UNIVERSITY OF CALIFORNIA SAN DIEGO

Self-Organizing Wireless Networks: Challenges, Design, and Implementation

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Electrical Engineering (Communication Theory and Systems)

by

Hans Chinghan Yu

Committee in charge:

Professor Ramesh R. Rao, Chair
Professor Dinesh Bharadia
Professor Sujit Dey
Professor James Friend
Professor Aaron Shalev

2021

The dissertation of Hans Chinghan Yu is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2021

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Thanks to whoever deserves credit for Blacks Beach, Porters Pub, and every coffee shop in San Diego.

Chapter 2, in full, is a reprint of the material as it appears in H. C. Yu, G. Quer and R. R. Rao, "Wireless SDN mobile ad hoc network: From theory to practice", *2017 IEEE International Conference on Communications (ICC)*, Paris, 2017. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, is a reprint of the material as it appears in H. C. Yu, K. Alhazmi and R. R. Rao, "Wi-Fi Roaming as a Location-based Service", *2020 IEEE International Conference on Communications (ICC)*, Dublin, Ireland, 2020. The dissertation author was the primary investigator and author of this paper.

VITA

2000-2004	B. S. in Electrical Engineering, National Taiwan University
2004-2006	M. S. in Communication Engineering, National Taiwan University
2012-2013	M. Eng. in Electrical and Computer Engineering, Cornell University
2014-2020	Ph. D. in Electrical Engineering (Communication Theory and Systems), University of California San Diego

PUBLICATIONS

H. C. Yu, G. Quer and R. R. Rao, "Wireless SDN mobile ad hoc network: From theory to practice", *2017 IEEE International Conference on Communications (ICC)*, Paris, 2017, pp. 1-7.

H. C. Yu, K. Alhazmi and R. R. Rao, "Wi-Fi Roaming as a Location-based Service", *2020 IEEE International Conference on Communications (ICC)*, Dublin, Ireland, 2020, pp. 1-7.

M. Meng, M. Dunna, H. Yu, S. Kuo, P.-H. P. Wang, D. Bharadia, and P. P. Mercier, "Improving the Range of WiFi Backscatter Via a Passive Retro-Reflective Single-Side-Band-Modulating MIMO Array and Non-Absorbing Termination", *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, Vol. 64, pp. 202-204.

ABSTRACT OF THE DISSERTATION

Self-Organizing Wireless Networks: Challenges, Design, and Implementation

by

Hans Chinghan Yu

Doctor of Philosophy in Electrical Engineering (Communication Theory and Systems)

University of California San Diego, 2021

Professor Ramesh R. Rao, Chair

A self-organizing network refers to a computer network that can configure, manage, and optimize itself. To achieve this goal, it first collects the network information for performance analysis at some control entities. After the control entities determine the optimized parameters, they push the settings back to the network again using the same channel they use to collect the information. Network optimization has been extensively studied over the last two decades. Most works ignore actual mechanisms for data collection and parameter update. This is partially true because device vendors, such as Cisco

or Juniper, have their own control interfaces. Unfortunately, these interfaces are generally not compatible with each other, and so a unified control protocol, OpenFlow, emerged.

OpenFlow defines a set of commands for data collection and parameter update for wired networks. However, wireless networks, or the most richly existing Wi-Fi, were not the primary targets of OpenFlow. In this dissertation, we first showed OpenFlow protocol could be migrated from wired networks to wireless networks with minor software modification and use to build a software-defined mobile ad hoc network (SD MANET) prototype.

Seeing the tight constraints in our SD MANET, we moved on to loosen these limitations. We came up with an approach to support most nowadays smart devices using the standard IEEE 802.11v/r protocol to replace the last mile connection between the devices and their associated access points. Our solution did not require any hardware or software modification on users' devices but only need the built-in IEEE 802.11v/r support, which can be found in many mainstream smart devices.

In addition to smart devices having rich network capability. We noticed another group of wireless connectivity devices but did not support advanced protocols such as IEEE 802.11v/r due to their simplified architectures. These devices include the ever-popular internet-of-things devices built on simple, low-power, and low-cost Wi-Fi system-on-chip solutions. Because Wi-Fi connection consumes lots of power, today's Wi-Fi IoT devices generally require external power support, and we can hardly see battery-powered Wi-Fi IoT devices. Seeing the demand, we design a non-coherent wake-up receiver that takes over the channel monitoring task of a power-consuming Wi-Fi interface so that the Wi-Fi

interface can be completely turned off. Our wake-up receiver consumes only 20-40 μW when monitoring the channel, whereas a general Wi-Fi interface can easily drain more than 100 mW. We also came up with an extendable finite-state-machine design that supports multicast wake-up. Multiple receivers can wake up through a single, carefully selected wake-up signal.

Practicality is the main idea of this dissertation. We have seen solutions with amazing performance but required either huge investment or complicated hardware design. In this dissertation, we chose the other way around by first analyzing the capabilities of existing frameworks and design solutions installed as overlays. Through this process, practicality is guaranteed.

Chapter 1

Introduction

Self-Organizing Network (SON) is an emerging concept in which a network manages itself with little human intervention. Two important factors establishing the base of a self-managed network are real-time control and real-time monitoring. Generally speaking, a self-managed network has a feedback control loop in which the network configures itself based on the network dynamics it collects. The control mechanism can be either centralized or decentralized depending on the network topology and the intended applications. Many enterprise-level networks already have centralized control mechanisms, whereas ad hoc/peer-to-peer (P2P) networks are mostly decentralized. SON offers some benefits. Take an in-home network as an example. Assume we start watching a Netflix drama on TV. A predefined rule is triggered when the controller on a router senses an increase in traffic throughput between Netflix's cloud server and the TV. The controller quickly applies congestion control algorithms to guarantee the bandwidth of Netflix streaming. When wireless communication is part of the network, given its mobile nature, channel character-

istics, and device locations, also need to be considered. Say a user would like to print out the photo from her smartphone. The controller uses the location information to determine the best available printer nearby and help the two devices set up their connection. Because wireless communication is already part of our daily lives, there is little reason to exclude wireless networks from the SON paradigm. Also, the above example implies that location information could serve as a key to optimization.

This dissertation is intended to show our answer to how a wireless SON (WSON) can be built on top of different types of existing networks with our software-oriented solutions and how the location information can be obtained at a close-to-zero cost. We agree a new hardware design will likely lead to even better performance, but it also creates extra cost and compatibility issues, preventing it from being widely accepted and deployed. On the other hand, software-based solutions generally have a much lower deployment cost and can more easily adapt to an existing setup. Throughout this dissertation, we weigh practicality as important as system performance, meaning that we plan not to argue how reasonably we can scarify some current features in exchange for a performance gain.

1.1 SON for Wireless Mobile Ad Hoc Networks

In the upcoming 5G scenario, we can expect an increase in local traffic, i.e., the traffic source and destination are close in proximity. Examples of such kinds include video casting. We stream a video clip from a smartphone to a TV and smart cities where vehicles learn and share local real-time traffic information to optimize speed and route

planning. In the current architecture, even local data are first sent to the cloud before they are redistributed. This approach is sub-optimal, especially in sharing time-sensitive information like local traffic. Not only because it suffers from a longer delay but also because it creates congestion at base stations, road-side units, or access points (APs). A solution to this problem would be direct device-to-device (D2D) communications [1], in which local nodes together form a wireless ad hoc network and run the same protocol.

One of the most well-known D2D protocols is ad hoc on-demand distant vector (AODV)[2], a reactive protocol finding path only when there is a demand. This approach inevitably suffers from long delay and pronto fail when there is a rapid topology change. On the other hand, the optimized link-state routing protocol (OLSR)[3] maintains a link-state among a node and its neighboring nodes by periodically exchanging control messages and updating the network topology at each node. A major disadvantage of this protocol is that a change might take a long time for nodes at the other end to learn in a large network. Also, frequent topology changes could create excessive control messages, resulting in significant overhead. A hybrid solution called zone routing protocol (ZRP) was proposed to reduce the number of control messages and balance the performance. ZRP divides nodes into clusters based on their geographical locations. The protocol assumes there is generally more intra-cluster traffic than inter-cluster traffic. Hence, nodes belong to the same cluster can exchange their data with an OLSR-like protocol, and inter-cluster traffic is transmitted over an AODV-like protocol. This hybrid approach significantly reduces the number of control messages because updated topology information is maintained only among the nodes within each cluster, and not so surprisingly, a significant delay is

introduced when a packet is destined to a node in a different cluster because the new packet needs to be searched reactively.

The above findings show that we cannot entirely rely on reactive path-finding if we want an acceptable delay. In the meantime, we also need to limit the use of a proactive path-finding approach in a large network to avoid excessive overhead. One might think that there might be an optimal network size in terms of the number of nodes to balance traffic delay and overhead. We chose not to follow the same thought process but have found a solution by using the concept of software-defined networks (SDN),[4] in which we put control messages and data in different network planes and made the size of the network more scalable.

Admittedly, SDN needs two different networks instead of one, and this is not a fair comparison to existing D2D networks. However, we believe this assumption is far from impractical, providing many smart devices today have two or more network interfaces. For example, we can route control messages over cellular networks with a local controller installed at a base station, forming a star-topology network and let all the local traffic transmitted over the D2D network. We did see a significant performance gain in our experiments as we bench-marked our wireless mobile ad hoc network (MANET) [5] against an OLSR counterpart.

A major reason why we could realize our software-defined MANET (SD-MANET) in this manner was that we confirmed the existence of a one-to-one mapping between an Ethernet frame and Wi-Fi ad hoc frame, i.e., an Ethernet frame can be converted to its Wi-Fi ad hoc version and vice versa without losing any information. This mapping bridges

the gap between wired and wireless networks, and we were able to port OpenFlow [6] SDN protocol to wireless networks with minor fixes. The implementation is present in Chapter 2.

1.2 SON for Infrastructure Wi-Fi Networks

After explaining the details of building our SD-MANET in Chapter 2, we focus on porting the SDN paradigm to infrastructure mode Wi-Fi in Chapter 3 because the infrastructure mode Wi-Fi is more widely seen today. In contrast, the ad hoc mode is seldom used. We soon encountered two challenges. The first roadblock we had was no one-to-one relation between an Ethernet frame and an infrastructure mode Wi-Fi frame. Enforcing similar conversions in our SD-MANET design would violate the standard, making our design complicated and less compatible with existing devices. We decided not to follow this path but to dedicate our time to a standard-compatible software-oriented solution. The second challenge was that unlike nodes in ad hoc networks in which beacons are broadcasted periodically, station devices (STA) in infrastructure mode Wi-Fi do not broadcast beacons, making it hard to estimate their locations.

To deal with location issues, we made a device that generates expected signal patterns to identify its neighboring nodes. One way to have a device make sound is to disassociate it from the AP intentionally. After being disassociated, the device will start looking for other APs for connection by sending probe requests on all the channels, giving us a hint for triangulation. However, during this AP probing period, all the ongoing traffic

has to stop. An all-channel probe could take up to several hundred milliseconds, long enough to break VoIP calls or video conferencing[7, 8].

A more practical solution would be not to disassociate the device from the AP but still trigger the probe. We found that the 802.11v [9] BTM control frame best fits our needs and becomes a key to our goal. By sending unsolicited BTM control frames in an orchestrated way through the associated AP, the controller suggested a list of candidate APs, triggering the AP probes. Because almost all current smart devices support compatible with the 802.11v standard, our solution does not require any additional hardware or software support.

Our solution consists of two parts. First, we could obtain real-time location information. Then we use it as an input to the controller. Combining our Wi-Fi tracking technique with the existing Bluetooth Low-Energy (BLE) tracking one, a radio-based contact tracing system that helps against the ongoing COVID-19 pandemic would be possible.

1.3 SON and Wi-Fi Internet of Things Devices

Now we have covered solutions for both ad hoc mode and infrastructure mode Wi-Fi. Another group of devices does not either form ad hoc networks, nor do they support 802.11v protocol. For example, Internet of Things (IoT) devices such as Wi-Fi switches and smart plugs. These devices are built with micro-controllers and only support infrastructure mode Wi-Fi in general for simplicity.

In addition to Wi-Fi, BLE and LoRa[10] are two other popular radio standards for IoT communications. These standards are designed for low-power applications because they generally consume much less power than Wi-Fi and are more suitable for battery-powered scenarios. However, to collect data from these devices, we need to build a new infrastructure, which might not be cost-efficient because they are unlikely to have a high traffic demand. One way to resolve this challenge is to try to make a Wi-Fi interface consume less power by introducing a standard-compatible wake-up receiver circuit (WuRx) as a front end to the original Wi-Fi interface. Such a circuit will turn on the Wi-Fi interface only after seeing a specific radio pattern in a Wi-Fi channel. Based on our measurements, the circuit consumes only 20-40 uW of power, whereas a general Wi-Fi interface consumes 100-150 mW.

In our design, a wake-up radio pattern can be generated through the Wi-Fi frame injection technique from a user application with compatible hardware and does not require any hardware change. In our experiments, we confirmed that at least two brands of Wi-Fi cards could be used to wake up the circuit.

The COVID-19 pandemic has largely reshaped our life. People are strongly encouraged to practice social distancing and wear masks. We have yet to know when a vaccine will be available and put an end to this pandemic. Many people started working remotely when they were told to shelter at home, and they stayed connected to their communities and the whole world through computer networks.

As network traffic increases, problems and challenges that have been overlooked for a long time are hungry for solutions. Through documenting my answers in this dis-

sertation, I think it will, hopefully in large, at least in part, light up a way to a better network design in the future.

Chapter 2

Wireless SDN Mobile Ad Hoc Network

As mentioned in Section 1.1, OpenFlow protocol [6] was originally designed for wired networks and did not consider frame structure or mobile characteristics of wireless networks. However, its well-defined application program interfaces (APIs) and cross-layer support make it suitable for dealing with wireless networks' fast-changing environment. This chapter starts by identifying our building blocks and then putting these building blocks to build the whole system. First, let's look into the format differences between an Ethernet [11] frame and a Wi-Fi [9] frame.

2.1 Wi-Fi Frame Structure

Unlike wired links that electromagnetic signals propagate in copper wires and one link only goes to one destination, wireless connections use a shared medium. Of course, we can put links on different channels to be physically apart like their wired counterpart, limiting the number of sender/recipient pairs and resulting in inefficiency. Alternatively, people use addresses as keys to identify links. Each node has a unique address called the medium access (MAC) layer address. It should only capture and process messages with an address related to itself while ignoring all others. In Wi-Fi, messages are packed into the frame format whose header holds the addresses and control parameters necessary for the recipient to process correctly.

There are two different Wi-Fi service modes: 1) basic service set (BSS) mode is commonly known as infrastructure mode, whereas 2) independent BSS (IBSS) mode is called ad hoc mode. This section will first cover our solution to extending SDN to ad hoc mode Wi-Fi networks.

Figure 2.1 shows headers of an Ethernet frame, ad hoc Wi-Fi frame, infrastructure Wi-Fi frames, and mesh frames. Although they look slightly different, there are still common fields shared by all four kinds of frames. For example, both the Ethernet frame and ad hoc Wi-Fi frame consist of only two addresses in their headers, whereas the infrastructure Wi-Fi frame has three addresses. An infrastructure Wi-Fi frame's additional address is used to determine the relationship between an AP and a station node (STA). This is because the AP is may not be the final destination of a frame, but it is more

likely to be an intermediate node. The third address is necessary because there might be multiple APs within the range of access, and the STA has to use the third address field to identify the targeted AP for frame sending. Similarly, there could be multiple STAs within the range of a single AP. The AP also uses the third address to identify the receiving STA without touching the source and destination addresses.

We first focused on the shared fields between the Ethernet frame and ad hoc Wi-Fi one. We quickly confirmed if the BSSID is known in advance, an Ethernet frame can be translated into an ad hoc Wi-Fi frame without losing any information. A backward conversion is also possible.

Figure 2.2 shows the forward and the backward conversion of frames. Because of this possibility, it is possible to replace a wired link with an ad hoc Wi-Fi link in a network with minor modifications. In fact, in Linux systems, this conversion is done automatically when necessary in the kernel. More detail will be covered in Section 2.3. This gives us a perfect chance to extend the OpenFlow protocol to wireless networks.

2.2 MAC Address Rewriting

We mentioned that wireless senders use MAC addresses to identify frame recipient in Section 2.1. To inform the sender that the frame has correctly arrived at the recipient, the recipient replies an acknowledgment (ACK) by putting the sender's MAC address in it. Therefore, if the sender does not hear the ACK associated with a particular frame in time, it assumes that the frame was lost and will resend it again. This feedback mechanism

provides reliability to wireless communications.

In many nowadays Wi-Fi interfaces, due to timing constraints, this feedback mechanism is usually implemented at the hardware or firmware level to avoid a possibly long delay between the interface and the CPU. To build a multi-hop network, we need to follow what has been hard-coded in the hardware. For example, assume there are three nodes sS, sH, and sD, and a frame wants to make two hops, from sS to sH and then from sH to sD, as shown in Figure 2.3.

We need to make sure that for the sS-sH link, all the frames should have sS as their source MAC address and sH as destination MAC address. After these frames arrive at sH, we rewrite their headers so that sH becomes their source address and sD becomes the destination for them to follow the sH-sD link. There are two other host nodes in Figure 2.3 hS and hD, serving as source host and destination host. These two machines are introduced in our testbed for debugging purposes and can be easily integrated into sS and sD, respectively. This setup also implies that the multi-hop network is transparent to hS and hD because they see no difference and treat the incoming frames as if they were sent over an Ethernet cable.

A significant benefit in this setup is that through this MAC address rewriting mechanism, we can control a particular flow route. Assume there is another intermediate node sK that is also reachable from sS and sD; we can intentionally redirect the sS-sH-sD link to the sS-sK-sD link when sH node is out of service, and the overall system robustness is enhanced.

To efficiently utilize this setup, we need first to resolve the following two chal-

lenges:

- How do we figure out the reachability of each node?
- How can we systematically rewrite headers?

We use beacons to test the reachability. All the nodes periodically broadcast beacons and put their own MAC addresses as identifiers. At the same time, they also pay attention to beacons from other nodes and report a list of nodes they can see beacons to the SDN controller, which will be covered in Section 2.4. The SDN controller, after collecting these data, should update the connected graph of the network. After that, the controller uses that graph to determine the optimal route for each traffic flow and instruct nodes with rewriting rules.

2.3 SDN Protocol and Switch

In the last section, we covered that MAC addresses were changed according to rules systematically. If there was an update to the rules, new rules were applied at their earliest convenience. In OpenFlow protocol, we can assign priority and an expiration period to each rule, telling an OpenFlow switch that rule is only valid for a limited time for fail-safe reasons. For example, we set updated rules with a higher priority and a short expiration period and back up rules with low priority and a long expiration period. When a switch does not receive the updated rules in time, those out-of-date will expire and eventually be removed. The backup rules will take place and become effective.

How does a rule look like, and how does it work? An OpenFlow rule generally consists of two major parts, filter and command, and is pretty similar to a firewall rule. A filter is used to identify frames with some particular characteristics. Take Figure 2.3 as the example again, we set a filter at the incoming interface of sS, marking all the frames having hS as the source MAC address and hD as the destination address in their headers. The stream of these frames is called a flow, and we want to modify the frames in this flow-through executing a set of commands, such as replacing the source MAC address with sS and the destination address with sH. We need to apply various filters and rules on sH and sD to allow multiple hops and recover the frames with their original headers. In this example, we customize the rules of a particular flow on each node along its route.

In a Linux system, there are two main approaches to modify the routing paths in a network. The first approach (A1) is based on a direct modification of each node's routing table. This requires a protocol that modifies the Linux kernel network layer based on the SDN application's information running in the node. The second approach (A2) is based on the use of an SDN module (a wireless switch), which is the software component on the top of the Linux kernel that can put into action the routing decisions made at the SDN controller, i.e., it actually appends the new destination address to each packet.

We have chosen (A2) for its compatibility with the previous SDN framework to provide a more general testbed for the networking community.

Among the supported wireless switches, we selected Open vSwitch (OVS) [12] and Centro de Pesquisa e Desenvolvimento em Telecomunicações (CPqD) switch [13]. Both provide the needed functionalities in our scenario. Still, we implemented OVS because it

has a simpler architecture than CPqD, supports more versions of the Linux kernel, and allows further flexibility in node devices' choice.

We chose to use Raspberry Pi 2 Model B+ (RPI) [14] as the OVS device because its hardware consists of one Ethernet port and four USB ports. We used USB-to-Ethernet adapters and USB Wi-Fi dongles to covert USB ports into wired and wireless interfaces. RPI also runs Linux and has all the toolchains for us to build the OVS from the source. The installation process started from source compilation. One should check the list of compatible OVS and Linux kernel versions. In our case, our Linux kernel version was 3.18, so we selected OVS-2.4.0. We directly built the OVS on our target machine rather than doing a cross-compilation. The compiled binary executables into installation packages are packed as installation packages to copy these files onto different nodes without recompiling the source.

Figure 2.4 shows changes in the Linux kernel network stack before and after installing the OVS module. OVS inserts a new layer called SDN bridge between kernel's Ethernet stack and drivers in Figure 2.4(b). At this position, it can first filter out the traffic that needs to be processed according to the SDN rules while allowing all unrelated traffic to be forwarded following their original path in the kernel. A user-space application is also introduced as a bridge, hooking a remote SDN controller and OVS module in the kernel. This design is for a fail-safe concern, allowing us to set up some backup rules to handle the traffic should the SDN application lose the connection to the remote controller. For example, applying autonomous routing policies.

Figure 2.4(a) also shows how the Linux kernel handles Ethernet and Wi-Fi con-

nections. We notice that an incoming Wi-Fi frame will be converted to an Ethernet format before being sent to the upper layer. The conversion is done by replacing the Wi-Fi header with an Ethernet header; conversely, an outgoing Wi-Fi frame has its Ethernet header replaced with a Wi-Fi header added in the MAC layer. This design allows the Linux kernel to handle different interfaces in the same manner.

2.4 SDN Controller

After selecting the southbound protocol and setting up the switches, we need to build the controller. The controller should continue to monitor the network dynamics and uses the information it collects to compile rules to be pushed to the SDN switches, forming a feedback control loop. Again, the control policy can be classified into a proactive, reactive, or mixed one. In a proactive control policy, the controller asks all the nodes to broadcast beacons and uses beacon information to construct the network graph. It then installs rules for optimal routes onto each switch, even if there is no traffic demand. This kind of design minimizes the end-to-end delay and creates excessive traffic in the control plane (CP), the network between the controller and the controlling switches. Another extreme policy is to use an entirely reactive control policy, which only reacts to the traffic demand. It will result in a long delay because the controller only starts to collect the information necessary for coming up with the rules when a new request arrives.

Based on these concerns, one can conclude an optimal point between a fully proactive design and an entirely reactive design and is a mixed design for general applications.

In fact, in our design, we keep the beaconing feature in the proactive policy to update the network topology in real-time, but our controller reactively installs rules. More precisely speaking, once a traffic demand appears at one node, the controller should calculate the optimal route for that traffic and establish rules on each node along the path. Therefore, a longer delay will only happen at the beginning of the traffic.

There are several options available for SDN controllers compatible with OpenFlow [15, 16, 17, 18, 19], which differ based on the routing protocols implemented. Among the controllers supported by the developer community, Open Network Operating System (ONOS) [20] developed by ON.Lab was chosen because it provides a large number of resources that can contribute to this project.

Figure 2.5 shows the architecture of ONOS. In the form of link information, the network information is collected by OpenFlow and communicated to the distributed core of ONOS through the southbound connections. The distributed core is responsible for translating this information into a format that the routing algorithm can process. This new information is then sent up (through the northbound connections) to the routing algorithm's application. Several available examples of applications, including the Border Gateway Protocol (BGP) [21] and the Open Shortest Path First (OSPF) [22]. In general, any centralized routing algorithm can be implemented here.

2.5 Control Application

ONOS made the "operating system" part of its name because it tried to borrow the concept of an operating system. A typical operating system talks to the hardware, handle inputs/outputs and provide function libraries for a user to carry out their work. Similarly, ONOS supports multiple southbound protocols such as SNMP, NetConf, and OpenFlow and provide function libraries for northbound applications. The idea is to increase the portability. For example, if one particular feature can be found in both NetConf and OpenFlow protocols, a northbound application relying on that feature should run without modification. This is done by not exposing all the protocol commands directly to northbound applications. ONOS wraps these commands into its own APIs, making itself similar to an operating system, as shown in Figure 2.5. One significant benefit to this architecture is that a user only needs to be familiar with one set of APIs instead of many protocols.

We built our application by extending the reactive forwarding application provided by ONOS. The example application was originally design for the wired networks and did not have the MAC address rewriting feature mentioned in Section 2.2. Also, it works in an entirely reactive manner, meaning that the first frame/packet of each new traffic flow might suffer a long delay due to rule installation. We also modified this part so that our control application will proactively install rules onto nodes along the route.

The topology finding task is handled by a separate application called ProxyARP, a dependency of our SD MANET control application. In an IP network, a node uses an

address resolution protocol (ARP) for discovering the link-layer address, such as MAC address. The node uses cache memory to keep track of all the IP-MAC relations known by itself. Should a MAC address of an IP address not be found in the table, it will send out a broadcasting ARP probe to its on-hop neighbor, asking if they have the record. If not, the neighboring nodes will keep forwarding the probe request until a node with the record returns a reply.

This strategy is not suitable for wireless communications because these duplicated probe requests are sent over a shared medium, resulting in a huge overhead. ProxyARP application installs rules on all the nodes, asking them no to forward ARP probes if they do not have the MAC address record but instead redirecting them to the controller, who keeps track of all the IP-MAC records, through the control plane. ProxyARP can quickly look up the correct address and reply to it. Through this design, an ARP probe has to make two hops at most and will not be flooded throughout the data plane.

2.6 Putting Pieces Together

Now we have our software switches and controller ready. It time to connect them.

At each RPI node, we connected two USB Wi-Fi dongles A and B. Dongle A was put in Wi-Fi ad hoc mode, whereas Dongle B was set as an infrastructure STA. If an RPI was connected to a host machine, we added a USB-to-Ethernet adapter to extend one Ethernet port. RPI's built-in Ethernet port ran a secure shell server (SSH) and was used for troubleshooting.

Secondly, we installed OVS-2.4.0 on each switch node as a Linux kernel module, added Dongle A and USB-to-Ethernet adapter to the OVS switch, and set the switch to connect to the controller using Dongle B.

Thirdly, we connected the ONOS controller running our SD MANET application. The controller was also associated with a Wi-Fi AP to receive connections from our OVS switches and form a star topology control plane.

Figure 2.6 shows the final look of the SDN switches and the controller.

2.7 Evaluations

In this section, we showcase a practical implementation of our SDN framework with commercial devices. We set up the network, and we send some data traffic with a multi-hop topology. To show our framework's potential advantages, we run the same network with a distributed routing protocol. We show the benefits of the SDN framework in the case of sudden topology changes.

2.7.1 Network Setup

The experimental scenario is composed of an SD MANET with three SDN nodes, labeled S, H, and D, as in the previous section, deployed as shown in Fig. 2.7. Each of these nodes is composed of one RPi Model B+ and one Wi-Fi adapter (Ralink RT5370 USB), and the transmissions in the DP use the IEEE 802.11g ad-hoc mode (on channel 6). The three SDN nodes are equipped with OVS-2.4.0 and are connected to the CU running

ONOS (the chosen OpenFlow controller) and our MANET application[23].

A second network, named OLSR MANET, is set up for comparison using the same three nodes (S, H, and D) in the same location and with the same topology. In this second network, the nodes are not equipped with our SDN framework, but they are running a distributed routing strategy, OLSR. In particular, `olsrd-0.8.8`, which the HSMM-Pi Project provides[24], is installed in the three nodes. The parameters of OLSR are set up as follows. The hello message interval, the hello validity time, and the topology control message intervals are set to 10, 1, and 0.5 seconds.

In both networks, the data traffic is generated at node S using the traffic generator `iPerf3` [25], which creates a random TCP flow towards the destination, node D. The length of the experiment is N seconds, and time is divided into intervals of 1 second. For each interval τ_n, with $n = 1, \ldots, N$, the end-to-end throughput is measured in bit-per-second (bps) as:

$$T(\tau_n) = \frac{\texttt{TCP RWND} \times 8}{\texttt{RTT}} ,$$

where `TCP RWND` is the average receiving window size of TCP session during interval τ_i, and `RTT` is the average round-trip time, i.e., the elapsed time between the transmission of the first bit of a TCP segment sent and the receipt of the last bit of the corresponding TCP acknowledge.

To compare the behavior of the SD MANET and the OLSR MANET in the case of a sudden topology change, we alternate the fully-connected topology, shown in Fig. 2.7-(a), with the multi-hop topology, without a direct connection between S and D, shown in

Fig. 2.7-(b).

Since it is not easy to perfectly control a link failure by changing the location of the nodes, we emulate a link failure between S and D by designing a module at the MAC layer of the nodes that can reject all the packets coming from S (for node D), or D (for node S). In this way, we can perfectly control in our experiment when the link between S and D fails or when it is up again.

In the following, we describe three experiments that compare SD MANET and OLSR MANET in the case of a change in the topology (due to the failure of an existing link or the restoration of a previously non-existing). Each experiment is repeated $M = 20$ times for both the SD MANET and the OLSR MANET. The average throughput, shown in the results, is obtained as:

$$\overline{T}(\tau_n) = \frac{\sum_{m=1}^{M} T_m(\tau_n)}{M} ,$$

where $T_m(\tau_n)$ is the throughput obtained during the time interval τ_n for the m-th experiment.

2.7.2 Link-down Experiment

In the first experiment in Fig. 2.8, we observe the effects of a link failure event, which changes our network's topology. The experiment starts at $t = 0$ with the topology of Fig. 2.7-(a), where the three nodes (S, H, and D) are fully connected. S starts sending TCP traffic towards D, and both the SD MANET and the OLSR MANET choose the direct link between S and D.

Then, at time $t = 10$, the direct link between S and D fails. Thus the topology becomes the one in Fig. 2.7-(b). The SDN controller is immediately notified about this event, and it promptly reacts, imposing a new SDN rule to nodes S and H. In this way, node S sends all the packets destined to D towards H, and H forwards these packets towards D. The throughput for SD MANET is immediately restored to half the initial throughput value since the new path from S to D now has two hops.

The OLSR MANET can identify the link failure and react to it by changing the path to D only at $t = 25$, with a delay of about 15 seconds, causing a significant throughput outage. This result is expected since OLSR has a fully distributed routing algorithm, which takes significant time to update. On the other hand, SD MANET can exploit the CP, which allows the routing algorithm to be run in a centralized fashion at the CU, where all the information about link conditions are promptly collected.

2.7.3 Link-up Experiment

In the second experiment, in Fig. 2.9, we observe the averaged throughput experienced by SD MANET and OLSR MANET when the initial topology is the one in Fig. 2.7-(b), i.e., a two-hop path between S and D. At $t = 10$ the direct link between S and D is also activated, as in Fig. 2.7-(a). As expected, we observe that in the case of SD MANET, the network can react promptly to the change in topology, and the throughput almost doubles for $t > 10$. On the other hand, OLSR MANET has a delay of about 20 seconds before it can fully use the direct link, reaching the maximum throughput.

2.7.4 Fast-changing Topology Experiment

In the third experiment, in Fig. 2.10, we have a series of consecutive topology changes. At $t = 0$, the topology is the one in Fig. 2.7-(a) (direct link between S and D), then at time $t = 30$ the topology becomes the one in Fig. 2.7-(b) (two hops), then it switches again to Fig. 2.7-(a) at $t = 60$ and finally to Fig. 2.7-(b) at $t = 90$.

In this case, the experiment is repeated 20 times, and the results are averaged over all the trials. For each topology change, we observe how the SD MANET can react almost immediately to the topology change. In contrast, OLSR MANET reacts to the changes with a certain delay, causing a significant throughput loss, as expected.

2.8 Conclusion and Discussions

In this work, we presented a practical implementation of an SD MANET that provides all the advantages of D2D data transmissions and, at the same time, has the flexibility of centralized network management. We described the SDN architecture details, and we overviewed and referenced all the software components that we adopted. We contributed to this effort by providing new components for the wireless networking community.

To show the advantages of the SD MANET and the validity of all the software provided, we compared our SD MANET with an ad hoc network managed in a distributed way. With few simple examples, we highlighted the significant advantages of our approach, particularly for fast-changing network topology.

However, there were still missing pieces in our work. First, we did not cover the controller's sensitivity to the network changes, nor did we cover the controller's best policy to react against a fast-changing scenario. Moreover, our work showed the OpenFlow could be extended to a MANET running Wi-Fi ad hoc mode. Still, the solution to the more popular infrastructure mode remained undetermined in this work.

2.9 Acknowledgments

Chapter 2, in total, is a reprint of the material as it appears in H. C. Yu, G. Quer and R. R. Rao, "Wireless SDN mobile ad hoc network: From theory to practice", *2017 IEEE International Conference on Communications (ICC)*, Paris, 2017. The dissertation author was the primary investigator and author of this paper.

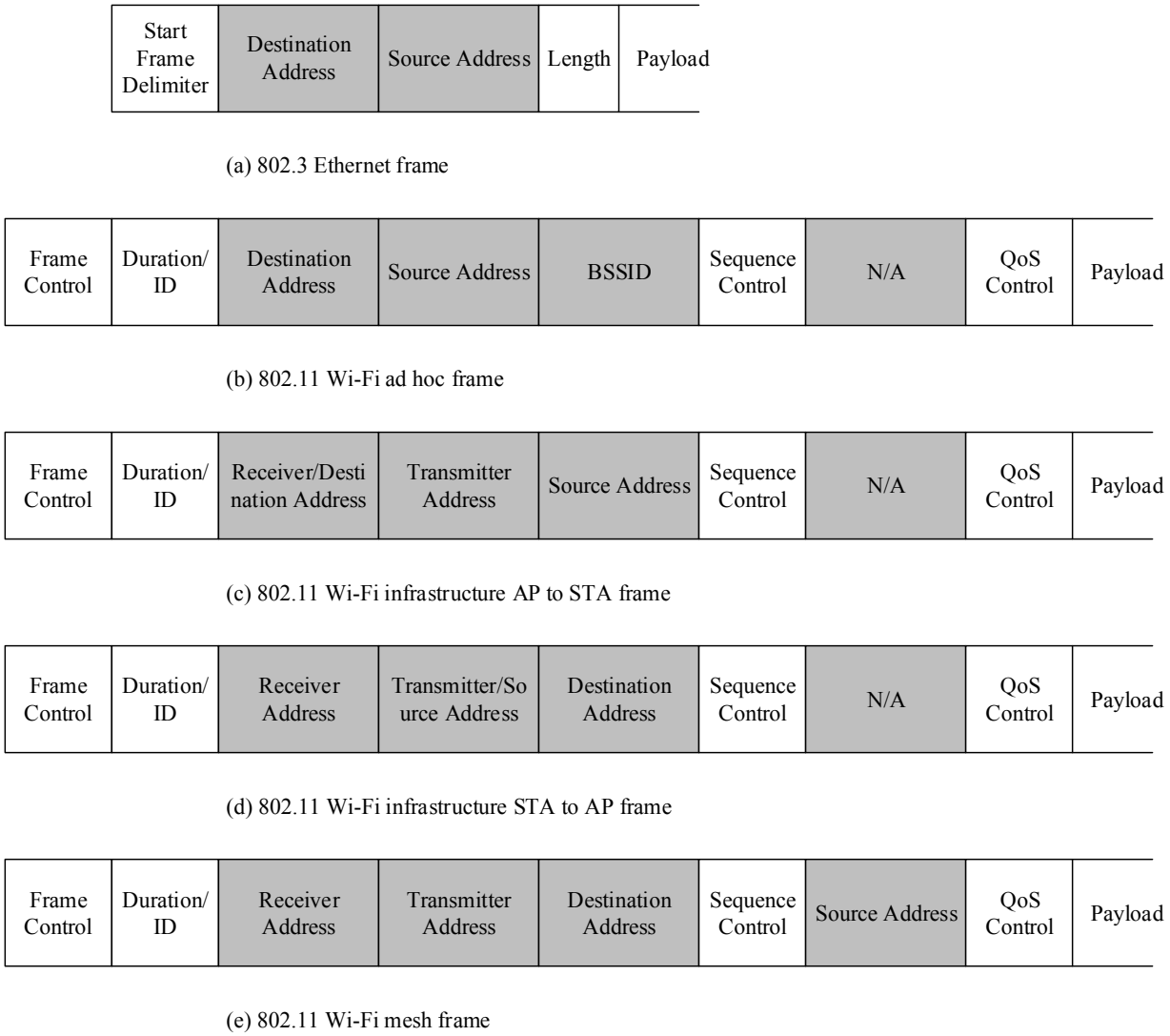

Figure 2.1: Differences among Ethernet and Wi-Fi headers

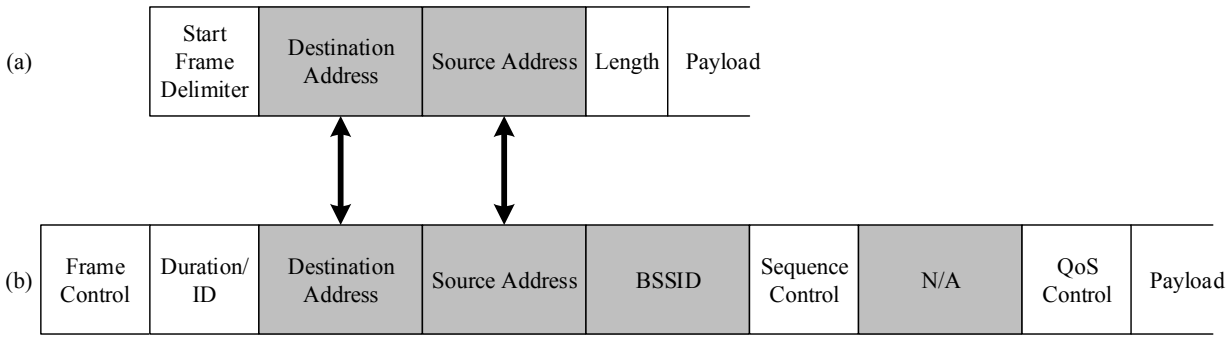

Figure 2.2: One-to-one conversion between an Ethernet frame and an ad hoc Wi-Fi frame

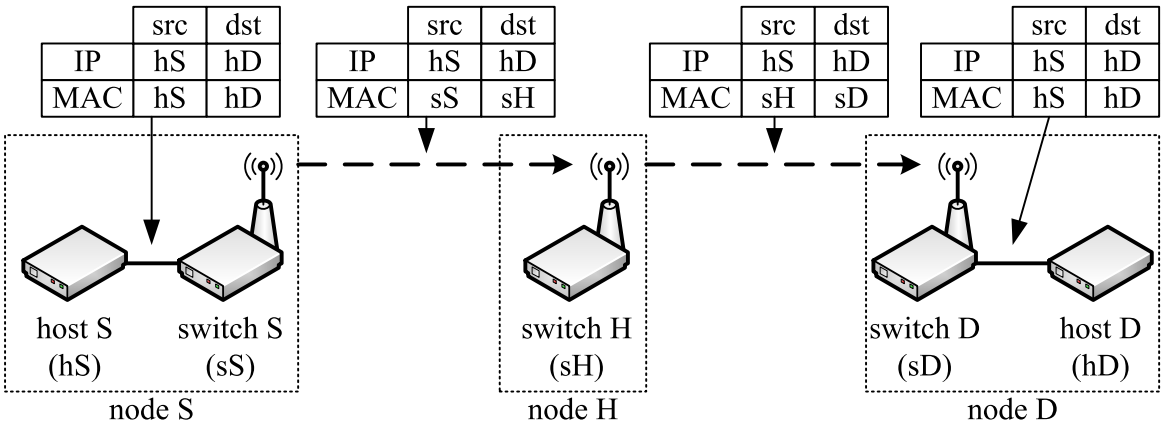

Figure 2.3: Modifications in the MAC header for a packet generated at node S, relayed by node H, and destined for node D. Wired connections are denoted by solid lines, whereas wireless connections are represented with dashed lines.

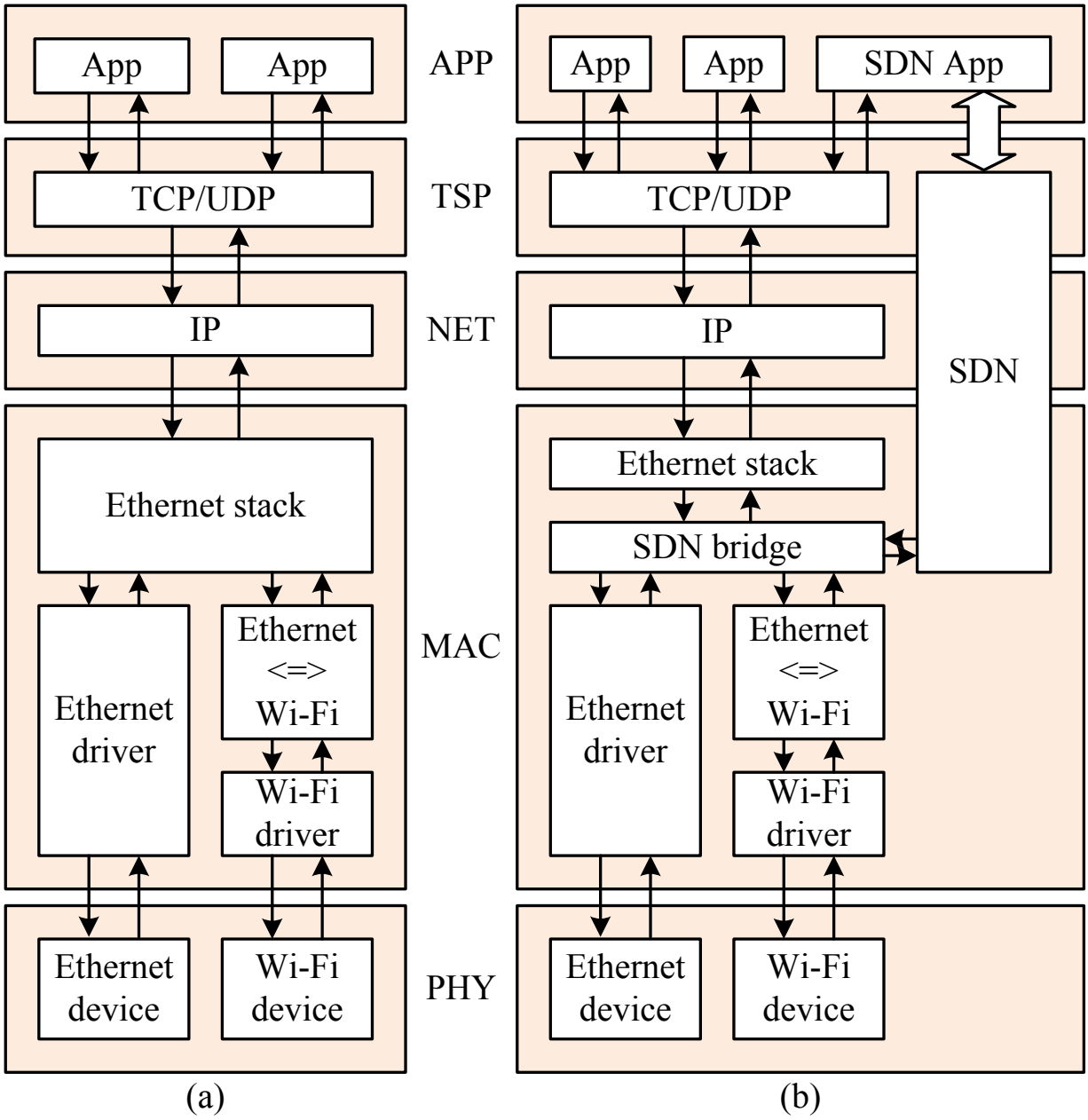

Figure 2.4: Data flow in the Linux kernel network stack: (a) without the SDN modules; (b) after SDN modules are attached.

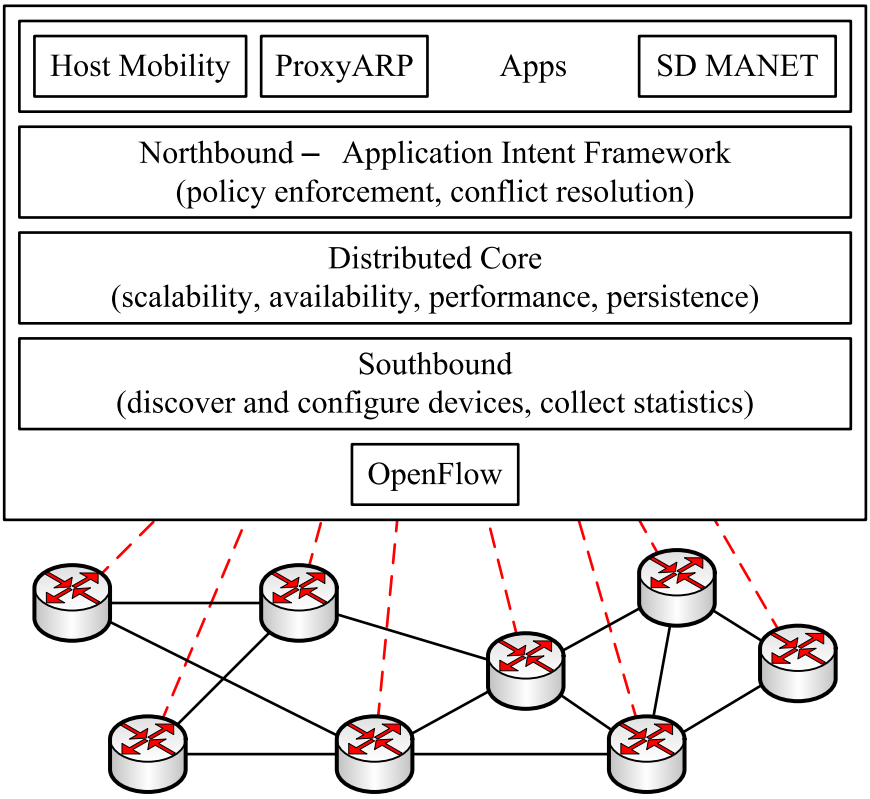

Figure 2.5: Architecture of our ONOS controller.

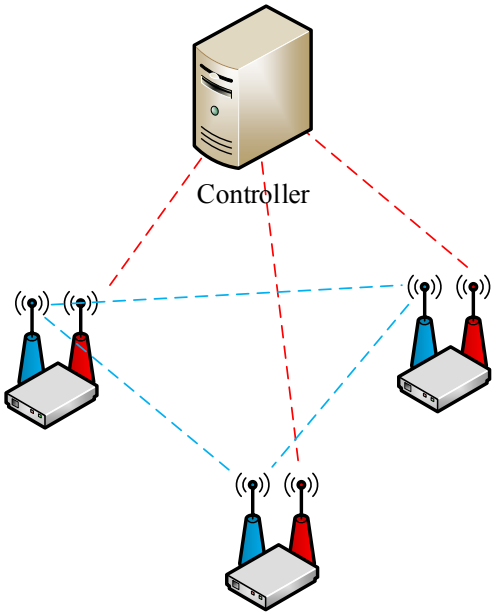

Figure 2.6: SD MANET testbed

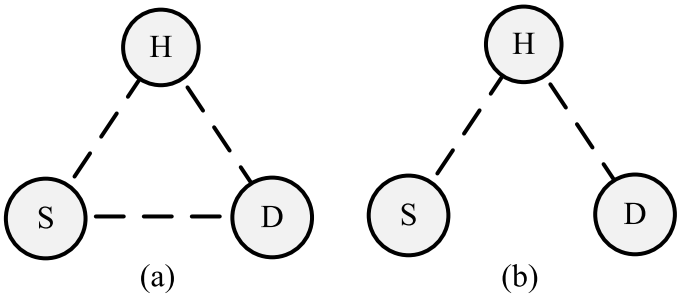

Figure 2.7: Network topology for testing the throughput: (a) both a one-hop link and a two-hop link are available between S and D; (b) only the two-hop link is available.

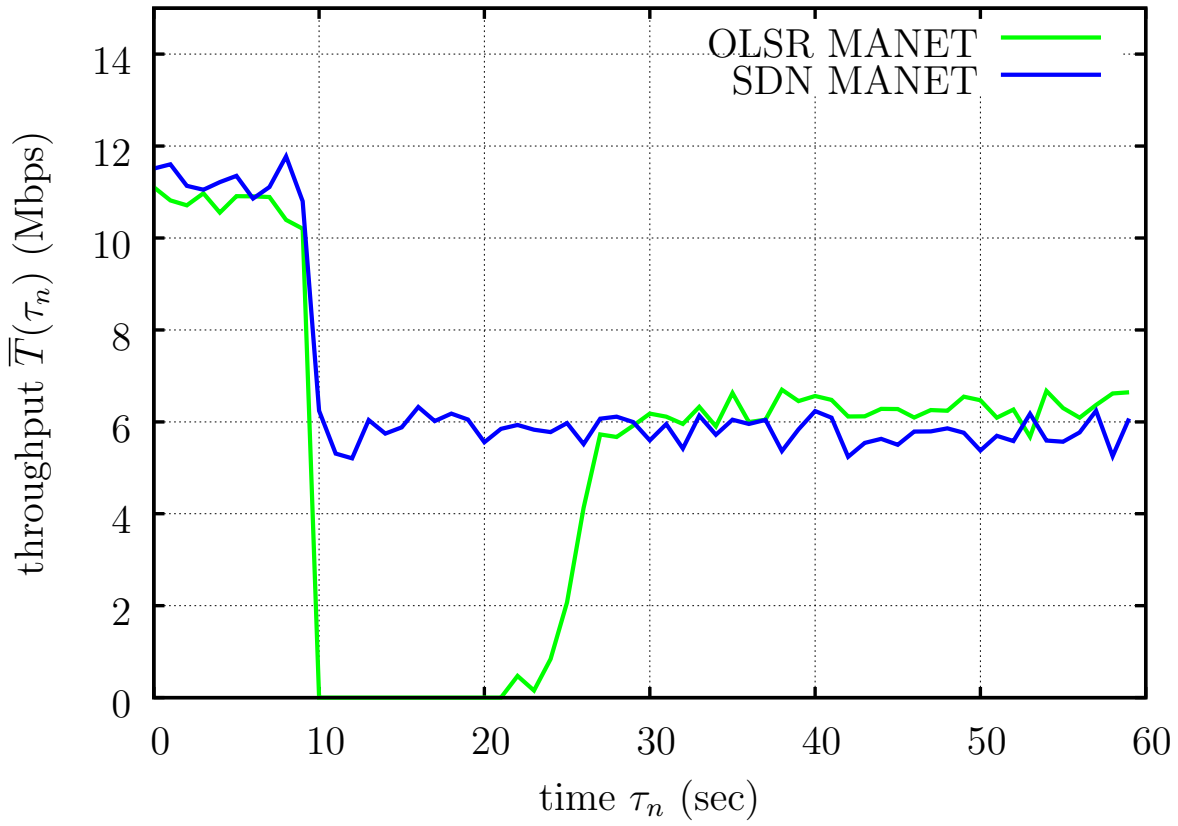

Figure 2.8: Throughput as a function of time for SD MANET and OLSR MANET for the "Link-down" experiment.

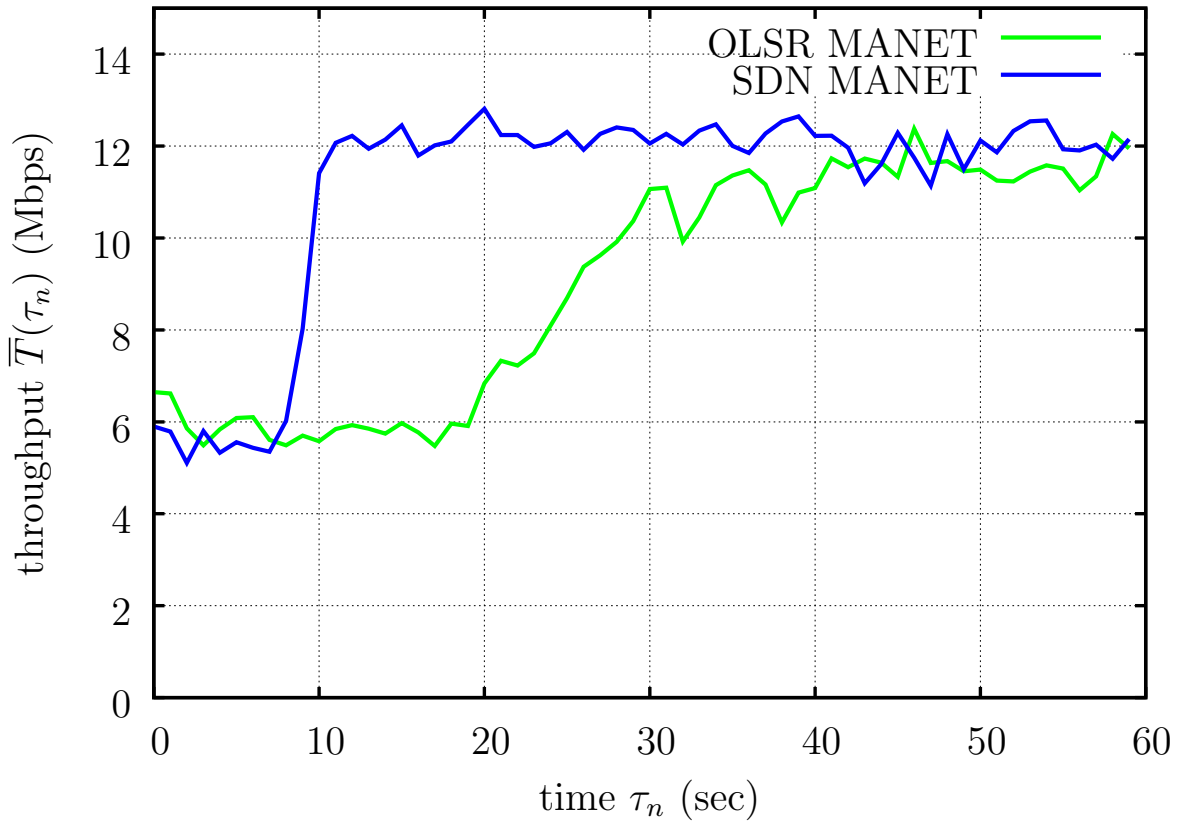

Figure 2.9: Throughput as a function of time for SD MANET and OLSR MANET for the "Link-up" experiment.

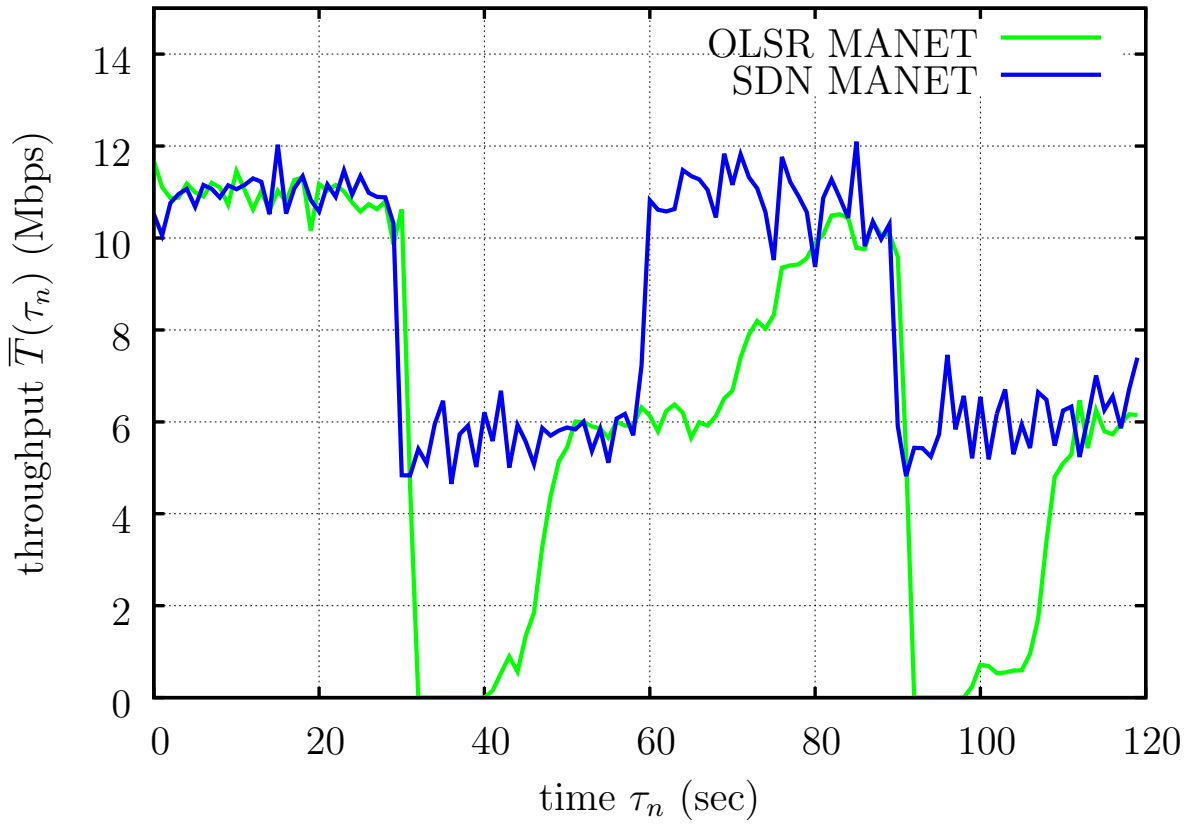

Figure 2.10: Throughput as a function of time for SD MANET and OLSR MANET for the "Fast-changing Topology" experiment.

Chapter 3

Wi-Fi Roaming as a Location-based Service

After uncovering a software-defined solution for MANET, we changed our course to a more popular and widely seen scenario, the infrastructure mode Wi-Fi. Ranging from in-home wireless networks, on-campus networks to many Wi-Fi HotSpots in shopping malls, coffee shops, and grocery stores, The infrastructure mode dominates our current Wi-Fi architecture. Thus, it becomes challenging if we want to bring in new hardware or new standards not compatible with existing ones. On the contrary, if we can develop a fully software-oriented and logically practical solution, we may leverage all the existing Wi-Fi frameworks and clear the path to popularity.

3.1 Challenges in Infrastructure Wi-Fi

As mentioned in Section 2.1, there is no easy way to establish a one-to-one mapping between an Ethernet frame and an infrastructure Wi-Fi frame because there are additional address fields in the frame header. The transmitter/receiver addresses in an infrastructure Wi-Fi frame determine the frame's intermediate recipient, as shown in Figure 2.1-(c) and (d).

It turns out that the thought process of expanding the OpenFlow protocol to SD MANET is not fully applicable to infrastructure mode Wi-Fi, and additional modifications are required. One approach (A1) [26] is to extend OpenFlow's API set, adding new functions that deal with addressing issues, channel selections..., etc. One shortcoming of this approach is that it would lead to a new protocol design that is not fully compatible with the existing one. Another disadvantage, which might be even more challenging, requires all the STAs to have wireless OpenFlow support. If they do not have built-in support from their operating systems, extra software packages are required, making itself less practical.

However, our idea (A2) was that we chose not to extend SDN protocol to all the STAs but limited the SDN support at the APs. We only turned our APs into SDN-enabled APs and left all the STAs untouched. STAs such as users' laptops, smartphones, or tablets would see no difference between a regular AP and our SDN-enabled APs.

Figure 3.1 illustrates the difference between approach A1 and A2. Devices with SDN support are marked in blue, and the support is provided through SDN software

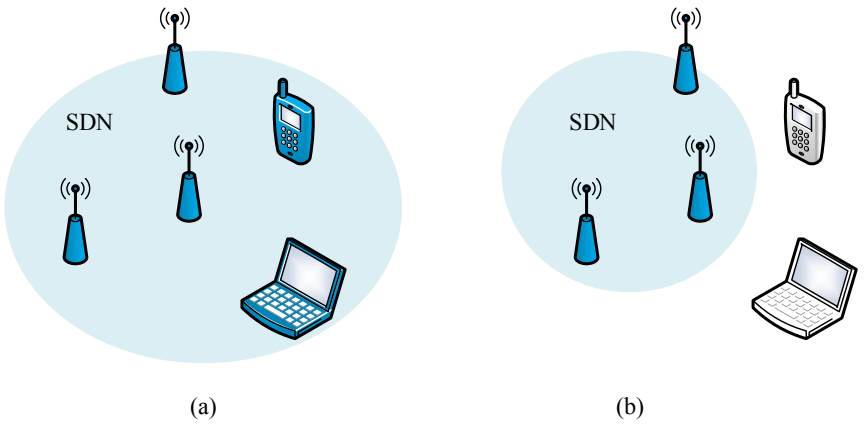

Figure 3.1: (a) both APs and STAs have SDN support (marked in blue); (b) only APs have SDN support

packages. As shown in Figure 2.4, SDN is done by parameter change in network frames/-packets and generally requires kernel-level privileges. This, unfortunately, could raise some concerns. In A2, we build a smart infrastructure that supports all the features that an existing one has but uses SDN to uncover valuable information such as location estimations. Because now wireless client does not necessarily support SDN, we cannot use the approach in Chapter 2 but also because a client is connected to our infrastructure, it must be one-hop away from its associated AP, we still can use the association information to estimate the network topology and infer the location.

Many smartphone applications require to know the location information to work. Location-based computing can be done in portable devices to improve performance and provide a better quality of service (QoS). For example, when a user brings a smartphone close to a wireless printer, the infrastructure can inform both smartphone and printer to be ready for a print. Moreover, when a smartphone moves away from its currently associated AP, the infrastructure quickly reacts against the change. It helps the phone locate a new

AP for a handoff, preventing ongoing traffic from being dropped.

Since many devices are equipped with Wi-Fi functionality, it would be very cost-efficient if topology information could be retrieved directly through existing Wi-Fi infrastructure rather than extra hardware. Topology information, such as locations, BSSIDs, operating channels, or traffic load on neighboring APs, is valuable to clients but more easily accessible from the infrastructure side because infrastructure is generally managed by a single entity, whose devices share a unified interface in general. A good example is roaming. Tseng et al. [27] assumed the location of clients is known and showed that location information could play an important role in helping clients find correct APs. They thought location information was available and proposed to use a location server to keep track of clients and suggest probable next APs, adding another degree of certainty in the roaming decision-making metrics.

The Wi-Fi roaming process is usually initialized by clients who have no prior knowledge of network topology. Still, they are unfairly tasked with quickly finding correct APs before they are disconnected. Many clients collect necessary information with time-consuming full-channel scans, during which no ongoing traffic can be sent. Fig. 3.2 illustrates a standard Wi-Fi handoff process. A client does a full-channel scan first, evaluates the results, and then tries to connect to a selected AP. However, a full-channel scan takes a relatively long time, so many clients only start scanning after they detect packet loss or their current links have become unstable, but that would be too late because a full-channel scan can take up to 500 ms, long enough to interrupt a video streaming or a VoIP call services.

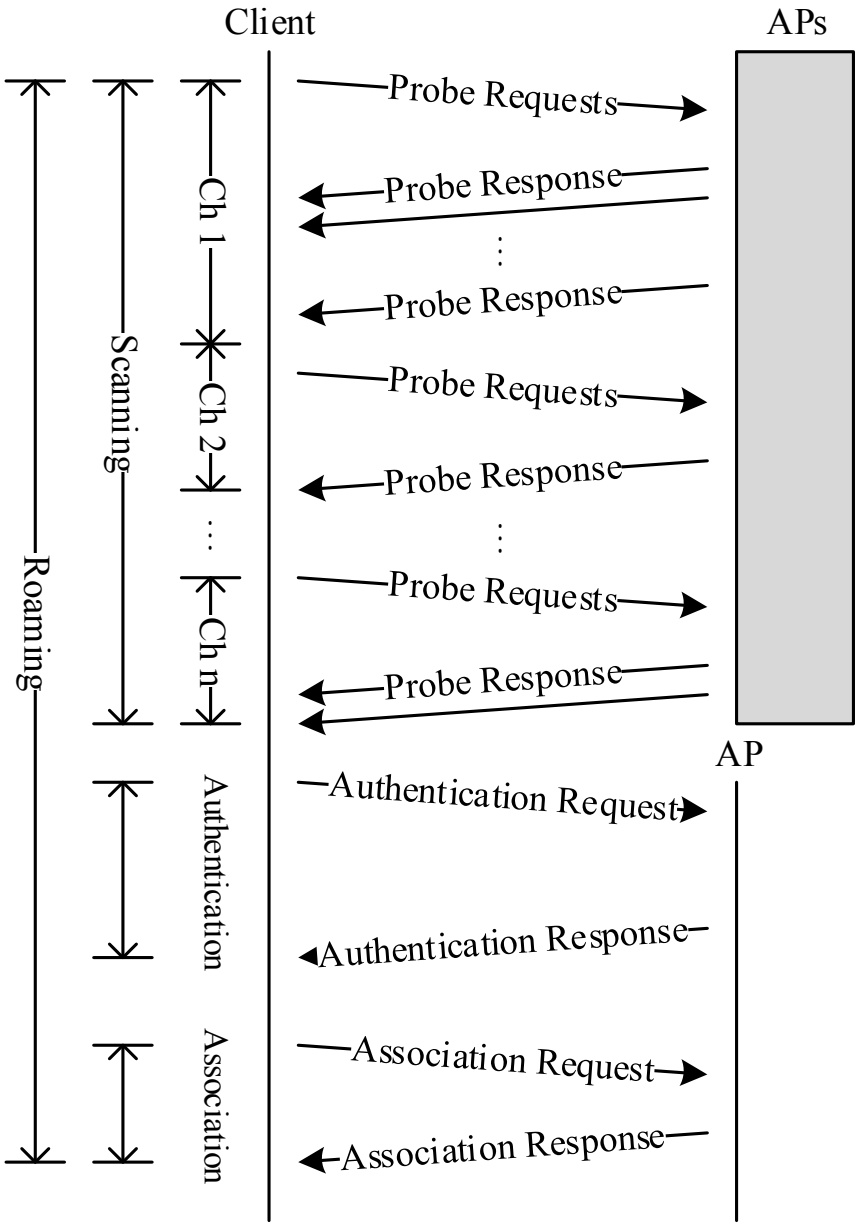

Figure 3.2: IEEE 802.11 standardized hand off process.

As we can see, a full-channel scan is a culprit for a prolonged interruption. To speed up the scanning process, a client device should avoid full-channel scans as much it can. But how? If the infrastructure can share such information with clients, clients can do a partial scan and select an AP for roaming promptly.

3.2 Location Information for Roaming

Localization is the heart of location-based services. One localization method is triangulation, which requires obtaining channel information from at least three APs almost simultaneously. That, however, could be challenging because Wi-Fi APs generally operate on various channels. SyncScan [28] overcame this challenge by synchronizing beacon timers so that beacons' arrival times in neighboring channels differ by a fixed amount of time. For example, assume the beacon interval is T, beacons in channel 1 arrive at t, those in channel 2 arrive at $t+d$, and those in channel n arrive at $t+(n-1)d$. If $(n-1)d$ is smaller than T, then a client can quickly scan all the channels in T. With this solution, however, firmware-level support is required to make timers work coherently across APs.

Another solution seen in the literature was to use multiple radios [29] in which some radios were used for background scanning for potential handoff candidates, and some were used for data transmission. In an ideal situation, as the node gets closer to a new AP, the scanning interface got associated with that new AP because of a stronger RSSI and started data transmission, providing a seamless handoff. The data sending interface disconnected from the old AP and started scanning. This solution added another layer

of complexity and needed an organized way to maintain routing and ARP tables. A few works [30, 31] proposed the use of Multi-path TCP (MPTCP) [32] to address this issue. However, MPTCP has not yet been integrated into Linux's mainline kernel and is not widely seen on Android devices.

The third solution we saw in the literature was ClientMarshal [33]. However, the paper focused on seamless roaming. The authors' techniques could be used for triangulation. In ClientMarshal, all the APs shared the same BSSID, which served as each AP's unique identifier. In general, no two AP should have used the same BSSID in the same channel if their radios covered the same area; otherwise, a device connecting to either AP would lose synchronization in data transmission and a power-saving mode. Due to this issue, the authors put nearby APs onto different channels, making them physically isolated. To achieve seamless roaming, they used the channel switching announcement (CSA) frame to tell a particular client that the AP had moved to a different channel. Because the target AP also had the same BSSID, it looked the same as the previous one from the device perspective. As for the beacon synchronization problem, ClientMarshal used unicast beacons and tweaked the beacon interval field to tell the recipient when to expect the next beacon and maintain the synchronization. One of the biggest challenges was that their controller also needed to track the difference of beacon timers across APs to maintain the synchronization.

To avoid this overhead while achieving the same goal, we explore another opportunity offered by the IEEE 802.11v protocol [9] that allows us to trigger selective scans at the client-side without beacon synchronization.

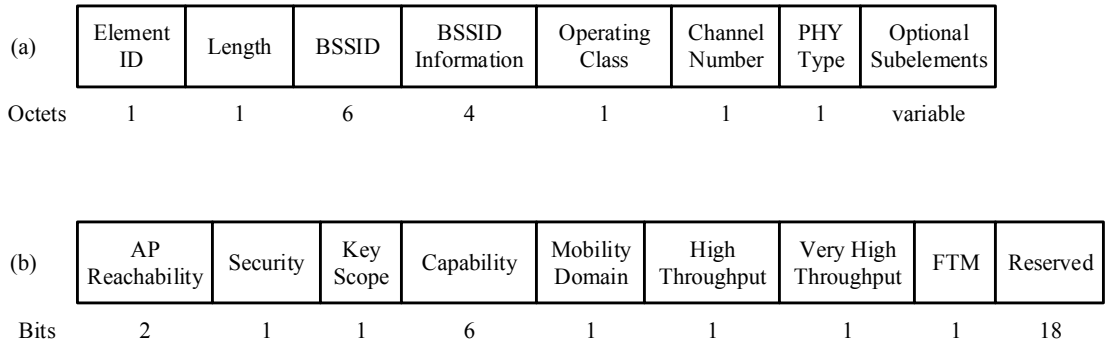

Figure 3.3: The entry format of PCL in a BTM request frame: (a) Figure-9.295 in [9]: neighbor report element format and (b) Figure-9.296 in [9]: BSSID information field

3.3 Selective Scans for Triangulation

The 802.11v protocol defines a set of new management frames. Among these frames, an unsolicited BSSID Transition Management (BTM) request frame sent from an AP to a client may attach a preferred candidate list (PCL) along with their priority, BSSID, and operating channels. Figure 3.3 shows the format of a neighbor entry in a PCL. An entry contains parameters necessary for a device to connect to the target AP, so there is no need to maintain synchronization as required in [28, 33]. When a client receives a BTM frame, it knows the associated AP might cease the service shortly. Hence, it extracts the PCL information and quickly starts an off-channel scan according to the suggested priorities at its first availability. Because the client only needs to verify the existence of APs in the list rather than all the APs in all the channels, a selective scan is enough to collect the necessary information for roaming.

We have confirmed this behavior in the source code of `wpa_supplicant-2.6` [34], a Wi-Fi manager widely found in many Linux and Android devices. Its `ieee802_11_rx_bss_trans_mgnt_req()` function defined in `wnm_sta.c` inspired us and es-

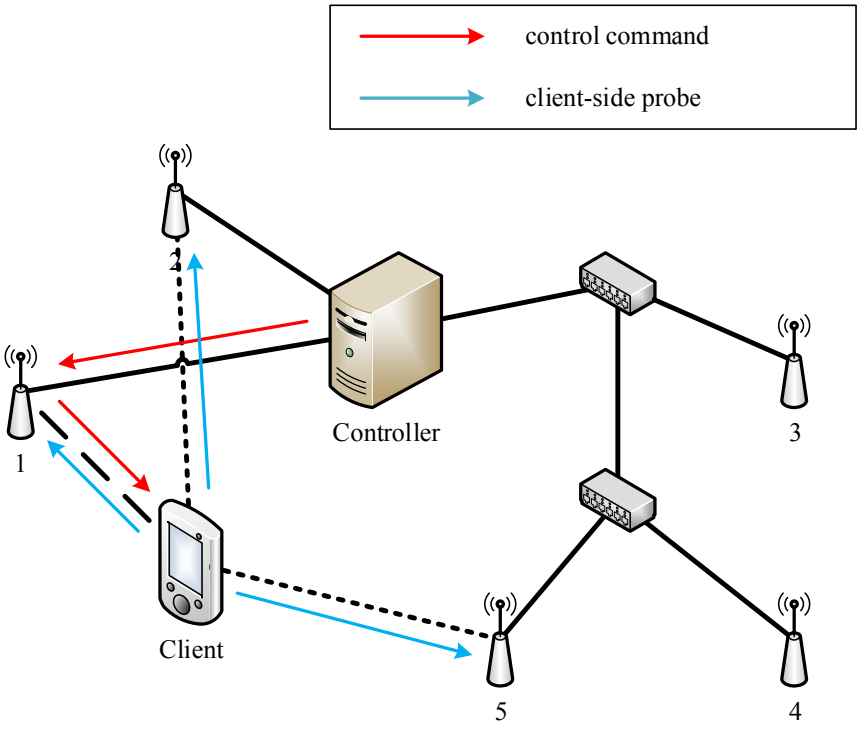

Figure 3.4: Triggering a selective scan with a unsolicited BTM request frame

tablishes the base of our work. After a BTM request frame arrives at the client, its PCL is unpacked and processed. `wpa_supplicant` will first try to look up the most recent scan results from the kernel. If the scan results are fresh enough, it checks if the roaming candidates suggested by the PCL are also in the scanning results. If so, then there is no need to schedule a proactive scan; otherwise, the roaming candidates are proactively probed. We can always include a non-existing phantom candidate in the PCL to make sure the scan is always triggered. After that, `wpa_supplicant` will then ask the interface to probe these candidates, allowing us to measure RSSI at the AP side.

Figure 3.4 shows how our customized BTM request frame triggers a selective scan. Assume there are three software-defined APs under the controller's control. These

APs need not be on the same channel, and the device is currently associated with A. The controller asks A to send the BTM request frame, putting B, C, and a phantom AP, D into the frame's PCL section. Upon receiving the BTM request frame, the device cannot find D's record in its most recent scan results and would decide to schedule a new scan at its earliest availability.

There is a chance that the device might actually roam and move its connection to B or C. This roaming process generally involves disassociation and re-association and can cause traffic interruption. Here we suggest also enabling IEEE 802.11r [9] that speeds up the roaming process from several hundred milliseconds to less than a hundred milliseconds should it happens.

3.4 Controller Design

Now we uncover the way to trigger selective scans at a device without interrupting the ongoing traffic. We need to orchestrate these selective scans to make sure that we trigger the scan only when necessary and efficiently.

We have come up with the following algorithms. Algorithm 1 shows a process of finding the roaming candidate. First, the associated AP of a particular client device can be easily looked up. Because there is likely to be ongoing traffic, the signal strength seen from the AP is also available. Second, if the location of all the APs is known in advance, we can use that knowledge to find out the neighboring APs. To not make a client device scan too often and drain out its battery power, we suggest the client scan for the

roaming candidates only when the signal strength is degraded. When that happens, we use the neighbor information in the previous step to build the unsolicited BTM request frame. We may give its currently associated AP the highest priority. In this case, since we recommend that the client not roam, there will be a scan but no roam. An important reason we have to do this is that in our experiments, we found a client might spend more than 100 ms if it has to scan the channels and re-associate to a new AP. Suggesting the client not to roam by giving its currently associated AP the highest priority decouples the scanning and roaming into different tasks.

After we decide to steer the client device and increase the targeted AP's priority in the PCL, the chances are that the client decides not to follow the suggestion and stay connected with the current. We call the device becomes **sticky**. Apple's iOS devices are generally more sticky than those running Android because iOS has its own Wi-F roaming metrics, whereas Android entirely relies on `wpa_supplicant` package.

When an iOS device declines the suggestion, it might reply to a BTM response frame, arguing why it decides not to move. A BTM response frame may also come with a PCL, suggesting why the device thinks the currently associated AP is the best one. Because we cannot obtain the source code or Apple's implementation when this dissertation is being written, we rely on our observation. The good news is that Android has a larger market share than iOS does, and we know Google's Android system relies on `wpa_supplicant`, which is an open-source package. We can still optimize our design for many Android devices.

Algorithm 2 shows how we treat a sticky client. We set the `disassociation`

Algorithm 1: FindRoamingCandidate(Client)

```
1  AP = GetAssociatedAP(Client);
2  RSSI = GetAssociatedRSSI(Client);
3  Candidate = AP;
4  Location = GetLocation(AP);
5  if RSS < rss_threshold then
6      Neighbors = SelectNeighbors(GetAllNeighbors(AP));
7      BTM = BuildBTM(Neighbors, disassoc_imminent = False);
8      sendFrame(AP, BTM);
9      sleep(120 ms); /* wait for measurement */
10     RssiMap = GetRssiMap() ∪ (AP, RSSI);
11     Location = EstimateLocation(RssiMap);
12     Candidate = SelectCandidate(Location);
13 end
14 return (Candidate, Location);
```

`imminent` (DI) bit in the BTM request frame to `True`, telling the client that we are enforcing this suggestion. If the client decides to stay, it would be disassociated shortly. After that, we blacklist the client at the AP, preventing it from coming back for 3 seconds.

Algorithm 2: SteerClient(Client, Candidate)

```
1  AP = GetAssociatedAP(Client);
2  BTM = BuildBTM(Candidate, dissaoc_imminent = True);
3  sendFrame(AP, BTM);
4  sleep(100 ms);
5  if AP == GetAssociatedAP(Client) then
6      blacklistClient(AP, Client);
7      disassocClient(AP, Client);
8      sleep(3000 ms);
9      whitelistClient(AP, Client);
10 end
```

3.5 AP Implementation

To show our proposed approach is general enough and does not require any hardware or software modification at the client-side, we built a testbed. We chose to use two typical smartphones: Apple iPhone 7 (A1778) running iOS 12.4.1 and Samsung Galaxy S7 Edge (SM-G935T) running Android 8.0. Table 3.1 gives the specifications of the

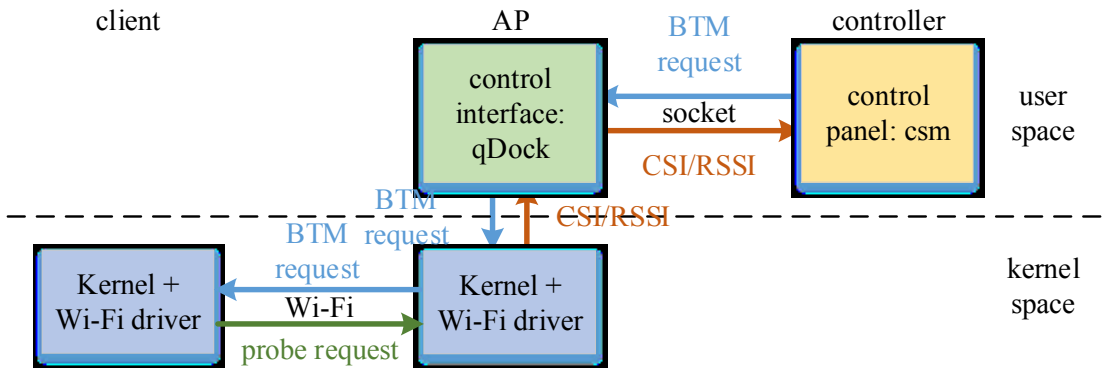

Figure 3.5: Control framework: the `csm` control application uses socket I/O to connect to qDock at the AP side, and qDock uses another socket to talk to Wi-Fi drivers on the same AP.

devices that participated in our tests. These client devices were not rooted or jailbroken to allow kernel modification.

We used a Quantenna/On Semiconductor QSR1000 reference AP design [35] as our infrastructure APs. The design was adopted by many vendors and can be found in their off-the-shelf devices as their 5 GHz Wi-Fi solutions [36]. It runs Linux and can give real-time RSSI reports for each associated client device with packet-level granularity. We implemented a control panel for customized 802.11v BTM frame sending and RSSI collecting with Quantenna's service development kit (SDK).

Figure 3.5 shows the architecture and the data flows among our controller, APs, and clients. The controller kept collecting the CSI or RSSI data. When it decided to locate the client, it asked the associated AP to compile an unsolicited BTM frame and sent it to the client. After the client received the BTM request, it probed the APs listed in the PCL. The APs being probed then updated the RSSI, and the controller came and collected these samples for determining a roaming candidate.

Table 3.1: Device Specifications

Device	**AP**	**Client**	
	Quantenna QSR1000	***Apple iPhone 7***	***Samsung Galaxy S7 Edge***
Wi-Fi module	Quantenna QT2518B	Murata 339S00199	Murata KM5D18098
Band	5 GHz	2.4/5 GHz	2.4/5 GHz
Standard	a/n/ac	a/b/g/n/ac	a/b/g/n/ac
Antenna configuration	4×4 MIMO	2×2 MIMO	2×2 MIMO
OS	Linux 2.6.35	iOS 12.4.1	Android 8.0
Wi-Fi manager	hostapd	SBWiFiManager	WifiManager + wpa_supplicant
802.11v	supported	supported	supported
802.11r	supported	supported	supported

Table 3.2 shows our control interface. The numbered MAC addresses on the left under the `STA` column are client stations, and the addresses under the `Seen By` column shows their RSSI values measured by the APs; `Age` values indicate the time elapsed since the last RSSI update in second.

3.6 Performance Evaluations

With our controller and APs ready, we were able to connect them. To show that our design is general enough, we set up a controller PC as a gateway, and our AP and devices are put in a private subnet. All the traffic that comes from and goes through the internet has to first pass through the gateway, and we run our controller application on the PC. First, we wanted to show that our design has little to no negative throughput impact on the devices. This drew our baseline, i.e., meaning the smartphones should have at least the same throughput as if we were not there. Secondly, we showed the performance gain.

3.6.1 Impact of Frequent BTM Requests

In the first experiment, we wanted to find out how frequently we could get updated RSSI samples and use them to construct a base for our proposal to test how drivers react against unsolicited BTM requests at a fixed location. A BTM request contains three fields: 1) PCL, 2) abridged bit (A), and 3) disassociation imminent bit (DI) for us to customize. Abridged bit implies the strength of roaming suggestion, and disassociation imminent bit informs whether the client will be disassociated if not moved. Note that even

Table 3.2: Control Panel

		--- STA Max Capabilities ---					---------- Current Association Info ----------						------------- Seen by -------------			
	STA	band	SS	bw	phyrate	BSSTr	AssocWithBSSID	MDID	RSSI	MaxPhyR	Avg TX/RX	Last TX/ RX	MDID	BSSID	RSSI	Age
0001.	38:de:ad:a2:20:0b	5G	2	80	867	Yes	00:26:86:f0:c5:d9	e612	-66	866	390/ 215	1/ 2	e612	00:26:86:f0:c6:2b	-82	48
													e612	00:26:86:f0:c5:d9*	-66	0
													e612	00:26:86:f0:c5:a5	-84	165
0002.	b8:53:ac:4c:c4:29	5G	2	80	867	Yes	00:26:86:f0:c5:a5	e612	-49	866	767/ 867	0/ 0	e612	00:26:86:f0:c5:a5*	-50	0
													e612	00:26:86:f0:c5:d9	-68	3
													e612	00:26:86:f0:c6:2b	-64	14
0003.	1c:87:2c:b7:6c:33	5G	3	80	1300	No	00:26:86:f0:c6:2b	e612	-73	1299	252/ 494	1/ 1	e612	00:26:86:f0:c5:a5	-66	33
													e612	00:26:86:f0:c6:2b*	-74	0
													e612	00:26:86:f0:c5:d9	-82	33
0004.	a8:96:75:9a:d4:55	5G	1	40	150	Yes	00:26:86:f0:c6:2b	e612	-69	150	121/ 125	4/ 1	e612	00:26:86:f0:c6:2b*	-71	0
													e612	00:26:86:f0:c5:d9	-68	46
													e612	00:26:86:f0:c5:a5	-56	46

when we set DI, we did not disassociate our client after the BTM request. We performed the following steps:

- Setup traffic with `iPerf3` [25].
- Prepare a BTM request with a customized PCL, A, and DI fields. For example, (PCL = 3*, A = 1, DI = 1) can be interpreted to mean there are three APs in the PCL; the AP strongly suggests the client roam, and it disassociates from it. We use '*' to indicate that the currently associated AP is the most recommended one; otherwise, the AP suggests the client roam to a different AP.
- Send the BTM request frame to the client.
- Check the timestamps of RSSI measurement at the controller. If a particular AP's timestamp is not refreshed, it implies that the client did not scan the AP.

The results are shown in Table 3.3.

Based on our test results, we noticed that as long as there were multiple APs in the PCL, both clients only scanned after 300 and 210 seconds, whereas when there was only one AP, both clients scanned that AP immediately after receiving the BTM request, triggering RSSI updates. We also noticed that iPhone was more sticky and our Galaxy S7 Edge always roamed in our tests.

Table 3.3: Test results of RSSI update frequencies

BTM			**Devices Reaction**	
AP	***A***	***DI***	***iPhone 7***	***Galaxy S7 Edge***
3	0	0	scanned every 300 sec.; not roamed	scanned every 210 sec. roamed
3	1	0	scanned every 300 sec.; not roamed	scanned every 210 sec. roamed
3	1	1	scanned every 300 sec.; roamed	scanned every 210 sec. roamed
3*	0/1	0/1	no scan, no roam	no scan, no roam
1	0	0	scanned immediately; not roamed	scanned immediately; roamed
1	1	0	scanned immediately; not roamed	scanned immediately; roamed
1	1	1	scanned immediately; roamed	scanned immediately; roamed
1*	0/1	0/1	no scan, no roam	no scan, no roam

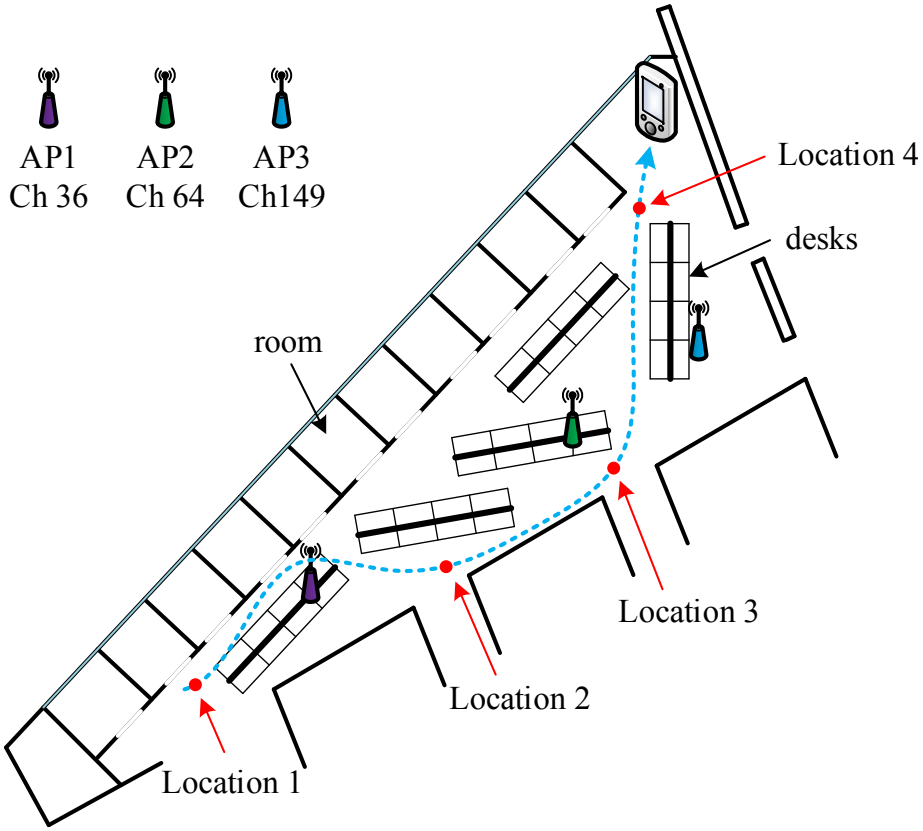

Figure 3.6: Experiment setup on Atkinson Hall 4th floor.

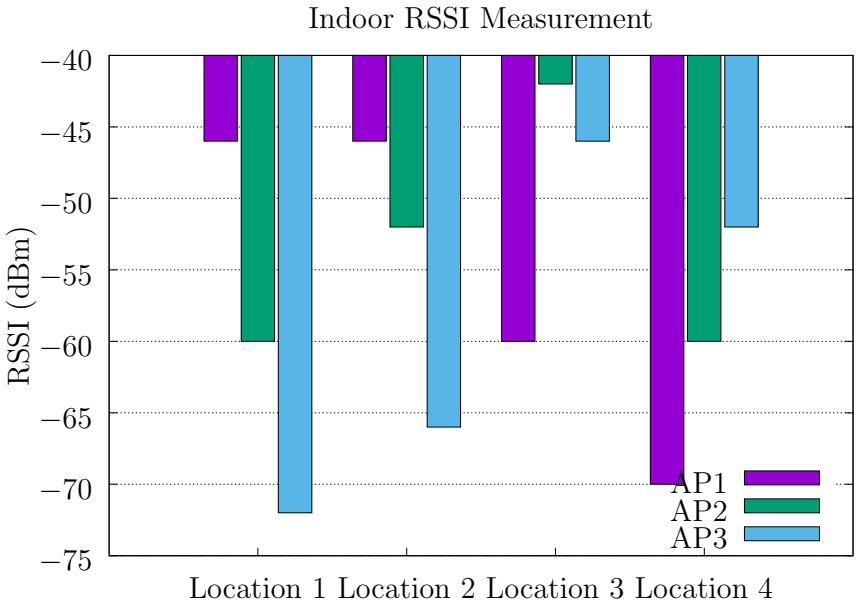

Figure 3.7: RSSI values of the Galaxy S7 Edge

3.6.2 Multi-channel Localization

Some localization approaches to[37, 38] require multiple APs and the client to be on the same channel so that the traffic sent by the client can be overheard by APs, allowing RSSI or CSI to be measured. We relaxed this constraint by using BTM-triggered selective scans, allowing APs not to be on the same channel but the client goes to different channels.

Our indoor office environment consists of multiple rooms and desks shown in Figure 3.6. A client smartphone was connected to AP1 on channel 36 initially and traveled at walking speed along the dotted blue path, passing through another two APs. We triggered the scans and measured RSSI values at four different locations.

The visualized RSSI map seen by the controller is shown in Figure 3.7. The

controller could only detect a relative location of the smartphone among all three APs. For example, RSSI values measured by AP1 between Locations 1 and 2 were almost the same, but those by AP2 and AP3 were significantly different. The controller could thus infer Location 2 is closer to AP2 and AP3 and came up with a hot zone where the smartphone is most likely to be. This is enough for handoff decisions because a higher RSSI generally leads to a better throughput in the MAC layer. However, the overall throughput might be affected by other factors in the upper layers.

3.6.3 Impact of Selective Scans

We showed a way to trigger selective scans, and we can utilize the results for localization and roaming from our previous experiments. In this experiment, we aimed to determine the impact of our approach on ongoing streaming traffic. More precisely, we wanted to determine if there would be a noticeable streaming delay; if so, is there a solution to mitigate such an impact? We reused the same setup as described in Section 3.6.1, captured Wi-Fi frames with Wireshark [39] installed on a PC, and measured the real-time throughput with a modified `iPerf3` at 100 ms granularity.

The real-time TCP throughput measurement of two smartphones are shown in Figure 3.8. Before the traffic started, we made the phone connect to a distant AP and generated a TCP flow. Traffic started at $t = 0$ second. At $t = 5$, the controller sent out the first BTM request with one non-associated AP in the PCL, unset abridge bit, and unset disassociation imminent bit (AP = 1, A = 0, DI = 0). The purpose of this BTM request was to make the phone probe the target AP and update its information in the

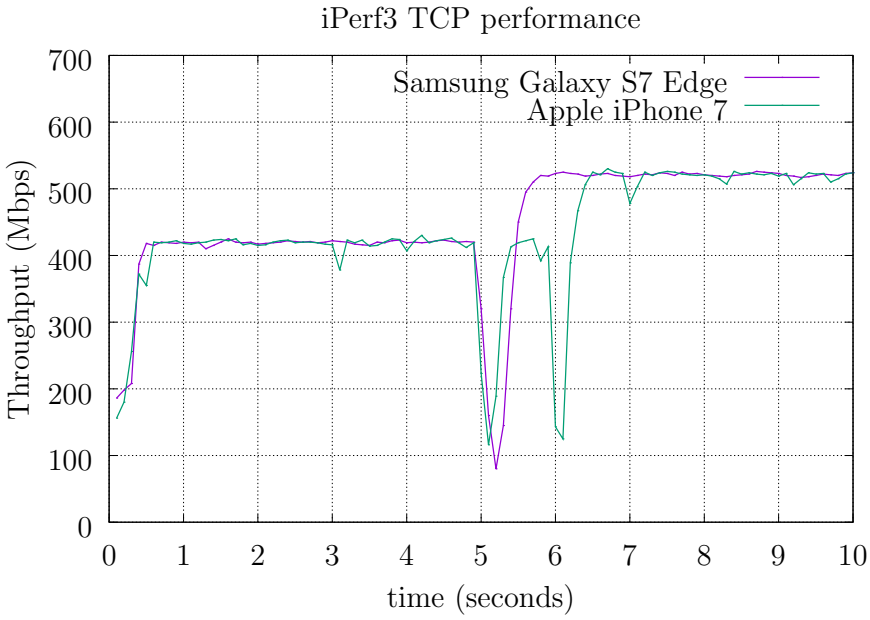

Figure 3.8: Real-time traffic throughput interrupted by 802.11v BTM frames

local database.

Our Galaxy S7 Edge always roamed in our tests, and it took 150 ms for traffic to recover. This was the expected behavior of `wpa_supplicant` and implied Google did not make too many changes to it but included it into its Android SDK. On the contrary, our iPhone 7 did not roam in most of the tests and was generally more sticky than our Galaxy S7 Edge. Our results showed that its throughput dropped and recovered within 100 ms. To force the iPhone 7 to connect to the targeted AP, we sent another BTM request with A $= 1$ and DI $= 1$ at $t = 6$. The second throughput was also shorter than 100 ms, meeting the requirement.

Figure 3.9 visualizes the above-mentioned approach of decoupling the scanning from roaming by toggling the A bit and the DI bit in two consecutive BTM requests.

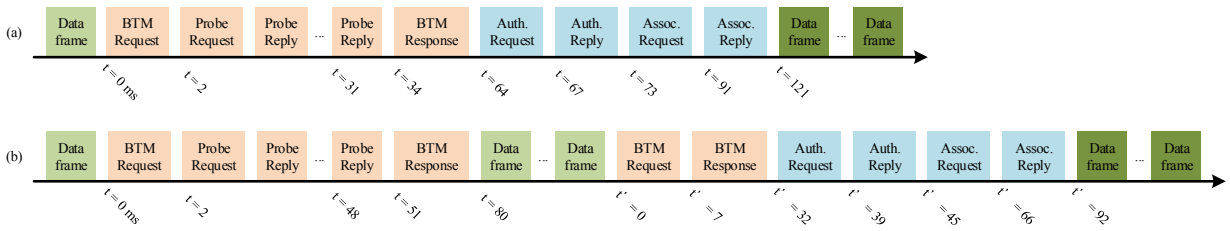

Figure 3.9: Meeting 100 ms requirement: (a) When abridged bit and disassociation imminent bit is set in an 802.11v BTM request frame, the client should handoff and connect to a new AP right after the scan, making the overall process longer than 100 ms to complete. (b) We can implement the decouple scan and handoff process by clearing both the abridge bit and the disassociation imminent bit. The client will not hand off after the scan, making the traffic gap shorter than 100 ms. If a handoff is necessary, the controller sends another BTM request with only the targeted AP and sets both the abridged bit and the disassociation imminent bit.

Although the overall traffic downtime caused by two BTM requests was more significant than that caused by a single request, each traffic gap was shorter than 100 ms. Moreover, in cases where we do not want a client to roam but want to find out its location, we can send out only the first BTM request.

3.7 Discussion and Conclusion

Our results showed different clients react differently to the same set of BTM requests. For example, our Galaxy S7 Edge behaved more compliantly than did iPhone 7. A similar observation was also mentioned in a Cisco tech report [40]. It pointed out that Apple built its own Wi-Fi scoring system, and iOS devices would only roam when the RSSI of the suggested AP is at least 8 dBm better than its currently associated one. This prevented us from freely steering iOS devices even when there is a load balancing demand.

Therefore, if we really want to force an iOS device to connect to the desired AP,

we have to send back-to-back BTM requests to force a handoff. However, we may not want to steer that iOS device if we know the targeted AP is not 8-dBm greater in signal strength than the device's currently associated AP. Android devices rely on open-sourced `wpa_supplicant`, but they are more likely to roam even when we merely want it to perform a selective scan. We checked the source code of `wpa_supplicant-2.9` (v2.6 and v2.9) and confirmed that 1) abridged bit (A) is not involved in any part of its roaming decision, 2) disassociation imminent bit (DI) will force it to schedule a scan and it is up to the driver's decision on whether to perform a scan and 3) when there is only one AP in PCL (AP = 1), the scheduled scan will also include targeted BSSID. This matches our results in Section 3.6.1, and we look forward to seeing the abridged bit become involved in the decision-making process to refine our control policies further.

A detailed analysis of how `wpa_supplicant` decides to scan the AP can be found in Appendix A. It turns our `wpa_supplicant` will try to avoid excessive scans by reusing previous results first because each off-channel scan would drain energy and be likely to interrupt ongoing traffic. Our results showed that even if `wpa_supplicant` has a sophisticated decision making policy, we were still able to trigger the scan by putting the only one targeted AP in the PCL of a BTM request frame.

Switching connections inevitably causes traffic stops. Our goal was to find an efficient and systematic way to minimize traffic stops before the buffer is depleted to boost the overall user experience.

In Chapter ,2 we showed a way to extend the SDN paradigm to wireless networks. We found the relative location among nodes but provided that the infrastructure and the

client or all the moving nodes have to support SDN. In this chapter, we borrowed the concept of SDN and exploited Wi-Fi standards so that without the device's native support of SDN, we were still able to track it and provided roaming as a location-based service. Here comes a new question – how if a device does not support SDN or IEEE 802.11v protocol? Today, there have been many internet-of-things devices that are built on micro-controllers with Wi-Fi capabilities. Unlike smartphones, tablets, or laptops, these devices generally perform simple tasks and have no 802.11v support, and the minimum requirement for us to track them will be discussed in the next chapter.

3.8 Acknowledgments

Chapter 3, in total, is a reprint of the material as it appears in H. C. Yu, K. Alhazmi and R. R. Rao, "Wi-Fi Roaming as a Location-based Service", *2020 IEEE International Conference on Communications (ICC)*, Dublin, Ireland, 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Design and Tracking Energy-Saving Wi-Fi Internet of Things Devices

In the earlier chapters, we covered different approaches to track wireless devices. For those having the full support of SDN, signal strength measurements among nodes are available almost all the time, so performing triangulation was straightforward; for smart devices not having SDN support, we used Wi-Fi standard protocols to orchestrate the selective scans and collected data required for triangulation. The two examples above imply that interactions among nodes are necessary for tracking, i.e., we need information for inferring geological distance among nodes. The data can easily be obtained from external sources; however, our work's value was that we showed a software-oriented approach to using the infrastructure, assuming that a device has SDN or necessary capabilities. There is another group of devices whose number grows exponentially, and they do not follow our assumption. Such devices include smart home devices and the internet-of-things devices

built on top of microcontrollers (MCUs) with Wi-Fi/BLE radios and connected to wireless networks. Thanks to the Wi-Fi system's advance in semiconductor technology, these devices can now be obtained at a meager cost.

These devices generally do not have SDN support for cost and complexity concerns, nor do they have built-in IEEE 802.11v/r support. In this case, we cannot expect the approaches in earlier chapters would always work. Still, we should exploit the possibility of interacting with these devices under infrastructure mode Wi-Fi.

4.1 Energy-Saving Wi-Fi IoT Devices

Internet-of-Things generally describes devices with network capabilities embedded with software and sensors to communicate with other devices or systems. One way to communicate with other devices is to use radios. There are many kinds of radio standards such as Wi-Fi, Bluetooth [41], Bluetooth Low-Energy (BLE) [42], LoRa [10], ZigBee [43], ..., etc. Wi-Fi would not be the most preferred one, mainly due to its power-consuming nature for its high throughput and far-reaching capabilities. However, Wi-Fi's rich existence around the globe could offset this shortcoming. Wi-Fi is almost available at every corner in metropolitan areas, whereas BLE, LoRa, or ZigBee are not. Although other radio standards might help save more energy, people would have to reinvest in the infrastructure for small traffic. In contrast, Wi-Fi IoT devices could reuse the current infrastructure with modifications proposed in the previous chapters.

Clearly, once we have a solution to reducing Wi-Fi energy consumption, we can

Table 4.1: Power consumption test for Bluetooth, ZigBee, and Wi-Fi devices. We chose ESP8266 as our Wi-Fi reference device.

	Bluetooth	**ZigBee**	**Wi-Fi**
IEEE Spec.	IEEE 802.15.1	IEEE 802.15.4	IEEE 802.11b
Sleeping Mode	10 μA	15 μA	1.2 mA
Awake Mode	35 mA	50 mA	150 mA

make it as competitive as other radio standards and make Wi-Fi IoT devices more quickly and widely deployed.

Table 4.1 gives average power consumption levels of different radio standards. Wi-Fi's higher power consumption originates from its 1) larger channel bandwidth (20 MHz), 2) higher transmitting power level, 3) higher data rate, and 4) higher protocol overhead. One can use a lower data rate to save more power; however, the receiver still has to be continuously in the active state because transmission can happen at any time. To further save power, an on-demand wake-up scheme that allows the receiver to turn off its Wi-Fi interface when there is no traffic was proposed in [44]. The delay and energy tradeoff was evaluated in [45, 46].

To realize such a scheme, an additional wake-up receiver (WuRx) circuit is attached. For uplink (from device to AP) transmissions, the device sends the data, so no wake-up is required; for downlink (from AP to device) transmissions, the sender first needs to wake-up the device through the WuRx circuit before it starts the transmission. Now, in the Wi-Fi interface, the WuRx circuit keeps monitoring the channel for a wake-up signal while consuming little power.

Two implementation proposals of the WuRx circuit design have been seen in the literature – coherent detection and non-coherent detection. In a coherent detection approach, the receiver demodulates the IEEE 802.11 orthogonal frequency division multiplexing (OFDM) into symbols where the wake-up and identity information is carried. For example, in [47], the author proposed to use modified OFDM to carry the address and identifier information. In [48], the author proposed to build data patterns of OFDM subcarriers that produced an amplitude modulated signal after OFDM modulation. Unfortunately, these approaches would require some firmware-level modifications at the sender and would be far from being practically implemented. Another example of coherent detection is Wi-Fi itself. Wi-Fi station devices synchronize themselves with the AP via AP's beacons and only wake up before a beacon arrives. When there is downlink traffic, the AP sets the traffic indication map in the beacon to wake up the intended station device for transmission. As mentioned earlier, Wi-Fi still consumes excessive power even through this approach.

In a non-coherent detection approach, a WuRx circuit does not demodulate the signal but uses an envelope detector (a 1-bit A/D convertor) and a clock to measure frame durations. Authors of [49] developed an optimized frame duration-based modulation approach that maximized the Hamming distance between equivalent wake-up messages. Each equivalent wake-up message consisted of $N = N_1 + N_2$ frames where N_1 was the number of message frames, and N was that of the dummy frames, making N a constant for Hamming distance calculation. They claimed the maximum Hamming distance among these messages helped lower the false positive wake-ups against the interference. Their

AP was turned off for power-saving during a long idle period. Hence, before a Wi-Fi IoT device sent the AP messages, it had to send the wake-up signal to the WuRx circuit attached to the AP and turn on the AP. The authors chose to use a burst transmission, i.e., consecutively sending message frames in a short period of time. However, this feature was not commonly used, so they had to modify the driver to enable this function. To ensure their message constellation, they disabled the Wi-Fi card's back-off feature so that frames being sent were always gapped by the 50 μs DCF inter-frame spacing (DIFS). Unfortunately, this modification limited the selection of supported Wi-Fi chips. In the meantime, their maximized Hamming distance design did not consider the possibility of multicast wake-up, i.e., to wake up multiple devices of the same group simultaneously.

In this chapter, we will also propose a non-coherent detection wake-up approach that uses a frame duration-based modulation. Rather than the design present in [49], we search in the frame duration distribution chart for less frequently used frame durations to avoid interference. Wi-Fi's frame durations are not uniformly distributed; in fact, some durations appear much less frequently than others. We call the set of these infrequent durations quiet zones. The reason why frame durations distribute unevenly will be discussed in Section 4.2.

Figure 4.1 explains how we plan to attach the wake-up circuit to a device. The output of the wake-up circuit will be connected to the device as an external wake-up signal. After receiving the wake-up signal frames, the device wakes up and makes the Wi-Fi interface connect to the AP, downloading the data, and then goes back to sleep. The `OR` logic implies that the Wi-Fi interface can be turned on by either the circuit for

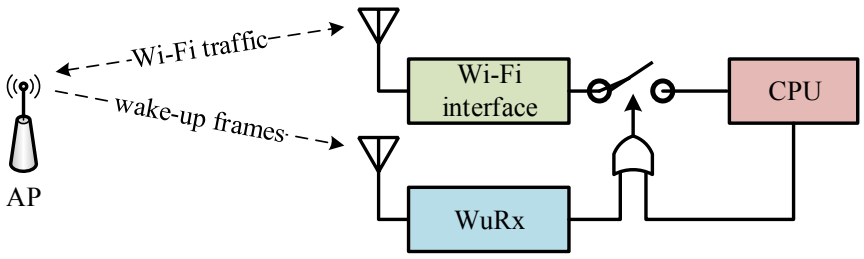

Figure 4.1: WuRx circuit as an add-on switch to the Wi-Fi interface.

downlink traffic or the CPU for uplink traffic. Ideally, the wake-up circuit can share the same antenna with the Wi-Fi interface because only one will be active simultaneously.

The add-on design adds gear to the existing Wi-Fi interface as a super power-saving mode that consumes little power to monitor the channel for wake-up frame patterns, which is one of the keys to our power-saving solution. Moreover, our approach does not use the bust mode, so driver modification is not necessary; we will use the frame injection technique instead and completely follow the standard.

The key contribution of our work in this chapter includes:

1. We design a wake-up signal pattern compatible with the IEEE 802.11 standard and can be generated with commercial Wi-Fi devices without hardware, firmware, or driver modification.

2. We use the high-speed frame injection feature available in the Linux kernel. Kernel module modifications may be necessary because these features are still under development and have not yet been integrated into the kernel.

3. Our proposed scheme can be easily adapted to the change in the number of devices

and extended to support multicast wake-up, which was not covered in [49]. Besides, we introduce a mask number to inform the number of message frames the receiver should expect, so the use of dummy frames is not necessary.

The chapter is organized as follows: in Section 4.2, we will first cover our counter-based non-coherent detector's architecture. An algorithm that helps find the parameters for injecting frames with intended durations is then present. We continue to explain the challenges brought by Wi-Fi's CSMA/CA congestion control mechanism and how we plan to overcome these challenges in Section 4.3 and maintain the compatibility at the same time. The architecture of our unicast WuRx design is also proposed in this section. Section 4.4 explains an extended version of our WuRx that supports multicast wake-up. Section 4.5 will show how we can connect our WuRx to a microcontroller platform and realize Wi-Fi IoT device tracking using the same framework present in Chapter 3.

4.2 Injecting Wake-up Frame Patterns

An envelope detector circuit is an RF circuit, outputting the instantaneous power level in a binary format after sampling. It tells whether the power level is above or below the threshold. When a Wi-Fi frame arrives, it carries energy and makes the detector output high (logic one). The detector will output low (logic zero) if the channel is low energy. The time duration of a Wi-Fi frame can be obtained by adding a clock and a counter. Figure 4.2 illustrates the logic of such a design.

An incoming frame will make the counter count up by one at each clock rising

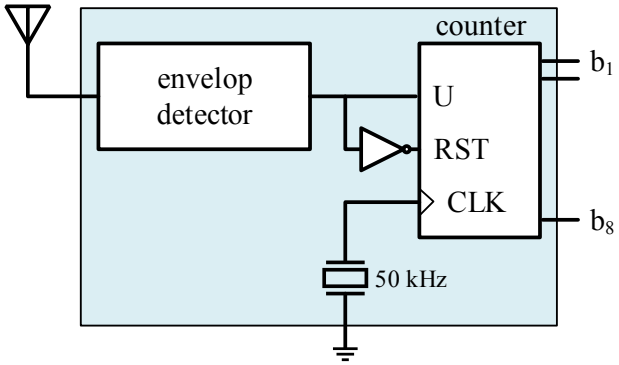

Figure 4.2: The counter-based envelop detector

edge. After the end of the frame, the channel goes back to low energy, and the output of envelop detector will reset the counter to output zero. We should store the output snapshot before it is reset, and the snapshot value represents the frame duration in the form of clock cycle numbers. The envelop detector performs non-coherent detection and does not look into the content of frames. It tells no difference if two frames share the same time duration and signal strength. In this case, we need to use multiple frames to perform a frame duration-based modulation scheme.

4.2.1 Injecting Frames of Specific Duration

Frame injection is a feature provided by the Linux kernel and the chipset driver to test and develop new Wi-Fi protocols. If the frame format is not yet supported by the kernel or is still under development and subject to change, a user can put the Wi-Fi interface into monitor (injection) mode so that the kernel will redirect the frame inputs/outputs to the user.

If we want to send a non-standard Wi-Fi frame over the channel, we first put the

message into a buffer and then build the radiotap header specifying that frame's physical layer parameters. Linux kernel's convention does not allow a user to talk to Wi-Fi hardware directly but has to do so through the radiotap header. This gives us control over the length and the sending rate of the frame to be injected. Some patches to the Linux kernel are necessary for it to support rate assignment. Stephan M. Günther provided these patches in `libmoep80211` [50, 51] project for his high-speed frame injection experiments and made them public. According to his work, not all the hardware currently supports high-speed injection. We have confirmed Qualcomm-Atheros AR9380 (also known as AR5BHB112) [52] and MediaTek-Ralink RT5370 [53] chip has such support after kernel fixes [54]. These kernel fixes allow the kernel to pass the user's high-speed injection parameters to the hardware. Those without such support will always send the injected frame at 1 Mbps, the lowest available rate. For the devices support rate assignment, we can make the hardware send at a higher rate by specifying the sending rate in the radiotap header [55].

Equation 4.1 shows the relation among the preamble duration T_p and the symbol duration T_s in microseconds (μs), the minimum frame length L_0 and the payload length L in byte, the sending rate r in bit/sec (bps), and the frame duration T_f. The ceiling function $\lceil a, b \rceil$ in Equation 4.1 means to round up variable a to the closest multiple of variable b such that $\lceil a/b \rceil - 1 < a/b \leq \lceil a/b \rceil$ where $\lceil \cdot \rceil$ is the normal ceiling function. There could be multiple combinations of (L, r) that give the same T_f. For example, the frame duration would remain the same if we send twice the amount of data at a doubled rate. Also, there could be a duration that any combination cannot achieve. This might

be due to either the frame's duration being too short and cannot be achieved by sending the shortest frame at the highest rate or too long and cannot be achieved by sending the longest frame at the lowest rate. The boundaries of these parameters together form a multi-dimensional set of solutions, allowing us to add extra value to our design.

$$T_f = T_p + \left\lceil \frac{(L_0 + L) \times 8}{r}, T_s \right\rceil \tag{4.1}$$

In the IEEE 802.11b standard, T_p is 192 μs and T_s is 1 μs/symbol; in the IEEE 802.11g, preambles can be either short or long. A short preamble lasts for 20 μs, and its symbol duration is 4 μs/symbol. We only injected data frames for simplicity and convenience to reuse the data structure defined in the Linux kernel. The minimum frame length L_0 is 28 bytes. Because we have control over the payload length L and the data rate r, the frame duration is determined. Equation 4.1 also implies that a 50-μs frame duration is not achievable in the IEEE 802.11g because a valid frame duration has to be a multiple of 4 μs. The closest choices would be 48 or 52 μs.

In our application, we need to interpret Equation 4.1 in a reverse way to answer the question: given a desired frame duration T_f, what are the achievable payload length L and sending rate r combinations? We first rearrange Equation 4.1 into Equation 4.2 as

$$T_f - T_p = \left\lceil \frac{(L_0 + L) \times 8}{r}, T_s \right\rceil \tag{4.2}$$

The ceiling function implies its values has to be a multiple of T_s, so we have

$$(n-1) \times T_s < (T_f - T_p) \leq n \times T_s \tag{4.3}$$

where n is the number of symbols constituting the payload part of the frame. We can then use Equation 4.3 to look for all possible n's. For example, in IEEE 802.11g, if we want a

frame to be 52 μs. After plugging in the numbers, we solve n with

$$(n-1) \times 4 < (52-20) \leq n \times 4 \Rightarrow n = 8 \tag{4.4}$$

So we know that the payload is sent across 8 symbols and will last for 32 μs in the IEEE 802.11g standard. Assume we can pad the payload length till the maximum bytes allowed in 8 symbols, we find the shortest acceptable payload length L as follows:

$$\frac{(L_0 + L) \times 8}{r} = 4 \times 8 = 32 \Rightarrow L = 4 \times r - L_0 \tag{4.5}$$

The possible rates in the IEEE 802.11g are 6, 9, 12, 18, 24, 32, 48, and 54 Mbps. Table 4.2 gives possible combinations of (L, r) for sending a 52-μs frame. To send a 52-μs Wi-Fi frame, we can inject an 8-byte data frame and send it at 9 Mbps or inject a 20-byte data frame and send it at 12 Mbps. One can easily verify these choices by plugging numbers into Equation 4.1.

We use the frame injection technique to inject frames meeting the requirements into the channel.

4.2.2 Avoiding False Positives with Multiple Frame Injections

Wi-Fi frame durations follow a distribution related to types of traffic. This is reasonable because control messages are usually short, whereas web browsing and video streaming data are long. Today's computer networks are mostly built up with Wi-Fi and Ethernet. Messages are converted between them and sometimes sliced into smaller pieces to avoid hitting the maximum transmission unit (MTU) limit, which is 1,500 bytes

Table 4.2: Payload length L for sending a 52-μs frame with different rates r in the IEEE 802.11g.

sending rate r (Mbps)	**payload length L (byte)**
6	-4 (not achievable)
9	8
12	20
18	44
24	68
36	116
48	164
54	188

for Ethernet and 9,000 bytes for Wi-Fi. This explains why we can see a large number of small size frames and another peak at frame size around 1,500 bytes. The histogram in Figure 4.3 clearly shows such a phenomenon. The frames were captured through a Qualcomm-Atheros AR9380 Wi-Fi card within one hour on the 4th floor in Atkinson Hall. The bar width is 50 bytes. One might note that the sharp drop happened at a position greater than 1,500 bytes. We looked into the raw data and confirmed around 500 frames whose lengths were between 1,575 and 1,600 bytes. This might result from some protocol overhead, and there was no frame longer than 1,600 bytes.

Figure 4.4 shows the duration statistics of the same captured frames. We put Y-axis on a logarithmic scale to help up focus on low peak regions. As expected, a typical in-door Wi-Fi channel contains much more short-duration frames than long-duration ones.

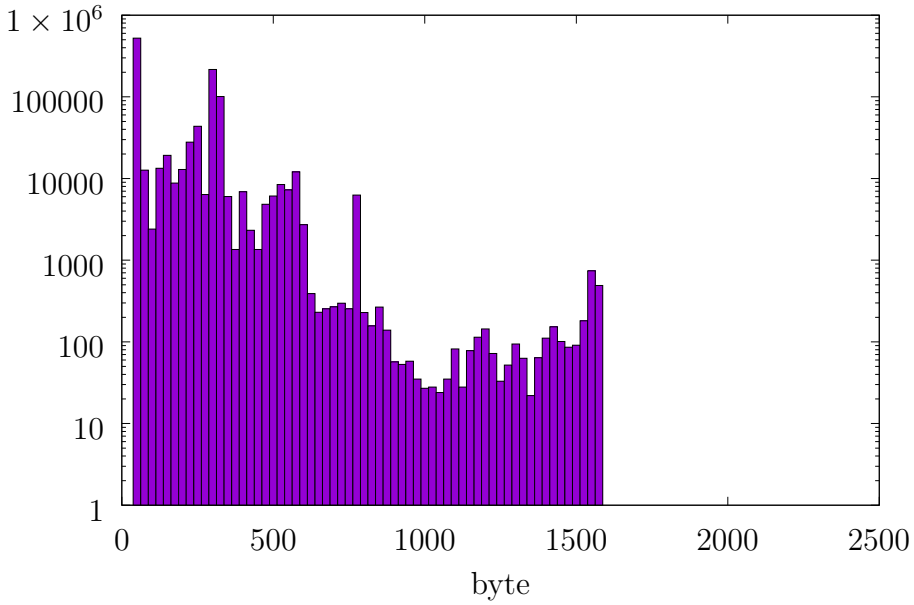

Figure 4.3: Length distribution of Wi-Fi frames captured in an 1-hour period

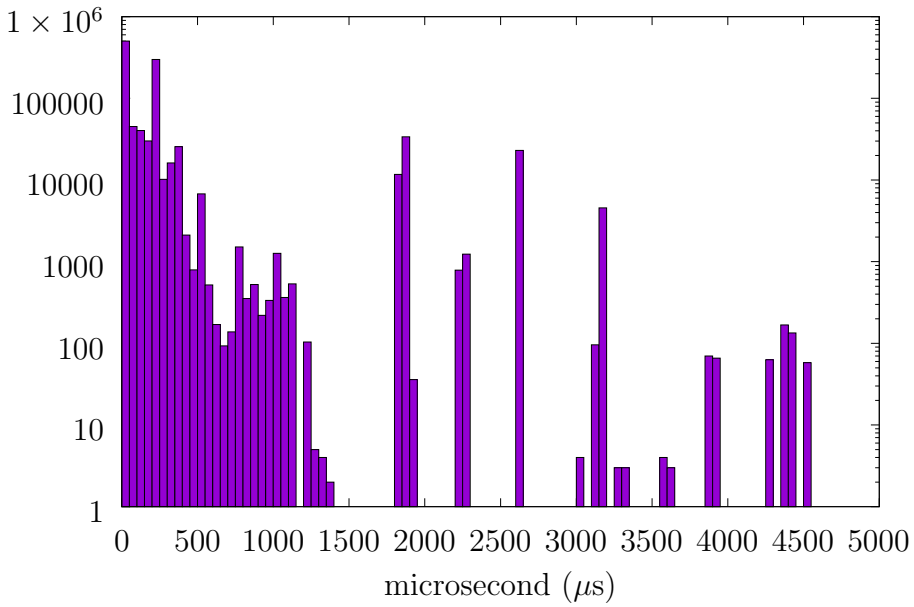

Figure 4.4: Time duration distribution of Wi-Fi frames captured in an 1-hour period

However, we noticed some peaks. For example, peaks around 1,900, 2,200, 2,600, and 3,100 bytes were caused by beacon frames from APs. Beacon frames can easily reach 200 bytes long, and they are generally sent at the lowest rate, 1 Mbps, using the IEEE 802.11b standard to make sure that the old devices can hear them. However, many nowadays APs send their beacons at 12 Mbps to increase efficiency.

Besides peaks, there were quiet zones in which no frame had such durations in these time intervals. This finding suggests that we can put our wake-up frame durations in these intervals to lower the odds of false positives. But we cannot guarantee the number of frames having these durations is always zero, meaning there could still be false positives if we only send one frame. The problem can be solved statistically by sending multiple frames of predefined durations in these regions. Each device will have a unique combination of predefined frame durations, which can also work as an identifier. The device should wake up only if it sees frame durations matching predefined patterns before it times out; otherwise, it should remain asleep. In our design, we assigned four integers representing the duration of wake-up frames for each device.

Figure 4.5 shows how these predefined frame patterns in terms of durations can work as identifiers. WuRx 1 and WuRx 2 belong to two different devices with different wake-up frame patterns. When there is downlink traffic to WuRx 2 at the AP, it broadcasts WuRx 2's wake-up pattern to call up WuRx 2; WuRx 1, upon receiving such a pattern, will not generate a wake-up signal to the CPU and continue to sleep to save the power.

The wake-up circuit's detection precision limits the maximum number of devices that a single AP can support. For example, in the IEEE 802.11g, the symbol duration is

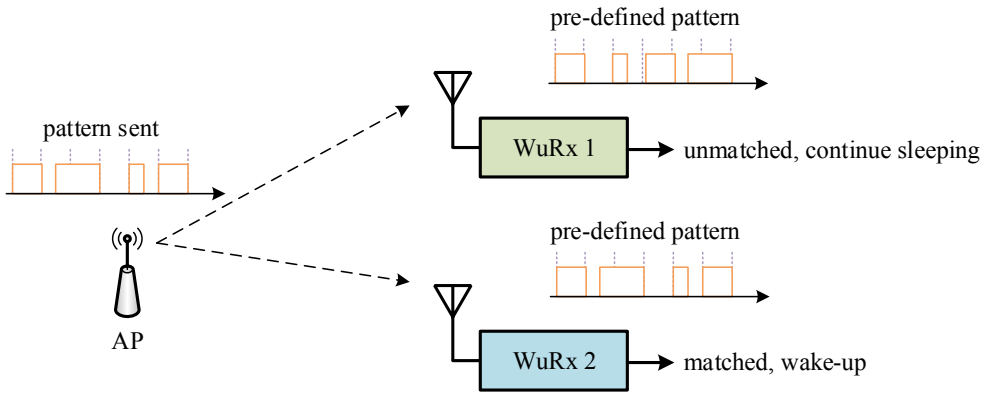

Figure 4.5: Unique wake-up pattern for device identification.

4 μs. To tell a 1-symbol difference, the sampling period should be shorter than one-half of the symbol duration according to the sampling theorem, and the value is 2 μs for the IEEE 802.11g.

To have a shorter than 2-μs sampling period, we need a clock to run at a frequency faster than 50 kHz, making the time interval between two consecutive rising edges less than 2 μs. The power consumption is squarely proportional to the clock frequency in electronic circuits. Therefore, if we intentionally ignore one-half of the durations by only sending frames with an even number of symbols, we can use a slower clock, but that will also reduce the number of devices that a single AP can support reduced.

Wi-Fi jumbo frames have 2,346 bytes as their MTU. They are also the longest frame we can inject. The shortest data frame is 28 bytes. According to Equation 4.1, the shortest duration we can obtain is 28 μs with $(L, r) = (0, 54)$ and the longest duration is 3,148 μs with $(L, r) = (2318, 1)$. (Note that there is always a 28-byte header L_0.). Between 28 and 3,148, we have 780 possible choices of durations for a single frame. Ideally, using

Table 4.3: Valid frame durations given by different clock frequencies

clock frequency (kHz)	**sampling period (μs)**	**detection precision (μs)**	**valid durations**
12.5	8	16	195
25	4	8	390
50	2	4	780
100	1	2	780

a 4-frame wake-up frame allows up to 780^4 combinations. Still, we recommend selecting durations within the quiet zones in Figure 4.4 and make the duration of the first frame a constant to lower the odds of false positives. Based on our observation, there was one-half of the durations had no frame. These regions are for us to take advantage of.

Table 4.3 gives the relation between clock frequencies and the number of valid frame durations. Note that increasing the clock frequency to 100 kHz does not help boost the number of duration choices because the number of options is also limited by the 4-μs symbol duration of the IEEE 802.11g standard. Eventually, our design choices are as follows:

- Use a 50-kHz clock so that we can have the per-symbol resolution in the IEEE 802.11g standard;
- Limit the duration choices of frame 1. For example, always set the duration of the first frame (frame 1) to 1,500 μs to avoid false positives;

- Allow a flexible number of frames in a wake-up pattern. We use a 4-frame pattern now, but we might use 3, 5, or more frames for different purposes.

4.3 Exploiting CSMA/CA Mechanism

CSMA/CA is Wi-Fi's congestion control algorithm, which triggers a random back-off on sending devices when detecting a collision. In a shared medium, we need a congestion control mechanism to handle traffic scheduling; otherwise, a collision happens when two or more devices begin their transmissions at or close to the same time, making their signals undecodable. CSMA/CA asks all the senders to follow a listen-before-send policy and only begin their transmission when the channel is clear for some period of time; if the channel is busy, back-off and continue to wait for the channel is clear again.

Many commercial Wi-Fi devices have this mechanism hard-coded at the firmware or hardware level. A direct outcome is that a device cannot send frames gaped by specified time interval, but it has to ensure the channel becomes clear again before the subsequent transmission.

We did not cover the detail about this issue in previous sections when mentioning a PWM-like wake-up pattern. Now we are going to show how we plan to overcome this challenge. The chances are that our wake-up frames could be collided by another node's signal, or there could be other frames inserted between two wake-up frames, disrupting our predefined wake-up frame pattern. There are two possible solutions:

- A1: Use a CTS-to-self control frame to hold the channel for a duration long enough

for the transmitter to send the entire wake-up pattern;

- A2: Ignore the timing gap between consecutive frames. As long as the next expected frame comes in time, the wake-up detection process continues; otherwise, go back to sleep.

The idea of using a CTS-to-self frame is intended to simplify the circuit design a lot because once a slot is secured, the transmitter does not need to worry about any collision. But in many commercial devices, CTS-to-self frames are primarily generated in the hardware and beyond our control. In this case, the best thing we can do is set a flag, tell the hardware that the frame has to be protected with a CTS-to-self frame, and let the hardware take care of the reset. However, we still cannot hard code the duration field of a CTS-to-self frame to hold the channel for an intended duration because the hardware will automatically fill in the field based on the duration of the frame it wants to protect. During the time this dissertation is being written, we still cannot identify any off-the-shelf device supporting customized CTS-to-self frame injection. To our best knowledge, using software-defined radios is the only solution, but that would significantly increase the cost.

Another challenge of using a CTS-to-self frame is that there are devices designed not to honor the CTS-to-self frame's duration value if they find the value to be unrealistically large. If so, they will go ahead and begin transmitting their frames in any case. This measure is to counter the so-called "CTS-to-self attack", in which a malicious node keeps broadcasting CTS-to-self frames to hold the channel for long durations, preventing other nodes from sending their traffic. Beacons from the AP will still be sent, so all the devices

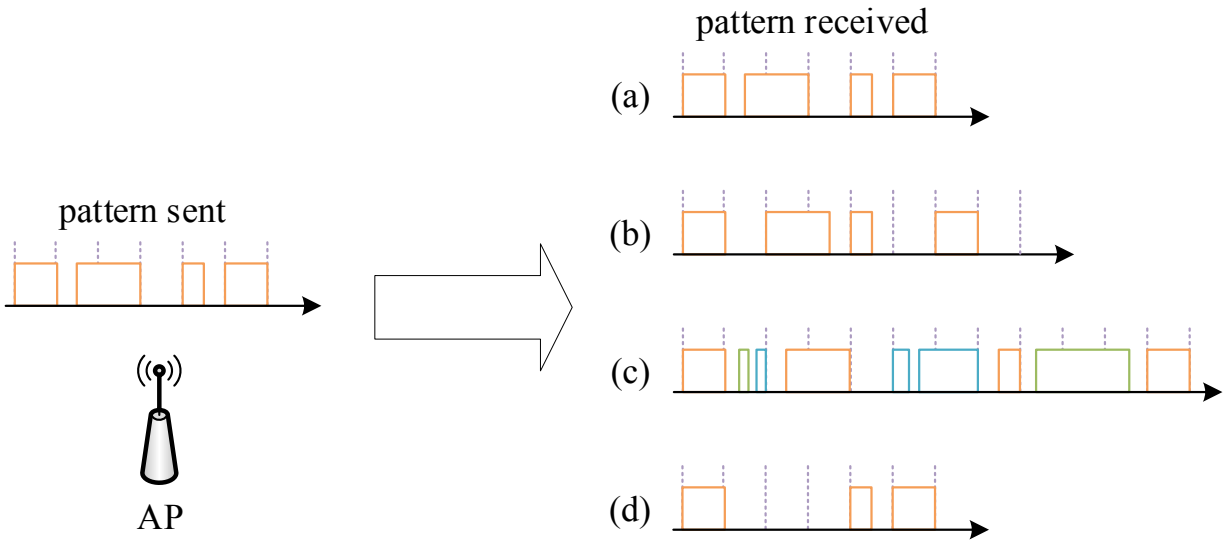

Figure 4.6: A wake-up frame pattern may vary because of congestion. There are four scenarios: (a) it arrives as it is because there is light traffic; (b) it arrives with gap variations, but frame durations are the same; (c) it arrives with insertions of other frames; (d) it arrives with some loss.

are still connected but would experience low throughput due to the attack.

Based on the above analysis, using CTS-to-self frames in commercial hardware becomes less practical. However, without CTS-to-self frames, we cannot guarantee that the timing gaps between two consecutive frames would be fixed. This implies that our WuRx should only pay attention to the frame durations and allow some timing variations to the inter-frame intervals, as long as the next frame of the desired duration comes in time.

Figure 4.6 gives four different scenarios when our injected wake-up pattern arrives at the circuit. In a light traffic scenario (a), the back-off of CSMA/CA is not triggered, so the pattern is not changed upon arrival. However, the long delay between a userspace application and the hardware may distort the pattern's gap durations. In this case, the

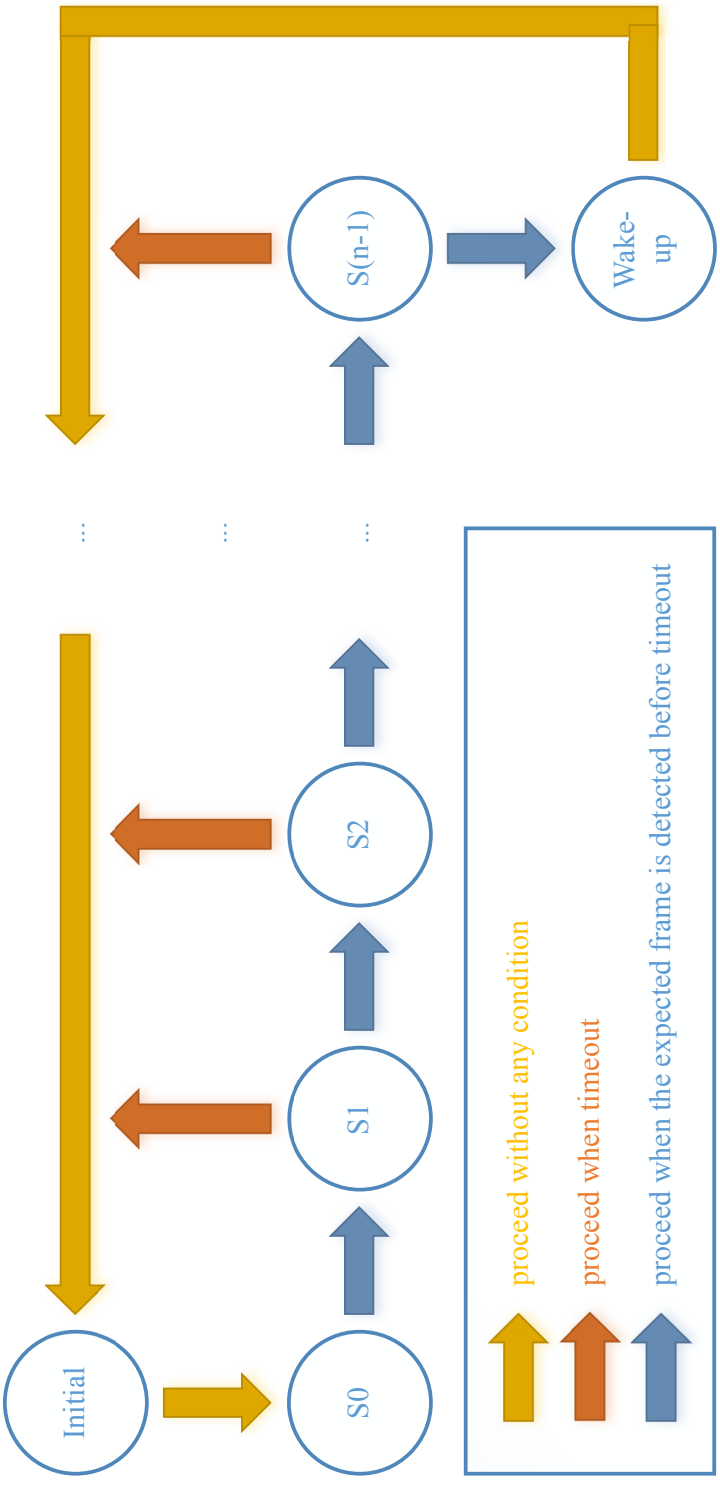

Figure 4.7: The finite state machine model of the wake-up procedure.

received pattern will look like (b), in which the frame durations are the same as those in (a), but gaps are not. If there is traffic among other devices, CSMA/CA mechanism has to be activated for congestion control. Device competing for transmission slots results in a received pattern like that in (c), in which all the wake-up frames are still there but mixed with frames from other devices. We should also consider (b) and (c) as valid patterns. In scenario (d), some frames in the pattern are missing due to noise, interference, or collisions, and the wake-up process will fail.

Figure 4.7 depicts a finite state machine diagram to explain the algorithm. Note that we leave room to use up to n frames and set $n = 4$ in our example. Our wake-up circuit consists of a clock, a counter, a countdown timer, and storage holding the predefined count of frame durations based on the above parameters. We expect a circuit with these components will not consume more than 50 μW of power.

In our choice, the maximum allowed time duration between two consecutive wake-up frames is set to 400 μs based on our experiments and environment. In a channel with light traffic, we can inject two frames separated by 250 μs; however, we expect a busier channel, due to the traffic, this separation could be larger.

4.4 Multicast Wake-up

There are cases where we want to wake up multiple devices at once. For example, we want to get a snapshot of the distribution of air pollutants at a particular time—a straightforward way to spend some time and wake up the sensor device one by one. Alter-

Algorithm 3: wake-up procedure

```
1  State Initial
2      set timeout = 400 μs;
3      set clock_freq = 50 kHz;
4      set frame_durations [4] = [x, y, z, w];
5      goto S0;
6  end
7  State S0
8      repeat
9      until duration of the incoming frame == frame_durations [0];
10     goto S1;
11 end
12 State S1
13     set count-down-timer = timeout;
14     while count-down-timer > 0 do
15         if duration of the incoming frame == frame_durations [1] then
16             goto S2;
17         end
18     end
19     goto Initial;
20 end
```

```
21 State S2
22     set count-down-timer = timeout;
23     while count-down-timer > 0 do
24         if duration of the incoming frame == frame_durations [2] then
25             goto S3;
26         end
27     end
28     goto Initial;
29 end
30 State S3
31     set count-down-timer = timeout;
32     while count-down-timer > 0 do
33         if duration of the incoming frame == frame_durations [3] then
34             goto Wake-up;
35         end
36     end
37     goto Initial;
38 end
```

```
39 State Wake-up
40 |   output wake-up signal;
41 |   goto Initial;
42 end
```

natively, we can implement multicast support so that the same group's device will wake up altogether after receiving a particular pattern of wake-up frames promptly.

4.4.1 Mimicking IP Multicast Solution

IP protocol [56] supports multicast in the way that the destination address in a packet header instructs some but not all the hosts in the same subnet to process the packet. It archives multicast by assigning relative IP addresses to the same group's hosts and uses a mask to inform the hosts whether it is doing a unicast, a multicast, or a broadcast.

Assume host A has 192.168.1.1 and host B has 192.168.1.2 as their IP addresses. Both hosts are given 255.255.255.0 as their mask so that if host C has 192.168.2.1 as its address, we can tell host C is not in the same subnet as A and B. However, if we set all their masks to 255.255.0.0, they will be in the same subnet.

The above example shows that we can carefully assign IP addresses to the hosts and use a mask to slice hosts into groups. The idea can be ported to our case because we design our wake-up pattern similar to IP addressing. Remember that we assign four integers representing the durations of wake-up frames to each device. These four integers

map to those in an IP address. In IPv4 protocol, a mask is an integer between 0 and 32, representing the number of 1's to work as the mask, whereas our previous design did not reserve the room for a mask, so an extension is required.

4.4.2 Adding the Mask

To simplify our design, we use the integer between 0 and 4, telling our devices to match the first n wake-up frames' duration while ignoring the last $(4 - n)$ ones. For example, assume the duration of the wake-up pattern of device A is [152, 252, 352, 452] and that of device B is [152, 252, 352, 472]. If we want to do masking, we need to tell the devices to only pay attention to the first n frames. Therefore, we need to send the mask before wake-up frames, and the pattern would be in the form of `[mask, frame_1, frame_2, frame_3, frame_4]`.

However, a mask could be a variable, and building a circuit that detects for a varying frame duration is more complicated than that for a frame with a fixed duration. A simple solution is to build four frame-1-detection loops instead of one. Another approach would be sending another frame of a fixed duration before the mask frame such as follows `[magic_number, mask, duration_1, duration_2, duration_3, duration_4]`.

The `magic_number` frame has a duration that is rarely seen in Figure 4.4 to prevent false positives. After receiving the magic number, all the devices should pay attention to the `mask` frame, determining how many wake-up frames they need to honor following the mask frame.

One challenge here is that the frame duration representing the mask is not fixed

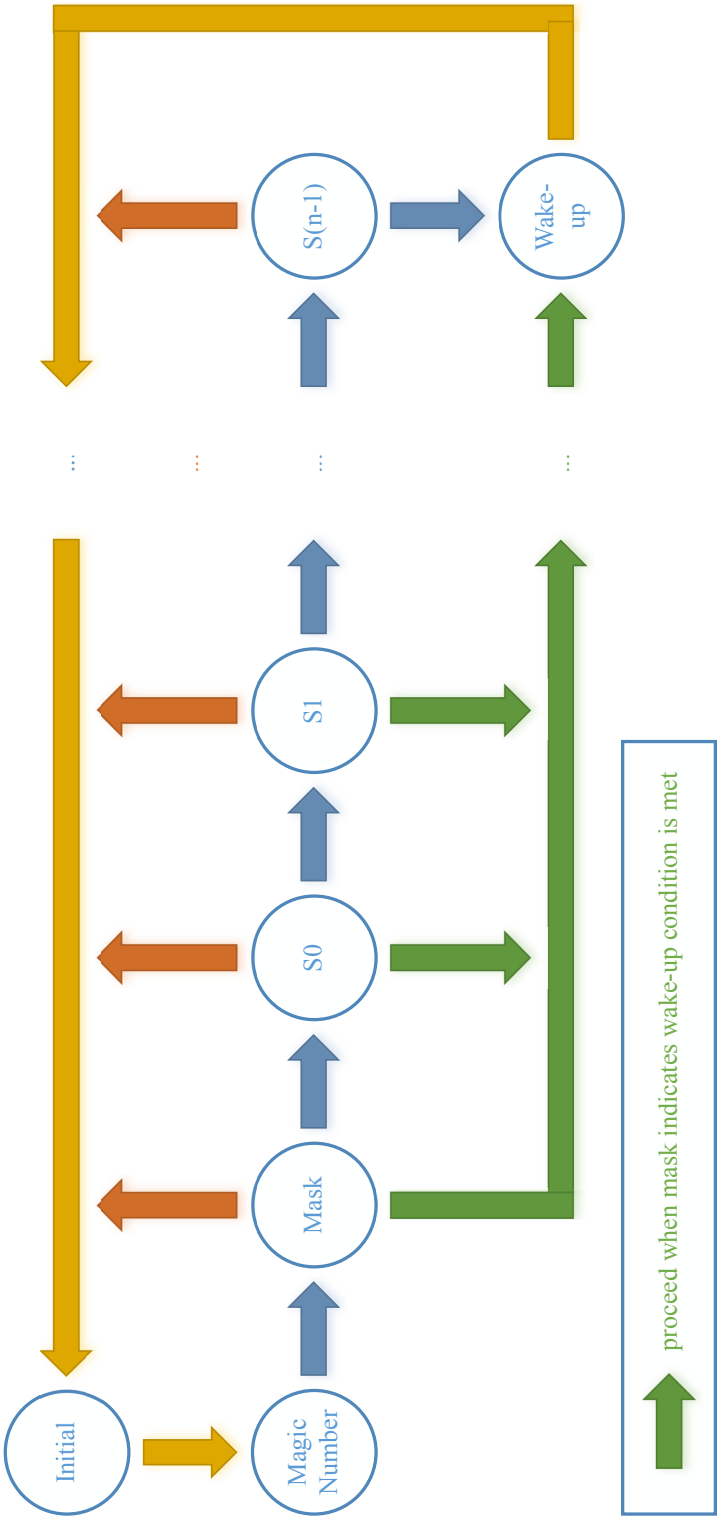

Figure 4.8: The finite state machine supporting multicast wake-up

but could be a constant from a known set. An extra circuit design may be necessary for detecting the `mask` frame. The modified finite state machine for multicast wake-up is shown in Figure ,4.8 and the detailed logic is given in Algorithm 4.

Algorithm 4: wake-up procedure with multicast support

```
1  State Initial
2      set timeout = 400 μs;
3      set clock_freq = 50 kHz;
4      set magic_number = n;
5      set masks [4] = [h, i, j, k];
6      set frame_durations [4] = [x, y, z, w];
7      goto Magic Number;
8  end
9  State Magic Number
10     repeat
11     until duration of the incoming frame == n;
12     goto Mask;
13 end
```

```
14 State Mask
15     set count-down-timer = timeout;
16     while count-down-timer > 0 do
17         if duration of the incoming frame ∈ masks then
18             set mask = masks.index(incoming frame duration);
19             goto S0;
20         end
21     end
22     goto Initial;
23 end
24 State S0
25     if mask == 0 then
26         goto Wake-up
27     end
28     set count-down-timer = timeout;
29     while count-down-timer > 0 do
30         if duration of the incoming frame == frame_durations [0] then
31             goto S1;
32         end
33     end
34     goto Initial;
35 end
```

```
36 State S1
37     if mask == 1 then
38         goto Wake-up
39     end
40     set count-down-timer = timeout;
41     while count-down-timer > 0 do
42         if duration of the incoming frame == frame_durations [1] then
43             goto S2;
44         end
45     end
46     goto Initial;
47 end
```

```
48 State S2
49     if mask == 2 then
50         goto Wake-up
51     end
52     set count-down-timer = timeout;
53     while count-down-timer > 0 do
54         if duration of the incoming frame == frame_durations [2] then
55             goto S3;
56         end
57     end
58     goto Initial;
59 end
```

```
60 State S3
61     if mask == 3 then
62         goto Wake-up
63     end
64     set count-down-timer = timeout;
65     while count-down-timer > 0 do
66         if duration of the incoming frame == frame_durations [3] then
67             goto Wake-up;
68         end
69     end
70     goto Initial;
71 end
72 State Wake-up
73     output wake-up signal;
74     goto Initial;
75 end
```

4.5 Tracking Wi-Fi IoT Devices

Now we had a power-saving solution for Wi-Fi IoT devices, making them more adaptable to general usage. We now move on to tracking these devices. Unlike smartphones that tend to be mobile, IoT devices are mostly static, so a real-time location discovery is less necessary. However, when we first deploy them on the field, we might need to localize them by finding their relative positions to establish a mesh network. Equipping the capability of being tracked will not add too much overhead and can still be done with a software solution.

4.5.1 Reusing Wi-Fi Infrastructure

From a cost point of view, we should avoid adding extra hardware unless the expected benefit could offset such a cost. We believe that adding new features that do not depend on new hardware would reduce deployment costs and create a better marketing strategy that gives people more incentive to try. With this idea in mind, we should reuse the framework we built in Chapter 3 and adopt the policy that the device to be tracked should react against the commands sent by its associated AP. For example, we can build a command that asks the device to perform off-channel scans similar to the purpose of a BTM request frame in Figure 3.4 because all we need is to have the device ping multiple APs within the range so that we can do triangulation.

In this design, besides APs, we can reuse the controller in the previous chapter and only need to replace the IEEE 802.11v support with our own command set for our

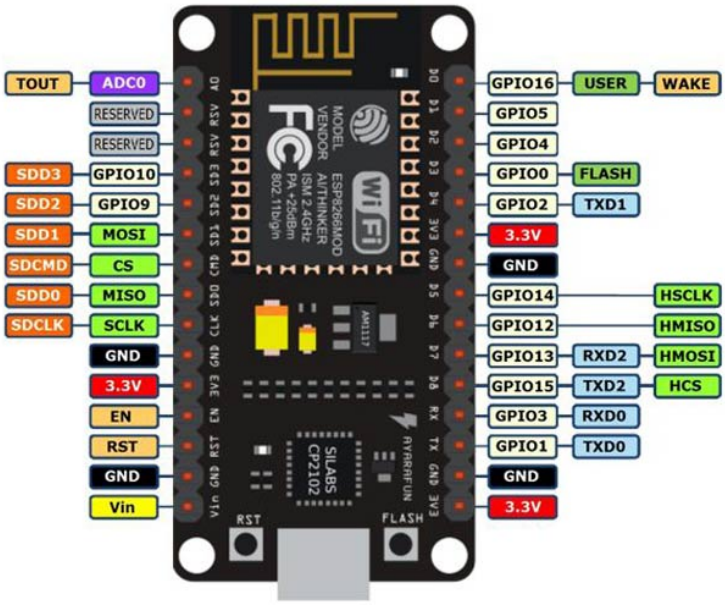

Figure 4.9: ESP8266 chip and its pin assignment

Wi-Fi IoT devices because these devices do not support the IEEE 802.11v standard in general. There is no need to implement full support.

We realized our idea on an ESP8266 chip, a low cost (around $3 each) Wi-Fi IoT platform welcomed by IoT and smart cities communities. Figure 4.9 shows the pin assignment of the chip. Its actual size is around one-third of a business card. Its manufacturer, Espressif Systems provide an open-source SDK. However, they did not release the firmware nor the driver source codes of its Wi-Fi module, but they instead provided the binary file and a C/C++ header file describing the available functions in the SDK. This measure is typically seen in many development boards. It blocks us from diving deep into their hardware and also implies that it would be challenging to enable the complete IEEE 802.11v support to the chip.

Building our own command sets is a workaround to this issue. What we need is a way to orchestrate selective scans at the device. The IEEE 802.11v standard was a way to achieve this goal, and the beauty of this approach is that we do not need to modify smartphones, thus reducing the privacy concerns; here, we do not have native support of the IEEE 802.11v, so we decide to implement our commands for orchestrated selective scans.

We observed that ESP8266 would not disassociate with its current AP when performing off-channel scans. It generally took 30-40 milliseconds for an ESP8266 to scan one channel in a 2.4 GHz band when doing active scans. Assume we only deploy our APs on the three non-overlapping channels in the 2.4 GHz band, in addition to the one it currently associates with, it only needs to scan two other channels to leave signal strength measurements on APs for triangulation, and that will take 60-80 milliseconds, which is shorter than the regular beacon interval. One can check Appendix B for example codes.

4.5.2 Putting Everything Together

The design's final step is to connect the wake-up circuit's output to the ESP8266 board as an external wake-up signal. The system is put in the power-saving mode most of the time. When there is downlink traffic to the system, we send the predefined frame pattern to wake it up. If it succeeds, the ESP8266 chip is set to connect to an available AP, reporting its status to the controller and ready for a commend.

We could not carry out a system-wide test for our wake-up circuit and ESP8266 chip when this dissertation is being written due to the pandemic. We believe this would

be logically possible because we managed to wake-up the circuit with a frame pattern generated through frame injection. However, more tests may still be required to verify the robustness of the wake-up circuit design.

Chapter 5

Wi-Fi Self-Organizing Networks and Their Future

This dissertation gave practical device tracking solutions that require software support and hereby laying down the base for self-organizing networks. The idea was to uncover the best possible solutions under the existing hardware infrastructure and only introduce new hardware design when necessary.

5.1 Retrospect to the Proposed Solutions

We turned them into wireless SDN nodes for the devices having enough computing power and open for full access. These devices formed an SD MANET or smart mesh networks. This network's key features include time-sensitive local data exchange, so P2P and multihop transmission best suit this traffic. The network topology may frequently

change, so an on-demand route searching would unlikely to guarantee short delays, and we introduced some proactive measures instead.

A locally centralized controller oversees the network dynamics and collects the data. We can also run machine learning on the controller to estimate upcoming topology changes and pre-cache control policies onto relative SDN nodes to minimize network traffic delay. To further boost the network's robustness, the controller may detect malicious nodes by monitoring the traffic pattern in the network. If suspicious behavior is found, the control isolates the node by installing policies onto the neighboring nodes.

The above-mentioned approach requires certain levels of control on software installation. One might doubt how a network operator can make their users install particular applications on their devices. A solution is to run a rent-and-subscribe mode instead of asking the user to buy off the hardware and software. The Rent-and-subscribe model makes users rent the hardware and subscribe to software updates in exchange for the services. These something-as-a-service models provide sustainable cash flow to the business and overcome the challenge of asking users to install particular applications.

In the meantime, we expect there are still privately owned smart devices that continue not to allow SDN module installation due to business decisions or security concerns. To track these devices, we have to use whatever the device support. Currently, we rely on the support of IEEE 802.11v/r standards. Through an orchestrated set of control frames, we were able to make the device expose their locations. Although our work seemed to rely on Quantenna's reference design and their SDK, which gave us a proof of concept, it can be further extended to `hostapd` application and compatible devices using the same

control logic, dropping the dependency of Quantenna's framework. `hostapd` is an open-source AP management application seen in almost every Linux-based AP. It is responsible for the AP-STA handshake and sending the BTM request frames to compatible devices. An extension would be to build a new protocol between `hostapd` the controller so that the controller can ask the AP to send BTM request frames when necessary.

Another possible extension would be at the device's end. Although all Apple's iOS devices and Windows-running devices have proprietary Wi-Fi manager, most Android and Linux-running devices use `wpa_supplicant` and their Wi-Fi manager. We noticed that there remains undid work in its BTM control frame handling process. That might explain why sometimes there were unexpected behaviors, and we needed to disassociate the devices. We believe there is a way to extend `wpa_supplicant` so that Android and Linux-running devices can react to our customized 802.11v BTM request frames more smoothly.

Lastly, we proposed an approach to track Wi-Fi IoT devices using almost the same setup as we used to track smart devices. Enabling a feature similar to 802.11v BTM request and reply handshake was straightforward but coming up with a standard-compatible energy-saving way was more challenging. We introduced a low-powered wake-up circuit as a front end to the regular, power-consuming Wi-Fi interface for channel monitoring. Our design also includes identification and multicasting features so that a single AP can wake up multiple devices simultaneously.

5.2 Potential Support on Future Wi-Fi Standards

The first commercially available Wi-Fi standard, IEEE 802.11b were proposed in 1999. It operates in the 2.4 GHz band using 20 MHz bandwidth and gives a maximum raw data rate of 11 Mbps. Although proposed almost simultaneously, IEEE 802.11a, which also uses 20 MHz bandwidth but operates in the 5 GHz band, became commercially available one year later in the year 2000. It was also the first Wi-Fi standard using frame-based OFDM modulation, giving its maximum raw data rate at 54 Mbps. As mentioned in Chapter 4, using the 5 GHz band gives 802.11a a significant advantage. First, neighboring channels in the 2.4 GHz band are overlapped against each other, causing inter-channel-interference, whereas those in the 5 GHz band are separate and free from inter-channel-interference. Second, there are communication standards other than Wi-Fi operating in 2.4 GHz, such a crowded environment makes connection unstable, and coexisting different standards becomes a headache. On the other hand, a higher carrier frequency of 5 GHz also brings some disadvantages. For example, the overall coverage of 802.11a is smaller due to path loss and fading characteristics. Fortunately, the fundamental propagation advantages of the OFDM signal can help offset these disadvantages.

Three years later, the 2.4 GHz version of 802.11a, 802.11g standard became available and soon dominated the market for almost six years until 802.11n/Wi-Fi 4/Wireless N appeared in 2009. 802.11n standard was the first standard supporting MIMO and took advantage of spatial diversity. Interestingly, in a 4-stream scenario, the theoretically achievable throughput is 800 Mbps rather than $4 \times 150 = 600$ Mbps. This is because

802.11n uses 256 QAM-OFDM modulation, which comes with an additional 30% spatial gain. The current mainstream 802.11ac standard is branded as Wi-Fi 5 by Wi-Fi Alliance. It reaches its maximum raw data rate of 1,733 Mbps when operated with an 80-MHz channel and four streams. Many commercial dual-band Wi-Fi APs sum the maximum raw data rate in 2.4 GHz band and 5 GHz band together and use "AC2600" for marketing. Table 5.1 summarizes the dates and features of each above-mentioned Wi-Fi standard.

The resource management protocols 802.11v and 802.11r first appeared as supplements for the 802.11n standard and continue to be supported by the 802.11ac standard. We believe these two protocols will also be supported by the future Wi-Fi standards for backward compatibility and only be replaced when more advanced ones become available.

The latest Wi-Fi 6 standards, 802.11ax, operates at the same frequency band as 802.11n and 802.11ac. The makes the channel characteristics of 802.11ax similar to 802.11n and 802.11ac. The future Wi-Fi 6E will run in the 6 GHz band, and we can expect it will have a smaller coverage than that of Wi-Fi 5 under the same transmission power and will require a denser deployment to cover the area of service. As discussed in earlier chapters, because we make AP as our sensors, AP's denser deployment brings us a denser deployment of sensors. The devices will undoubtedly experience more frequent handoffs, but that shortcoming will become our tracking performance.

In sum, we believe as long as the carrier frequency becomes higher and AP's deployment becomes denser, the idea covered in this dissertation will be more sustainable and perform better.

Table 5.1: Status, features, and commercially available dates of Wi-Fi standards

Name	**Standard**	**Commercially Available**	**Max Single-stream Speed**	**Channel Width**	**Frequency Band**	**Status**
N/A	802.11b	1999	11 Mbps	20 MHz	2.4 GHz	obsolete
N/A	802.11a	2000	54 Mbps	20 MHz	5 GHz	obsolete
N/A	802.11g	2003	54 Mbps	20 MHz	2.4 GHz	obsolete
Wi-Fi 4	802.11n	2009	150 Mbps	20/40 MHz	2.4/5 GHz	legacy
Wi-Fi 5	802.11ac	2012	433 Mbps	20/40/80 MHz	5 GHz	mainstream
N/A	802.11ad	2015	7 Gbps	2.16 GHz	60 GHz	limited use
Wi-Fi 6	802.11ax	2019	1200 Mbps	20/40/80/160 MHz	2.4/5 GHz	latest
Wi-Fi 6E	802.11ax	2020	1200 Mbps	20/40/80/160 MHz	6 GHz	upcoming

Appendix A

BTM Frame Process Logic

When an IEEE 802.11v control frame arrives at a device running `wpa_supplican-2.6` [34], several functions in `wnm_sta.c` are supposed to handle the response.

```
1516 void ieee802_11_rx_wnm_action(struct wpa_supplicant *wpa_s,
1517              const struct ieee80211_mgmt *mgmt, size_t len)
1518 {
1519   const u8 *pos, *end;
1520   u8 act;
     ...
1538   switch (act) {
1538   case WNM_BSS_TRANS_MGMT_REQ:
1538     ieee802_11_rx_bss_trans_mgmt_req(wpa_s, pos, end,
1538              !(mgmt->da[0] & 0x01));
1538     break;
```

If the frame's action code is `WNM_BSS_TRANS_MGMT_REQ`, it calls a handler function

to continue the process.

```
1538 static void ieee802_11_rx_bss_trans_mgmt_req(struct wpa_supplicant *
       wpa_s,
1538                 const u8 *pos, const u8 *end,
1538                 int reply)
1538 {
       ...
1203   if (wpa_s->wnm_mode & WNM_BSS_TM_REQ_DISASSOC_IMMINENT) {
1203     wpa_msg(wpa_s, MSG_INFO, "WNM: Disassociation Imminent - "
1203       "Disassociation Timer %u", wpa_s->wnm_dissoc_timer);
1203     if (wpa_s->wnm_dissoc_timer && !wpa_s->scanning) {
1203       /* TODO: mark current BSS less preferred for
1203        * selection */
1203       wpa_printf(MSG_DEBUG, "Trying to find another BSS");
1203       wpa_supplicant_req_scan(wpa_s, 0, 0);
1203     }
1203   }
       ...
1220   if (wpa_s->wnm_mode & WNM_BSS_TM_REQ_PREF_CAND_LIST_INCLUDED) {
1220     unsigned int valid_ms;
1220
1221     wpa_msg(wpa_s, MSG_INFO, "WNM: Preferred List Available");
         ...
1266       valid_ms = valid_int * beacon_int * 128 / 125;
1266     wpa_printf(MSG_DEBUG, "WNM: Candidate list valid for %u ms",
1266         valid_ms);
```

```
1266    os_get_reltime(&wpa_s->wnm_cand_valid_until);
1266    wpa_s->wnm_cand_valid_until.sec += valid_ms / 1000;
1266    wpa_s->wnm_cand_valid_until.usec += (valid_ms % 1000) * 1000;
1266    wpa_s->wnm_cand_valid_until.sec +=
          wpa_s->wnm_cand_valid_until.usec / 1000000;
1266    wpa_s->wnm_cand_valid_until.usec %= 1000000;
          ...
1287    /*
1287     * Try to use previously received scan results, if they are
1287     * recent enough to use for a connection.
1287     */
1287    if (wpa_s->last_scan_res_used > 0) {
1287      struct os_reltime now;
1287
1288      os_get_reltime(&now);
1287      if (!os_reltime_expired(&now, &wpa_s->last_scan, 10)) {
1287        wpa_printf(MSG_DEBUG,
1287             "WNM: Try to use recent scan results");
1287        if (wnm_scan_process(wpa_s, 0) > 0)
1287          return;
1287        wpa_printf(MSG_DEBUG,
1287             "WNM: No match in previous scan results - try a new scan
      ");
1287      }
1287    }
1287
```

```
1288    wnm_set_scan_freqs(wpa_s);
1287    if (wpa_s->wnm_num_neighbor_report == 1) {
1287      os_memcpy(wpa_s->next_scan_bssid,
1287          wpa_s->wnm_neighbor_report_elements[0].bssid,
1287          ETH_ALEN);
1287      wpa_printf(MSG_DEBUG,
1287           "WNM: Scan only for a specific BSSID since there is only a
    single candidate "
1287           MACSTR, MAC2STR(wpa_s->next_scan_bssid));
1287    }
1287    wpa_supplicant_req_scan(wpa_s, 0, 0);
        ...
1327 }
```

In line 1203, if `WNM_BSS_TM_REQ_DISASSOC_IMMINENT` bit is set, the device is to be disassociated soon and `wpa_supplicant` should request a scan quickly. However, at this point, the device has no idea whether there are preferred candidates or it has to do a full-channel scan in the worst case.

Starting at line 1220, `wpa_supplicant` process the preferred candidate list, uncovering more clue for the handoff. It then checks if the previous scan results in cache. If the result is still no more aged than 10 seconds, it will determine if all the candidates are in the result. Otherwise, it should request a new scan.

Finally, if there is only one candidate, `wpa_supplicant` should request a unicast scan, including the BSSID of the candidate in the probe request; otherwise, it will demand

a broadcast scan.

One interesting finding is that the abridged bit (A) is ignored in `wpa_supplicant`. This might be that the author has not yet come up with a way to process it. The abridged bit is used to indicate the severity of the roaming request according to the standard [9].

`wpa_supplicant_req_scan()` is the function use for scheduling a scan. It does not kick off the scan immediately after it is called but will put a request in the event loop because `wpa_supplicant` is a multi-thread application.

The declaration and the implementation of `wpa_supplicant_req_scan()` is in `scan.c` are in `scan.h` and `scan.c`, respectively. Here is how it was implemented.

```
1327 {
1327   int res;
1327
1328   if (wpa_s->p2p_mgmt) {
1327     wpa_dbg(wpa_s, MSG_DEBUG,
1327       "Ignore scan request (%d.%06d sec) on p2p_mgmt interface",
1327       sec, usec);
1327     return;
1327   }
1327
1328   res = eloop_deplete_timeout(sec, usec, wpa_supplicant_scan, wpa_s,
1327             NULL);
1327   if (res == 1) {
1327     wpa_dbg(wpa_s, MSG_DEBUG, "Rescheduling scan request: %d.%06d sec
      ",
```

```
1327        sec, usec);
1327    } else if (res == 0) {
1327      wpa_dbg(wpa_s, MSG_DEBUG, "Ignore new scan request for %d.%06d
        sec since an earlier request is scheduled to trigger sooner",
1327        sec, usec);
1327    } else {
1327      wpa_dbg(wpa_s, MSG_DEBUG, "Setting scan request: %d.%06d sec",
1327        sec, usec);
1327      eloop_register_timeout(sec, usec, wpa_supplicant_scan, wpa_s,
        NULL);
1327    }
1327 }
```

If there is already a pending scan request, `wpa_supplicant_req_scan()` should reschedule it; otherwise, it can go ahead and schedule the scan.

`wps_supplicant_scan()` function in `scan.c` handles scan requests. It consists of more than 400 lines of codes and is called by the event loop. Unlike `ieee802ch_11_rx_bss_trans_mgmt_req()` function in `wnm_sta.c` that primarily set the scan parameters according to the information in the BTM request frame, `wps_supplicant_scan()` function updates the scan parameters based on system settings as well as hardware capabilities. It would be challenging to override system settings or hardware capabilities on a user device because that would involve system-level modifications so we decide to let them be and focus on the information in BTM request frames.

Another noticeable finding is that `wps_supplicant_scan()` function calls `wpa`

_supplicant_extra_ies at line 982, a function defined at line 462 in scan.c. This function processes the information elements (IEs) defined by different vendors and opens a good way for the extension. For example, we can make ourselves a vendor that put all the roaming parameters in IEs and then modify wpa_supplicant to support our parameters without violating the standard. For the devices running unmodified software or those not relying on wpa_supplicant for Wi-Fi connections, they will ignore our customized IEs and continue to operate as they should.

Appendix B

Off-channel Scan Implementation on ESP8266

assoc_and_off_channel_scan.ino:

```
#include <ArduinoJson.h>
#include <ESP8266WiFi.h>
#include <ESPAsyncTCP.h>
#include <vector>

#define BLINK_PERIOD 250
long lastBlinkMillis;
bool ledState;

#define SCAN_PERIOD 5000
long lastScanMillis;
```

```
1327
1328 const char *ssid     = "MyPlace";
1327 const char *password = "22627298";
1327 int current_channel;
1327
1328 // command flags
1327 bool cmd_request_scan = false;
1327 bool cmd_scan_result = false;
1327 StaticJsonDocument<2048> ap_list;
1327
1328 // a global vector holding active clients
1327 static std::vector<AsyncClient *> clients; // a list to hold all
       clients
1327
1328 /* clients events */
1327 static void handleError(void *arg, AsyncClient *client, int8_t error)
       {
1327   Serial.printf("\n\r connection error %s from client %s \n", client
       ->errorToString(error), client->remoteIP().toString().c_str());
1327 }
1327
1328 static void handleData(void *arg, AsyncClient *client, void *data,
       size_t len) {
1327   // A buffer to hold the reply string. Sadly, we only have limited
       memory here so need to be careful.
1327   char reply[1024];
```

```
1327  // print out where the data come from
1327  Serial.printf("\n\rInfo: data received from client %s: ", client->
       remoteIP().toString().c_str());
1327
1328  // print out the received data
1327  Serial.write((uint8_t *)data, len);
1327  Serial.printf(", len = %d", len);
1327
1328  if (!memcmp(data, "scan_result", max((int) len, 11))) {
1327    sprintf(reply, "Show off-channel scan reults\n");
1327    if (ap_list.isNull())
1327      sprintf(reply, "Scan result not yet available\n");
1327    else {
1327      serializeJsonPretty(ap_list, reply);
1327      Serial.printf("\n\rcontent length = %d", measureJsonPretty(
       ap_list));
1327    //cmd_scan_result = true;
1327    }
1327  } else if (!memcmp(data, "scan", max((int) len, 4))) {
1327    sprintf(reply, "Request an off-channel scan\n");
1327    cmd_request_scan = true;
1327  }
1327
1328  // reply to client
1327  if (client->space() > strlen(reply) && client->canSend()) {
1327    client->add(reply, strlen(reply));
```

```
1327      client->send();
1327    }
1327  }
1327
1328  static void handleDisconnect(void *arg, AsyncClient *client) {
1327    Serial.printf("\n\rInfo: client %s disconnected \n", client->
          remoteIP().toString().c_str());
1327  }
1327
1328  static void handleTimeOut(void *arg, AsyncClient *client, uint32_t
          time) {
1327    Serial.printf("\n\rWarning: client ACK timeout ip: %s \n", client->
          remoteIP().toString().c_str());
1327  }
1327
1328  /* server events */
1327  static void handleNewClient(void *arg, AsyncClient *client) {
1327    Serial.printf("\n\rInfo: a new client has been connected to server,
            ip: %s", client->remoteIP().toString().c_str());
1327
1328    // add to list
1327    clients.push_back(client);
1327
1328    // register events
1327    client->onData(&handleData, NULL);
1327    client->onError(&handleError, NULL);
```

```
1327   client->onDisconnect(&handleDisconnect, NULL);
1327   client->onTimeout(&handleTimeOut, NULL);
1327 }
1327
1328 void scan_and_update_ap_list() {
1327   /* if async_scan is true, scanNetworks() will exit right away and
        we need to check the return value of scanComplete() and
1327    * see if the scan is complete. If async_scan is false, then
        scanNetworks() is blocking and scanComplete() will always
1327    * return the scan result. In our application, we need to scan only
         selective channels. For a single thread application
1327    * that only deals with Wi-Fi, we can make it blocking. According
        to the discussion on GitHub, Adruio has only single core
1327    * and is non-preemptive, so I better come up with a single thread
        solution.
1327    */
1327   bool async_scan = false, show_hidden = false, passive = false;
1327   int chan = 1, n = -1;
1327   long lastActionMillis;
1327   ap_list.clear();
1327
1328   // scan channel 1
1327   Serial.printf("\n\rStart scanning channel %u...", chan);
1327   lastActionMillis = millis();
1327   WiFi.scanNetworks(async_scan, show_hidden, chan, NULL, passive);
1327   n = WiFi.scanComplete();
```

```
1327
1328  Serial.printf(" %d network(s) found, %d ms spent", n, millis() -
       lastActionMillis);
1327  while (n > 0) {
1327    n--;
1327    Serial.printf("\n\r%d: %s, %s, Ch:%d (%ddBm) %s", n + 1, WiFi.
       SSID(n).c_str(), WiFi.BSSIDstr(n).c_str(), WiFi.channel(n), WiFi.
       RSSI(n), WiFi.encryptionType(n) == ENC_TYPE_NONE ? "open" : "");
1327
1328    JsonObject ap = ap_list.createNestedObject(WiFi.BSSIDstr(n));
1327    if (!ap)
1327      Serial.printf("\n\rError: out of memory.");
1327
1328    ap["SSID"] = WiFi.SSID(n);
1327    ap["channel"] = WiFi.channel(n);
1327    ap["RSSI"] = WiFi.RSSI(n);
1327    //ap["encryption"] = WiFi.encryptionType(n) == ENC_TYPE_NONE ? "
       open" : "";
1327  }
1327  WiFi.scanDelete();
1327
1328  // scan channel 6
1327  chan = 6;
1327  Serial.printf("\n\rStart scanning channel %u...", chan);
1327  lastActionMillis = millis();
1327  WiFi.scanNetworks(async_scan, show_hidden, chan, NULL, passive);
```

```
1327  n = WiFi.scanComplete();
1327
1328  Serial.printf(" %d network(s) found, %d ms spent", n, millis() -
       lastActionMillis);
1327  while (n > 0) {
1327    n--;
1327    Serial.printf("\n\r%d: %s, %s, Ch:%d (%ddBm) %s", n + 1, WiFi.
       SSID(n).c_str(), WiFi.BSSIDstr(n).c_str(), WiFi.channel(n), WiFi.
       RSSI(n), WiFi.encryptionType(n) == ENC_TYPE_NONE ? "open" : "");
1327
1328    JsonObject ap = ap_list.createNestedObject(WiFi.BSSIDstr(n));
1327    if (!ap)
1327      Serial.printf("\n\rError: out of memory.");
1327
1328    ap["SSID"] = WiFi.SSID(n);
1327    ap["channel"] = WiFi.channel(n);
1327    ap["RSSI"] = WiFi.RSSI(n);
1327    //ap["encryption"] = WiFi.encryptionType(n) == ENC_TYPE_NONE ? "
       open" : "";
1327  }
1327  WiFi.scanDelete();
1327
1328  // scan channel 11
1327  chan = 11;
1327  Serial.printf("\n\rStart scanning channel %u...", chan);
1327  lastActionMillis = millis();
```

```
1327   WiFi.scanNetworks(async_scan, show_hidden, chan, NULL, passive);
1327   n = WiFi.scanComplete();
1327
1328   Serial.printf(" %d network(s) found, %d ms spent", n, millis() -
        lastActionMillis);
1327   while (n > 0) {
1327     n--;
1327     Serial.printf("\n\r%d: %s, %s, Ch:%d (%ddBm) %s", n + 1, WiFi.
       SSID(n).c_str(), WiFi.BSSIDstr(n).c_str(), WiFi.channel(n), WiFi.
       RSSI(n), WiFi.encryptionType(n) == ENC_TYPE_NONE ? "open" : "");
1327
1328     JsonObject ap = ap_list.createNestedObject(WiFi.BSSIDstr(n));
1327     if (!ap)
1327       Serial.printf("\n\rError: out of memory.");
1327
1328     ap["SSID"] = WiFi.SSID(n);
1327     ap["channel"] = WiFi.channel(n);
1327     ap["RSSI"] = WiFi.RSSI(n);
1327     //ap["encryption"] = WiFi.encryptionType(n) == ENC_TYPE_NONE ? "
       open" : "";
1327   }
1327   WiFi.scanDelete();
1327 }
1327
1328 void print_ap_list() {
1327   if (!ap_list.isNull())
```

```
1327      serializeJson(ap_list, Serial);
1327    else
1327      Serial.printf("\n\rWarning: scan results not available.");
1327  }
1327
1328  void setup() {
1327    Serial.begin(115200);
1327    Serial.println();
1327
1328    pinMode(LED_BUILTIN, OUTPUT);
1327
1328    WiFi.mode(WIFI_STA);
1327    WiFi.disconnect();
1327
1328    // Connect to an AP and test if off-channel scans would break the
        connection.
1327    WiFi.begin(ssid, password);
1327    while (WiFi.status() != WL_CONNECTED) {
1327      delay(500);
1327      Serial.print(".");
1327    }
1327
1328    current_channel = WiFi.channel();
1327
1328    Serial.printf("\n\rInfo: Wi-Fi connected, local IP address is %s",
        WiFi.localIP().toString().c_str());
```

```
1327
1328    AsyncServer* server = new AsyncServer(7050); // start listening on
         tcp port 7050
1327    server->onClient(&handleNewClient, server);
1327    server->begin();
1327  }
1327
1328  void loop() {
1327    long currentMillis = millis();
1327
1328    // blink LED
1327    if (currentMillis - lastBlinkMillis > BLINK_PERIOD) {
1327      digitalWrite(LED_BUILTIN, ledState);
1327      ledState = !ledState;
1327      lastBlinkMillis = currentMillis;
1327    }
1327
1328    // trigger Wi-Fi network scan
1327    if (cmd_request_scan && (currentMillis - lastScanMillis >
         SCAN_PERIOD)) {
1327      scan_and_update_ap_list();
1327      lastScanMillis = millis();
1327      cmd_request_scan = false;
1327    }
1327
1328    // print scan result
```

```
1327   if (cmd_scan_result) {
1327     print_ap_list();
1327     cmd_scan_result = false;
1327   }
1327 }
```

Bibliography

[1] B. Kaufman and B. Aazhang, "Cellular networks with an overlaid device to device network," in *Asilomar Conference on Signals, Systems and Computers*, Oct. 2008, pp. 1537–1541.

[2] C. Perkins, E. Belding-Royer, and S. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing," IETF, RFC 3561, Jul. 2003.

[3] T. Clausen and P. Jacquet, "Optimized Link State Routing Protocol (OLSR)," IETF, RFC 3626, Oct. 2003.

[4] "Software-Defined Networking (SDN) Definition." [Online]. Available: https://www.opennetworking.org/sdn-resources/sdn-definition

[5] S. Corson and J. Macker, "Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations," RFC 2501, Jan. 1999.

[6] 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, Mar. 2008.

[7] International Telecommunication Union (ITU), "G.114 : One-way transmission time." [Online]. Available: https://www.itu.int/rec/T-REC-G.114

[8] Y. Chen, T. Farley, and N. Ye, "QoS Requirements of Network Applications on the Internet," *Inf. Knowl. Syst. Manag.*, vol. 4, no. 1, pp. 55–76, Jan. 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1234242.1234243

[9] "IEEE Standard for Information technology—Telecommunications and information exchange between systems Local and metropolitan area networks—Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," Dec 2016.

[10] M. Bor, J. Vidler, and U. Roedig, "Lora for the internet of things," in *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, ser. EWSN '16. USA: Junction Publishing, 2016, p. 361–366.

[11] "Ieee standard for ethernet," *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, pp. 1–5600, 2018.

[12] "Open vSwitch." [Online]. Available: http://openvswitch.org/

[13] "CPqD - An OpenFlow 1.3 switch." [Online]. Available: https://github.com/CPqD/ofsoftswitch13

[14] "Raspberry Pi Model B+," https://www.raspberrypi.org/products/model-b-plus/, Raspberry Pi Foundation, 2014.

[15] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an Operating System for Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, Jul. 2008.

[16] "POX." [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[17] "Beacon." [Online]. Available: https://openflow.stanford.edu/display/Beacon

[18] "Floodlight." [Online]. Available: http://floodlight.openflowhub.org/

[19] "Ryu." [Online]. Available: https://osrg.github.io/ryu/

[20] "Onos - open network operating system." [Online]. Available: http://onosproject.org/

[21] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)," IETF, RFC 4271, Jan. 2006.

[22] J. Moy, "OSPF Version 2," IETF, STD 54, Apr. 1998.

[23] "SDN MANET application for ONOS." [Online]. Available: https://github.com/chinghanyu/onos-wfwd

[24] "Project HSMM-Pi." [Online]. Available: https://github.com/urlgrey/hsmm-pi

[25] "iPerf - The ultimate speed test tool for TCP, UDP and SCTP." [Online]. Available: https://iperf.fr/

[26] W. paper, "SDN for WiFi. OpenFlow-enabling the wireless LAN can bring new levels of agility."

[27] Chien-Chao Tseng, Kuang-Hui Chi, Ming-Deng Hsieh, and Hung-Hsing Chang, "Location-based fast handoff for 802.11 networks," *IEEE Communications Letters*, vol. 9, no. 4, pp. 304–306, April 2005.

[28] I. Ramani and S. Savage, "SyncScan: practical fast handoff for 802.11 infrastructure networks," in *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, vol. 1, March 2005, pp. 675–684 vol. 1.

[29] V. Brik, A. Mishra, and S. Banerjee, "Eliminating Handoff Latencies in 802.11 WLANs Using Multiple Radios: Applications, Experience, and Evaluation," in *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 27–27. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251086.1251113

[30] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure, "Exploring mobile/wifi handover with multipath tcp," in *Proceedings of the 2012 ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design*, ser. CellNet '12. New York, NY, USA: ACM, 2012, pp. 31–36. [Online]. Available: http://doi.acm.org/10.1145/2342468.2342476

[31] Y. Lim, Y. Chen, E. M. Nahum, D. Towsley, and K. Lee, "Cross-layer path management in multi-path transport protocol for mobile devices," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, April 2014, pp. 1815–1823.

[32] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses," Internet Requests for Comments, RFC Editor, RFC 6824, January 2013, http://www.rfc-editor.org/rfc/rfc6824.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6824.txt

[33] A. Bhartia, B. Chen, D. Pallas, and W. Stone, "Clientmarshal: Regaining control from wireless clients for better experience," in *The 25th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '19. New York, NY, USA: ACM, 2019, pp. 6:1–6:16. [Online]. Available: http://doi.acm.org/10.1145/3300061.3300135

[34] J. Malinen, "Linux WPA/WPA2/IEEE 802.1X Supplicant." [Online]. Available: https://w1.fi/wpa_supplicant/

[35] Quantenna Communications, "QSR1000 reference AP design." [Online]. Available: http://www.quantenna.com/products/qsr1000/

[36] "Quantenna WiSoC Device List." [Online]. Available: https://wikidevi.com/wiki/Quantenna

[37] H. Lim, L. . Kung, J. C. Hou, and H. Luo, "Zero-configuration, robust indoor localization: Theory and experimentation," in *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, April 2006, pp. 1–12.

[38] M. Kotaru, K. Joshi, D. Bharadia, and S. Katti, "SpotFi: Decimeter Level Localization Using WiFi," in *Proceedings of the 2015 ACM Conference*

on Special Interest Group on Data Communication, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 269–282. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787487

[39] "Wireshark." [Online]. Available: https://www.wireshark.org/

[40] Cisco Systems, Inc., "iPhone 6 Roaming Behavior and Optimization." [Online]. Available: https://bit.ly/30R5a4U

[41] "IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 15.1a: Wireless Medium Access Control (MAC) and Physical Layer (PHY) specifications for Wireless Personal Area Networks (WPAN)," *IEEE Std 802.15.1-2005 (Revision of IEEE Std 802.15.1-2002)*, pp. 1–700, 2005.

[42] Bluetooth SIG (2020), "Specification of the BluetoothSystem - Covered Core Package version: 5.2." [Online]. Available: https://www.bluetooth.com/specifications/bluetooth-core-specification/

[43] "IEEE Standard for Low-Rate Wireless Networks," *IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015)*, pp. 1–800, 2020.

[44] Lin Gu and J. A. Stankovic, "Radio-triggered wake-up capability for sensor networks," in *Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004.*, 2004, pp. 27–36.

[45] W. L. Leow, H. Pishro-Nik, and D. Ni, "Delay and energy tradeoff in multi-state wireless sensor networks," in *IEEE GLOBECOM 2007 - IEEE Global Telecommunications Conference*, 2007, pp. 1028–1032.

[46] R. Jurdak, A. G. Ruzzelli, and G. M. P. O'Hare, "Multi-hop rfid wake-up radio: Design, evaluation and energy tradeoffs," in *2008 Proceedings of 17th International Conference on Computer Communications and Networks*, 2008, pp. 1–8.

[47] F. Hutu, A. Khoumeri, G. Villemaud, and J. Gorce, "Wake-up radio architecture for home wireless networks," in *2014 IEEE Radio and Wireless Symposium (RWS)*, 2014, pp. 256–258.

[48] H. Zhang, C. Li, S. Chen, X. Tan, N. Yan, and H. Min, "A low-power ofdm-based wake-up mechanism for ioe applications," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 2, pp. 181–185, 2018.

[49] S. Tang, H. Yomo, and Y. Takeuchi, "Optimization of frame length modulation-based wake-up control for green wlans," *IEEE Transactions on Vehicular Technology*, vol. 64, no. 2, pp. 768–780, 2015.

[50] S. M. Günther, M. Leclaire, J. Michaelis, and G. Carle, "Analysis of injection capabilities and media access of IEEE 802.11 hardware in monitor mode," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, 2014, pp. 1–9.

[51] S. M. Günther, "libmoep – packet injection library." [Online]. Available: https://moepi.net/phd/

[52] DeviWiki, "Atheros AR5BHB112." [Online]. Available: https://deviwiki.com/wiki/Atheros_AR5BHB112

[53] MediaTek, "RT5370 High-performance 802.11n Wi-Fi with antenna diversity switching." [Online]. Available: https://www.mediatek.com/products/broadbandWifi/rt5370

[54] 0x90, "wifi-arsenal." [Online]. Available: https://github.com/0x90/wifi-arsenal/tree/master/libmoep-1.1/patches

[55] "Radiotap." [Online]. Available: https://www.radiotap.org/

[56] J. Postel, "Internet Protocol," Internet Requests for Comments, RFC Editor, STD 5, September 1981, http://www.rfc-editor.org/rfc/rfc791.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc791.txt