Department of Computer Science
Series of Publications A
Report A-2022-11

Seamless Programmable Multiconnectivity

Seppo Hätönen

Doctoral dissertation, to be presented for public examination with the permission of the Faculty of Science of the University of Helsinki in Auditorium B123, Exactum Building, on the 30th of September, 2022, at 13 o'clock.

University of Helsinki
Finland

Supervisors

Sasu Tarkoma, University of Helsinki, Finland
Ashwin Rao, University of Helsinki, Finland

Pre-examiners

Anna Brunström, Karlstad University, Sweden
Mika Ylianttila, University of Oulu, Finland

Opponent

Michael Welzl, University of Oslo, Norway

Custos

Sasu Tarkoma, University of Helsinki, Finland

Contact information

Department of Computer Science
P.O. Box 68 (Pietari Kalmin katu 5)
FI-00014 University of Helsinki
Finland

Email address: info@cs.helsinki.fi
URL: http://cs.helsinki.fi/
Telephone: +358 2941 911

ISSN 1238-8645 (print)
ISSN 2814-4031 (online)
ISBN 978-951-51-8528-0 (paperback)
ISBN 978-951-51-8529-7 (PDF)
Helsinki 2022
Unigrafia

Seamless Programmable Multiconnectivity

Seppo Hätönen

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
Seppo.Hatonen@helsinki.fi
http://cs.helsinki.fi/u/shatonen/

PhD Thesis, Series of Publications A, Report A-2022-11
Helsinki, September 2022, 88+57 pages
ISSN 1238-8645 (print)
ISSN 2814-4031 (online)
ISBN 978-951-51-8528-0 (paperback)
ISBN 978-951-51-8529-7 (PDF)

Abstract

Our devices have become accustomed to being always connected to the Internet. Our devices from handheld devices, such as smartphones and tablets, to our laptops and even desktop PCs are capable of using both wired and wireless networks, ranging from mobile networks such as 5G or 6G in the future to Wi-Fi, Bluetooth, and Ethernet. The applications running on the devices can use different transport protocols from traditional TCP and UDP to state-of-the-art protocols such as QUIC. However, most of our applications still use TCP, UDP, and other protocols in a similar way as they were originally designed in the 1980s, four decades ago. The transport connections are a single path from the source to the destination, using the end-to-end principle without taking advantage of the multiple available transports.

Over the years, there have been a lot of studies on both multihoming and multipath protocols, *i.e.*, allowing transports to use multiple paths and interfaces to the destination. Using these would allow better mobility and more efficient use of available transports. However, Internet ossification has hindered their deployment. One of the main reasons for the ossification is the IPv4 Network Address Translation (NAT) introduced in 1993, which allowed whole networks to be hosted behind a single public IP address. Unfortunately, how this many-to-one translation should be done was not standardized thoroughly, allowing vendors to implement their own versions

of NAT. While breaking the end-to-end principle, the different versions of NATs also behave unpredictably when encountering other transport protocols than the traditional TCP and UDP, from forwarding packets without translating the packet headers to even discarding the packets that they do not recognize. Similarly, in the context of multiconnectivity, NATs and other middleboxes such as firewalls and load balancers likely prevent connection establishment for multipath protocols unless they are specially designed to support that particular protocol.

One promising avenue for solving these issues is Software-Defined Networking (SDN). SDN allows the forwarding elements of the network to remain relatively simple by separating the data plane from the control plane. In SDN, the control plane is realized through SDN controllers, which control how traffic is forwarded by the data plane. This allows controllers to have full control over the traffic inside the network, thus granting fine-grained control of the connections and allowing faster deployment of new protocols. Unfortunately, SDN-capable network elements are still rare in Small Office / Home Office (SOHO) networks, as legacy forwarding elements that do not support SDN can support the majority of contemporary protocols. The most glaring example is the Wi-Fi networks, where the Access Points (AP) typically do not support SDN, and allow traffic to flow between clients without the control of the SDN controllers.

In this thesis, we provide a background on why multiconnectivity is still hard, even though there have been decades worth of research on solving it. We also demonstrate how the same devices that made multiconnectivity hard can be used to bring SDN-based traffic control to wireless and SOHO networks. We also explore how this SDN-based traffic control can be leveraged for building a network orchestrator for controlling and managing networks consisting of heterogeneous devices and their controllers. With the insights provided by the legacy devices and programmable networks, we demonstrate two different methods for providing multiconnectivity; one using network-driven programmability, and one using a userspace library, that brings different multihoming and multipathing methods under one roof.

Computing Reviews (2012) Categories and Subject Descriptors:

Networks → Network components → Middle boxes / Network appliances

Networks → Wireless access networks
Networks → Network layer protocols
Networks → Transport protocols
Networks → Session protocols
Networks → Network Components → Bridges and switches
Networks → Control path Algorithms

General Terms:
Software-Defined Networking, Multipath, Multihoming, Multiconnectivity, Network orchestration, IoT, Wi-Fi Networks

Additional Key Words and Phrases:
Middleboxes, Legacy Hardware, Wireless Access Points, Programmable Networks

Acknowledgements

This journey that I have taken to reach my Ph.D. has been a long one, and without the support and the trust of my supervisors, Professor Sasu Tarkoma and Ashwin Rao, it would not have been possible. Over the years, there have been many ups and downs, and sometimes I have wondered if I will ever reach the end of this journey. However, the support from my supervisors, colleagues, and friends has made it possible to achieve this goal.

First of all, I want to thank my supervisors, Sasu Tarkoma and Ashwin Rao, who have guided and supported me through my Ph.D. studies. I also want to thank my pre-examiners, Anna Brunström and Mika Ylianttila, and my opponent, Michael Welzl.

I want to thank the Department of Computer Science for the years I have been part of it, and the colleagues I have worked with. Julien Mineraud, Samu Varjonen, Matteo Pozza, Pupu Toivonen, Tiina Niklander, Markku Kojo, Jussi Kangasharju, Maksym Gabielkov, Lauri Hyttinen, Aki Nyrhinen and many others deserve my thanks and gratitude. I also want to thank DoCS and Pirjo Moen in particular for making sure that I could complete this journey as smoothly as possible.

I also want to extend my thanks to the IT staff of the Department of Computer Science, IT4Science, and HIIT. Without their support, the experimental parts of this thesis would not have been possible. Especially Pekka Niklander, Pasi Vettenranta, and Pekka Tonteri have been invaluable over the years. Their support has made running the Nodeslab possible and allowed me to learn new things beyond the scope of my thesis.

The organizations that have funded my research also deserve my thanks. Namely, Business Finland, Academy of Finland, and Nokia Center for Advanced Research (NCAR) have supported me through various projects.

Finally, I want to thank my family and friends, who have supported me over the years. Without them, I would not be writing this.

Espoo, September 2022
Seppo Hätönen

Contents

List of Abbreviations

3G	3rd Generation
4G	4th Generation
5G	5th Generation
6G	6th Generation
AP	Access Point
ARP	Address Resolution Protocol
DCCP	Datagram Congestion Control Protocol
DHCP	Dynamic Host Control Protocol
DPI	Deep Packet Inspection
HE	Happy Eyeballs
HIP	Host Identity Protocol
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IoT	Internet-of-Things
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISP	Internet Service Provider
LAN	Local Area Network
MAC	Medium Access Control
ML	Machine Learning
MPQUIC	Multipath QUIC
MPTCP	Multipath TCP

NAT	Network Address Translation
OF	OpenFlow
OVS	Open vSwitch
PVLAN	Private VLAN
QoE	Quality of Experience
RFC	Request for Comments
RTT	Round-Trip Time
SCTP	Stream Control Transmission Protocol
SDN	Software-Defined Networking
SoC	System on a Chip
SOHO	Small Office / Home Office
TAPS	IETF Transport Services Working Group
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VLAN	Virtual Local Area Network
VPN	Virtual Private Network
WAN	Wide Area Network
Wi-Fi	Wireless Fidelity
WLAN	Wireless Local Area Network
WLC	Wireless LAN Controller
WNC	Wireless Network Controller

Chapter 1

Introduction

Internet connectivity is a mainstay of the modern world. We are all almost always connected to the Internet, usually through our smartphones. The advances in technology have made it possible for us to be connected even when we are moving, and at the same time, fixed landline connections have become rarer over the years. Nonetheless, our devices still connect to the Internet using a single active network connection, even though our networked devices could use multiple different transport technologies and protocols. We are still unable to leverage on these for seamless multiconnectivity, as the choices of how the Internet was designed limits our choices for implementing multiconnectivity. At best, our devices can switch from one transport to another, but this transition is not seamless, nor can we easily control it through programmatic approaches.

To understand why this is so, we need to examine the history of the Internet and re-evaluate past design decisions. Since the 1990s, Internet connectivity has become faster, cheaper, and available almost everywhere. Many of the regions that are sparsely populated are increasingly being connected to the Internet. These regions are mostly covered with some form of broadband, be it either cellular networks or satellite-based connections.

Earlier, fast Internet connectivity was mainly limited to fixed places such as homes, workplaces, or Internet cafes. However, when the mobile networks started deploying 3G networks, the increased network speeds allowed us to start using the Internet anywhere where there was connectivity. With the 4G networks, the mobile Internet was brought closer to the level of fixed broadband, and with the 5G networks being currently deployed and 6G in the future, mobile broadband has already surpassed many of the older fixed broadband technologies.

This proliferation of Internet connectivity has changed our lives dramatically. We can now access almost every content available on the Internet

from almost everywhere. At the same time, most of the tasks that we need to do to live our lives normally have moved to the Internet. Services such as banking, mails, teleconferencing, and watching TV have moved to the Internet, and social media and video streaming use up a large part of the bandwidth we use daily.

However, how we define an Internet connection has not changed much since the ARPANET. Our devices still use a point-to-point connection to our Internet Service Provider (ISP), which provides us with a single IPv4 address to use. Although IPv6 is gaining a foothold, especially in mobile networks, only a few services on the Internet are accessible with both IPv4 and IPv6 [1, 2]. At the same time, while broadband technologies have evolved into much faster connections, what remains the same as the traditional connections is the nature of the link; a single line to the ISP.

Our modern devices typically have at least two or more ways to connect to the Internet. All of our laptops, tablets, and smartphones support Wi-Fi; while the cellular network is supported by phones and laptops, at least via tethering or USB modem dongles. Ethernet is supported by laptops, and many phones and tablets can actually use USB Ethernet dongles. If we look at how connectivity has changed over the years, wireless networks have started to surpass wired networks. Both Wi-Fi and cellular have become faster and there are more wireless devices than ever before, including Internet-of-Things devices [3]. But, due to various reasons, we cannot use available networks together, *i.e.*, we are bound to the single connection paradigm instead of being able to combine them.

1.1 Motivation

Internet usage has grown exponentially over the years, and with the higher speeds of 5G and beyond, the broadband speeds are expected to increase even more [4]. At the same time, the Internet and transport protocols have remained mostly the same as they have been for decades. We use cable and DSL modems, Wi-Fi, Ethernet, and cellular broadband for connectivity. Even though the available bandwidth and the latency of especially cellular broadband have advanced dramatically, the way we use them remains the same. Almost all of our network traffic uses a single network interface connected to one of the above technologies as before. We do not aggregate or use other available interfaces for backup connections, instead, our devices only switch between different interfaces depending on the connectivity, and end up breaking the existing connections.

For transport protocols, even though there are newer protocols such as QUIC [5], we still mostly use both Transmission Control Protocol (TCP) [6] and User Datagram Protocol (UDP) [7] as we have done since the 1980s. The changes in the transport protocols are essentially optimizations to the existing protocols, such as better congestion control, but very few new protocols have been deployed on the Internet. For example, while multiple new protocols have been introduced, such as Stream Control Transmission Protocol (SCTP) or Datagram Congestion Control Protocol (DCCP), or a multipath variant of TCP called Multipath TCP (MPTCP), most of these protocols have not been taken into use on a large scale. Instead, some of them, such as MPTCP, are used in controlled environments or not at all [8].

There are several factors why the current status quo of Internet connectivity has remained the same. The Internet is primarily operated with equipment that has a long life span. These routers and switches can operate for years without significant upgrades, or with upgraded modules, such as line cards. This comes with the cost of ossifying the network as the older equipment may not be capable of handling new transport protocols.

This is especially true with consumer Internet connections. In most cases, the ISP provide their consumers with home gateways such as broadband or cable modems. These devices connect the home networks to the ISP networks and remain largely untouched for years, *i.e.*, their capabilities remain the same as they were when first installed [9, 10].

To summarize, Internet connectivity has remained the same for decades, with only small steps taken to break free from the way connectivity has always been. This serves as the motivation to examine why things are currently as they are, and how the knowledge claimed could be used to bring better connectivity to users. The above factors serve as the motivation for this thesis.

1.2 Problem Statement and Methodology

The research questions can be divided into two categories that form the topic of the thesis, seamless programmable multiconnectivity. These questions delve into the current state of network connectivity and how we can extend the state-of-the-art methods to solve them.

The first set of questions explores how Internet connectivity works from the consumer point of view and what problems these home gateway devices bring. They also explore how these devices could be used in future networks, as they remain usable well beyond their expected life span.

RQ1 How do middleboxes hinder connectivity?

RQ2 How can we make contemporary non-programmable hardware evolvable through software extensions?

The first research question, RQ1, outlines one of the big problems of the current Internet: ossification. Over the years, there has been significant anecdotal and observed evidence that middleboxes such as the home gateways, and especially how home gateways perform Network Address Translation (NAT), are not well-defined and in some cases broken [11, 12]. We, therefore, explore what the NAT characteristics of the home gateways are and try to find out common features that would allow us to design new protocols.

The second question, RQ2 delves into how we can evolve existing network devices for future networks. Many of these devices have an expected life span of years. In many cases, updating them to the newest, most advanced equipment is cost-prohibitive, especially in large installations. As they fulfil the traditional network requirements, replacing them before their expected end-of-life date is not cost-effective. However, many of these devices could still be updated through other means than replacing them to have similar capabilities as newer equipment. We, therefore, explore how we can bring SDN to these devices without major modifications or customization.

The second category of research questions explores the topic of multiconnectivity.

RQ3 How can we offer multiconnectivity in a programmable network?

RQ4 How can we achieve user-driven multiconnectivity over arbitrary networks?

The third research question, RQ3, explores how we can use SDN to offer multiconnectivity in a programmable network. Traditionally connectivity is driven by the host device's network stack, which makes decisions based on pre-existing rules and heuristics on which network interfaces to use. While this usually allows the host device to be connected to the Internet, it is often not optimal. For example, a mobile network could offer better connectivity than a congested Wi-Fi network, but Wi-Fi is chosen as the predetermined rules give higher priority to Wi-Fi and do not take the network state into account. Similarly, switching between network interfaces causes existing connections to break, and recovering from those is left to the applications.

Research Paper	RQ1	RQ2	RQ3	RQ4
1. An Experimental study of Home Gateway Characteristics	✓			
2. SWIFT: Bringing SDN-Based Flow Management to Commodity Wi-Fi Access Points		✓		
3. Orchestrating Intelligent Network Operations in Programmable Networks		✓		
4. An SDN Perspective on Multiconnectivity and Seamless Flow Migration			✓	
5. Programmable Session Layer MULTI-Connectivity				✓

Table 1.1: **The research questions addressed in this thesis**. *For each question, we indicate which publication addresses it.*

The host device's network stack contains limited information of the networks. While some network characteristics can be gleaned through passive monitoring or probing, the network stack does not have the full view of the network. Here, the paradigm of programmable networks offer new venues for how networks and their controllers can be leveraged to perform multiconnectivity. With the network controller's information, the multiconnectivity decisions can be made with better knowledge.

The fourth research question, RQ4, explores multiconnectivity in heterogeneous networks. Unlike the networks that RQ3 targets, *i.e.*, networks that support multiconnectivity, the heterogeneous networks do not have a controller with whom the host device can communicate. Most networks fall into this category. As such, finding solutions to multiconnectivity in those networks remains an important topic.

1.3 Thesis Contributions

The research contributing to this thesis is described in the five included publications, which form the basis of the thesis. In the following, we show the details of the publications, highlighting the specific contributions of the author. Table 1.1 summarizes which publications addresses particular research questions.

Publication I: An Experimental Study of Home Gateway Characteristics. S. Hätönen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti, and M. Kojo. Published in the Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10), pages 260-265, Melbourne, Australia, November 1-3, 2010.

***Contribution**: Aki Nyrhinen and the author developed the Home Gateway testbed and were responsible for developing the tests, analysing, and reporting the results. All authors contributed to the test designs and were involved in all phases of the work.*

Publication II: SWIFT: Bringing SDN-Based Flow Management to Commodity Wi-Fi Access Points. S. Hätönen, P. Savolainen, A. Rao, H. Flinck, and S. Tarkoma. Published in the Proceedings of the 2018 IFIP Networking Conference (IFIP Networking) and Workshops, pages 1-9, Zurich, Switzerland, May 14-16, 2018.
***Contribution**: The author was in charge of designing and implementing the SWIFT system, gathering the evaluation data and writing the paper. The ideas behind the system were extensively discussed with all of the authors.*

Publication III: Orchestrating Intelligent Network Operations in Programmable Networks. S. Hätönen, I. Hafeez, J. Mineraud A. Rao, and S. Tarkoma. Published in the Proceedings of the IEEE Conference on Standards for Communications and Networking (CSCN'21), pages 148-154, Thessaloniki, Greece, December 15-17, 2021.
***Contribution**: The author developed both the SDN controller and the management bus of the modularized system. Julien Mineraud and the author developed the PraNA orchestrator together. The author was also in charge of evaluating the system. Ashwin Rao and the author were in charge of writing the paper, while all other authors provided feedback and text in all phases of the work.*

Publication IV: An SDN Perspective on Multi-connectivity and Seamless Flow Migration. S. Hätönen, T. Huque, A. Rao, G. Jourjon, V. Gramoli, and S. Tarkoma. Published in IEEE Networking Letters (Volume: 2, Issue: 1, pages 19-22, March 2020).
***Contribution**: The author was in charge of building the physical migration testbed and implementing the Meghna system with Ashwin Rao. The author was also in charge of writing, while all other authors provided both feedback and ideas behind the system in all the phases of the work.*

Publication V: Programmable Session Layer MULTI-Connectivity. S. Hätönen, A. Rao, and S. Tarkoma. Published in IEEE Access, vol. 10, pages 5736-5752, 2022.
***Contribution**: The author was in charge of writing the paper. Ashwin Rao and the author both developed MULTI, and the author was in charge*

of evaluating MULTI in a physical test environment. All authors were involved in writing the paper and contributed ideas.

1.4 Thesis Structure

The thesis is organized as follows. Chapter 2 provides the background knowledge to this thesis and discusses various reasons why certain aspects of the Internet are as they are today. We also take a look into **RQ1** in this chapter and present our contributions that address **RQ1**. Chapter 3 concentrates on **RQ2**, and explores how different legacy devices can be used in future networks. To achieve this, we present the SWIFT system to bring Software-Defined Networking to off-the-shelf legacy Wi-Fi devices in Section 3.1. We also present the PraNA framework for programmable network orchestration using off-the-shelf controllers in Section 3.2. Chapter 4 discusses multiconnectivity in general and presents two different ways to tackle the problems inherent in the multiconnectivity, namely Meghna in Section 4.1 and MULTI in Section 4.2, and attempt to answer **RQ3** and **RQ4.** Finally, Chapter 5 concludes the thesis by revisiting the research questions and summarising the contributions of the thesis and discusses future work.

Chapter 2

Background

Many of the challenges for seamless multiconnectivity stem from past design decisions on how networks work. When these decisions were made, the connected world was very different, and technological advances like mobile smartphones were decades in the future.

In this chapter, we present the background to the topics in this thesis. We start by describing what middleboxes are and how they affect Internet connectivity. We focus on the home gateways, *i.e.*, the devices that connect our home networks to the Internet. After discussing the middleboxes, we describe what programmable networks are, and how they work. We then touch on multiconnectivity by briefly describing different methods to achieve both multihoming and multipathing. These methods form the basis of multiconnectivity, as they allow us to break away from the traditional way we connect to the Internet. Finally, we present our Publication I, a study on home gateway characteristics and how they have affected the transport protocol designs over the years.

2.1 Middleboxes

Originally what we now call the Internet was a much simpler network. There were only a fraction of hosts connected to the network, and they could all communicate directly with each other using the End-to-End principle without having any devices in the network that would break the principle [13]. Many contemporary transport protocols were designed with this principle in mind in the 1980s, *i.e.*, the connections are between two hosts with fixed network addresses. Protocols such as Transmission Control Protocol (TCP) [6] and User Datagram Protocol (UDP) [7] use the End-to-End principle and are still in use in everyday Internet traffic. Both protocols

have seen modifications, but the basic operation principle is still the same, transfer data from one fixed network location to another network location.

The transport layer connections or datagram flows are defined as five-tuples, consisting of a source IP address and a protocol port number, a destination IP address and protocol port, and the protocol in use. While this five-tuple is a straightforward way to define a connection, it has some serious limitations. Over the years, the Internet has seen an influx of devices that do more than forward packets towards their destination. These middleboxes can alter the packet headers traversing them, redirect the flows to different destinations depending on their function, or drop them. Nonetheless, the middleboxes obfuscate the actual source and destination of the flows and thus break the End-to-End principle.

Next, we detail several types of the most widely used types of middleboxes that are relevant to this thesis.

2.1.1 Network Address Translation

Network Address Translation (NAT) allows whole private networks to be hosted behind a single public IPv4 address. The basic principle of NAT is simple; when a packet traverses a NAT device, either the source or destination IP address and port number is mapped either to the public IP address of the NAT device when the packet is traversing from the private network to the outside, or destination IP address is mapped to the destination IP address inside the private network. The NAT was originally proposed in 1993 [14], and standardized in 1994 by the Internet Engineering Task Force (IETF) in Request For Comments (RFC) 1631 and updated later [15, 16]. The NAT defines how a middlebox could translate a network address into another address and maintain a mapping between the addresses. The NAT was designed to alleviate the already foreseen IPv4 address space exhaustion, which eventually happened in 2017 when AFRINIC ran out of available general use /8 address blocks [17].

The NAT principle is illustrated in Figure 2.1. Here, the private network using 192.168.0.0/24 address space is connected to the Internet through a public IPv4 address 1.2.3.4. Now, when a client, with an IP address 192.168.0.2, connects to a destination at 8.8.8.8, the gateway performs the source address translation from 192.168.0.2 to 1.2.3.4, and adds this translation in the NAT mapping table. When a reply comes from 8.8.8.8 to 1.2.3.4, the gateway makes a translation lookup and finds that 192.168.0.2 had connected to 8.8.8.8 and replaces the destination address with 192.168.0.2.

This one-to-one NAT is not sufficient when there are multiple hosts behind the NAT. Normally, the NAT also uses the transport protocol and

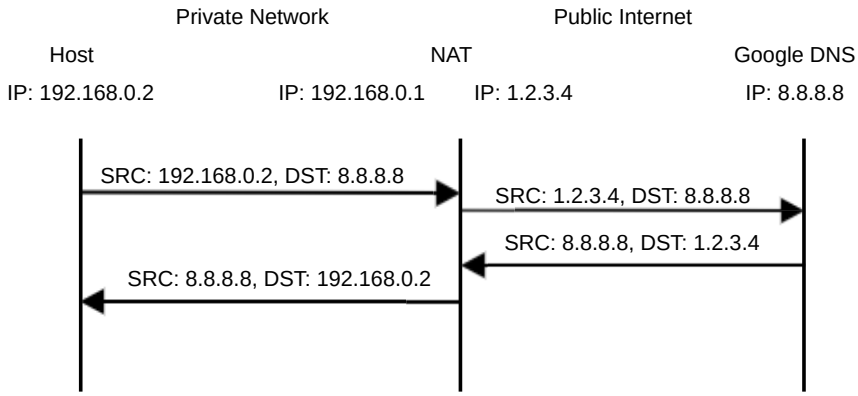


Figure 2.1: **Network Address Translation.** *The graph shows how a NAT device changes different fields of a packet when it traverses a NAT middlebox.*

protocol port in addition to the source and destination IP addresses to perform the translation. With this, the source port is also translated into an unused port by the gateway, allowing multiple hosts to connect to the same destination address and port.

In addition to extending the IPv4 address space, the NAT is sometimes seen as a security measure. From the outside of the NAT, none of the hosts behind the NAT are visible to the outside. This can be seen as a cheap firewall solution, although it is not a firewall.

However, the NAT process was not specified properly partly because NAT was only intended as a stop-gap measure before IPv6 would solve the IPv4 address shortage with a much larger address space and other features. IPv6 would have allowed the End-to-End principle to be used once more and allowed the Internet to have a clean design.

Instead, the Internet continued to grow explosively, and it was much cheaper for network device vendors and operators to embrace the NAT. As of 2021, IPv6 has only reached a portion of all Internet traffic, almost three decades after the NAT was standardized.

In Section 2.5, we detail the results of a study on home gateway characteristics, which examined multiple different NAT implementations. These results provide tangible proof of how varied different NAT implementations are over a large set of devices from different manufacturers. The results help us understand why Internet connectivity is still similar to what it was in the 1990s and what measures protocol designers need to take to ensure that their protocols work.

2.1.2 Firewalls

Another example of middleboxes is firewalls. Firewalls are security devices that monitor and control network traffic traversing them. The main goal of a firewall is to protect the hosts behind the firewall from outside by only allowing traffic that has been approved to pass through them. This is done through a set of rules, against which all traffic passing the firewall is matched. If there is a rule denying the traffic, the traffic is blocked.

A simple firewall will block all incoming traffic from the public network to the private network while allowing the traffic from the private network to the public. The firewalls carry a state that tracks outgoing connections so that returning traffic can pass the firewall. The firewalls can filter out outgoing traffic to specific ports that are deemed to either carry security risks or are used in denial of service attacks. For example, port 25 used by mail servers is typically blocked as it is used for sending spam emails. More complicated firewalls will have a larger, more sophisticated rule set that can allow different operations to be performed on the traffic.

2.1.3 Load Balancers and Reverse Proxies

Load balancers are a category of the middleboxes that distribute the incoming load to multiple destinations [18, 19]. A single server can become congested if there are more concurrent users at a given time than the server can handle. For example, a website served by a single server will reach the limit of how many requests it can serve when its popularity increases. Using a more powerful server would solve the issue for a while; however, this does not scale as powerful servers are expensive and will reach their maximum capacity at some point.

The load balancers alleviate this problem by distributing the load to multiple servers. This balancing can be done either based on the network or transport layer protocols or at the application layer based on the protocol headers. Regardless of the protocols, the balancing of incoming connections can be done using different methods. Common methods include Round-Robin, least connections, and least response time.

2.2 Programmable Networks

Programmable networks are networks whose network operations can be managed through software-defined rules. Traditional networks are controlled through command-line interfaces or employ routing protocols. Still, the scope of the control is closer to the individual switches instead of cen-

tralized control. For larger routed networks, different routing protocols decide how the traffic is forwarded. However, the routing protocols are slow to react to changes, and controlling them is not straightforward.

Software-Defined Networking (SDN) is a paradigm where the traditional network elements have been split into control and data planes [20, 21]. This split moves the forwarding decisions away from the network switches to a logically centralized network controller. Here, the switches no longer decide themselves how to forward the network traffic but ask the controller for instructions over a control protocol such as OpenFlow (OF) [21]. Using this logically centralized approach allows a more fine-grained approach to network traffic management. The SDN controllers can achieve a full view of the network state and affect changes on how the network forwards traffic without individually configuring each switch.

While there are different types of networks that can be called software-defined, in this thesis, we use the term SDN to mean networks that use OpenFlow for the control.

2.2.1 Programmable Switches

For decades, the network switches have supported configuration over different methods. These switches contain both the control and the data planes, allowing them to forward traffic independently but making the configuration unwieldy and prone to mistakes.

The Software-Defined Networking removes the control plane from the switches and moves the configuration elements to an SDN controller. This allows switches to be relatively simple as they do not need the control plane.

In SDN, switches and controllers use Northbound and Southbound APIs to perform the SDN-based control. The SDN switches and controllers use the Southbound API to communicate with each other, while the Northbound is reserved for the communications between users and the servers. The typical Southbound API used in the context of this thesis is OpenFlow [21].

With OpenFlow, the switches report their state changes to the controller. Such state changes are, for example, changes in which switch ports are active. The switches also do not make autonomous decisions on how to forward the traffic. Instead, they use an OpenFlow *match-action* rule set, against which the traffic entering the switch is matched. If the traffic matches a rule in the ruleset, the appropriate action defined by the rule is taken to process the traffic. If the traffic matches no rules, the switch sends a request to the SDN controller on how to handle the traffic and does not forward the traffic before receiving an answer.

The OF also allows the SDN switches to change headers in the network packets. For example, the SDN controller could perform the NAT function described in Section 2.1.1, or remove and add new header fields.

2.2.2 Programmable Network Controllers

The second part of the SDN are the SDN controllers. In short, the SDN controller act a as the controlling intelligence of the network, and communicates with the SDN switches using southbound APIs like OpenFlow [21]. The SDN controllers control and manage the traffic passing SDN switches by installing the *match-action* rules either proactively or when a switch makes a request to the controller about how a packet should be treated. A simple SDN controller might only install forwarding rules based on the topology of the network, but more advanced controllers can use more intelligent approaches, and also perform other functions.

Most controllers also expose the network state to the outside using a northbound API, for example, a REST API. As they have the full view of the network, they can optimize the traffic forwarding, or forward it through third party services such as traffic analysis systems like Snort [22].

2.3 Multihoming and Multipathing

Traditional network connections between hosts are single path connections that are defined by the 5-tuple. This ties the connections to particular network interfaces, as used IP address defines which interface is used. However, if multiple addresses and interfaces are used, the hosts become multihomed and can use multiple paths between them.

In general, multiconnectivity is traditionally separated into two categories, namely multihoming and multipathing. A host is multihomed if it is connected to the Internet through more than one network interface and IP address. For example, a multihomed server could be connected to two or more Internet Service Providers (ISP) and would have multiple IP addresses from both of the ISPs.

Multipathing, on the other hand, means using multiple paths between hosts. This usually requires one or both of the hosts to be multihomed, but in some cases network topology and routing rules can also achieve multipathing through the network.

When a multipath protocol is used, usually, one available path between the hosts is chosen as the primary path. This path is used to initiate the connection, and when the connection is made, other possible paths are probed and opened between the hosts. After the paths are open, the

multipath schedulers on the hosts decide how to use the available paths and send packets accordingly. One prominent example of a multipath transport protocol is Multipath TCP (MPTCP) [23]. The MPTCP is implemented over the regular TCP with special MPTCP options. These options carry Connection IDs and other information required to exchange and manage the multipath connections.

2.4 Internet of Things

Over the last decade, we have seen an influx of small, Internet-connected devices in our homes and offices. These devices range from small sensors such as temperature sensors, small appliances like remote-controlled power switches and smart fridges, to larger devices like smart TVs, tablets, and to an extent, laptops and computers. Common to all of them is that they are all connected to the Internet, either directly over the local network or through specialized hubs and gateways. These devices are commonly known as Internet-of-Things (IoT) devices [24, 25, 26].

In general, a typical IoT device has only a few functions. For example, sensors, which are one of the most common types of IoT devices, measure their values and publish them either through their respective IoT hubs or through a cloud service, accessible over the Internet [27]. The devices use either low-power and short-range communication technologies such as ZigBee [28], Z-Wave [29], and Bluetooth [30], Wi-Fi, or even Ethernet to communicate with the Internet or their hubs.

There are also vendor-specific ecosystems built around the IoT devices known as silos [31, 32]. For example, Philips Hue's ecosystem includes devices ranging from lights and power switches to window blind controls and sensors. These ecosystems can sometimes also be tied with other vendor systems such as Logitech Harmony or Home Assistant [33], which allow the building of larger home automation systems.

The IoT devices are typically designed to be as simple as possible to reduce the manufacturing costs [3]. This drives the manufacturers to use as cost-effective parts as possible. This limits the computational capacity of the IoT devices, and, depending on the manufacturer, the amount of work done on securing the IoT devices [34, 35].

2.5 Characteristics of Home Gateways

Over the years, there has been a swath of both anecdotal and observed evidence that home gateways, and in particular, their NAT function does

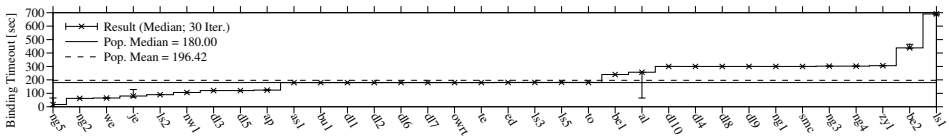
not function well [11, 12]. In many cases, the NATs drop packets that they do not understand, such as new transport protocols, translate them improperly, or show other unexpected behaviours. These issues are largely due to NAT originally not being defined well, and the home gateways had been largely ignored by standardizing bodies like IETF for years. Only in around 2009 were specifications on how the home gateways should behave published, over a decade after NAT was introduced in 1993 [36, 37, 38].

In Publication I, we performed an extensive study of 34 home gateways and their characteristics [9]. The publication was later extended into two technical reports for both Layers 3 and 4 [39, 40]. In Publication I, we explored how some of the most widely used protocols like TCP, UDP, and Internet Control Message Protocol (ICMP) were treated by the gateways and how long the gateways kept the NAT mapping entries in the mapping tables before removing them. Our findings confirmed how poorly the NATs behaved. In most cases, it was difficult to find two NAT devices with the same characteristics. This was not vendor-independent as devices from the same vendor had different characteristics, and in one case, three different firmware versions for a single device had different characteristics.

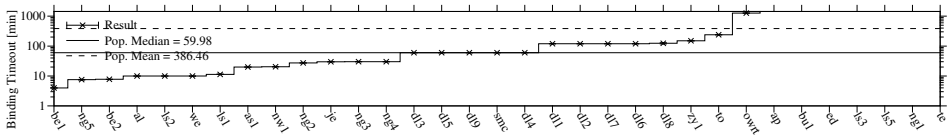
When we compared the characteristics against the NAT behavioural requirement RFCs for UDP [36] and for TCP [37], we found out that in many cases, the gateways manufactured after the publication of the RFCs did not follow them. One of the key discoveries was how varied the NAT binding timeouts were. The timeouts manage how long the NAT devices keep translation entries in their translation tables, and when the timeout expires, that particular binding is removed and cannot be used again. This kind of behaviour is particularly important to those connections that can be dormant for long time and then see activity. For example, many IoT sensors only publish their data periodically, and if the binding has expired between transmissions, a new data connection has to be established every time. While outgoing connection establishment is straightforward, no connection can be established to the device from the outside through the NAT.

In Figure 2.2, we show the results for NAT binding timeouts for both UDP and TCP. For unicast UDP, the IETF requires that NAT bindings must not expire in less than 120 seconds and recommends 300 seconds or longer. Here, we observe that many NAT devices use much shorter binding expiration values, and only some use the recommended value or larger, with the median being 180 seconds.

Similarly, IETF requires established TCP bindings to last at least two hours and four minutes due to the requirements for Internet hosts defined in [41]. We observed that more than half of the devices tested used timeouts



(a) UDP timeouts.



(b) TCP timeouts.

Figure 2.2: **NAT binding timeouts.** *These results show that NAT devices use different timeouts depending on the vendor. Device models shown at the X-axis can be found in Appendix A. When the binding timeout is reached, the NAT closes the binding and incoming traffic cannot be mapped to its destination. These figures were taken from Publication I.*

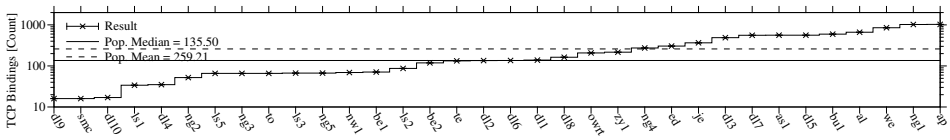


Figure 2.3: **TCP Connections.** *The maximum number of concurrent TCP connections. If the maximum number of connections is reached, the NAT must either drop an existing mapping, or it cannot create a new mapping. This figure was taken from Publication I.*

that were less than the IETF requirement, and in one case, one device used less than four minutes before the binding expired.

While the NAT binding characteristics may not be critical for typical Internet usage, *i.e.* for short-lived connections like web browsing, or for active connections like video streaming and gaming, they can cause problems for multiconnectivity. If a multiconnectivity solution does not use any keepalives or take other measures like using identifiers in the protocol payload like QUIC [42], the connections that are used for backup connections, *i.e.*, connections that are mainly dormant, may be closed by the NAT devices. If this is not detected, it may cause problems when the connection is assumed to be open and instead it is closed.

In Figure 2.3 we show our results for the maximum number of simultaneous TCP connections. While many of the devices supported over a hundred simultaneous connections, some devices supported less than thirty

zy1	we	to	te	smc	owrt	nw1	ng5	ng4	ng3	ng2	ng1	ls5	ls3	ls2	ls1	je	ed	dl9	dl8	dl7	dl6	dl5	dl4	dl3	dl2	dl10	dl11	bu1	be2	be1	as1	ab	al	Tag
																																		DCCP: Conn.
	•	•			•			•	•							•	•			•	•				•			•				•	•	DNS over TCP
•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	DNS over UDP
•	•	•	•		•		•	•	•	•	•	•	•	•	•	•	•		•	•	•	•		•	•		•	•			•	•	•	ICMP: Host Unreach.
	•	•	•		•							•	•			•	•		•	•	•				•		•	•			•	•	•	SCTP: Conn.
•	•	•	•		•			•	•	•	•	•	•			•	•		•	•	•	•		•	•		•	•			•	•	•	TCP: Reass. Time. Ex.
•	•	•	•		•			•	•	•	•	•	•		•	•	•		•	•	•	•		•	•		•	•			•	•	•	TCP: Frag. Needed
•	•	•	•		•			•	•	•	•	•	•			•	•		•	•	•	•		•	•		•	•			•	•	•	TCP: Param. Prob.
•	•	•	•		•			•	•	•	•	•	•			•	•		•	•	•	•		•	•		•	•			•	•	•	TCP: Src. Route Fail.
•	•	•	•		•			•	•	•	•	•	•			•	•		•	•	•	•		•	•		•	•			•	•	•	TCP: Source Quench
•	•	•	•	•	•			•	•	•	•	•	•			•	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•	TCP: TTL Exceeded
•	•	•	•		•			•	•	•	•	•	•			•	•		•	•	•	•		•	•		•	•			•	•	•	TCP: Host Unreach.
•	•	•	•		•			•	•	•	•	•	•			•	•		•	•	•	•		•	•		•	•			•	•	•	TCP: Net Unreach.
•	•	•	•	•	•			•	•	•	•	•	•			•	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•	TCP: Port Unreach.
•	•	•	•		•			•	•	•	•	•	•			•	•		•	•	•	•		•	•		•	•			•	•	•	TCP: Proto. Unreach.
•	•	•	•		•		•	•	•	•	•	•	•	•	•	•	•		•	•	•	•		•	•		•	•	•	•	•	•	•	UDP: Reass. Time Ex.
•	•	•	•		•			•	•	•	•	•	•	•	•	•	•		•	•	•	•		•	•		•	•	•	•	•	•	•	UDP: Frag. Needed
•	•	•	•		•			•	•	•	•	•	•		•	•	•		•	•	•	•		•	•		•	•			•	•	•	UDP: Param. Prob.
•	•	•	•		•		•	•	•	•	•	•	•	•	•	•	•		•	•	•	•		•	•		•	•	•	•	•	•	•	UDP: Src. Route Fail
•	•	•	•		•			•	•	•	•	•	•	•	•	•	•		•	•	•	•		•	•		•	•			•	•	•	UDP: Source Quench
•	•	•	•	•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	UDP: TTL Exceeded
•	•	•	•		•		•	•	•	•	•	•	•	•	•	•	•		•	•	•	•		•	•		•	•	•	•	•	•	•	UDP: Host Unreach.
•	•	•	•		•		•	•	•	•	•	•	•	•	•	•	•		•	•	•	•		•	•		•	•	•	•	•	•	•	UDP: Net Unreach.
•	•	•	•	•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	UDP: Port Unreach.
•	•	•	•		•		•	•	•	•	•	•	•	•	•	•	•		•	•	•	•		•	•		•	•	•	•	•	•	•	UDP: Proto. Unreach.

Table 2.1: **DCCP, SCTP and ICMP results.** *The table shows if the NAT devices support protocols like DCCP and SCTP, and if different ICMP messages are properly translated. This table was taken from Publication I.*

connections. In earlier days, when the websites were simpler, the maximum number of connections was not a problem. However, these days, many websites consist of hundreds of elements. If the maximum number of concurrent connections is reached, it will cause problems for browsers when fetching the web pages, leading to either delays as multiple tries are needed to fetch all elements or partial web pages.

Table 2.1 shows if different NAT devices translated ICMP messages correctly and if other protocols like Datagram Congestion Control Protocol (DCCP) or Stream Control Transmission Protocol (SCTP) work. None of the devices supported DCCP, and for the SCTP, we were unable to determine if the devices that allowed SCTP connections to be established really supported SCTP or just translated the IP headers.

Similarly, the table shows how the NAT devices translated ICMP messages traversing the devices. While most of the devices translated some ICMP messages properly like *TTL Exceeded* and *Port Unreachable*, many devices do not translate other messages. The worst device did not translate *any* of the ICMP messages, and one device generated TCP resets for all

TCP-related ICMP messages. In addition to these, roughly half of the devices did not translate transport headers inside the ICMP messages. These results show that relying on external protocols like ICMP for messaging network state is not reliable as many of the messages can be lost.

We also studied different TCP options were treated by home gateways [40]. Our findings were that the NAT gateways either dropped the TCP packets with lesser used or unknown TCP options, forwarded them, or even added their own options. These findings suggest that even extending well-known protocols is hard with NAT devices. Similar findings have been reported in other publications [10, 43]. As such, IPv4 NAT devices contribute hugely to the ossification of the Internet. In most cases creating a new transport protocol is practically impossible as the NAT gateways will not work with them. This means that the only truly viable path left for protocol designers is to create protocols on top of already existing protocols like UDP so that the NAT devices only see well-known protocols. For example, QUIC [5] has taken this path. It is implemented completely over UDP and carries its own headers inside the UDP payload.

While the NAT gateways typically behaved with IPv4 and basic TCP or UDP, *i.e.*, all devices could be used to connect to the Internet, the same could not be said about other protocols. In most cases, protocols such as DCCP, SCTP, and similar did not work at all; the NAT devices either did not forward the packets or did some address translation for the IPv4 headers but not for the transport headers. Our conclusion was that as no devices supported the less-known protocols properly, they cannot be relied on to work properly on the Internet, and application developers need to aim for the lowest common denominator, the basic TCP or UDP.

2.6 Summary

In this chapter, we have introduced several concepts and technologies that lay the groundwork for this thesis. The middleboxes discussed here have greatly influenced how the Internet currently works and have an effect on how transport protocols need to be designed in the modern world. In many cases, the middleboxes hinder the protocol designs as they are not originally designed to work with those protocols. In Publication I, we presented our study on home gateway characteristics and discussed how our measurements could be used to influence the protocol designs.

In addition to middleboxes, we also explore programmable networks and their principles. We also introduce the basic concepts of multiconnectivity, namely multipath and multihoming. These concepts will be touched upon

in all parts of the thesis, and the insights from the gateway study allow us to answer the research questions in Chapters 3 and 4.

Chapter 3

Current Devices in Future Networks

The middleboxes discussed in Chapter 2 and other devices like Wi-Fi APs are usually designed with a well-defined purpose. However, these can be versatile devices, especially if they support the installation of custom firmware or new software modules. In this chapter, we show how some of these older APs and middleboxes can be extended to work in future networks and discuss what they can achieve. One tangible way that they can be used is to deploy SDN on them. Using the existing devices can, for example, cut deployment costs as they would not need to be replaced.

In this chapter, we focus on SDN solutions that are based on OpenFlow (OF) [21]. OpenFlow is one of the implementations of programmatic networks. Depending on the definition, for example, Cisco's Wireless LAN Controller (WLC) [44] can be defined as software-defined, as the networks defined through the WLC are automatically provisioned over the APs, and the WLC can automatically handle channel assignments. In this thesis, we limit the term SDN to mean OF-based software-defined networks.

SDN has seen deployments in almost all main types of networks that we encounter. In the Wide Area Networks (WAN), Google introduced their B4 solution in 2010 [45]. Similarly, many vendors such as Cisco, HP, and Dell have introduced SDN in their Ethernet switches over the years. However, one significant area where SDN has not gained traction is Wi-Fi. Over the years, there have been several attempts to bring SDN to Wi-Fi networks. The seminal work was done in OpenRoads [46]. Following the work done in OpenRoads, the BeHop [47], ÆtherFlow [48], and OpenSDWN [49] refined the approaches to the problem. However, all these approaches have limitations; most commonly, they require specific agents running on the APs. While these agents allow a more detailed view into the network and

more fine-grained control, in many cases, they make the usage of these approaches very hard in commodity networks. Another approach that some of these solutions took was to use multiple Service Set Identifiers (SSID). Each Wi-Fi network is identified by a name, which is provided as the SSID. In this case, each client associated with the AP is given its own virtual AP, usually realized through an SSID. While these allow the particular SDN controller to control the client traffic, the multiple SSIDs pollute the air with beacons and control frames that limit the usable bandwidth [50].

In this chapter, we focus on **RQ2**: how can we make contemporary, non-programmable hardware evolvable through software extensions? To answer this, we present SWIFT, a solution for bringing SDN-based flow management to the existing commodity access points. The SWIFT is designed to provide as straightforward as a possible way to deploy SDN on existing devices with minimal software changes. As such, SWIFT brings the benefits of a programmable network to networks that are already in existence and would be cost-prohibitive to replace with programmable hardware. We detail the SWIFT architecture and implementation in Section 3.1, with discussion of the possible use cases in Section 3.1.6. We also present PraNA, a framework for network orchestration that brings together different controllers, IoT devices, and SDN. We then present the SWIFT evaluation in Section 3.1.5 and finally conclude the chapter in Section 3.3.

3.1 SWIFT: Bringing SDN to Commodity Wi-Fi Access Points

In Publication II, we present SWIFT, our solution for bringing SDN-based flow management to commodity off-the-shelf Wi-Fi access points. The goal of SWIFT is not to fully manage the Wi-Fi network; instead, it focuses on traffic control. This work was presented as a demo in ACM SIGCOMM 2016, and the article describing it is in Appendix B [51].

We focused on the commodity APs as replacing existing hardware is costly for large deployments, and to the best of our knowledge, there are almost no OF-capable APs available. Similarly, another main goal was to use the commodity APs with minimal changes. The more specialized solutions like BeHop [47] and others used firmware that was customized to the particular APs used. This dependency may cause them to be undeployable in the future if newer AP models do not support those changes. Keeping this in mind, SWIFT was designed to use existing software as much as possible to keep the changes SWIFT required minimal.

SWIFT is not meant to be the Wi-Fi management system for the network; instead, it is envisioned to work alongside other dedicated platforms such as Cisco Wireless LAN Controllers (WLC) [44]. The WLC would handle the provisioning and management of the access points, while SWIFT controls the traffic flows traversing the APs.

The off-the-shelf APs can be divided into two main categories; namely, those which are supported by open firmware like *OpenWrt*, and those whose firmware is locked by their vendors and to support customization. To support both types of APs, we designed two approaches: Intelligent AP and Thin AP. The Intelligent AP is for those APs that allow us to modify the software of the AP directly. The Thin AP is meant for those APs that do not support custom firmware or software.

3.1.1 Client Isolation

The key obstacle for bringing SDN-based flow management to the Wi-Fi was that all Wi-Fi clients associated with the same AP could communicate freely with each other. In a normal wired Ethernet network, each client is connected to a fixed switch port. The switches in the network build their own local forwarding tables based on which ports they have seen different MAC addresses and forward packets accordingly. As such, each client has their unique physical location in the network, for example, a workstation on an office desk.

An SDN-based network operates in a similar manner, with the exception of the SDN controller keeping track of where each client is connected to the network instead of the switches. When a packet originating from a client enters a port on an SDN switch, the switch first checks if there are any matching SDN rules to the packet, and if there are none, the switch forwards the packet to the controller and requests instructions. This allows the controller to build a topology map of the network with the location of each connected client.

However, in Wi-Fi networks, multiple clients are connected to the same SSID, for example *eduroam*. In practice, each AP in the network broadcasts the SSID, and when the client connects to the SSID, the client associates with the nearest AP. The SSID is visible to the network stack as a wireless interface, behind which all associated clients are connected.

Typically, the Wi-Fi drivers used by the APs optimize traffic forwarding inside the wireless interface. If the destination of the traffic is not local, the drivers forward the traffic to the network stack of the AP [52]. However, if the destination of the traffic is a client associated with the same SSID and AP, the drivers forward the traffic directly between the clients, without

the traffic passing through the AP or an SDN switch. This means that the SDN switch and the controller never see the traffic, making it impossible for the SDN controller to manage and control the traffic between Wi-Fi clients.

While conducting our research, we noticed that many, if not all, access points support a feature called *client isolation*.[1] This feature blocks clients connected to the same SSID and AP from communicating with each other, thus providing a simple layer of security in the network.

When we examined the client isolation more closely, we found that there are at least three different variants of how it is implemented:

- *Permissive Isolation:* When the isolation is enabled, in Permissive Isolation, the AP sends all traffic from the clients to the AP's network stack but does not forward the traffic between clients.
- *Restrictive Isolation:* The AP sends only the ARP broadcasts to the network beyond the AP while discarding other traffic between clients.
- *Total Isolation:* All packets between the clients are discarded by the AP, including ARP.

In general, *OpenWrt*-based APs use *Permissive Isolation*, but enterprise APs may use any of the above implementations. While there is no way to know the isolation style beyond testing the behaviour, the manuals of the APs may give hints on the isolation implementation.

Both *Permissive Isolation* and *Restrictive Isolation* can be used to allow the SDN controllers to manage the traffic. However, their implementation details need to be taken into account when designing the SDN controller. When using *Permissive Isolation*, the Wi-Fi driver sends all traffic to the AP's network stack. The network stack compares the traffic to its ARP table, and if the destination is also associated with the AP, the network stack discards the packets as they would be sent back out from the network interface where they came from. This behaviour is prevalent in all network devices since if a packet is sent back out from the port it arrives at, a network loop is created, leading to a packet storm that can, and has brought down networks. Now, if there is an SDN switch running on the AP, the switch and the SDN controller can perform SDN actions to the packets and, for example, send them back to the interface where they came from. How this behaviour can be used is detailed in Section 3.1.2.

[1] Client isolation has different names between various vendors. For example, Cisco calls it Peer-to-Peer Blocking in their Wireless LAN Controller [44].

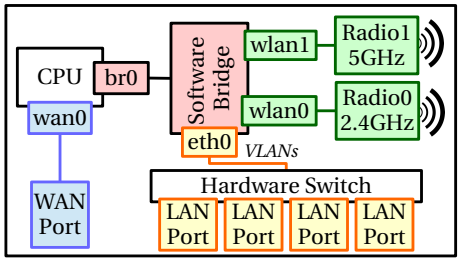


Figure 3.1: **Interfaces on an** *OpenWrt* **router**. *Regardless of the internal wiring, OpenWrt uses a* Software bridge *to manage the Wi-Fi network and the LAN. This figure was taken from Publication II.*

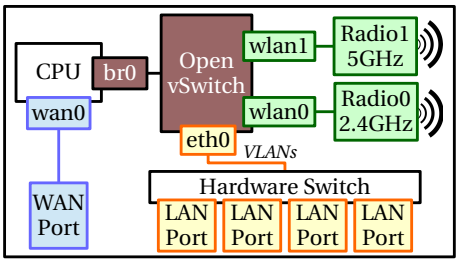


Figure 3.2: **AP configured for the Intelligent AP**. *OVS replaces the default bridge provided by OpenWrt, and Client Isolation is enabled on the AP. This allows the OVS to manage the flows traversing the AP. This figure was taken from Publication II.*

Restrictive Isolation on the other hand allows broadcasts to leave the AP while discarding other packets. This allows the SDN controller to detect the hosts behind the AP and take actions based on the network topology. We detail how this behaviour can be leveraged to bring SDN to those APs that do not allow custom firmware in Section 3.1.3.

Total Isolation prevents all traffic between Wi-Fi clients from reaching the network stack of the AP or beyond. If the AP uses Total Isolation, it cannot be used for SDN using the methods of SWIFT.

3.1.2 Intelligent AP

The Intelligent AP technique is aimed at those APs that allow installation of custom firmware or software and use Permissive Isolation. The typical examples of these are *OpenWrt*-based APs, of which a typical design is shown in Figure 3.1. Normally, the *OpenWrt*-based AP includes a System on a Chip (SoC), which has several Ethernet and Wi-Fi interfaces included on the chip. In addition to these, the APs usually have a small hardware switch with 802.1Q Virtual Local Area Network (VLAN) capabilities installed on the device [53]. The devices run a Linux bridge, which has all Wi-Fi interfaces and an Ethernet interface installed into it. The Ethernet interface (*eth0* in Figure 3.1) is connected to the hardware switch that allows several other network devices to be connected to the AP with cables.

In this technique, as illustrated in Figure 3.2, we replace the normal Linux Bridge in the AP with the Open vSwitch (OVS) [54]. We then plug

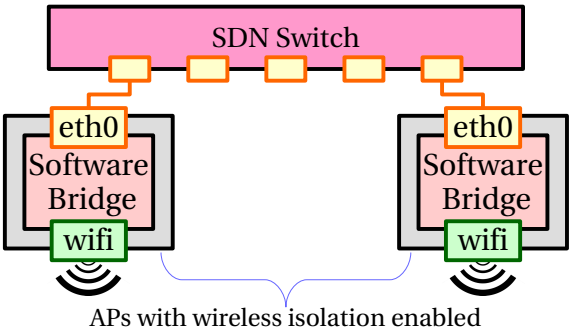


Figure 3.3: **APs configured for the Thin AP**. *The APs have Client Isolation enabled, and an external SDN switch manages the network traffic flows traversing the AP. This figure was taken from Publication II.*

all Wi-Fi interfaces that were plugged in the original Linux Bridge into OVS and enable the Permissive Isolation on the Wi-Fi interfaces. The Permissive Isolation then forwards all packets to the OVS, allowing the SWIFT SDN controller and the *OVS* to manage all traffic between Wi-Fi clients and the wired network.

To also allow the *OVS* to manage the traffic traversing the LAN interfaces, each LAN port is assigned a separate VLAN. These VLANs show as separate Ethernet interfaces on the OS and are also plugged into the *OVS*.

This technique also supports multiple Wi-Fi networks, *i.e.*, SSIDs on the AP. Each SSID appears as a logical Wi-Fi interface on the AP, which can be inserted into OVS. For the SWIFT SDN controller to manage the traffic properly, enough metadata on what SSID is connected to which OVS port needs to be provided to the controller. This requires either manual configuration or parsing OpenWrt configurations.[2]

3.1.3 Thin AP

The Thin AP approach is suitable for those APs that do not allow installation of custom firmware but support restrictive isolation. This approach is also suitable for those APs that have hardware limitations, such as too small a flash memory that does not allow installation of larger custom firmware or software packages.

How the Thin AP works is that the AP works as a remote interface for an SDN switch as illustrated in Figure 3.3. Each of the SSIDs configured

[2] All traffic with Ethertype 0x888e (EAP over LAN) needs to be forwarded to the AP to allow encrypted Wi-Fi to work.

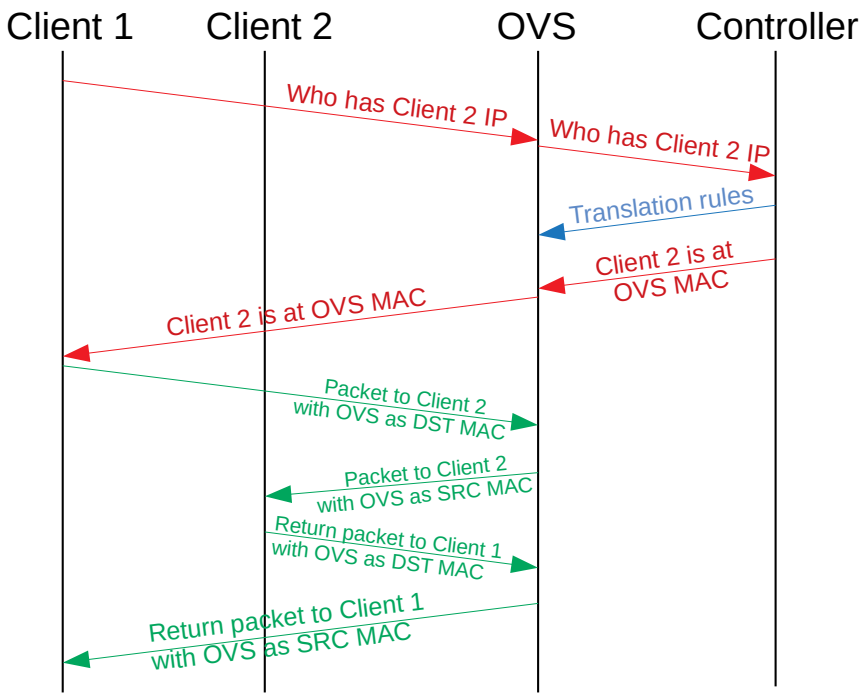


Figure 3.4: **SWIFT Thin AP sequence**. *When Client 1 needs to send a packet to Client 2, the following sequence of events occurs. Red denotes the ARP Requests and Responses, blue denotes the OpenFlow rules, and green the actual packet transfer.*

on the AP becomes a port on that switch through VLANs. In essence, the combination of the AP and the switch port create the Thin AP.

As discussed above, the key to this is how *restrictive* client isolation is implemented. The AP sends the Address Resolution Protocol (ARP) broadcasts to the wired network, *i.e.*, to the SDN switch and beyond. The controller keeps track of where each client is located and the client's MAC address. If two or more clients share the same SDN switch port that is defined to be a Thin AP, the controller knows that the clients are connected to the Wi-Fi.

After the controller has learned the client's MAC addresses, the controller can then manage the flows using the MAC address. We draw inspiration from Proxy ARP for Private VLANs (PVLAN), also known as VLAN Aggregation, to implement the Thin APs [55, 56]. The sequence of how Thin AP works is illustrated in Figure 3.4. In this technique, when the SDN controller sees an ARP request from a Wi-Fi client for another

Wi-Fi client's address, the SDN controller replies with the MAC address of the SDN switch.

Now, the first client has a MAC address that it uses to send the packets to the other client. When these packets arrive to the switch, the switch performs the MAC address translation by replacing the destination MAC address with the real destination MAC address and the source MAC address with its own MAC address. Now the destination client learns the SDN switch's MAC address as the MAC address of the original Wi-Fi client and uses it to send packets back to the original client. The controller adds relevant rules to perform this MAC address translation to the SDN switch. With these, the Wi-Fi clients think that they are communicating with each other while in reality, they are using the SDN switch as a relay.

3.1.4 SWIFT SDN Controller

The heart of SWIFT is its SDN controller. SWIFT is a set of techniques that can be implemented using any SDN controller. We now detail the main steps the SDN controller needs to take beyond the normal traffic management in the wired network also to support Wi-Fi.

Intelligent AP Support. As discussed above, the Intelligent AP is realized through an SDN switch running on the AP, on which each SSID is represented as a port in the switch. This information, *i.e.*, which SSID is on which port, needs to be available to the controller. When the controller detects traffic from this port, the hosts beyond the port need to be labelled as clients associated with the AP and the SSID. This information is then used to send relevant packets back to the port where they came from if the clients need to communicate with each other or perform other actions such as isolating any clients by dropping their traffic.

Thin AP Support. Thin APs require more actions to be taken by the controller than the Intelligent APs. In addition to the above requirements, the SDN controller also needs to track the ARP messages sent by the clients. The SDN controller needs to examine the ARP requests and keep track of the IP and MAC addresses of each of the clients. If the clients associated with the Thin AP want to communicate with each other, the SDN controller has to reply to the ARP requests with a pseudo-MAC address having that particular IP address. When the client communicates with the other client using the pseudo-MAC address, the SDN controller translates the pseudo-MAC address with the appropriate MAC address so that the clients think they are communicating with each other instead of using the SDN switch and the pseudo-MAC address as a relay.

Client Mobility. The clients associated with the APs may roam between APs, unlike clients in the wired network. To allow this, the SDN controller needs to track where the client is connected at any point in time and update the location and the SDN rules if the client roams to another AP.

Isolation Policies. The above steps give the SDN controllers the basic steps to control the traffic in the Wi-Fi network. In addition to these, to realize the programmatic client isolation, the SDN controller needs to implement isolation-specific policies. Each client needs to have a default network isolation policy assigned to it. In our example implementation, the policy can be *permitted*, *restricted*, *denied*, or *custom*. These levels allow SWIFT to support client isolation from everything allowed to everything denied or something in between. This is a minimal set of policies that allow us to demonstrate the capabilities of SWIFT.

When the clients need to communicate with each other, the controller needs to check the level of isolation policy each of the hosts is assigned. These policies are network-wide, *i.e.*, they affect all clients in the Wi-Fi regardless of the AP they are connected to. For example, a video-streaming device like a Chromecast could have a *custom* level, which would restrict its communications only to those hosts that are nearby, *i.e.*, in the same room and to the Internet. If the policy is *restricted*, the client is effectively isolated from the network.

Any SDN controller can be used to implement the above steps. We have aimed to keep the steps as simple as possible to provide a stepping stone for more advanced techniques. With these steps, and the AP techniques detailed in Section 3.1.2 and Section 3.1.3, many of the existing Wi-Fi networks can be converted to support SDN-based flow management.

3.1.5 SWIFT Evaluation

Replacing the original software bridge on the APs in the case of Intelligent APs, or forcing the traffic to traverse an SDN switch beyond the AP when using the Thin approach, may cause overheads in the packet forwarding and processing. To evaluate the overheads caused by our approaches, we designed a testbed that included both Intelligent and Thin APs, SWIFT controller, and multiple Wi-Fi clients. The full details are available in Publication II [57]. The testbed schema is presented in Figure 3.5. We used three different APs; Netgear WNDR-4300v1 (ng), TP-Link WR1043NDv2 (tp), and Cisco 1131ag (cisco), for our evaluation. Both the Netgear and the TP-Link are *OpenWrt*-based APs and thus support the three different

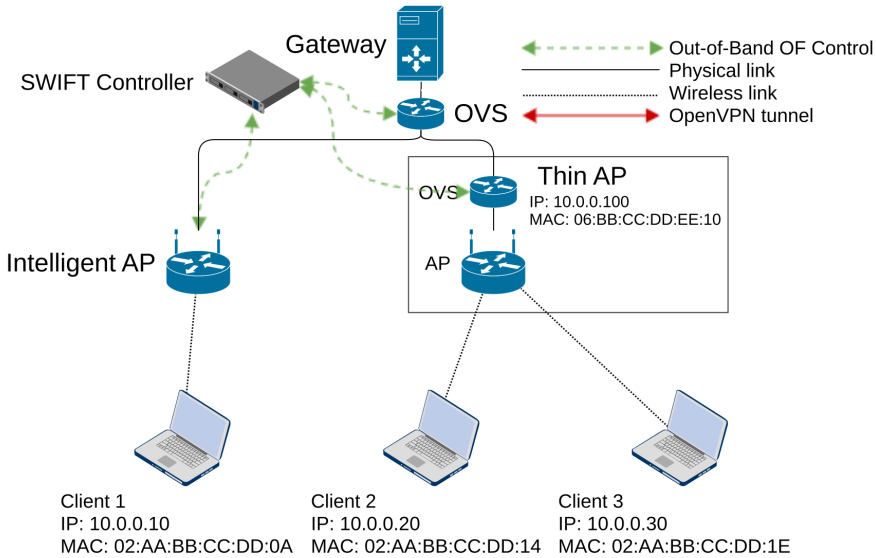


Figure 3.5: **Example SWIFT topology**. *The SWIFT controller manages the flows traversing the network. The Intelligent AP consists of an AP with OVS installed, while the Thin AP consists of an OVS switch and an AP without SDN support. This figure was taken from Publication II.*

configurations: Stock, Intelligent, and Thin APs. In Stock configuration, the APs used their stock *OpenWrt* firmware. In Intelligent configuration, we applied our Intelligent AP technique, and in Thin mode, the Thin AP. The Cisco is a representative of a device that does not allow firmware or software modifications, so it can only be configured as Stock or Thin AP. In addition to the APs, we use two identical laptops as clients to measure the changes our techniques cause.

The evaluation had several main goals. The first goal was to show that SWIFT could be operated as described in Section 3.1. To achieve this, we used 20 different devices connected to both Intelligent and Thin APs. As expected, the SWIFT controller was able to allow or deny clients from communicating with each other dynamically.

The second goal was to evaluate the overheads caused by the SWIFT approach. Both the Intelligent and Thin APs have different implementations, either of which adds overheads in different places in the system.

For the Intelligent APs, the addition of a software SDN switch, *i.e.*, the *OVS*, requires modifications to the AP firmware. The *OVS* replaces the traditional Linux bridge on the AP. While both operate in the kernel of

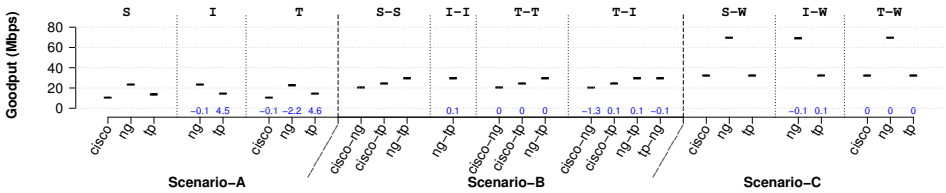


Figure 3.6: **Performance of SWIFT Techniques.** *The error bars represent 99% Confidence Intervals for the mean TCP goodput. S, I, T, and W denote Stock, Intelligent, Thin, and Wired configurations of the three APs. The figure was taken from Publication II [57].*

the operating system, the Linux bridge is a more mature implementation and may benefit from hardware accelerators. The *OVS*, on the other hand, operates without hardware support, which may limit its performance. With SWIFT, all traffic must traverse the *OVS* on the device and not be directly forwarded by the Wi-Fi driver of the AP.

For the Thin APs, the overheads come from two different places. The main reason for the overheads is that as there is no SDN switch on the AP, the traffic must first be forwarded to an external SDN switch. This brings two penalties. First, as with the Intelligent AP, the Thin AP no longer has the benefit of directly bridging clients that are connected to the AP as all traffic must pass through the external SDN switch. Second, the external SDN switch has several costs. The Round-Trip-Time (RTT) between the switch and the AP causes a slight additional increase in the total RTT. Similarly, the traffic between the clients congests the link between the AP and the switch.

To evaluate these, we used *Flent* to generate test traffic [58]. *Flent* is a tool specifically developed to expose buffers in the network and measure the effect of the *bufferbloat* [59]. *Flent* is designed to fill all buffers in the networking equipment, allowing us to gauge the effect of SWIFT on the performance of the APs.

The details of the evaluation are presented in Publication II [57]. In general, the effects of our modifications to the APs were negligible compared to when the AP had their stock configurations. Figure 3.6 highlights the effects of our Intelligent and Thin AP techniques. In the figure, we have plotted the changes in goodput of APs and clients in different combinations; for example, in *S-S* the *cisco-ng* pair denotes that both APs are in their stock configuration, and one of the client laptops is connected to the Cisco AP and the other to the Netgear. The numbers under the error bars show the percentage change compared to the relevant Stock configuration.

The testing highlighted that the performance of the APs mainly depends on their hardware. In some cases, especially with a slower AP, the Thin and Intelligent AP techniques slightly increased their throughput. This is mostly due to the lower hardware requirements of Thin AP, *i.e.*, the traffic can traverse directly from Wi-Fi to wired network and back, and more optimized queue management of the *OVS*. These results show the benefit of offloading the flow management to external switches when using commodity APs.

In general, the results presented in Publication II show that both Intelligent and Thin AP techniques are viable for converting commodity APs into SDN-capable APs. This in turn enables the benefits of the SDN to be brought into existing networks.

3.1.6 Applications

In this section, we present several use cases for SWIFT. These use cases highlight the possibilities that Wi-Fi combined with SDN can bring to both enterprises, campus networks, and home users.

Dynamic Client Isolation.

The typical client isolation supported by the APs and Wi-Fi controllers is a blunt instrument. In most cases, it is a binary on or off implementation, *i.e.*, either the Wi-Fi clients can or cannot communicate with each other. This can prevent wireless streaming devices, such as Google Chromecast, from working if no clients can communicate with them.

The SWIFT approach makes the client isolation programmable. The SWIFT controller can decide which clients can communicate with which other clients network-wide, *i.e.*, the isolation is no longer limited to a single AP or full network.

1) Location Based Services. Some network devices advertise their existence by broadcasting their presence in the network using, for example, multicast DNS (mDNS) [60]. These devices include networked printers and multimedia devices such as Google Chromecast.

The following example shows how this could be a problem and how SWIFT can help. The Chromecast is typically connected to a screen and broadcasts its presence and name. The user can then see the Chromecast in a browser menu or in a media player application. However, if there are multiple Chromecasts, the user needs to find out which device is the one connected to a nearby screen. If there are many screens with Chromecast

across the building, the list can be very long and may cause showing media on the wrong screen across the building.

With SWIFT, this can be alleviated. The Chromecasts and their location can be registered with SWIFT. When a user device searches for Chromecasts, SWIFT can filter out those responses that are not near the user, thus limiting the list of devices to a large degree. The filtering can be done through the OpenFlow *match-action* rules, which match on well-known fields of the response packets from the Chromecasts.

2) Device Based Services. Another similar feature is device-based services. When SWIFT has recognized a registered device, it can allow or deny different services for that particular device. Example services could include access to restricted services for employee devices while at the same time denying those for guest devices. For example, local file servers should only be accessible to employees but restricted from guests.

3) Network Slicing. Network slices can offer different levels of service to the devices. For example, some slices can offer guaranteed Quality of Services (QoS) or lower latency, while a general slice could offer best-effort network service. While SWIFT cannot directly affect the radio interfaces, SWIFT can prioritize traffic flows. With the prioritization, SWIFT can offer traffic-based slicing, and with coordination with a WLC or similar, also perform it in the radio links.

4) Network Security. Since all traffic needs to be checked first at the controller due to SDN switches and client isolation, SWIFT can provide support for network security. When a new device first joins the network and is not previously registered to the SWIFT controller, the controller can redirect the traffic to different network functions such as security functions. One example is *Snort* [22], which can perform deep packet inspection (DPI) on the traffic.

After the inspection, the security function will report its findings to the SWIFT controller, and the controller can then decide how to handle the traffic. Typically these would include actions such as forward, block, or restrict.

Bring SDN to Existing Networks.

One major obstacle to SDN proliferation is the existing hardware that does not support SDN. While major network hardware vendors have slowly brought SDN capabilities to their switches, many existing networks still

use older equipment. In many cases, the existing hardware can handle the network needs of the venue, and replacing them would not be cost-efficient.

For example, a home network may consist of only a few devices, including the home gateway. The general cost of a new access point or home gateway is typically between 100 and 200 Euros. However, off-the-shelf SDN-capable devices either cost several times more than these devices or are not available at all. In most cases, consumers are not willing to spend the extra money to replace working equipment unless the speeds and other common features exceed the performance of the current equipment.

Similarly, more extensive networks have the same issue. The networking equipment is usually more expensive than consumer equipment, and in many cases, especially with the more extensive networks, is supplied by a single vendor. Replacing these networks with devices that have native support for SDN is not financially possible.

However, as we have demonstrated with the Thin and Intelligent AP designs, using these methods would not require replacing all of the network equipment. Instead, only the key points in the network would require new hardware that would allow deployment of the SWIFT. For example, to get the SDN capabilities to the Wi-Fi network would only require replacing a few traditional switches where the APs are connected to with SDN switches. This would allow the Thin AP approach to be deployed in the network.

Use Existing Wi-Fi Testbeds for SDN Experiments

Similar to bringing SDN to the existing network, SWIFT can bring SDN to existing Wi-Fi testbeds. While Wi-Fi and other network testbeds are common in universities and research institutes, they do not typically support SDN. However, these commonly already use *OpenWrt*-capable hardware due to the ease of experimenting that *OpenWrt* brings. With our methods, these testbeds can be converted into SDN testbeds without requiring to buy new hardware.

Client Mobility

Over the years, there has been much work to support client mobility in Wi-Fi networks. In non-managed networks, the mobility is handled by the Wi-Fi clients themselves by deciding when the Wi-Fi signal is weak enough to trigger roaming to another AP whose signal is stronger. In managed networks, the roaming may be initiated by the wireless controller.

In both cases, the switches in the network need to update their forwarding tables. This will take some time, attributing to lost packets when the

client switches between the networks. SWIFT can alleviate this by coordinating with the other controllers by either rerouting the traffic to the new location or duplicating the traffic to the AP where the client is connected and to the AP where the client is expected to connect.

Limit Number of SSIDs

In many SOHO networks, there are typically at least two network IDs broadcasted by the APs, namely *Guest* and *Private*. In addition to these, there may be multiple additional networks for additional services, such as services that are limited to certain locations or for particular user groups.

Similarly, vulnerable devices like the IoT devices are sometimes restricted to their own VLAN and SSID, where they cannot be accessed by other devices, at least not directly [61]. To make matters worse, in some large venues like fair centres or shopping malls, there could be even more SSIDs for different shops and cafes. While the location of these is restricted to small areas, the Wi-Fi signal can travel well beyond the premises.

However, each SSID broadcast uses a bit of available bandwidth as they are control frames and not data frames [50]. In the end, these broadcast beacons use up all available bandwidth. Using client isolation and, for example, security groups, the number of SSIDs broadcasted by the APs can be reduced. The security groups would allow registering the client device to certain groups with different permission into different networks. This would allow dispensing some of the SSIDs as the controller takes care of which networks the devices can communicate with.

3.2 PraNA: Programmable Network Analytics

Our networks have grown from what they used to be just a decade ago, and this growth is expected to continue at even increasing pace [4, 62]. Traditional networks typically consist of regular network elements such as switches, routers, and APs. Alongside the regular devices, the networks have also witnessed a new category of devices connected to the network, namely the Internet-of-Things (IoT) devices. These IoT are generally designed for a limited set of functionalities such as temperature sensing, remote-controlled switches, and similar, simple use cases for home automation and remote sensing. As such, their capabilities, including computing power and network connectivity, have been optimized to make them as low-cost as possible [3]. At the same time, the IoT hubs are bringing more network services inside the network [27, 63]. All these network elements, from IoT to APs and routers, are either manually managed, or in the case of

Vendor	SNMP	Proprietary	REST	Command Line
Cisco Wireless LAN Controller	✓	✓		✓
Aruba Mobility Controller	✓	✓	✓	✓
Ubiquiti UniFI	✓	✓	✓	✓

Table 3.1: **Control interfaces of Wi-Fi Controllers.** *All controllers support both SNMP and command line interfaces in addition to their proprietary APIs. Some controllers also support REST APIs. This table originally appeared in Publication III.*

large Wi-Fi networks, controlled by specific Wireless Network Controllers (WNC). For example, Cisco offers Wireless LAN Controller [44], with similar offerings from Aruba and Ubiquiti [64, 65]. Similarly, the IoT devices installed in the network are typically controlled by their own hubs or controllers [32], creating silos in the network that operate individually.

This growth has already started to reach the point where managing these networks is starting to be an impossible task, as the management has fragmented into multiple autonomous controllers. To regain control, we present PraNA in Publication III [66]. PraNA is a framework for programmable network management, where we tie together individual controllers using their Northbound APIs through a shared management bus. Using the SWIFT SDN controller allows us to manage the traffic flows, and with the help of WNCs, IoT hubs, and other elements, allows us to bring these heterogeneous networks under the management of the PraNA orchestrator. We envision PraNA as a network orchestrator that can communicate with other controllers inside the network and provide network management and orchestration.

However, the challenge is how PraNA can communicate with the controllers. Previously, network management systems from prominent vendors have concentrated on their own ecosystems [67]. However, as shown in Table 3.1, different controllers do support standardized APIs such as Simple Network Management Protocol [68], or vendor-specific REST APIs. At a minimum, at least the advanced systems also support command-line configuration, which allows configuration tools like Ansible [69] to access them. With these APIs, we can build agents that allow PraNA to communicate with the controllers and devices over a shared management bus.

PraNA is not limited just to the network orchestration. We designed PraNA to be a highly modular system that allows new features to be

brought to the network using third-party services. To exemplify this, we focused on using PraNA to secure the network from the inside. As our networks have witnessed the influx of devices, especially the IoT platforms and devices have brought new security vulnerabilities to the network [34]. Many of these IoT devices are cheap, with varying levels of support from their vendors, and typically little effort has been made to secure them [70].

To minimize this attack surface, PraNA uses SDN to secure the network. Over the years, multiple different SDN-based security systems have been designed [35, 71, 72]. These systems use different approaches, from Machine Learning (ML) fingerprinting to deep packet inspections to detect malicious traffic and devices.

Our PraNA framework uses the capabilities provided by the SWIFT [57] described in Section 3.1 to allow PraNA to isolate devices inside both the wired and Wi-Fi network. To know which devices are vulnerable, PraNA uses ML-based device classification to detect vulnerable devices [73]. However, PraNA is not limited to this approach, instead, it can also use other approaches for providing security and other services.

3.2.1 PraNA Design

The PraNA framework is a modular system, where the control plane of the networks consists of the PraNA orchestrator, the SWIFT SDN controller with PraNA augmentations, and elastic computing engines that can perform different operations such as machine learning-based device classification. The modularity allows PraNA to communicate with other controllers and network equipment to bring centralized network orchestration to the network. To achieve this, we separate the control plane and the data plane of PraNA as described below.

Control Plane

The control plane of PraNA consists of all the different controllers inside the network. In Figure 3.7, we showcase an example control plane for PraNA. The control plane consists of autonomous elements tied together with a common bus. The PraNA deployment can include the following elements, though only a subset of them are required for the network orchestration:

- *SDN Controller* is responsible for running the network and providing the network view and events to the PraNA Orchestrator. The SDN controller is also responsible for implementing the network policies set by the PraNA Orchestrator.

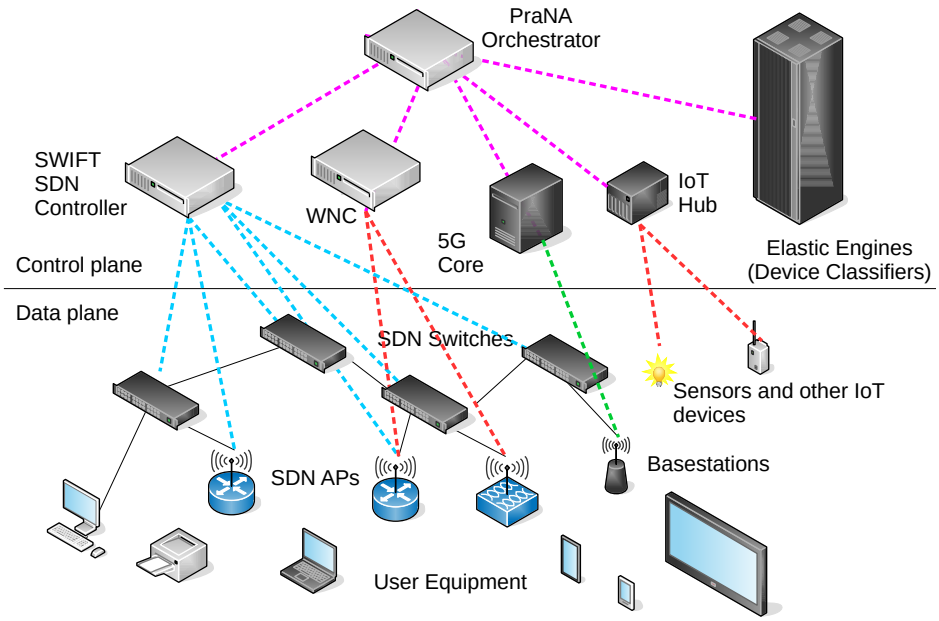


Figure 3.7: **PraNA architecture**. *Flows traversing the network are controlled by our SDN Controller, while Wi-Fi APs are managed by the Wi-Fi Controllers and IoT devices are managed by their respective controllers. This illustration is taken from Publication III.*

- *Wireless Network Controller (WNC)* manages the Wi-Fi network and its characteristics. The WNC provisions the managed APs points, including provisioning configurations and updates to the APs, and on which channels each AP should be to provide the best coverage. The WNC also provides Wi-Fi-related events to the PraNA Orchestrator.

- *Cellular Core* manages the local 5G network and its successors. It is expected that enterprises, factories, and other venues will run their own network slices to provide the best local connectivity [62, 74, 75]. The base stations in these networks are expected to provide both Wi-Fi and cellular networks [76], while the cellular core can provide open northbound APIs for integration [77].

- IoT Hubs manage their own IoT equipment. Many of the IoT devices are vendor-specific, and typically do not integrate easily with other IoT devices [63]. Nonetheless, protocols such as ZigBee [28] provide

standardized interfaces for devices to use, which in turn allow solutions like Home Assistant [33] to provide an aggregation point for the PraNA framework.

- *Elastic Engines* are computing units that provide different services to PraNA. In the testbed presented in Publication III, the Elastic Engines provide device fingerprinting and classification service to the Orchestrator [73]. They can also provide other services such as security services provided by the IoT Sentinel or other network functions [72].

- *PraNA Orchestrator* is the heart of PraNA. The PraNA Orchestrator is the main source of network orchestration and has the complete view of the network. The orchestrator aggregates the information provided by the other control plane elements and disseminates the results over the management bus. It provides network policy decisions to the SDN controller and presents the view of the network to the network administrators.

- *Management Bus* ties the above control plane elements together by providing a common publish-subscribe bus. The current topics provided by PraNA are shown in Table 3.2, which include topics for network events such as "new device" or "new switch," classification-related topics, and network policy topics. Each control plane element can subscribe to the relevant topics and publish new events or responses to commands it has received. In our proof-of-concept PraNA deployment, we use MQTT to implement the bus [78].

- *PraNA agents* are software components that allow the PraNA orchestrator and different controllers to communicate over the management bus. The agents translate messages from different entities to a form that particular control plane elements understand, creating a shim between them and PraNA.

PraNA is not limited to the above list of elements, nor does it require all of them to operate. The modular design with preconfigured default rules and policies allows PraNA to operate with a reduced capacity. For example, if the elastic engines are offline, the PraNA orchestrator will not receive classification updates. The orchestrator can still assign predefined policies to hosts, both known and unknown.

Topic	Description
prana/status	Status of network elements.
prana/network/device/switch/new	New switch added.
prana/network/device/switch/update	Update switch information.
prana/network/device/switch/remove	Switch removed from network.
prana/network/topology/links/new	New link between switches or between a switch and a device.
prana/network/topology/links/update	Update link information.
prana/network/topology/links/remove	Removed link from the network.
prana/network/device/host/new	New host added to the network.
prana/network/device/host/update	Update host information.
prana/network/device/hosts/remove	A host is removed from the network.
prana/classification/result	Result of the classification.
prana/classification/update	Update classification information.
prana/classification/pcap/request	Request a capture from a host.
prana/classification/pcap/response	Capture available for classification.
prana/policy/allow	Change device policy to Allow.
prana/policy/block	Change device policy to Block.
prana/policy/restrict	Change device policy to Restrict.
prana/policy/implemented	A policy has been implemented.

Table 3.2: **Example PraNA MQTT topics.** *The MQTT topics shown in this table are currently implemented in our PraNA framework. These topics allow network orchestration and management. This table is taken from Publication III.*

Data Plane

The data plane of PraNA consists of both non-SDN and SDN capable switches, APs, and routers. However, our deployment requires some SDN capabilities, as the SDN-capable devices are used to detect and capture traffic from the hosts in the network for classification and applying network policies inside the network. Still, traditional network equipment can be used for similar purposes, although not as efficiently. The WNC can inform the PraNA orchestrator of new devices through SNMP messages, and regular switches allow the usage of monitor ports, where network traffic is mirrored from other ports. Nonetheless, SDN is required at the network level to get the full benefits of PraNA.

Network policies

Network policies are inherent in every network. In a small home network, the main network policy inside the network is to allow all traffic between

the hosts, and only Internet access is limited by the firewall of the home gateway, *i.e.*, the NAT device. In more extensive networks, a more complex set of policies is required for the network to operate properly. A typical office network usually has at least two policies beyond the firewall rules, namely a guest network and employee network. The devices inside the guest network are barred from communicating with the devices inside the employee network and vice versa. Beyond these, large networks can have more complex rules for network operations.

For our PraNA framework, we created a minimal set of policies to demonstrate the capabilities and the dynamicity the SDN offers. This set of policies is not comprehensive but serves as a foundation for more complex policies. The PraNA policies include the following policies: *allow, restrict*, and *block*. Each of these policies is applied to individual devices, but the PraNA orchestrator can change the policy of each device at will.

The *allow* policy allows any host it is applied to have full access to the network, *i.e.*, the access is completely unrestricted. In contrast, the *block* policy denies all traffic to or from the device, for example, a malicious host can be quarantined from infecting other hosts using this policy. Lastly, the *restrict* policy allows the host only limited access to the network. In our example deployment, the *restrict* policy allows the host to receive an IP address from the DHCP server, communicate with the DNS server, and have access to the Internet. No other services are available for the host.

The SDN controller enforces these policies through OF rules installed on the SDN switches. All known devices will be applied with its default policy, and when an unknown device joins the network, it is first given the default *Restrict* policy, which is then updated when the device is classified if needed. When the PraNA orchestrator applies a policy update to a host, the SDN controller updates the relevant rules in the switches and informs the orchestrator when the policy is in place.

This is only a very basic set of rules. A more fine-grained set of policies is required for larger networks, where hosts have different levels of network access or some other requirements. For example, the applications described in Section 3.1.6 could be implemented through the PraNA orchestrator and the SDN controller.

3.2.2 PraNA Evaluation

We evaluated PraNA using a testbed that contained different types of IoT devices from simple power switches to video streaming devices, and even a smartphone and a laptop. The devices were chosen to represent different categories of IoT devices and had varying levels of both computing and

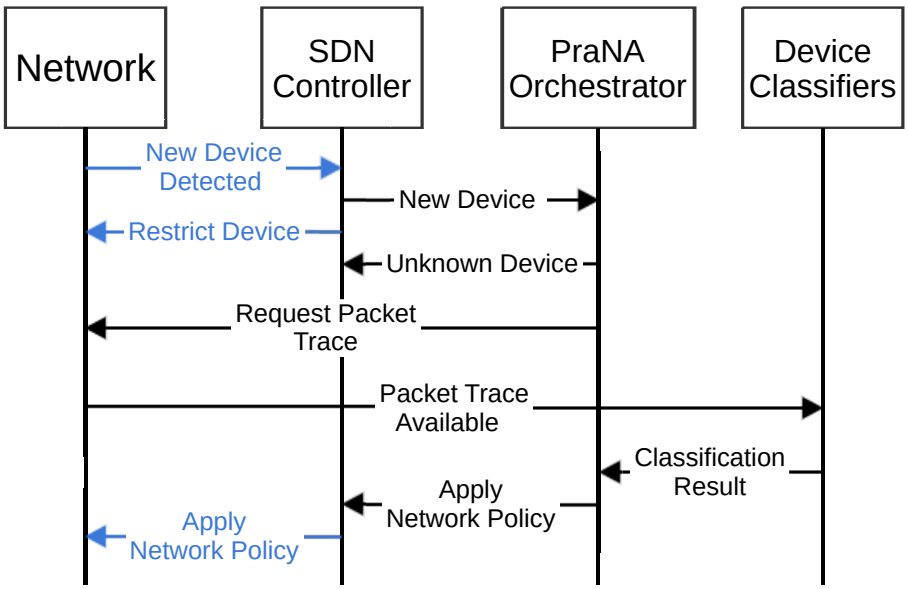


Figure 3.8: **Sequence of messages over the management bus.** *OpenFlow control messages between the SDN controller and the network are denoted in blue. This diagram was taken from Publication III.*

networking capacities. Each of the devices generates a different number of packets when it powers on and joins the network, including ARP, DHCP, and other packets if and when it connects to the Internet.

The control plane of the testbed consisted of the PraNA orchestrator, SDN controller, and device classifiers. The different control plane elements were tied together using MQTT-based management bus. The data plane of the testbed included multiple SDN-capable switches and APs, which used the SWIFT techniques to control the traffic between devices, and provided traffic captures for our ML-based device fingerprinting.

For our evaluation, we powered the devices and measured how long it took for PraNA to detect the device, get a packet capture of 30 packets, classify it, and apply a network policy to it. The sequence of events and management bus messages is shown in Figure 3.8, and the results are shown in Figure 3.9. When the network, *i.e.*, the SDN switches, detects a new device, they inform the SDN controller over OF messages. The SDN controller assigns a default policy to the device and informs the PraNA orchestrator. The orchestrator requests a packet trace from the device for the classifiers. When the trace is available, the classifiers classify the device

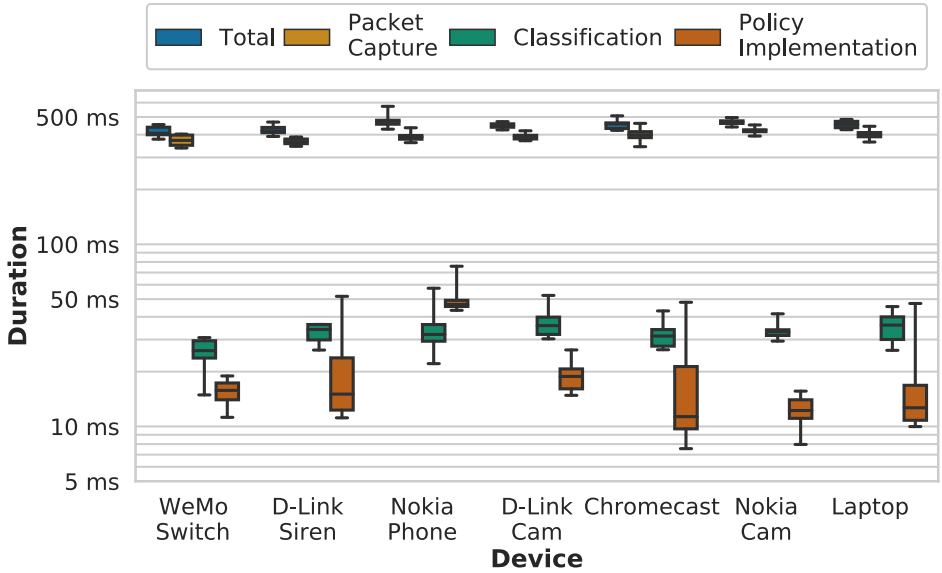


Figure 3.9: **Total time.** *We present a) total time taken by the PraNA Orchestrator from the moment a device is detected to the time a network policy has been implemented, b) time for the packet capture, and c) time for classification, and d) time to implement the policy. The figure was taken from Publication III.*

and inform the orchestrator of their results. The orchestrator then assigns a revised network policy to the device through the SDN controller.

In general, the full sequence of events took roughly 500 ms, out of which the packet capture used the most time, roughly 400 ms. The rest of the events, *i.e.*, classification and policy implementation, used on average 80 ms, with the classification taking 40 to 50 ms.

This testbed is only a small-scale testbed with negligible network latency. However, in the current form, many of the PraNA elements are single-threaded systems. If we compare the performance of the testbed to a *eduroam* dataset from 2014 [79], where at the peak hour there were 9.6 network association events per second, the PraNA framework can easily be made to handle the load. The packet captures are handled by individual switches and APs, and the device classifiers can be made to operate in parallel. With these tweaks, PraNA should be able to handle the loads presented in the dataset.

3.3 Summary

In this chapter, we have shown a way to deploy SDN on existing devices. The previous approaches for converting existing devices to support SDN have limitations such as allocating too many resources for each client or requiring custom firmware. Our SWIFT approach, namely the Intelligent and Thin AP, does not require massive changes to the devices. The main requirement is the client isolation in either permissive or restrictive form. The Intelligent AP approach also requires the AP to support OpenWrt or similar firmware; however, the firmware does not need to be customized in any other way than installing *OVS* on the device and replacing the default Linux bridge with it. The Thin AP technique, on the other hand, does not require changes to the device, but it requires the AP to be connected to an SDN switch. Our evaluation shows that the techniques do not have a significant impact on the device performance, and in some cases, may actually increase the performance of the less powerful APs.

We also discussed different applications that SDN-based flow management could bring to the Wi-Fi networks. These applications allow fine-grained control over the Wi-Fi clients and can bring different services such as location-based services to the client without special network designs. To exemplify the benefits of the SDN, we presented PraNA, a framework for network orchestration. PraNA ties together different network controllers, including SDN and Wi-Fi controllers, over a common management bus. This allows PraNA to orchestrate the network operations and bring other services like ML-based security services to the network.

Chapter 4

Seamless Multiconnectivity

In this chapter, we discuss the challenges of multiconnectivity and our proposed solutions to solve some of them. In general, almost all of our devices are capable of using more than one communication interface to connect to different networks [80]. For example, a typical laptop has at least three communication interfaces, namely Wi-Fi, Ethernet, and Bluetooth [30]. In addition to these, the laptops may also include a cellular modem to connect to the mobile network, either as a built-in modem or as a USB dongle. A smartphone usually has the same communication interfaces as laptops, with the difference of having the cellular modem built-in and the Ethernet as a USB dongle, although using a wired network with a phone is uncommon [81]. Other devices such as smartwatches and tablets either have the above interfaces available or a subset of them.

However, all these devices offer only a limited control to the users on which communications interface to use beyond toggling them on or off. In most cases, the communication interface is chosen by predefined metrics in the operating system. If we consider the three most common interfaces, namely Ethernet, Wi-Fi, and mobile networks like 5G, all of these have their own priorities. Traditionally, the order from the highest to the lowest priority is 1) Ethernet, 2) Wi-Fi, and 3) cellular. Why this order is such is based on multiple reasons. Earlier, Ethernet was always the fastest network available, while Wi-Fi was slowly catching up, and 3G had not become available yet. Then there are different cost metrics, such as energy, bandwidth, and monetary costs. They can be defined in different ways [82, 83]. Typically, the costs are expressed as power usage, bandwidth, latency, and money. For example, Ethernet is typically considered the fastest available network, while the mobile network is most expensive due to data plan costs.

In Table 4.1, we present pros and cons for different network interfaces. The list is approximate, but it illustrates the reasoning behind the metrics.

Interface	Pros	Cons
Ethernet	Fast, cheap	Requires cable
Wi-Fi	Cheap, good speeds	Limited range, can be congested, energy
Mobile	Good coverage	Data plan costs, speed varies, energy

Table 4.1: **Pros and cons of the different interfaces.** *Different network interfaces have different characteristics that affect what are benefits and disadvantages of using them are.*

Ethernet is typically very fast both in bandwidth and latency, at least in the local area network (LAN). However, Ethernet requires a physical connection to the LAN, *i.e.*, a cable, which limits the mobility of the connected device. This can also be considered a boon for Ethernet. There is usually also power available near the physical network sockets, allowing devices to be more powerful than fully wireless devices.

Typically Wi-Fi networks are available on an on-premises basis, *i.e.*, a home network, campus network, or workplace. However, these days Internet connectivity is considered to be almost a basic need; many places such as cafes, hotels, and airports offer complimentary free Wi-Fi. Wi-Fi networks are typically slower than Ethernet as the users share the same wireless medium. On the other hand, as Wi-Fi networks do not require a cable, the mobility is greatly increased, as long as the device stays within the range of the Wi-Fi network. Also, with the recent advances in Wi-Fi technology, the achievable speeds of the Wi-Fi are starting to catch up with Ethernet, at least in optimal conditions [84].

However, the use of the wireless medium has some drawbacks. Especially the older Wi-Fi networks can easily become congested as the available spectrum is shared by multiple devices. The Wi-Fi also covers only the premises within the range of the APs, and within this area, there can be areas where the signal strength is low due to thick walls, limiting the speeds of the network. Finally, using Wi-Fi consumes energy. While Wi-Fi itself may use less energy than Ethernet, there is typically a power socket available with Ethernet. Due to this, in practice, Ethernet does not require energy from the device's battery, while Wi-Fi consumes it.

These days, mobile networks are available almost everywhere. This makes them very attractive for Internet connectivity and is the reason why the world has changed into a digital one. They allow connectivity from places where no one would have thought connectivity would be possible only a few decades ago. 5G and future networks have also improved the speeds and latency dramatically. While they may not yet be on a par with the above technologies, the mobile network is fast closing in on Wi-Fi.

However, these come with costs. Mobile data plans are not cheap. The speed and available amount of gigabytes per month can drive the cost of mobile networks very high. As with Wi-Fi, mobile networks also require energy, which can be modelled as battery costs. In addition to the above, the coverage of mobile networks varies. Depending on the mobile operator, cities and suburbs are usually covered with fast networks, but when moving to more rural areas, the download speeds can drop dramatically from tens of megabits per second to only a few. Upload speeds may only be a few kilobytes per second at worst.

The above gives insights into why there are different priorities for different interfaces. Ethernet has the highest priority, followed by Wi-Fi. Mobile networks have the lowest priority, as they are considered to be the most expensive. Typically when a device enters a known Wi-Fi range, the device automatically switches to the Wi-Fi network, and existing connections are broken as the five-tuple defining the connection change. In most cases, users have no control over the change beyond shutting Wi-Fi off before entering the Wi-Fi range. Application developers may have some control over it, but usually, it is very limited or requires getting superuser privileges.

Another reason why automatically switching to Wi-Fi is not always the best idea is the network conditions. A Wi-Fi network may be very congested, limiting the available bandwidth to less than the mobile network. The uplink of the local network to the Internet can be a slow one, or it may have so many other users that it is also congested. These and other reasons make the automatic switching to Wi-Fi or Ethernet problematic at best.

In this chapter, we present two different approaches for multiconnectivity. In both cases, the aim is to make multiconnectivity programmable, allowing users and the network to influence the choice of interfaces and protocols to be used.

The first approach, Meghna, uses SDN to achieve multiconnectivity at Layer 3. The second approach, MULTI, is agnostic to underlying protocols and interfaces and is targeted to the session layer.

4.1 Meghna: An SDN Perspective on Multiconnectivity

Currently, our devices make the decision on which network interface to use to transfer data locally, without a proper network view beyond the first link. As discussed earlier, this decision is based on predefined metrics that do not take network state into account. SDN controllers, on the other hand, have this view as they control all traffic in the network. SDN allows

the SDN controllers to steer the traffic flows to paths that are best at any point in time, allowing the traffic to avoid congestion if possible. However, using SDN for multiconnectivity requires enabling the SDN on the host device itself so that it also becomes a part of the network, not just a host connected to the network. This allows the SDN controller to take over the decisions on which network interface to use at any given time.

Earlier in this thesis, we presented SWIFT in Section 3.1. While SWIFT brings SDN-based traffic management to Wi-Fi networks, it in itself is not a multiconnectivity solution. Although SWIFT does not support multipathing or multihoming, it can be used as a starting point for programmable seamless multiconnectivity.

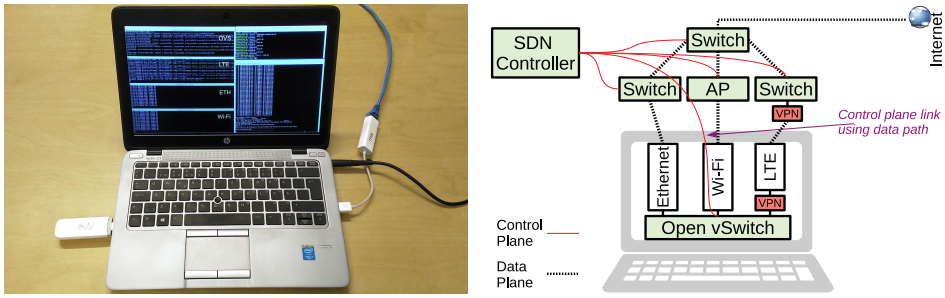
In a typical case, each network interface is assigned its own set of IP addresses. When an application creates a connection to a remote host, the routing elements of the device's operating system check which of the available interfaces can be used to connect to the destination and chooses one accordingly. This choice is influenced by which network the destination is located at, and if more than one interface can be used, the highest priority interface is chosen. However, as discussed before, this choice is not always the best one.

In Publication IV, we present Meghna, an SDN perspective on user-centered multiconnectivity. Meghna takes the ability of SWIFT to manage all network traffic one step closer to the user [85]. Meghna moves the software SDN switch to the host device itself, with the Meghna SDN controller located in the network.

Having the SDN switch at the host device simplifies traffic management. It allows SDN controllers to leverage the known network state fully and manage traffic accordingly. Moreover, as the host device is connected to the network through multiple interfaces, the SDN controller can also perform traffic migration based on different network inputs, such as signal strengths and congestion notifications. When the SDN controller makes the decision to move the traffic from one interface to another, the controller performs *make-before-break* migration, *i.e.*, install the required *match-action* rules for the traffic inside the network before switching interfaces [86].

4.1.1 Meghna Design

The key design decision for Meghna is to leverage the knowledge that an SDN controller has on the network state and not just the local information available to the host. The SDN controller, namely the Meghna controller, has the full information on the current network state, *e.g.*, which links are congested and how to optimize the network paths.



(a) Laptop with 3 interfaces. (b) Network topology.

Figure 4.1: **Multiconnectivity Setup for Meghna.** *The Ethernet, Wi-Fi, and LTE interfaces of the laptop are plugged to an SDN switch running on the laptop. These figures were taken from Publication IV.*

Meghna design has two main assumptions. First, all interfaces of a device are connected to a "home" network, either through local Ethernet or Wi-Fi links. If the device is not within the home network range, the Meghna can use a Virtual Private Network (VPN) over the Internet to connect to the home network. This home network shares a single IP address space that all devices use and is SDN capable.

The second assumption is that devices participating directly in the Meghna system are also SDN capable. This means that 1) they have an SDN switch installed locally on the device, and all network interfaces are connected to this switch, and 2) this SDN switch is registered to the Meghna SDN controller. This allows the traffic to be controlled by the Meghna controller when the device joins the network.

The above is shown in Figure 4.1. In Figure 4.1a, a laptop with multiple connections is shown, and in Figure 4.1b, we show how the laptop is configured internally. Each of the Ethernet, Wi-Fi, and cellular are plugged into the OVS, controlled by the remote SDN controller.

In addition to the above assumptions, we need to take one more step. Generally, each network interface of a device has one or more IP addresses. However, for Meghna to work, the SDN switch on the device is allocated a single IP address from the home network. As the network interfaces are directly connected to the *OVS*, they do not have separate IP addresses. This allows the switch to connect to the Meghna controller, and more importantly, allows applications to use the same IP address as their source address regardless of which interface is actually used. Since the IP address does not change, the applications can be agnostic on the underlying network and location, *i.e.*, the device can move from one location to another location, and as long as a connection to the home network is maintained,

Algorithm 1: The set of rules to add and delete

Data: P'': the new path, and P': the current path
Result: α: set of rules to add, and δ: set of rules to delete

```
S' = {s(r) | r ∈ P'}
S'' = {s(r) | r ∈ P''}
θ' = {⟨s(r), m(r), a(r)⟩ | r ∈ P', s(r) ∈ S' ⋂ S''}
θ'' = {⟨s(r), m(r), a(r)⟩ | r ∈ P'', s(r) ∈ S' ⋂ S''}
α = {r | r ∈ P'', s(r) ∈ S'' \ S'}
δ = {r | r ∈ P', s(r) ∈ S' \ S''}
foreach j ∈ S'' ⋂ S' do
    r_α = r | r ∈ P'', s(r) = j
    r_β = r | r ∈ P', s(r) = j
    if ⟨s(r_α), m(r_α), a(r_α)⟩ ∉ θ' ⋂ θ'' then
        π(r_α) ⇐ π(r_β) + 1
        α = α ⋃ r_α
        δ = δ ⋃ r_β
    end
end
```

the IP address does not change. As the IP address does not change, it allows the applications to maintain a constant connection to their destinations and do not have to recover from broken connections when the device moves and the IP address changes.

Devices that are not SDN-capable are not excluded from Meghna. They can connect to the network like any other device but do not participate directly on the Meghna. This means that they are plain clients with a single connection to the network; however, beyond the first link to the network, Meghna can control the traffic.

To facilitate the seamless multiconnectivity in Meghna, we use the algorithm shown in Algorithm 1. This algorithm is used by the Meghna controller when the controller decides to migrate all traffic from one interface into another interface, and the detailed description can be found in Publication IV.

The algorithm is implemented using the techniques of Huque *et al.* [87]. These techniques are built on the two-phase-commit [88], and the reverse rule update techniques [89]. However, using these techniques directly would also require updating rules on the switches that are common on both the new and old paths. We, therefore, improve on the rule update techniques so that we avoid modifying the rules that do not need modifications.

The rule update sequence is illustrated in Figure 4.2, where the initial path from source (SRC) to destination (DST) is illustrated in red and passes

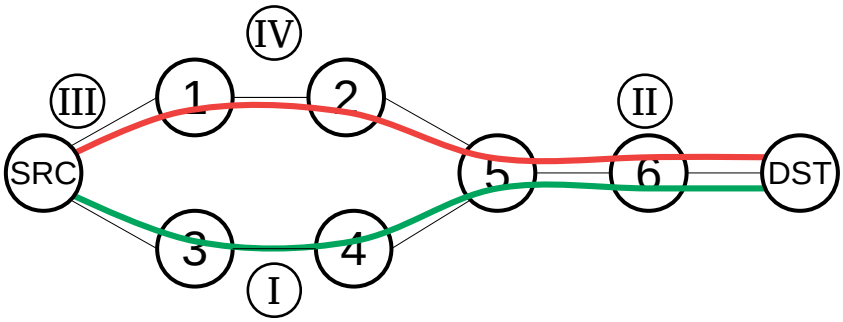


Figure 4.2: **Meghna rule update sequence**. *The figure depicts an old path and a new path, shown in red and green respectively.*

through Node 1. Each node, including SRC and DST, represents an SDN switch. The sequence of steps shown in Figure 4.2 shows how a one-way path, *i.e.*, the path from SRC to DST, is migrated from the old path, shown in red, to the new path. To allow full communications between SRC and DST, a reverse path from DST to SRC must also be installed. However, to simplify the figure, the reverse path is omitted.

When the Meghna controller initiates the migration to the new path, shown in green, it first calculates the path shown in Step I. Note that Node 6 is common to both paths, so the Meghna controller does not need to update the rules in Node 6 during Step II. After Step II has been completed, the Meghna controller redirects the outgoing traffic at SRC to use the new path through Node 3, completing Step III.

The last step, Step IV, is removing the old path. Here, the Meghna ensures that each hop of the path is clear, *i.e.*, there are no packets in flight on that hop. After each hop is clear, the Meghna controller removes the relevant OF rules from the nodes so that the set of rules present in each node does not grow too large.

As noted above, this sequence is for a single direction only. Similar operations also need to be performed in a reverse direction to allow two-way communications.

4.1.2 Meghna Evaluation

For our evaluation for Meghna, we built a migration testbed illustrated in Figure 4.3. The testbed can be split into three parts. First, the SDN test network contains multiple SDN switches marked OVS. These are based on Linksys WRT3200ACM APs that are running OpenWrt and OVS as described in Section 3.1.2. Each AP also includes four 1 Gbps Ethernet

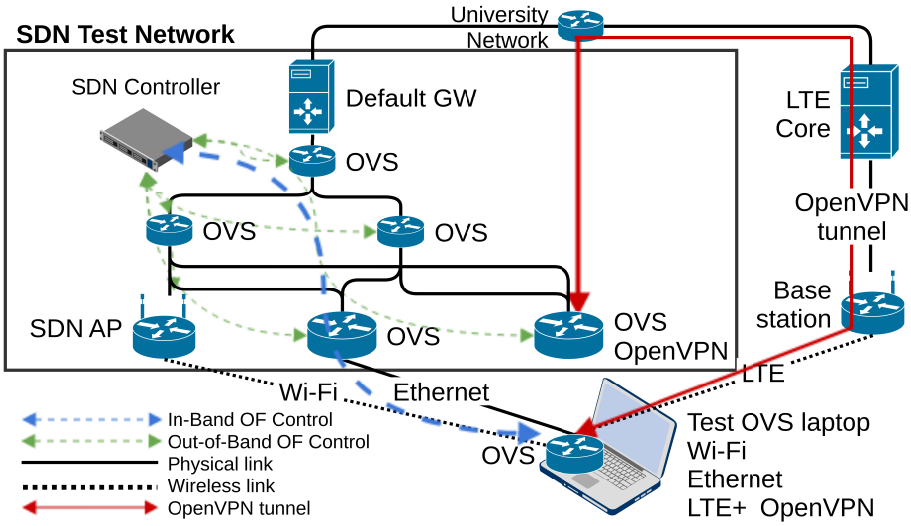


Figure 4.3: **Meghna Testbed.** *Our test network contains interlinked SDN switches, an SDN capable AP, and an LTE connection via a VPN tunnel. This illustration is taken from Publication IV.*

ports and a WAN port. The WAN ports connect the APs to the SDN controller to allow out-of-band management.

In addition to the SDN test network, we use an external LTE network to add a cellular capability to the testbed. The LTE is realized through a USB LTE dongle and OpenVPN tunnel through the external mobile network to a specific OVS AP inside the test network.

The last part of the Meghna testbed is the test laptop depicted in Figure 4.1a and Figure 4.1b. This laptop includes the following physical interfaces: Wi-Fi (802.11ac) interface, 1 Gbps Ethernet interface by default, and a USB LTE dongle to connect to the LTE network. To combine these, we installed an OVS switch on the laptop and plugged the Wi-Fi and Ethernet interfaces directly into the OVS.

Since the USB LTE dongle acts as a NAT device between the LTE network and the laptop, it gives out a private IP address to the host different from the address range used in the testbed. To work around this, we use an OpenVPN tunnel in TAP-mode to allow us to connect the VPN interface to the OVS. The TAP interface created by the OpenVPN is a virtual Ethernet device, and as such, can be added to the OVS.

After all interfaces are connected to the *OVS*, it acts as a bridge interface to the operating system. We allocate a single IP address to this bridge

interface from the testbed. As all interfaces are connected to the test network either directly or through the OpenVPN tunnel, we can migrate the traffic over any of the interfaces without breaking connectivity.

However, as the laptop is a mobile device, it does not have an out-of-band connection to the Meghna SDN controller. Instead, it has to use an in-band control channel through the data plane of the test network to connect the OVS to the controller. This requires specific default match-action rules to be installed into the laptop's OVS to allow it to forward traffic even if it is not connected to an SDN controller. These rules allow the OVS to reach and register to the SDN controller, after which the SDN controller can take over controlling the OVS bridge and manage all traffic flowing inside the network.

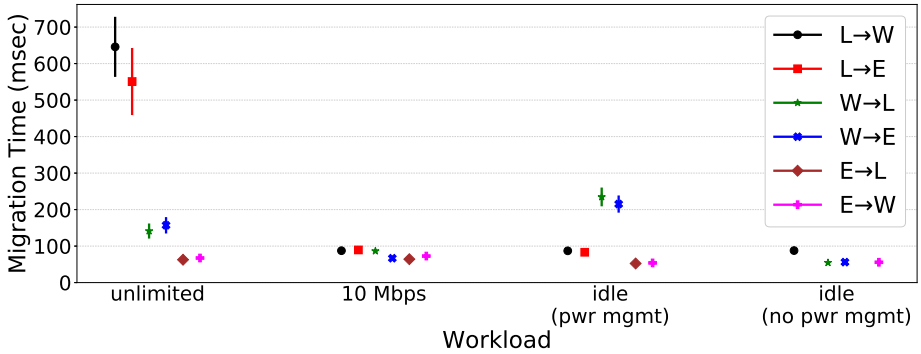
The evaluation results are shown in Figure 4.4, and the detailed analysis can be found in Publication IV [85]. The results show that when the network links are not congested, the migration time remains mostly under 100 ms. However, when the links are congested, the effect of scheduling wireless frames or the effect of bufferbloat, *i.e.*, large buffers in the network, can cause major latency in migration. This is due to the control messages from the Meghna server to the *OVS* running on the laptop being buffered alongside the other in-band traffic. A similar effect can be observed with the power-saving features of the wireless links. If there is no traffic, the transmitters do not need to be powered on and require waking up [90].

Similarly, the TCP retransmissions increase dramatically when the links are congested. While the Meghna ensures that all paths are clear before dismantling them, the large network buffers may cause the destination to discard packets during the migration if the packets on the new path start arriving well before the old path is clear. However, when the links are not congested, there are only under 30 retransmissions during the migration.

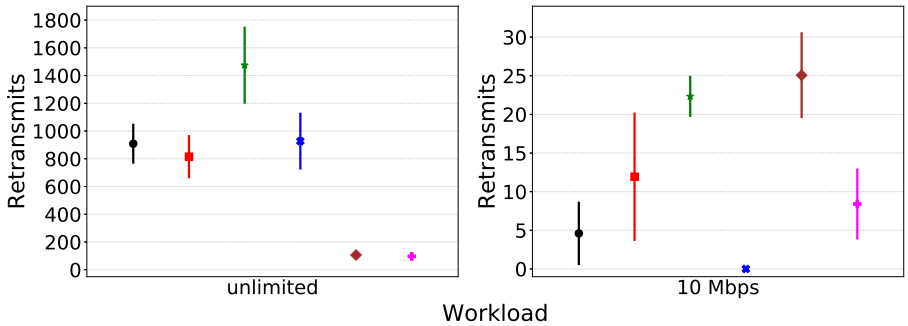
4.1.3 Meghna Discussion

Meghna is one way to perform multiconnectivity and traffic migration. The benefits of the network-triggered traffic migration are better network utilization and better Quality of Experience (QoE) as the disruption to network traffic is minimized. The main part of this is the algorithm that allows the Meghna controller to build a new path through the network before tearing up the old path.

Similarly, the Meghna design of using a single IP address assigned to the bridge interface on the host device simplifies the design of applications. This allows the application to maintain a network connection without breaking the 5-tuple that defines the connections.



(a) Mean time to migrate a flow.



(b) Number of TCP retransmissions during flow migration.

Figure 4.4: **Evaluation results.** *We consider all 6 combinations of flow migrations between Ethernet (E), LTE (L), and WiFi (W) links. The common legend is shown in the top figure and L→W implies migrating a flow from LTE to Wi-Fi. The error bars represent the 95% confidence interval across 30 iterations. These figures were taken from Publication IV.*

However, the main drawback of the Meghna is the home network. When the host device is on-premises of the home network, the Meghna performs optimally. However, if the host device is roaming in another network, the connection to the home network must be established over a VPN tunnel. While this allows retaining the IP address the application uses, the extra cost of tunnelling the traffic to the home network and back causes extra latency. This can be mitigated by having regional home networks that allow traffic to enter the network from a closer end-point.

However, if no connection to the home network can be established, the Meghna system must be able to operate without the home network. This is also a requirement for establishing the connection to the home network

in all cases, *i.e.*, when the host device, for example, wakes up from sleep, its old network connections are disconnected, and it must join the network.

This is achieved by installing low-priority default rules on the OVS running on the device. These allow normal connectivity to be established, and when the connection to the Meghna controller has been established, the controller can take over handling the traffic rules.

4.2 MULTI: Programmable Session Layer MULTI-Connectivity

Earlier in Section 4.1, we presented how SDN can be used to bring multiconnectivity to the host devices and perform seamless traffic migration from one interface to another interface. However, Meghna suffers from the fact that it requires an SDN-capable home network, thus making it unsuitable for general multiconnectivity. In addition to this, Meghna relies on relaying the traffic through the home network when the device is roaming.

While this kind of traffic relaying is not unheard of, it is suboptimal. To a degree, Meghna's approach is similar to Mobile IP, where the Mobile IP always relays the traffic through the home network [91]. Another example, Host Identity Protocol (HIP), uses a rendezvous server to keep track of the location of the host [92, 93].

The second problem with Meghna is that it is not an end-to-end multipath solution. While the traffic flows of Meghna can take multiple paths through the home network, beyond the home network, the traffic flows just like any other traffic. For an end-to-end multiconnectivity solution, the traffic must be able to use multiple paths to the destination. This can be achieved using transport protocols that are designed with multipathing and multihoming in mind. These protocols carry relevant information for multiconnectivity somewhere inside the packet headers instead of relying on the network infrastructure for multiconnectivity. At a minimum, the information carried must include a *Connection ID*, which identifies a connection to which the packet belongs. The receiver then uses this ID to associate all packets with the same ID to the same connection regardless of the source of the packets.

The seminal example is Multipath TCP (MPTCP) [23, 94]. MPTCP is an extension to TCP, and as such, operates in the TCP stack of the kernel. The support for multiple paths in MPTCP is implemented through MPTCP TCP options. When an MPTCP-capable host connects to another host, it adds MPTCP options to the SYN packet. If the destination also supports MPTCP, it replies with an MPTCP option acknowledging

that it also supports MPTCP. During the initial exchange, the two hosts exchange their available network addresses. After the initial exchange has been completed, the hosts can add parallel flows using different endpoints.

The MPTCP can use these subflows of the connection to aggregate different paths for added bandwidth, use them for backup connections, or find the lowest latency path. MPTCP is also backward compatible with regular TCP. If a non-MPTCP capable receives a TCP SYN with MPTCP options, it should reply without the options it does not understand. As such, if the reply, *i.e.*, the TCP ACK, does not contain the MPTCP option, the initiating host treats the connection as regular TCP.

However, the MPTCP has not yet been widely deployed. As discussed in Section 2.5, different middleboxes behave unpredictably when faced with packets with unknown headers. For example, a home gateway may drop the MPTCP packets, remove the option or otherwise behave erratically. Another problem is the load balancers. Load balancers distribute the incoming load to multiple servers, which allows for higher capacity than what a single server could provide. If the load balancer does not understand the MPTCP options, or there are multiple load balancers facing different IP address blocks, the load balancer cannot bridge different MPTCP subflows together, instead, it may distribute the subflows to different servers.

Another example of a protocol that provides mobility is QUIC [5, 42, 95]. QUIC is a transport protocol built over UDP to allow better connectivity and mobility. The main use case for QUIC is better performance for HTTPS. Many websites contain multiple elements and traditionally require the opening of a new TCP connection to the web server, which is costly. QUIC, on the other hand, is designed to multiplex multiple data streams into a single connection, allowing these elements to be fetched in parallel.

QUIC is not a multipath protocol in itself, but it supports mobility by using Connection IDs like MPTCP. These IDs are carried inside the QUIC headers, and as QUIC is implemented over UDP, the source address of the UDP datagrams is not fixed. When the host moves between networks and its IP address changes, the destination of the datagrams can associate incoming datagrams to existing connections using the Connection IDs.

One of the main benefits of QUIC is that it is implemented in userspace. This allows QUIC to be deployed faster than MPTCP as it does not require changes in the kernel of the underlying operating system. This, in turn, allows browsers and other applications to implement new features in QUIC to increase its capabilities.

There has also been work to add multipath capabilities to QUIC. This Multipath QUIC (MPQUIC) is an extension to the regular QUIC [96].

MPQUIC operates in a similar way to MPTCP, *i.e.*, when a connection is established, the endpoints determine if both support MPQUIC and exchange other possible paths.

Nonetheless, neither QUIC nor MPTCP is currently able to solve all issues of multiconnectivity by itself. Both are valid in their own area but are susceptible to middleboxes and other network constraints. As such, a method that combines different transports is required to solve multiconnectivity.

The IETF Transport Services Working Group (TAPS) is working on architecture for exposing a Transport Services API to applications [97]. Traditional transport protocols have their own APIs, requiring application developers to use protocol-specific calls to use them, even though conceptually similar protocols could share the calls. The transport API will allow applications to query what transport options are available between the destinations and use a standardized API for sending messages [97].

The NEAT library is an implementation of the TAPS framework [98, 99]. It is a userspace framework built on top of the socket API, allowing NEAT to offer a flexible platform and protocol-independent transport API to the applications. As the API is flexible, it can be easily extended to support new protocols as they become available.

NEAT also allows applications to request a certain set of transport options, including which protocols to use. When a connection is requested, NEAT performs Happy Eyeballs (HE) connection candidate gathering for each of the available transports [100]. This allows NEAT to build a list of working protocols, and when the HE process finds a working connection, it is returned to the application.

The MULTI framework introduced in Publication V [101], takes this further. In addition to bringing multiple transports under one roof, the MULTI also includes options to a) aggregate different transports together and b) allow applications to decide which interfaces are to be used for which transport protocols.

4.2.1 MULTI Design

The design of MULTI draws from the insights of both TAPS and NEAT, QUIC, and other sources like the UrlSession [102]. MULTI aims to provide a framework that brings different multiconnectivity approaches under one roof and provides simple interfaces for applications for transferring data over multiple transport interfaces and protocols.

Goals

With MULTI, we aim to address the following features:

1. End-to-end exchange of data streams. One of the largest obstacles for multiconnectivity is the five-tuple, discussed in Section 2.1. The five-tuple denotes both end-points of the connections, with the source address being the address of one of the device's network interfaces. If the device is connected to multiple networks or roams between networks, the five-tuple is broken as the IP addresses change. Meghna handles this with a bridge interface and is always connected to its home network. However, this is not possible with MULTI, as there is no home network.

MULTI takes another approach. Instead of directly binding to a network interface, MULTI is a connectivity layer between the application and the available networks. MULTI exposes the application *read* and *write* queues and handles required socket and other connectivity operations invisibly to the application. This provides MULTI with the capability to offer multiconnectivity in arbitrary networks.

2. Support for different transport, network, and link layer protocols. Even though most of our devices support multiple transport interfaces and protocols, not all the devices support all of them. Laptops typically support Wi-Fi and Ethernet for transport interfaces, while phones support Wi-Fi and cellular networks. All devices support the normal transport protocols like TCP and UDP, but may not have support for protocols that require kernel support like MPTCP. We have designed MULTI to be transport agnostic, *i.e.*, MULTI should be able to use whatever interfaces and protocols are available on the device. Depending on the situation, MULTI should be able to establish a connection to the remote host regardless of the network state, as long as there is a path available.

3. Allow applications to specify and suggest configurations. With MULTI designed to be a network agnostic library, we also allow applications to have more control over how different networks and protocols are used. As discussed earlier, different networks and protocols have different costs or connectivity issues. For example, cellular data could be more expensive than Wi-Fi, however, MPTCP might not work in the Wi-Fi network. MULTI should allow users and applications to specify their connectivity preferences and fulfil them as well as possible.

4. Simultaneously use multiple transport protocols and interfaces. Simultaneously using multiple types of transport allows MULTI to achieve and retain connectivity over heterogeneous networks. As middleboxes are prevalent in practically all networks [103], the set of transport protocols that will work is not known beforehand [9, 10]. For example, if

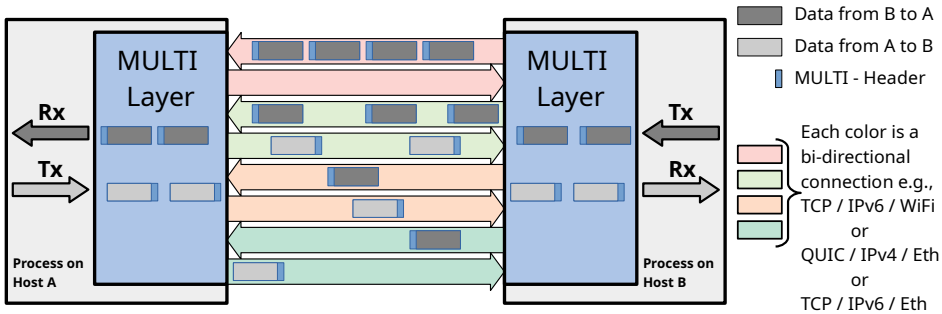


Figure 4.5: **Example MULTI Session.** *Processes can use MULTI to exchange bi-directional data streams. MULTI multiplexes the data over multiple bi-directional transport connections. This figure was taken from Publication V.*

QUIC has been selected as the transport protocol, and the device changes into a network that discards QUIC packets, the connectivity is lost even though there could be other protocols that would retain it. Using multiple protocols prevents this, especially if some protocols are specifically chosen as a backup from well-known protocols like TCP or UDP.

5. Seamlessly react to network changes. The different connections available to a device will change when the device roams between networks. MULTI needs to be able to both prepare and recover from the connection changes caused by the roaming. To achieve this, MULTI needs to be able to detect network changes and react to them.

6. Userspace implementation. As we have discussed before, using protocols that require changes to the operating system slows their deployment. To prevent this, we draw from QUIC and NEAT to implement MULTI in userspace. This allows faster deployment and the introduction of new features.

Architecture

In Figure 4.5, we show how two MULTI-enabled hosts transfer data. MULTI exposes *Read (RX)* and *Write (TX)* queues to the application. As discussed below, when the application requests MULTI to open a connection to the destination, MULTI negotiates bidirectional streams over the requested transports. When the application sends data, the data is encapsulated in MULTI segments to be transferred over the available connections.

The header fields of the MULTI segments used in our prototype implementation are shown in Table 4.2 and are influenced by QUIC. The key

Field	Description
Version	Version number
Segment type	Indicates the type of segment. Our prototype for MULTI currently supports segments for data, acknowledgement, keep-alives, and for closing a MULTI session.
Header length	The length of the header. This is used to indicate when the payload begins.
Session ID	The session id to which the segment belongs.
Sequence Number	Contains the sequence number if the segment contains data.
Segment length	The total length of the segment including the segment header.

Table 4.2: **Fields in the MULTI header for each MULTI segment.**

```
{'connection_priority': [('mptcp','ipv6',['eth', 'wifi']),
                         ('quic','ipv6', 'cellular'),
                         ('tcp','ipv6','cellular')],
'multi_config': {'connect':'seq', 'scheduler':'RR'},
'mptcp_config': {'scheduler':'RR'},
'quic_config' : {'idle_timeout':5},
'remote_host' : 'mcserver',
'session_id'  : 'random',
'ssl_config'  : {'cert':'cert_name', 'tls':'TLS_v3'},
'tcp_config'  : {'nagle':false, 'idle_timeout':10},
'wifi_config' : {'low_latency':true}}
```

Figure 4.6: **Example MULTI Configuration.** *Applications can specify the priority for the protocols along with the configuration for each protocol.*

fields inside the header are the *Session ID* and *Sequence Number*. These allow MULTI to associate incoming connections and segments to their particular session. The *Version* field allows MULTI to distinguish between different versions of MULTI. This field allows newer versions of MULTI to remain compatible with earlier versions of MULTI. If a vulnerability has been found in a particular version, it can be mitigated by choosing a non-vulnerable version.

In Figure 4.6 we show a sample configuration for MULTI. The configuration example is inspired by URLSession, which allows fetching URLs with different connection configurations [102].

A more detailed explanation of the configuration can be found in Publication V. The configuration can be divided into several main parts. First, the *connection_priority* defines which transport layer protocols over which

interfaces should be used. Each connection entry has three parts, namely transport protocol, IP version, and a list of network interfaces to use. How the connections are opened and used are defined in the subsequent options. For example, the *multi_config* defines how the connections should be established and used.

In the example, the connections are opened simultaneously, and a Round-Robin scheduler is to be used to distribute the data packets over the connections The example also provides configuration options for the protocols to be used. In the example, MPTCP is set to prefer the Round-Robin scheduler, and the QUIC timeout is set to five seconds.

4.2.2 MULTI Evaluation

Using a multipath protocol always brings overheads. The additional headers take space and the schedulers who decide which connections to use cause overheads in scheduling the traffic [96]. As these are inherent to any multipath system, MULTI also suffers from them. The main details of the evaluation can be found in Publication V, including connection establishment times and duration to transfer both small and large files. Here, we focus on aggregating different transport over wireless links.

For evaluating MULTI, we use the testbed shown in Figure 4.7. The testbed contains a laptop with multiple network interfaces, namely Ethernet, Wi-Fi, and tethered 5G through a mobile phone. Each of the wired network links shown in Figure 4.7 is at least 1 Gbps, with the server using a 10 Gbps link to the network. We use 802.11ac for the Wi-Fi, reaching 200 Mbps network speeds. The 5G connection reached 190 Mbps when downloading, and 35 Mbps when uploading. This eliminates link bottlenecks and allows the full utilization of the network.

Each connected link provides both IPv4 and IPv6 addresses to the laptop. While the Ethernet and Wi-Fi addresses are internal to the testbed,

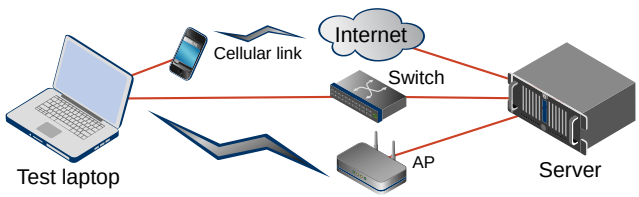
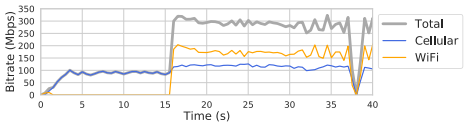
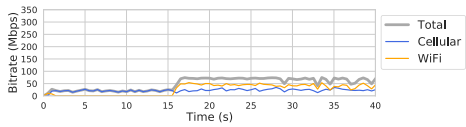


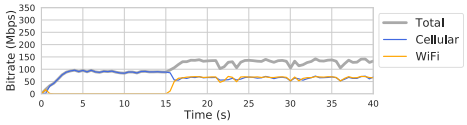
Figure 4.7: **MULTI testbed.** *The MULTI testbed contains a laptop with an Ethernet, a Wi-Fi, and a 5G connection, through which the laptop is connected to the server. This figure was taken from Publication V.*



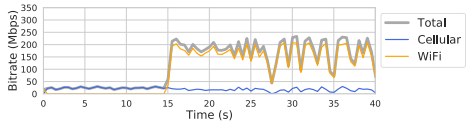
(a) **Two aggregated transport links over TCP.**



(b) **Aggregating, both QUIC.**



(c) **Two aggregated transport links, 5G-TCP, Wi-Fi-QUIC.**



(d) **Aggregating 5G-QUIC, Wi-Fi-TCP.**

Figure 4.8: **MULTI performance when aggregating connections.** *Figures (a) and (b) show how MULTI behaves when moving from 5G network to the range of a Wi-Fi network using TCP or QUIC. Figures (c) and (d) show how MULTI behaves when using either TCP over Wi-Fi and QUIC over 5G or vice versa. These figures are taken from Publication V.*

the 5G phone offers public addresses through a cellular operator. These allow us to evaluate both IPv4 and IPv6 with the MULTI system.

We studied how MULTI performs with different transport combinations in Publication V. We examined the performance using TCP and QUIC for the transport protocols, and Ethernet, Wi-Fi, and 5G for the network connections. We show here several key findings, and the detailed evaluation can be found in Publication V.

Figure 4.8 shows our results for aggregating our wireless links in the testbed. In this test, we began the test initially using only 5G and then aggregating the Wi-Fi. This simulates the case when the device is first only connected to the 5G network, and then moves into the Wi-Fi range.

For the first 15 seconds, we only used one transport for the data transfer. This was done to establish the baseline and allow transports to reach maximum speeds. At 15 seconds, we began aggregating the other transport with the original transport.

In the figures, we can see the benefits of aggregating streams. However, there are several notable observations to be made here. In Figures 4.9a and 4.9b, the total bandwidth is roughly the sum of separate streams. However, in Figure 4.8a, TCP stream over 5G takes a longer time to reach its sustained throughput, while the stream over Wi-Fi reaches it almost instantly. This is mainly due to the larger RTT over 5G link, 46 ms over 5G versus 2 ms over Wi-Fi. Another observation is that the Python asyncio

QUIC version we used cannot reach the capacity of the links. We believe this to be due to the nature of Python, *i.e.*, the userspace QUIC process performance is capped due to the single threaded implementation of our prototype and the signaling overheads caused by the TLS encryption and the RTT over 5G.

When we used different transport protocols in Figures 4.9c and 4.9d, we observed that in 4.9c the TCP stream over 5G loses some of its throughput when the QUIC over Wi-Fi is included. Still, the total throughput is larger than the individual stream.

We also observed some abnormal drops in the throughput in all tests shown here. While we could not reach any conclusive reason, we believe these could be due to various factors, including full network buffers and background traffic in the network.

The tests shown here, and the other tests in Publication V, highlight both benefits and disadvantages of aggregating different transport protocols. Clearly, the benefit of aggregating links allows us to reach higher throughput, but the asymmetry of the links can cause problems.

4.2.3 MULTI Discussion

MULTI exemplifies the benefit of using a transport library for handling multiconnectivity. As MULTI is agnostic to both the available networks and transport protocols, it can achieve connectivity regardless of the situation.

Another benefit of MULTI is that it hides the network operations from the application. While the application defines how it should be connected to the remote host, MULTI performs the connectivity establishment operations by itself. The application is provided with handles for reading and writing data, but beyond that, the application has no part in the communications. This allows MULTI to handle seamless multiconnectivity, as the five-tuple defining a single network connection is rendered meaningless from the application point of view. Under the hood, all connections have their own five-tuples. However, as long as there is a working five-tuple available, MULTI will be able to provide connectivity to the application even if some of the connections break.

MULTI does come with drawbacks. While deploying MULTI is straightforward as it operates in userspace, it is affected by the overheads of userspace. For example, the default Python asyncio implementation hinders the performance of MULTI, especially when using QUIC. Our current proof-of-concept prototype also requires both end-points to support MULTI. While this limits the current usage of MULTI, it is straightforward enough to add a protocol negotiation similar to MPTCP or TAPS

to MULTI. If the other end-point cannot use MULTI, this would allow MULTI to fall back to TCP or UDP. Finally, implementing an optimized scheduler for sending traffic over multiple transports requires more work. Our implementation uses simple schedulers, but a more intelligent scheduler that takes different characteristics of transport protocols and interfaces into account is needed. In any case, as MULTI operates in userspace, the schedulers are straightforward to upgrade.

4.3 Summary

In this chapter, we presented Meghna and MULTI, two different approaches for multiconnectivity from different perspectives. Meghna provides an SDN-based approach for multiconnectivity, in which a local SDN switch on the host device is used to steer traffic through various network interfaces based on the rules given by the remote SDN controller. MULTI is a session layer framework that acts as an umbrella for various existing multiconnectivity protocols and methods. MULTI also allows users and applications to compose requirements and preferences for used connections, for example, which protocols to use over which interfaces.

Both approaches have their benefits and disadvantages. While Meghna has more stringent requirements for the network infrastructure, MULTI, on the other hand, suffers the typical multipath protocol problems such as Head-of-Line blocking and increased scheduling latencies. Nonetheless, both have their benefits. Meghna does not carry the extra bandwidth costs of adding additional headers, while MULTI is agnostic to the underlying network and can determine which protocols are available.

Both Meghna and MULTI draw from the insights of the earlier work presented in this thesis. The design of MULTI attempts to be resilient to different middleboxes by using multiple different transport protocols to achieve end-to-end connectivity. Similarly, Meghna uses the lessons learned from SWIFT to take a step further and bring SDN to the host device itself.

Chapter 5

Conclusion

In this chapter, we summarize the research and outcomes of the work presented in this thesis. First, we revisit the research questions presented in the beginning of the thesis in Section 5.1. We then highlight the scientific contributions presented in this thesis in Section 5.2. Finally, we discuss the future work envisioned based on the findings of this thesis in Section 5.3.

5.1 Research Questions Revisited

RQ1 How do middleboxes hinder connectivity?

The home Internet connections still largely use NAT for IPv4. In Publication I, we studied 34 different home gateway devices and their NAT characteristics. This testbed was later increased in size to incorporate almost 100 devices. The testing done with both the original testbed and the larger set showed almost limitless variations in how different home gateways perform the NAT function. The testing showed that there is no way to predict how the devices will react to packets that are unknown to them; the devices may forward them unmodified, drop them, or perform some NAT operations to the packets. As such, they ossify the Internet as any new protocol will face a significant amount of problems traversing these NAT devices. Nonetheless, the study performed in Publication I has helped design new state-of-the-art transport protocols like QUIC. The study has brought to light the lowest common denominators of NAT binding timeouts, and how badly protocols are treated. The study also shows that unless a protocol behaves outwardly like TCP or UDP, it may not work well over the Internet.

RQ2 How can we make contemporary non-programmable hardware evolvable through software extensions?

In Publication II, we presented two methods for bringing SDN-based traffic management to Wi-Fi networks using legacy devices. The two methods, namely Intelligent and Thin APs, allow the usage of older, originally non-SDN-capable APs with SDN. The Thin AP method is viable for those APs that do not allow custom firmware or software installation, while the Intelligent AP is viable for, for example, *OpenWrt*-based APs.

There may be some performance loss with the Intelligent AP approach as the computing capacity of especially the older APs is not as powerful as newer APs, and running the *OVS* on AP consumes computing power. However, based on our testing, the performance hit of the *OVS* is not a major issue. The *OpenWrt*-based APs can also serve as small switches, so the benefits of the SDN are not limited just to the Wi-Fi.

Meanwhile, the performance of the Thin APs is largely unaffected by the Thin technique. As our changes do not directly change how the APs operate, the APs themselves do not take a performance hit. However, for the Thin APs, the main bottleneck will be the Ethernet connection to the external SDN switch. All traffic needs to traverse the external SDN switch, this indirection may cause the Ethernet link between the AP and the switch to become congested. Still, this bottleneck problem also affects new APs that support the higher speeds of Wi-Fi6 or newer standards.

We also exemplified how these devices can be used for network orchestration in Publication III. The PraNA framework uses the SWIFT SDN controller and both the Intelligent and the Thin AP techniques to support the PraNA network orchestrator. The fully SDN-capable network allows PraNA to achieve a full view of the network, and with the help of other controllers and network functions, protect the network using SDN-based traffic management.

PraNA also exemplifies the advantages gained from a common network orchestrator. Our networks are growing, and specific controllers control different parts of the networks. Without a common network orchestrator like PraNA, managing these networks is almost impossible.

RQ3 How can we offer multiconnectivity in a programmable network?

Seamless multiconnectivity is hard to achieve. One of the key problems of multiconnectivity is that the devices have only a limited view of the network state. The programmable networks, on the other hand, can achieve this view through the programmable switches and their controller, which can be leveraged to achieve better multiconnectivity.

The key component is the network controller, which allows the logically centralized approach to utilize the benefits of the full network view available to the controller. In Publication IV, we present a way for SDN to manage multiconnectivity. Meghna utilizes an SDN switch installed on the host devices to steer the traffic between the host and the Internet over the links that are most suitable at any given time.

The Meghna allows applications to be agnostic to the underlying network connections. This is achieved through the SDN switch on the device, which is assigned a static IP address from the home network. As long as the device is connected to the home network either locally or through a VPN tunnel, the Meghna can route the traffic through the home network, and the IP address that applications uses never changes. These allow Meghna to achieve seamless programmable multiconnectivity, with the caveat that Meghna requires a home network to route the traffic.

RQ4 How can we achieve user-driven multiconnectivity over arbitrary networks?

Heterogeneous networks consist of different transport technologies and network policies, which make it impossible for a single multiconnectivity solution to fulfil all application needs. A typical router or a firewall is likely to reject or drop packets that it does not recognize. This can be due to different reasons, including unknown protocols or protocol options, or the device could not associate the packet with an existing connection. If we also include multiple networks from different service providers, the problem is even harder.

In Publication V, we presented MULTI, which combines different multiconnectivity solutions and is designed with the end-to-end principle in mind. By using multiple solutions, MULTI can determine which solutions work in a particular network or networks. Similarly, MULTI can use traditional transport protocols such as TCP as they currently operate; the necessary metadata for multiconnectivity is carried inside the userspace protocol carried inside the packet payloads. This allows MULTI to operate in environments where the network prevents protocols like QUIC or MPTCP from working properly and achieve multiconnectivity.

5.2 Scientific Contributions

In the works presented in this thesis, we provide a number of contributions to the network research community. First, our measurements and analysis of how different middleboxes behave provide better stepping stones than

just anecdotal evidence for new transport protocol development, and what challenges different middleboxes pose for multiconnectivity. Understanding how past decisions affect the Internet of today and why they were made is important. This knowledge will allow us to learn from the past to design better solutions in the future.

Second, we show how SDN can be deployed on existing, off-the-shelf equipment. The methods presented in Section 3.1 allow already deployed equipment to be converted into supporting SDN-based traffic management without major modifications. Using the existing hardware cuts down costs and increases the life span of the existing devices, as they do not need to be replaced with newer equipment and also allows converting wireless test environments to support SDN research. We also show how our approach can be used to orchestrate state-of-the-art networks that use a combination of legacy and new devices.

Third, our two approaches for multiconnectivity tackle it from different directions. Our Meghna system allows the network to decide which interfaces to use, allowing better service as the network state can be fully taken into account. On the other hand, MULTI explores how different multiconnectivity protocols and transports could be combined into a single solution. While there still remains a large amount of work to be done, MULTI highlights both the benefits and the problems that combining different interfaces and protocols will have. These insights will allow for better multiconnectivity solutions to be developed in the future.

5.3 Future Work

This thesis provides several stepping stones for programmable seamless multiconnectivity. The work presented in Chapter 3 allows existing networks to adopt a more programmable approach to network management and orchestration. Similarly, the work presented in Chapter 4 leverages on the insights gained from deploying SDN on existing networks and our proof-of-concept network orchestration.

The network orchestration could be extended beyond the normal network operations presented in Section 3.2.1. The modular system is not limited just to the network controllers, it could be extended to, for example, containers or similar services. The presented classification-based use case is straightforward to extend to encompass containers and other services, allowing our networks to become more secure.

Nonetheless, our solutions are not yet ready for wide-scale adoption. When we look at our solutions for multiconnectivity, there is a gap between

the SDN-based multiconnectivity and MULTI; namely, these solutions do not operate together. One clear way forward would be defining a local controller running on hosts that would act as the coordinator between the user, the applications, and the network. Here, we envision the local controller and the controllers in different networks to be able to exchange network state information and pass the requests from the users and applications to the network controllers, so that we could offer better connectivity. For example, the IETF Multi-Access Management Services (MAMS) could be used as a starting point [104].

MULTI also exposes other underlying problems of combining different transport technologies and protocols. As those have been designed individually, combining them is not always optimal. For example, using MULTI with QUIC can lead to encrypting the data flows twice, which in turn may cause load balancers and other network elements to send traffic to the wrong destination. Similarly, managing the Head-of-Line blocking and packet scheduling requires more work before MULTI can be adopted for wider use.

If we think about how future networks such as 6G would operate, there are venues where this kind of cooperation would be beneficial. Current networks are from the application point of view only pipes that provide network connectivity. If we can devise a way that would allow other benefits to be gained from the networks, for example, estimates on the current network capacity, this could benefit both the network operators and the users. If a network cannot provide the requested capacity due to congestion or other reasons, exchanging state information could allow the application to use alternative paths and allow the congested network to recover. However, this depends on how our access networks continue to evolve. Even now, the operators are removing landlines as offering wireless connectivity is easier and cheaper. It remains to be seen how this will affect multiconnectivity if there are only limited ways to connect to the Internet. Nonetheless, the work presented in this thesis shows that seamless programmable multiconnectivity can be achieved, but the work is not yet finished.

References

[1] V. Bajpai, S. Ahsan, J. Schönwälder, and J. Ott, "Measuring YouTube Content Delivery Over IPv6," *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 5, pp. 2–11, 2017.

[2] E. Pujol, P. Richter, and A. Feldmann, "Understanding the share of IPv6 traffic in a dual-stack ISP," in *Proceedings of the International Conference on Passive and Active Network Measurement*. Springer, 2017, pp. 3–16.

[3] M. Bauer, M. Boussard, N. Bui, J. D. Loof, C. Magerkurth, S. Meissner, A. Nettsträter, J. Stefa, M. Thoma, and J. W. Walewski, "IoT Reference Architecture," in *Enabling things to talk*. Springer, Berlin, Heidelberg, 2013, pp. 163–211.

[4] "Total Wi-Fi device shipments to surpass ten billion this month," http://www.wi-fi.org/news-events/newsroom/total-wi-fi-device-shipments-to-surpass-ten-billion-this-month, January 2015, [Accessed 2021-12-27].

[5] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://rfc-editor.org/rfc/rfc9000.txt

[6] "Transmission Control Protocol," RFC 793, Sep. 1981. [Online]. Available: https://rfc-editor.org/rfc/rfc793.txt

[7] "User Datagram Protocol," RFC 768, Aug. 1980. [Online]. Available: https://rfc-editor.org/rfc/rfc768.txt

[8] O. Bonaventure and S. Seo, "Multipath TCP deployments," *IETF Journal*, vol. 12, no. 2, pp. 24–27, 2016.

[9] S. Hätönen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti, and M. Kojo, "An Experimental Study of Home Gateway Characteris-

tics," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, 2010, pp. 260–266.

[10] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, "Is it still possible to extend TCP?" in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, 2011, pp. 181–194.

[11] L. D'Acunto, J. Pouwelse, and H. Sips, "A measurement of NAT and firewall characteristics in peer-to-peer systems," in *Proceedings of the 15th ASCI Conference*, vol. 5031. Citeseer, 2009, pp. 1–5.

[12] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-Peer Communication Across Network Address Translators," in *Proceedings of the USENIX Annual Technical Conference, General Track*, 2005, pp. 179–192.

[13] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 277–288, 1984.

[14] P. F. Tsuchiya and T. Eng, "Extending the IP Internet through address reuse," *ACM SIGCOMM Computer Communication Review*, vol. 23, no. 1, pp. 16–33, 1993.

[15] K. B. Egevang and P. Francis, "The IP Network Address Translator (NAT)," RFC 1631, May 1994. [Online]. Available: https://rfc-editor.org/rfc/rfc1631.txt

[16] K. B. Egevang and P. Srisuresh, "Traditional IP Network Address Translator (Traditional NAT)," RFC 3022, Jan. 2001. [Online]. Available: https://rfc-editor.org/rfc/rfc3022.txt

[17] L. Prehn, F. Lichtblau, and A. Feldmann, "When wells run dry: the 2020 IPv4 address market," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 2020, pp. 46–54.

[18] V. Cardellini, M. Colajanni, and P. S. Yu, "Dynamic load balancing on web-server systems," *IEEE Internet Computing*, vol. 3, no. 3, pp. 28–39, 1999.

[19] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.

[20] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A Clean Slate 4D Approach to Network Control and Management," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 5, pp. 41–54, 2005.

[21] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[22] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks." in *Lisa*, vol. 99, no. 1, 1999, pp. 229–238.

[23] C. Paasch and O. Bonaventure, "Multipath TCP," *Communications of the ACM*, no. 4, p. 51–57, 2014. [Online]. Available: https://doi.org/10.1145/2578901

[24] K. Ashton *et al.*, "That 'Internet of Things' Thing," *RFID journal*, vol. 22, no. 7, pp. 97–114, 2009.

[25] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[26] S. Li, L. Da Xu, and S. Zhao, "The Internet of Things: A Survey," *Information Systems Frontiers*, vol. 17, no. 2, pp. 243–259, 2015.

[27] J. Mineraud and S. Tarkoma, "Toward interoperability for the Internet of Things with meta-hubs," *arXiv preprint arXiv:1511.08063*, 2015.

[28] "Zigbee," visited 2021-10-25. [Online]. Available: https://csa-iot.org/

[29] "Z-Wave," visited 2021-10-25. [Online]. Available: https://z-wavealliance.org/

[30] J. C. Haartsen, "The Bluetooth radio system," *IEEE Personal Communications*, vol. 7, no. 1, pp. 28–36, 2000.

[31] O. Mazhelis, E. Luoma, and H. Warma, "Defining an internet-of-things ecosystem," in *Internet of Things, Smart Spaces, and Next Generation Networking*, S. Andreev, S. Balandin, and Y. Koucheryavy, Eds. Springer Berlin Heidelberg, 2012, pp. 1–14.

[32] J. Mineraud, O. Mazhelis, X. Su, and S. Tarkoma, "A gap analysis of Internet-of-Things platforms," *Computer Communications*, vol. 89, pp. 5–16, 2016.

[33] "Home Assistant," visited 2021-10-25. [Online]. Available: https://www.homeassistant.io/

[34] I. Hafeez, A. Y. Ding, L. Suomalainen, A. Kirichenko, and S. Tarkoma, "Securebox: Toward safer and smarter iot networks," in *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*, 2016, pp. 55–60.

[35] I. Hafeez, M. Antikainen, A. Y. Ding, and S. Tarkoma, "IoT-KEEPER: Detecting Malicious IoT Network Activity Using Online Traffic Analysis at the Edge," *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 45–59, 2020.

[36] C. Jennings and F. Audet, "Network Address Translation (NAT) Behavioral Requirements for Unicast UDP," RFC 4787, Jan. 2007. [Online]. Available: https://rfc-editor.org/rfc/rfc4787.txt

[37] B. Ford, S. Guha, K. Biswas, S. Sivakumar, and P. Srisuresh, "NAT Behavioral Requirements for TCP," RFC 5382, Oct. 2008. [Online]. Available: https://rfc-editor.org/rfc/rfc5382.txt

[38] S. Guha, B. Ford, S. Sivakumar, and P. Srisuresh, "NAT Behavioral Requirements for ICMP," RFC 5508, Apr. 2009. [Online]. Available: https://rfc-editor.org/rfc/rfc5508.txt

[39] S. Hätönen, Y. Li, and M. Kojo, *Study of Middle-box Behavior on Network Layer Protocols*, Department of Computer Science, Series of Publications C. University of Helsinki, Finland, 2012, no. C-2012-3.

[40] S. Hätönen, Y. Li, and M. Kojo, *Study of Middle-box Behavior on Transport Layer Protocols*, Department of Computer Science, Series of Publications C. University of Helsinki, Finland, 2012, no. C-2012-4.

[41] R. T. Braden, "Requirements for Internet Hosts - Communication Layers," RFC 1122, Oct. 1989. [Online]. Available: https://rfc-editor.org/rfc/rfc1122.txt

[42] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. R. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W. Chang, and Z. Shi, "The QUIC Transport Protocol: Design and Internet-Scale Deployment," in *SIGCOMM '17*. ACM, 2017, p. 183–196. [Online]. Available: https://doi.org/10.1145/3098822.3098842

[43] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson, "Netalyzr: Illuminating the edge network," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, 2010, pp. 246–259.

[44] "Cisco Wireless LAN Controller," visited 2021-10-25. [Online]. Available: https://www.cisco.com/c/en/us/products/wireless/wireless-lan-controller/index.html

[45] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[46] K.-K. Yap, M. Kobayashi, R. Sherwood, T.-Y. Huang, M. Chan, N. Handigol, and N. McKeown, "OpenRoads: Empowering Research in Mobile Networks," *SIGCOMM CCR*, vol. 40, no. 1, pp. 125–126, Jan. 2010.

[47] Y. Yiakoumis, M. Bansal, A. Covington, J. van Reijendam, S. Katti, and N. McKeown, "BeHop: A testbed for dense WiFi networks," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 18, no. 3, pp. 71–80, 2015.

[48] M. Yan, J. Casey, P. Shome, A. Sprintson, and A. Sutton, "ÆtherFlow: Principled Wireless Support in SDN," in *Proceedings of the 2015 IEEE 23rd International Conference on Network Protocols (ICNP)*, 2015, pp. 432–437.

[49] J. Schulz-Zander, C. Mayer, B. Ciobotaru, S. Schmid, and A. Feldmann, "OpenSDWN: Programmatic Control over Home and Enterprise WiFi," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015, pp. 1–12.

[50] "SSID Overhead - How Many Wi-Fi SSIDs Are Too Many?" visited 2021-12-31. [Online]. Available: http://revolutionwifi.blogspot.com/2013/10/ssid-overhead-how-many-wi-fi-ssids-are.html

[51] S. Hätönen, P. Savolainen, A. Rao, H. Flinck, and S. Tarkoma, "Off-the-Shelf Software-Defined Wi-Fi Networks," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, p. 609–610. [Online]. Available: https://doi.org/10.1145/2934872.2959071

[52] "hostapd: IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator," visited 2021-12-25. [Online]. Available: http://w1.fi/hostapd/

[53] "IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks," *IEEE Std 802.1Q-1998*, pp. 1–214, 1999.

[54] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The Design and Implementation of Open vSwitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 117–130.

[55] B. A. Dykes and D. R. McPherson, "VLAN Aggregation for Efficient IP Address Allocation," RFC 3069, Feb. 2001. [Online]. Available: https://rfc-editor.org/rfc/rfc3069.txt

[56] M. Foschiano and S. HomChaudhuri, "Cisco Systems' Private VLANs: Scalable Security in a Multi-Client Environment," RFC 5517, Feb. 2010. [Online]. Available: https://rfc-editor.org/rfc/rfc5517.txt

[57] S. Hätönen, P. Savolainen, A. Rao, H. Flinck, and S. Tarkoma, "SWIFT: Bringing SDN-Based Flow Management to Commodity Wi-Fi Access Points," in *Proceedings of the 2018 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, 2018, pp. 1–9.

[58] T. Høiland-Jørgensen, C. A. Grazia, P. Hurtig, and A. Brunstrom, "Flent: The FLExible Network Tester," in *Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools*, 2017, p. 120–125. [Online]. Available: https://doi.org/10.1145/3150928.3150957

[59] J. Gettys and K. Nichols, "Bufferbloat: Dark Buffers in the Internet," *Queue*, vol. 9, no. 11, pp. 40:40–40:54, Nov. 2011.

[60] S. Cheshire and M. Krochmal, "Multicast DNS," RFC 6762, Feb. 2013. [Online]. Available: https://rfc-editor.org/rfc/rfc6762.txt

[61] S. A. Alabady, F. Al-Turjman, and S. Din, "A novel security model for cooperative virtual networks in the IoT era," *International Journal of Parallel Programming*, vol. 48, no. 2, pp. 280–295, 2020.

[62] A. Pouttu, F. Burkhardt, C. Patachia, L. Mendes, G. R. Brazil, S. Pirttikangas, E. Jou, P. Kuvaja, F. T. Finland, M. Heikkilä *et al.*, "6G White Paper on Validation and Trials for Verticals towards 2030's," *6G Research Visions*, no. 4, 2020.

[63] T. Zachariah, N. Klugman, B. Campbell, J. Adkins, N. Jackson, and P. Dutta, "The Internet of Things Has a Gateway Problem," in *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, 2015, pp. 27–32.

[64] "Aruba Networks Mobility Controller," visited 2021-10-25. [Online]. Available: https://www.arubanetworks.com/products/networking/controllers

[65] "Ubiquiti Networks UniFi," visited 2021-10-25. [Online]. Available: https://www.ubnt.com/enterprise

[66] S. Hätönen, I. Hafeez, J. Mineraud, A. Rao, and S. Tarkoma, "Orchestrating Intelligent Network Operations in Programmable Networks," in *Proceedings of the 2021 IEEE Conference on Standards for Communications and Networking (CSCN)*. IEEE, 2021, pp. 148–154.

[67] N. F. Saraiva de Sousa, D. A. Lachos Perez, R. V. Rosa, M. A. Santos, and C. Esteve Rothenberg, "Network service orchestration: A survey," *Computer Communications*, vol. 142-143, pp. 69–94, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366418309502

[68] M. Fedor, M. L. Schoffstall, J. R. Davin, and D. J. D. Case, "Simple Network Management Protocol (SNMP)," RFC 1157, May 1990. [Online]. Available: https://rfc-editor.org/rfc/rfc1157.txt

[69] "Ansible," visited 2021-10-25. [Online]. Available: https://www.ansible.com/

[70] S. L. Keoh, S. S. Kumar, and H. Tschofenig, "Securing the Internet of Things: A Standardization Perspective," *IEEE Internet of things Journal*, vol. 1, no. 3, pp. 265–275, 2014.

[71] F. Restuccia, S. D'Oro, and T. Melodia, "Securing the Internet of Things in the Age of Machine Learning and Software-Defined Networking," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4829–4842, 2018.

[72] M. Miettinen, S. Marchal, I. Hafeez, N. Asokan, A.-R. Sadeghi, and S. Tarkoma, "IoT SENTINEL: Automated Device-Type Identification for Security Enforcement in IoT," in *Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2177–2184.

[73] N. Aluthge, "IoT device fingerprinting with sequence-based features," 2018. [Online]. Available: https://helda.helsinki.fi/handle/10138/234247

[74] X. Li, M. Samaka, H. A. Chan, D. Bhamare, L. Gupta, C. Guo, and R. Jain, "Network slicing for 5g: Challenges and opportunities," *IEEE Internet Computing*, vol. 21, no. 5, pp. 20–27, 2017.

[75] S. Zhang, "An overview of network slicing for 5g," *IEEE Wireless Communications*, vol. 26, no. 3, pp. 111–117, 2019.

[76] "Nokia - Flexi zone," visited 2021-10-25. [Online]. Available: https://www.nokia.com/networks/mobile-networks/flexi-zone/

[77] "OpenAir Cloud RAN (C-RAN)," visited 2021-10-25. [Online]. Available: https://openairinterface.org/use-cases/cloud-ran-c-ran/

[78] "MQTT," visited 2021-10-25. [Online]. Available: https://www.mqtt.org/

[79] L. Pajevic, V. Fodor, and G. Karlsson, "Revisiting the modeling of user association patterns in a university wireless network," in *Proceedings of the 2018 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2018, pp. 1–6.

[80] P. Bahl, A. Adya, J. Padhye, and A. Wolman, "Reconsidering Wireless Systems with Multiple Radios," *SIGCOMM CCR*, vol. 34, no. 5, pp. 39–46, Oct. 2004.

[81] K.-K. Yap, T.-Y. Huang, M. Kobayashi, Y. Yiakoumis, N. McKeown, S. Katti, and G. Parulkar, "Making Use of All the Networks Around Us: A Case Study in Android," *SIGCOMM CCR*, vol. 42, no. 4, pp. 455–460, Sep. 2012.

[82] C. A. Gizelis and D. D. Vergados, "A survey of pricing schemes in wireless networks," *IEEE Communications Surveys & Tutorials*, vol. 13, no. 1, pp. 126–145, 2010.

[83] E. Peltonen, E. Lagerspetz, P. Nurmi, and S. Tarkoma, "Energy Modeling of System Settings: A Crowdsourced Approach," in *Proceedings of the 2015 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 2015, pp. 37–45.

[84] E. Khorov, I. Levitsky, and I. F. Akyildiz, "Current status and directions of ieee 802.11be, the future wi-fi 7," *IEEE Access*, vol. 8, pp. 88 664–88 688, 2020.

[85] S. Hätönen, M. T. I. ul Huque, A. Rao, G. Jourjon, V. Gramoli, and S. Tarkoma, "An SDN Perspective on Multi-connectivity and Seamless Flow Migration," *IEEE Networking Letters*, vol. 2, no. 1, pp. 19–22, 2019.

[86] K. Ramachandran, S. Rangarajan, and J. C. Lin, "Make-Before-Break MAC Layer Handoff in 802.11 Wireless Networks," in *Proceedings of the 2006 IEEE International Conference on Communications*, vol. 10. IEEE, 2006, pp. 4818–4823.

[87] M. T. I. ul Huque, G. Jourjon, and V. Gramoli, "Garbage Collection of Forwarding Rules in Software Defined Networks," *IEEE Communications Magazine*, vol. 55, no. 6, pp. 39–45, 2017.

[88] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for Network Update," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 323–334, 2012.

[89] D. M. F. Mattos, O. C. M. B. Duarte, and G. Pujolle, "Reverse Update: A Consistent Policy Update Scheme for Software-Defined Networking," *IEEE Communications Letters*, vol. 20, no. 5, pp. 886–889, 2016.

[90] A. Rice and S. Hay, "Decomposing power measurements for mobile devices," in *Proceedings of the 2010 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2010, pp. 70–78.

[91] C. Perkins, "Mobile IP," *IEEE Communications Magazine*, vol. 35, no. 5, pp. 84–99, 1997.

[92] R. Moskowitz, T. Heer, P. Jokela, and T. R. Henderson, "Host Identity Protocol Version 2 (HIPv2)," RFC 7401, Apr. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7401.txt

[93] P. Nikander, A. Gurtov, and T. R. Henderson, "Host Identity Protocol (HIP): Connectivity, mobility, multi-homing, security, and privacy over IPv4 and IPv6 networks," *IEEE Communications Surveys & Tutorials*, vol. 12, no. 2, pp. 186–204, 2010.

[94] A. Ford, C. Raiciu, M. J. Handley, O. Bonaventure, and C. Paasch, "TCP Extensions for Multipath Operation with Multiple Addresses," RFC 8684, Mar. 2020. [Online]. Available: https://rfc-editor.org/rfc/rfc8684.txt

[95] Y. Cui, T. Li, C. Liu, X. Wang, and M. Kühlewind, "Innovating Transport with QUIC: Design Approaches and Research Challenges," *IEEE Internet Computing*, vol. 21, no. 2, pp. 72–76, 2017.

[96] Q. De Coninck and O. Bonaventure, "Multipath QUIC: Design and Evaluation," in *Proceedings of the 13th International Conference on Emerging Networking Experiments and Technologies*, 2017, pp. 160–166.

[97] T. Pauly, B. Trammell, A. Brunstrom, G. Fairhurst, C. Perkins, P. S. Tiesel, and C. A. Wood, "An Architecture for Transport Services," Internet Engineering Task Force, Internet-Draft draft-ietf-taps-arch-11, Jul. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-taps-arch-11

[98] N. Khademi, D. Ros, M. Welzl, Z. Bozakov, A. Brunstrom, G. Fairhurst, K.-J. Grinnemo, D. Hayes, P. Hurtig, T. Jones, S. Mangiante, M. Tuxen, and F. Weinrank, "NEAT: A Platform- and Protocol-Independent Internet Transport API," *IEEE Communications Magazine*, vol. 55, no. 6, pp. 46–54, 2017.

[99] P. Hurtig, S. Alfredsson, A. Brunstrom, K. Evensen, K.-J. Grinnemo, A. F. Hansen, and T. Rozensztrauch, "A NEAT Approach to Mobile Communication," in *Proceedings of the Workshop on Mobility in the Evolving Internet Architecture*, 2017, p. 7–12. [Online]. Available: https://doi.org/10.1145/3097620.3097622

[100] D. Schinazi and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency," RFC 8305, Dec. 2017. [Online]. Available: https://rfc-editor.org/rfc/rfc8305.txt

[101] S. Hätönen, A. Rao, and S. Tarkoma, "Programmable Session Layer MULTI-Connectivity," *IEEE Access*, vol. 10, pp. 5736–5752, 2021.

[102] Apple Inc., "URLSession - Apple Developer Documentation," https://developer.apple.com/documentation/foundation/urlsession, [accessed 2020-08-26].

[103] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[104] S. Kanugovi, F. Baboescu, J. Zhu, J. Mueller, and S. Seo, "Multiple Access Management Services Multi-Access Management Services (MAMS)," RFC 8743, Mar. 2020. [Online]. Available: https://rfc-editor.org/rfc/rfc8743.txt

Appendix A

List of Middleboxes

The following middleboxes were used in the home gateway study presented in Publication I.

Vendor	**Model**	**Firmware**	**Tag**
A-Link	WNAP	e2.0.9A	*al*
Apple	Airport Express	7.4.2	*ap*
Asus	RT-N15	2.0.1.1	*as1*
Belkin	Wireless N Router	F5D8236-4_WW_3.00.02	*be1*
	Enhanced N150	F6D4230-4_WW_1.00.03	*be2*
Buffalo	WZR-AGL300NH	R1.06/B1.05	*bu1*
D-Link	DIR-300	1.03	*dl1*
	DIR-300	1.04	*dl2*
	DI-524up	v1.06	*dl3*
	DI-524	v2.0.4	*dl4*
	DIR-100	v1.12	*dl5*
	DIR-600	v2.01	*dl6*
	DIR-615	v4.00	*dl7*
	DIR-635	v2.33EU	*dl8*
	DI-604	v3.09	*dl9*
	DI-713P	2.60 build 6a	*dl10*
Edimax	6104WG	2.63	*ed*
Jensen	Air:Link 59300	1.15	*je*
Linksys	BEFSR41c2	1.45.11	*ls1*
	WR54G	v7.00.1	*ls2*
	WRT54GL v1.1	v4.30.7	*ls3*
	WRT54GL-EU	v4.30.7	*ls5*
	WRT54G	OpenWRT RC5	*owrt*
	WRT54GL v1.1	tomato 1.27	*to*

Vendor	**Model**	**Firmware**	**Tag**
Netgear	RP614 v4	V1.0.2_06.29	*ng1*
	WGR614 v7	(1.0.13_1.0.13)	*ng2*
	WGR614 v9	V1.2.6_18.0.17	*ng3*
	WNR2000-100PES	v.1.0.0.34_29.0.45	*ng4*
	WGR614 v4	V5.0_07	*ng5*
Netwjork	54M	Ver 1.2.6	*nw1*
SMC Barricade	SMC7004VBR	R1.07	*smc*
Telewell	TW-3G	V7.04b3	*te*
Webee	Wireless N Router	e2.0.9D	*we*
ZyXel	P-335U	V3.60(AMB.2)C0	*zy1*

Table A.1: Home gateway models used in Publication I.

Appendix B

ACM SIGCOMM 2016 Demo

S. Hätönen, P. Savolainen, A. Rao, H.Flinck, and S. Tarkoma

Off-the-Shelf Software-Defined Wi-Fi Networks

In *Proceedings of the 2016 ACM SIGCOMM Conference*,
ACM, Florianopolis, Brazil, August 22-26, 2016, pages 609-610.

Off-the-Shelf Software-defined Wi-Fi Networks

Seppo Hätönen*, Petri Savolainen†, Ashwin Rao*, Hannu Flinck+, Sasu Tarkoma*†
*University of Helsinki, †Helsinki Institute for Information Technology HIIT, +Nokia Bell Labs

ABSTRACT

Wi-Fi networks were one of the first use-cases for Software-defined networking (SDN). However, to deploy a software-defined Wi-Fi network today, one has to rely on research prototypes with availability, documentation, hardware requirements, and scalability issues. To alleviate this situation, we demonstrate two simple techniques to bring SDN functionality to existing Wi-Fi networks and discuss their benefits and short-comings. Researchers can use our techniques to convert their existing Wi-Fi testbeds into software defined Wi-Fi testbeds. Our two techniques thus significantly lower the barrier-to-entry for deploying software-defined Wi-Fi networks.

CCS Concepts

•**Networks → Wireless local area networks;**

1. INTRODUCTION

Wi-Fi is becoming synonymous with Internet connectivity. However, in spite of its growing importance, Wi-Fi has received limited attention in the SDN community. Wi-Fi access points (APs) including those created using open-source solutions, such as *OpenWrt* [2] and *hostapd* [1], act as bridges or hubs between Wi-Fi clients. These Wi-Fi APs cannot take intelligent forwarding decisions because they cannot be programmed to process the complex match/action rules used by SDN switches such as Open vSwitch (*OVS*) [4].

In this paper, we present two simple techniques, namely Intelligent Edge and Thin Edge, that bring SDN functionality to Wi-Fi networks. These techniques leverage on wireless isolation and SDN switches such as *OVS* to simplify the integration of Wi-Fi networks and SDN.

SIGCOMM '16, August 22–26, 2016, Florianopolis, Brazil

DOI: http://dx.doi.org/10.1145/2934872.2959071

The Intelligent Edge technique empowers devices running *OpenWrt* with *OVS*. In contrast, the Thin Edge technique allows existing Wi-Fi APs which support wireless isolation to offload the flow management to SDN switches and be integrated as-is into SDN.
Our key contributions are as follows.

• Our two techniques enable existing SDN controllers to manage the flows traversing Wi-Fi networks built using off-the-shelf components. Using these components, *we significantly lower the barrier-to-entry to deploy and experiment on software-defined Wi-Fi networks.*

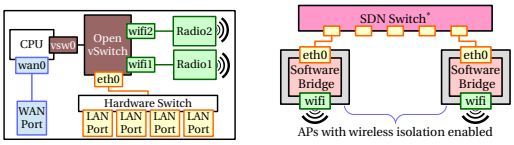
• Our Intelligent Edge technique allows SDN controllers to leverage the processing power of the APs for managing the edge of Wi-Fi networks. Furthermore, by combining *OVS* with *OpenWrt*, this technique opens avenues for SDN research on existing Wi-Fi testbeds which use devices running *OpenWrt*.

• Our Thin Edge technique offloads the flow management to either SDN switches or hosts running *OVS*. This technique is useful for managing Wi-Fi networks and testbeds which use APs and routers that either cannot support SDN, or are not powerful enough to run *OVS* and process the complex match-action rules supported by OpenFlow [3] and other SDN protocols.

The seminal work on bringing SDN to Wi-Fi networks was OpenRoads [7], which used protocols such as the Simple Network Management Protocol (SNMP) to manage the Wi-Fi APs. Based on the insights of OpenRoads, several solutions such as BeHop [8], ÆtherFlow [6], and OpenSDWN [5] have been proposed. However, these solutions suffer from deployability issues which make it hard to convert existing Wi-Fi networks and testbeds into software-defined Wi-Fi networks. We therefore focus on enabling SDN in Wi-Fi networks built using off-the-shelf components.

2. OUR SOLUTION

We present two techniques to create software defined Wi-Fi networks using off-the-shelf components. Our techniques build on Wireless Isolation, a feature that configures an AP to process all the wireless frames in the network stack instead of just bridging wireless clients associated with that AP. Wireless Isolation is supported by many APs including those running *OpenWrt*, and some enterprise APs such as the Cisco 1131 AP.



(a) **Intelligent Edge**. (b) **Thin Edge**.

Figure 1: **Our Intelligent Edge and Thin Edge techniques**. *For the Intelligent Edge, OpenWrt's software bridge is replaced with an OVS and wireless isolation is enabled on the AP; a patched* hostapd *uses the OVS while allowing Wi-Fi clients to encrypt their Wi-Fi frames. For the Thin Edge, Wireless Isolation is enabled on the APs and the forwarding decisions are taken on an external SDN switch; all the data encapsulated in Wi-Fi frames traverse the external SDN switch.*

Intelligent Edge. In this technique, we run *OVS* on an AP running *OpenWrt*. As shown in Figure 1(a), we replace the software bridge created by *OpenWrt* with an *OVS*; all the interfaces that used the software bridge now use the *OVS*. This step is not enough for the *OVS* to manage flows between two Wi-Fi clients associated with the AP. To manage this Wi-Fi traffic, we configure the Wi-Fi interfaces to send all the packets encapsulated inside the Wi-Fi frames to the *OVS*. We also use a patched *hostapd* to use the *OVS* while allowing Wi-Fi clients to encrypt their Wi-Fi frames. The Intelligent Edge technique also supports multiple (virtual) Wi-Fi networks on the same AP. Each virtual network appears as a logical Wi-Fi interface on *OpenWrt*, which is added to the *OVS*. These interfaces can be treated as virtual APs and the network flows are controlled by *OVS*.

Thin Edge. In this technique, an external SDN switch manages all the flows traversing the Wi-Fi APs. Thin Edge can be used with APs which support Wireless Isolation but do not support custom firmware such as *OpenWrt*, allow installing an SDN switch, or have hardware limitations such as CPU or flash memory size. Furthermore, as this technique only requires Wireless Isolation, many older devices which support *OpenWrt* can now be used for SDN research. At the same time, enterprise equipment which do not support custom firmware but support Wireless Isolation can also be integrated into these testbeds. Thus the Thin Edge can bring SDN to existing Wi-Fi networks or testbeds. In this technique, the AP acts as a remote Wi-Fi interface for the SDN switch and each Wi-Fi network of the AP becomes a port on that switch. A key shortcoming of this technique is that all the Wi-Fi traffic needs to traverse an external switch; the flows are not routed optimally and the network can become congested.

Discussion. Our techniques combine widely deployed technologies, Wireless Isolation and SDN switches such as *OVS*, and can therefore be used in almost all existing networks including Wi-Fi testbeds and also enterprise networks. Furthermore, our techniques enable networks running legacy Wi-Fi devices such as Wi-Fi testbeds to be powered by SDN, thus lowering the barrier-to-entry for doing SDN research on Wi-Fi networks.

3. DEMO DESCRIPTION

We use an *OpenWrt* AP with *OVS* for the Intelligent Edge, and an enterprise AP (Cisco 1131 Autonomous AP) without any firmware changes for the Thin Edge.

We demonstrate our techniques using Google Chromecast. A Chromecast device is typically visible to all clients in its same network, which may not be desirable in campus or enterprise networks. Our two techniques that leverage on SDN and Wireless Isolation restrict access to the Chromecast device to a given Wi-Fi client. In its default mode, Wireless Isolation blocks all Wi-Fi clients associated to a given AP from communicating with each other. We use SDN to extend Wireless Isolation to selectively enable a given Wi-Fi client to communicate with the Chromecast device.

We use Chromecast to exemplify that our techniques can be used to programmatically manage the Wi-Fi flows in smart spaces. Furthermore, we demonstrate that bringing SDN to Wi-Fi networks does not require specialized hardware, large changes in devices or software, and can be accomplished using existing devices whether they support custom firmware or not. Our techniques open existing Wi-Fi testbeds for experiments on next generation applications which leverage the programmability offered by SDN. The steps required for using our two techniques for bringing SDN to Wi-Fi networks are documented on the following web page: https://wiki.helsinki.fi/display/WiFiSDN/

To conclude, our techniques can be used to convert existing Wi-Fi networks and testbeds created using off-the-shelf devices into software-defined Wi-Fi networks.

Acknowledgments. This work has been supported by the Tekes 5GTrek research project, Digile Cyber Trust project, Academy of Finland Cloud Security Services project, and the Nokia Center for Advanced Research (NCAR).

4. REFERENCES

[1] hostapd: IEEE 802.11 AP. http://w1.fi/hostapd/.
[2] OpenWrt. https://openwrt.org/.
[3] McKeown, N., et al. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR.* (2008).
[4] Pfaff, B., et al. The Design and Implementation of Open vSwitch. In *Proc. of NSDI*, (2015).
[5] Schulz-Zander, J., et al. OpenSDWN: Programmatic Control over Home and Enterprise WiFi. In *Proc. of ACM SOSR '15* (2015).
[6] Yan, M., et al. Ætherflow: Principled Wireless Support in SDN. In *Proc. of IEEE ICNP* (2015).
[7] Yap, K.-K., et al. OpenRoads: Empowering Research in Mobile Networks. *ACM SIGCOMM CCR.* (2010).
[8] Yiakoumis, Y., et al. BeHop: A Testbed for Dense WiFi Networks. In *Proc. of WiNTECH '14* (2014).