenarios. *CoRR*, abs/1406.2470, 2014.

[15] Serge Fdida, Thanasis Korakis, Harris Niavis, Stefano Salsano, and Giuseppe Siracusano. The express sdn experiment in the openlab large scale shared experimental facility. In *2014 International Science and Technology Conference (Modern Networking Technologies)(MoNeTeC)*, pages 1–7. IEEE, 2014.

[16] Mohamed Labraoui, Michael Boc, and Anne Fladenmuller. Self-configuration mechanisms for sdn deployment in wireless mesh networks. In *2017 IEEE 18th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–4. IEEE, 2017.

[17] Iulisloi Zacarias, Luciano P Gaspary, Andersonn Kohl, Ricardo QA Fernandes, Jorgito M Stocchero, and Edison P de Freitas. Combining software-defined and delay-tolerant approaches in last-mile tactical edge networking. *IEEE Communications Magazine*, 55(10):22–29, 2017.

[18] Fei Xiong, Aijing Li, Hai Wang, and Lijuan Tang. An sdn-mqtt based communication system for battlefield uav swarms. *IEEE Communications Magazine*, 57(8):41–47, 2019.

[19] C Yu Hans, Giorgio Quer, and Ramesh R Rao. Wireless sdn mobile ad hoc network: From theory to practice. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2017.

[20] Mohamed Labraoui, Michael Mathias Boc, and Anne Fladenmuller. Software defined networking-assisted routing in wireless mesh networks. In *2016 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 377–382. IEEE, 2016.

[21] Syed Sherjeel A Gilani, Amir Qayyum, Rao Naveed Bin Rais, and Mukhtiar Bano. Sdnmesh: an sdn based routing architecture for wireless mesh networks. *IEEE Access*, 8:136769–136781, 2020.

[22] Paolo Bellavista, Alessandro Dolci, and Carlo Giannelli. Manet-oriented sdn: motivations, challenges, and a solution prototype. In *2018 IEEE 19th International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*, pages 14–22. IEEE, 2018.

[23] Ziyao Zhang, Qiaofeng Qin, M Liang, Konstantinos Poularakis, Franck Le, K Leung, Sastry Kompella, and Leandros Tassiulas. Routing performance in distributed sdn under synchronization constraint. *DAIS-ITA Project, New York, NY, USA, Tech. Rep*, 2485, 2018.

[24] Open vswitch. `https://www.openvswitch.org/`.

[25] Ryu docs. `https://ryu.readthedocs.io/en/latest/index.html`.

[26] python-etcd3. `https://python-etcd3.readthedocs.io/en/latest/usage.html`.

[27] Eventlet. `https://eventlet.net/`.

[28] Devu Manikantan Shila and Tricha Anjali. Load-aware traffic engineering for mesh networks. In *2007 16th International Conference on Computer Communications and Networks*, pages 1040–1045. IEEE, 2007.

[29] Sonia Waharte, Brent Ishibashi, Raouf Boutaba, and D Meddour. Interference-aware routing metric for improved load balancing in wireless mesh networks. In *2008 IEEE International Conference on Communications*, pages 2979–2983. IEEE, 2008.

[30] Liang Ma and Mieso K Denko. A routing metric for load-balancing in wireless mesh networks. In *21st international conference on advanced information networking and applications workshops (AINAW'07)*, volume 2, pages 409–414. IEEE, 2007.

[31] Networkx. `https://networkx.org`.

[32] Mininet-wifi. `https://mininet-wifi.github.io/`.

[33] Wmediumd. `https://github.com/ramonfontes/wmediumd`.

[34] Iperf docs. `https://iperf.fr/`.

[35] Sugam Agarwal, Murali Kodialam, and TV Lakshman. Traffic engineering in software defined networks. In *2013 Proceedings IEEE INFOCOM*, pages 2211–2219. IEEE, 2013.

[36] Klement Streit, Nils Rodday, Florian Steuber, Corinna Schmitt, and Gabi Dreo Rodosek. Wireless sdn for highly utilized manets. In *2019 International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 226–234. IEEE, 2019.

[37] James Nguyen and Wei Yu. An sdn-based approach to support dynamic operations of multi-domain heterogeneous manets. In *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 21–26. IEEE, 2018.