INF 2004
TRIMESTER 3 AY23/24

Group 29
Embedded Systems Programming Report

Fong Kai Wen Kobin[1], Kenneth Kuah Zhi Yong[2], Lee Yong Zhang[3], Winson Low Zi Cai[4], Ng Xu Len Keith[5]

[1]2103287; [2]2103301; [3]2103296; [4]2103272,[5]2003269

Design

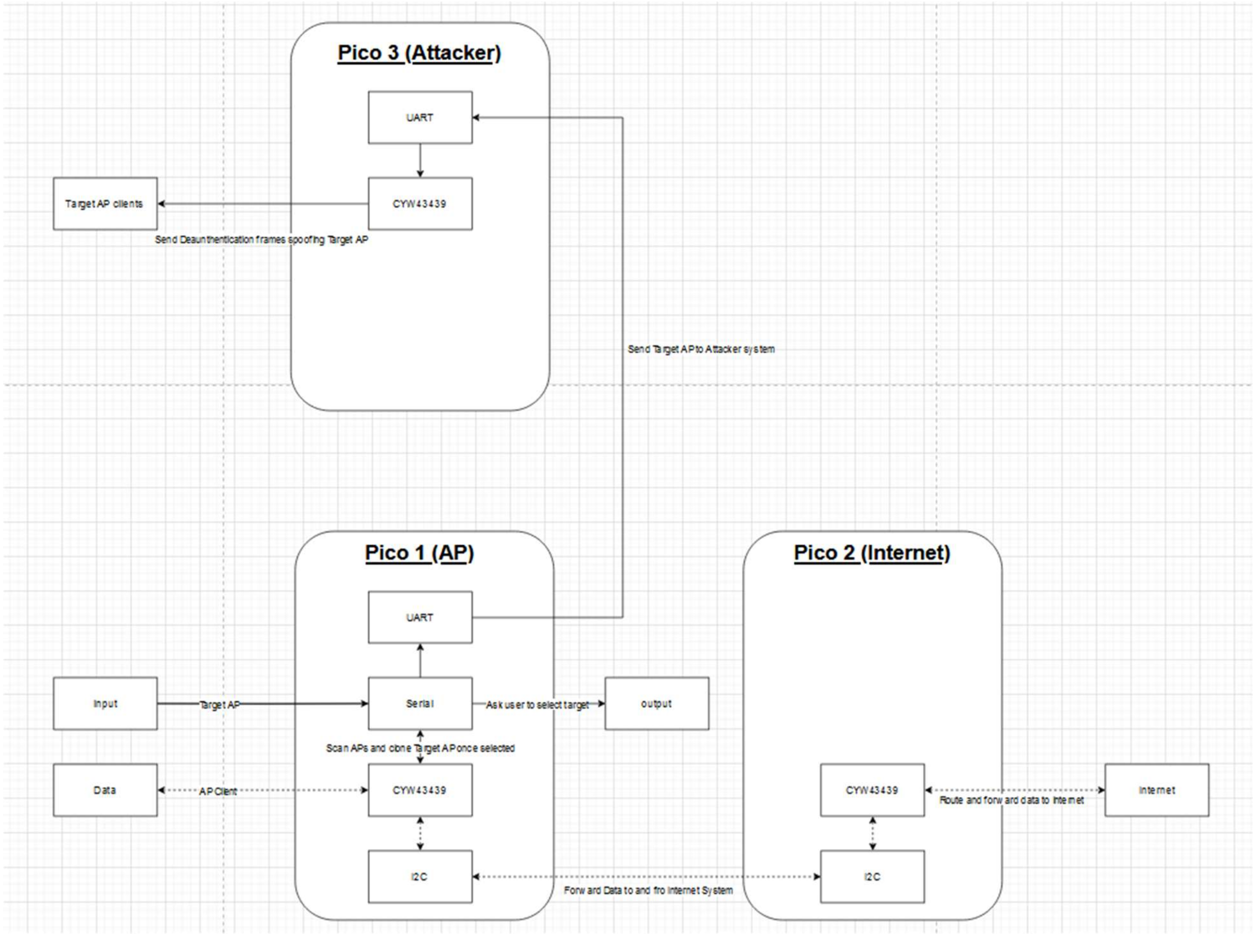

Pico 3 (Attacker)
UART
CYW43439
Target AP clients
Send Deaunthentication frames spoofing Target AP
Send Target AP to Attacker system
Pico 1 (AP)
Pico 2 (Internet)
UART
Input
Target AP
Serial
Ask user to select target
output
Scan APs and clone Target AP once selected
Data
AP Client
CYW43439
CYW43439
Route and forward data to Internet
Internet
I2C
I2C
Forward Data to and fro Internet System

Figure 1: Block Diagram

Our project functions according to the block diagram, with a series of orchestrated steps. Initially, the attacker initiates Wi-Fi network scanning using Pico 1, selecting the target wireless network. Subsequently, a command is transmitted from Pico 1 to Pico 3 via UART, triggering a deauthentication attack on the chosen access point. Once clients are disconnected from the legitimate AP, the evil twin attack is executed. In this phase, Pico 1 transforms into an access point, enabling client connections and routing all of the client's traffic to Pico 2 via I2C. Once connected, clients are able to browse the internet seamlessly, facilitated by Pico 2 routing client requests to the internet before relaying them back to Pico 1. Notably, on Pico 2, client's packets are stored into an SD card for subsequent data acquisition and analysis purposes.

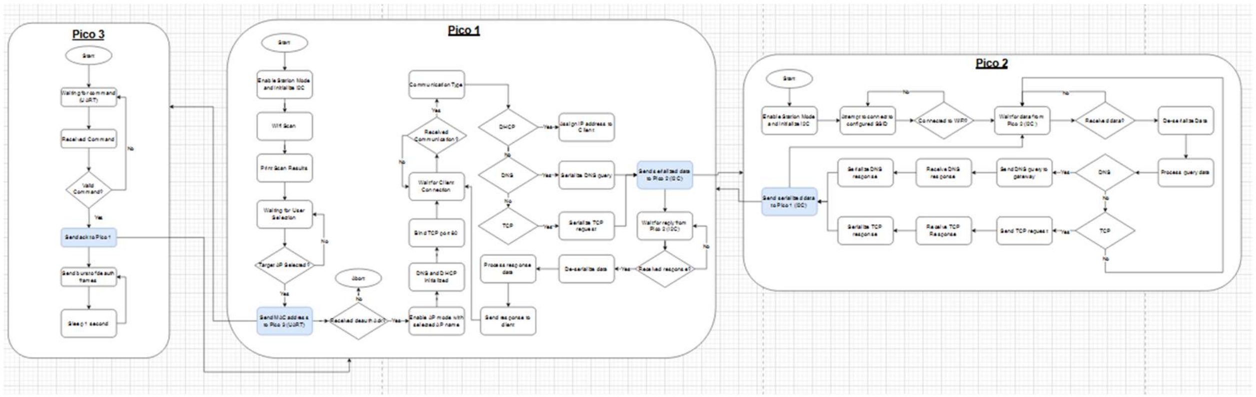


Figure 2: Flow Chart

The above flowchart provides a comprehensive overview of the detailed methodology employed in operating our system. It elucidates the various components that converge to constitute the complete attack flow.

Development Process

I2C

I2C, short for inter-integrated circuit, serves as a synchronous, multi-master/multi-slave, single-ended serial communication bus tailored for slow internal, short-range communications. This communication protocol utilizes only two wires – Serial Data (SDA) and Serial Clock (SCL). SDA facilitates data transmission from master to slave, while SCL synchronizes the clock between master and slave.

Our team has opted for I2C to facilitate communication between Pico 1 and Pico 2, managing the routing of client packets bidirectionally. While there are alternative communication protocols like UART and SPI, we decided to select I2C. This decision is rooted in the superior speed of I2C, which can reach up to 3.4MHz, compared to UART. Additionally, we avoided using SPI to prevent an increase in the number of wires connected to the Pico and the occupation of numerous GPIO pins associated with the SPI protocol.

SD Card Driver

The SD card driver leverages an external library to facilitate writing operations on the SD card. For proper functionality, it is crucial to configure the appropriate General-Purpose Input/Output (GPIO) pins to enable reading from the SD card slot on the maker Pico Board. After this setup, the SD card must be mounted to initiate the creation of a pcap file. This file will contain all client traffic, encompassing both incoming and outgoing packets.

When the client initiates DNS queries and TCP requests, these actions will trigger a write operation to the previously established pcap file. This process ensures that all client communications are accurately recorded and stored for analysis.

Pico 1

This particular Pico assumes the role of hosting the Evil Twin Access Point (AP) and is responsible for executing various functions within the system. These functions include conducting Wi-Fi scanning, transmitting a command to Pico 3 via UART to initiate a Wi-Fi attack aimed at disabling the legitimate AP and hosting an AP so that clients are able to connect to it and access the internet. Upon successful connection of clients to the attacker's AP, the client traffic is directed from the Evil Twin to Pico 2. Pico 2 will serve as the intermediary, forwarding the traffic to the internet, thereby facilitating client's access to online resources.

Pico 2

This Pico functions as a forwarding agent, managing the real-time traffic for both DNS queries and TCP requests to and from the internet. It then relays this traffic to Pico 1, which subsequently routes it to the client. This necessity for a second Pico arises from our inability to simultaneously route internet traffic and maintain an Access Point (AP) for clients to connect to the Pico.

Pico 3

Pico 3 was intended to be the Pico responsible for conducting the Deauthentication attack. However, after weeks of research and troubleshooting, the team came to the conclusion that the CYW43439 Chip on the Pico W does not have the ability to send raw 802.11 Management frames. Therefore, the ESP8266-01 Wi-Fi Chip was acquired to provide this functionality. However, custom firmware needed to be coded and flashed to the ESP8266-01, and there was not enough time to acquire the additional hardware required to flash the ESP8266-01.

As an alternative to the Deauthentication attack, a DHCP Flood attack was implemented. The final implementation results in DHCP Discover Packets being successfully sent by the Pico W. However, there are DHCP Flooding protection measures enabled by default on the Access Points available to the team. This prevents us from knowing if the attack is succeeding or not.

Due to the issues mentioned above, Pico 3 was not able to be implemented for the final submission of the project.

Attack Flow

The attack process unfolds as follows: Initially, the attacker conducts a WiFi scan to select a target Access Point (AP). Following this, a de-authentication attack is launched using Kali Linux, effectively disconnecting the client from the AP. Subsequently, an Evil Twin Access Point is established on Pico 1. As the client discovers the disconnection from their regular WiFi, they may inadvertently connect to this rogue AP. Attempts to reconnect to the legitimate AP are thwarted by the ongoing de-authentication attack from the attacker's machine, leading the client to remain on the rogue network.

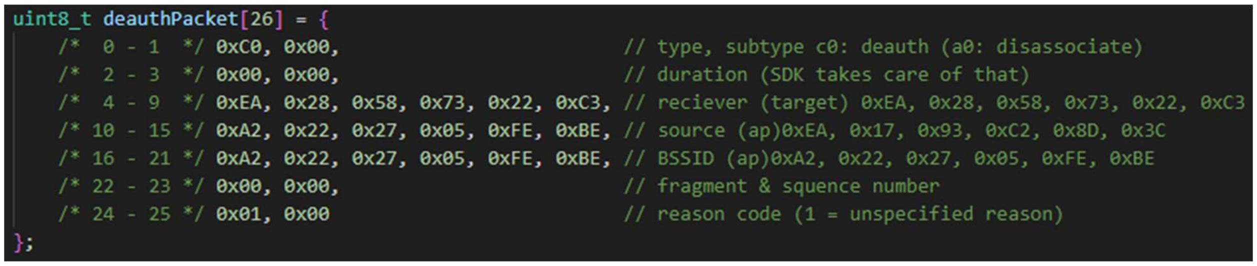
Once connected to the rogue AP, clients can access the internet. Their DNS queries and TCP requests are routed through Pico 1 to Pico 2, and from there, to the internet. Conversely, responses from the internet are channeled back through Pico 2 to Pico 1. Both incoming and outgoing packets in this setup are captured into a pcap file. These packets, which include both the data sent from Pico 1 to Pico 2 and vice versa, can be analyzed using Wireshark.

Technical Challenges

In the development of our Pico W project, Rogue AP-ico, we faced a distinct set of technical challenges that tested our expertise and resourcefulness. Firstly, we ventured into the task of crafting deauthentication frames in C for Pico W. The concept of a deauthentication attack is relatively simple. The IEEE 802.11 protocol defines "Management Frames" that are used to manage the state of a Wi-Fi connection. One of these frames is the deauthentication frame, which is the focus of this attack. When a deauthentication frame is sent to a Wi-Fi client or the Access Point, the recipient is notified that the sender is deauthenticating from the Wi-Fi network. This is not a request, but a notice. The recipient will then terminate the existing connection. The vulnerability comes from the fact that Management Frames are not encrypted and there is no authentication when receiving these frames. Therefore, attackers can easily conduct the attack by spoofing their MAC address and disconnecting any client they want.

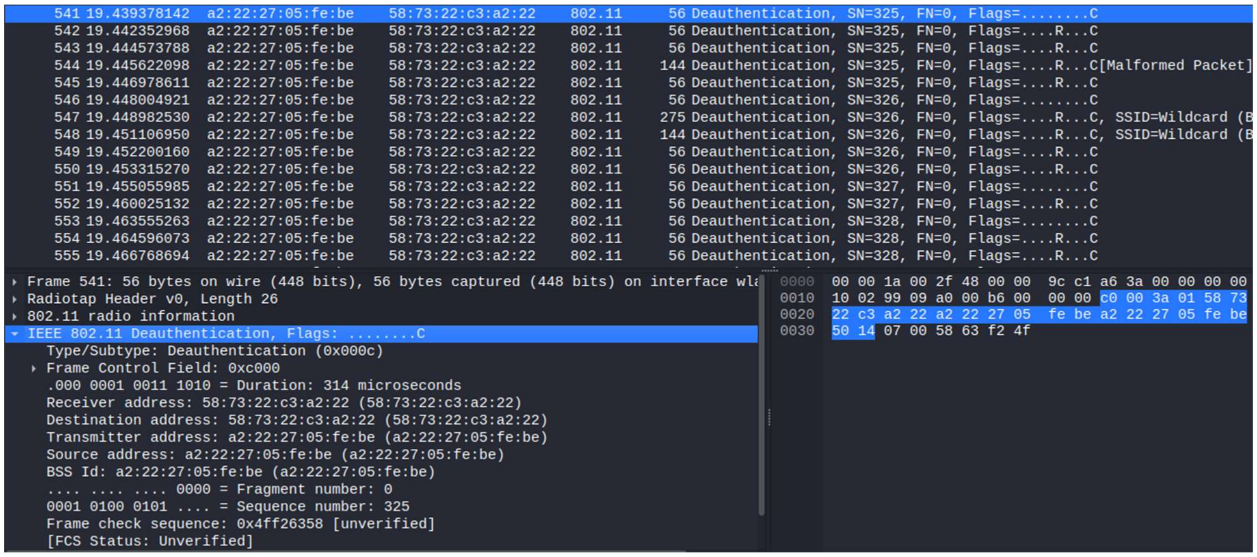
The format for the 802.11 deauthentication frame was studied in detail to generate these frames accurately. However, simply generating the frames was not the challenge. Extensive research showed that the Pico W has no existing implementations as a deauthentication device. There is little information regarding the Pico W's ability to send raw frames with the default drivers for the CYW43439 Wi-Fi chip as well. However, looking at the API documentation for the CYW43 library, the "cyw43_send_ethernet()" function was identified to be able to send out the raw frames. The documentation on how to use the function is extremely lacking, but it was able to be successfully implemented with the use of pbufs (Packet Buffers).

In a deauthentication attack, there is no need for the attacker to connect to the Wi-Fi AP to conduct the attack (tested successfully with separate hardware). Despite this, it has been observed that the Pico W will not send out any frames at all without first connecting to an AP. Additionally, even after connecting to the AP and sending out the deauthentication frames, the frame is malformed. The deauthentication frame that was crafted on the Pico W can be seen in Figure 3. The actual deauthentication frame that was observed on Wireshark (on a separate machine) can be seen in Figure 4.



```
uint8_t deauthPacket[26] = {
    /*  0 - 1  */ 0xC0, 0x00,                         // type, subtype c0: deauth (a0: disassociate)
    /*  2 - 3  */ 0x00, 0x00,                         // duration (SDK takes care of that)
    /*  4 - 9  */ 0xEA, 0x28, 0x58, 0x73, 0x22, 0xC3, // reciever (target) 0xEA, 0x28, 0x58, 0x73, 0x22, 0xC3
    /* 10 - 15 */ 0xA2, 0x22, 0x27, 0x05, 0xFE, 0xBE, // source (ap)0xEA, 0x17, 0x93, 0xC2, 0x8D, 0x3C
    /* 16 - 21 */ 0xA2, 0x22, 0x27, 0x05, 0xFE, 0xBE, // BSSID (ap)0xA2, 0x22, 0x27, 0x05, 0xFE, 0xBE
    /* 22 - 23 */ 0x00, 0x00,                         // fragment & squence number
    /* 24 - 25 */ 0x01, 0x00                          // reason code (1 = unspecified reason)
};
```

Figure 3: Crafted Deauthentication Frame



```
541 19.439378142  a2:22:27:05:fe:be   58:73:22:c3:a2:22   802.11    56 Deauthentication, SN=325, FN=0, Flags=........C
542 19.442352968  a2:22:27:05:fe:be   58:73:22:c3:a2:22   802.11    56 Deauthentication, SN=325, FN=0, Flags=....R...C
543 19.444573788  a2:22:27:05:fe:be   58:73:22:c3:a2:22   802.11    56 Deauthentication, SN=325, FN=0, Flags=....R...C
544 19.445622098  a2:22:27:05:fe:be   58:73:22:c3:a2:22   802.11   144 Deauthentication, SN=325, FN=0, Flags=....R...C[Malformed Packet]
545 19.446978611  a2:22:27:05:fe:be   58:73:22:c3:a2:22   802.11    56 Deauthentication, SN=325, FN=0, Flags=....R...C
546 19.448004921  a2:22:27:05:fe:be   58:73:22:c3:a2:22   802.11    56 Deauthentication, SN=326, FN=0, Flags=........C
547 19.448982530  a2:22:27:05:fe:be   58:73:22:c3:a2:22   802.11   275 Deauthentication, SN=326, FN=0, Flags=....R...C, SSID=Wildcard (B
548 19.451106950  a2:22:27:05:fe:be   58:73:22:c3:a2:22   802.11   144 Deauthentication, SN=326, FN=0, Flags=....R...C, SSID=Wildcard (B
549 19.452200160  a2:22:27:05:fe:be   58:73:22:c3:a2:22   802.11    56 Deauthentication, SN=326, FN=0, Flags=....R...C
550 19.453315270  a2:22:27:05:fe:be   58:73:22:c3:a2:22   802.11    56 Deauthentication, SN=326, FN=0, Flags=....R...C
551 19.455055985  a2:22:27:05:fe:be   58:73:22:c3:a2:22   802.11    56 Deauthentication, SN=327, FN=0, Flags=........C
552 19.460025132  a2:22:27:05:fe:be   58:73:22:c3:a2:22   802.11    56 Deauthentication, SN=327, FN=0, Flags=....R...C
553 19.463555263  a2:22:27:05:fe:be   58:73:22:c3:a2:22   802.11    56 Deauthentication, SN=328, FN=0, Flags=........C
554 19.464596073  a2:22:27:05:fe:be   58:73:22:c3:a2:22   802.11    56 Deauthentication, SN=328, FN=0, Flags=....R...C
555 19.466768694  a2:22:27:05:fe:be   58:73:22:c3:a2:22   802.11    56 Deauthentication, SN=328, FN=0, Flags=....R...C
```

```
▸ Frame 541: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface wla
▸ Radiotap Header v0, Length 26
▸ 802.11 radio information
▾ IEEE 802.11 Deauthentication, Flags: ........C
    Type/Subtype: Deauthentication (0x000c)
  ▸ Frame Control Field: 0xc000
    .000 0001 0011 1010 = Duration: 314 microseconds
    Receiver address: 58:73:22:c3:a2:22 (58:73:22:c3:a2:22)
    Destination address: 58:73:22:c3:a2:22 (58:73:22:c3:a2:22)
    Transmitter address: a2:22:27:05:fe:be (a2:22:27:05:fe:be)
    Source address: a2:22:27:05:fe:be (a2:22:27:05:fe:be)
    BSS Id: a2:22:27:05:fe:be (a2:22:27:05:fe:be)
    .... .... .... 0000 = Fragment number: 0
    0001 0100 0101 .... = Sequence number: 325
    Frame check sequence: 0x4ff26358 [unverified]
    [FCS Status: Unverified]
```

```
0000  00 00 1a 00 2f 48 00 00  9c c1 a6 3a 00 00 00 00
0010  10 02 99 09 a0 00 b6 00  00 00 c0 00 3a 01 58 73
0020  22 c3 a2 22 a2 22 27 05  fe be a2 22 27 05 fe be
0030  50 14 07 00 58 63 f2 4f
```

Figure 4: Observed deauthentication Frame

This shows that the receiver address of the deauthentication frame is being interpreted incorrectly by an offset of 2 bytes. Further experiments were conducted by changing the contents of the crafted frame and observing the results. Changes to the MAC addresses in the frame (at bytes number 4 – 21) resulted in the frame being interpreted as a QoS data frame instead of a deauthentication frame. This is extremely strange as the bytes responsible for dictating what type of management frame it is are bytes 0 and 1, which are unchanged. There are no error messages observed from the Pico W. The use of a third-party library (picowi), whose author claims is designed specifically to allow the sending of raw packets, yields the same results.

A decision was made to utilize MicroPython, which may potentially make the implementation of the deauthentication attack easier. Unfortunately, the MicroPython port for the rp2 (short for rp2040, which is the CPU Hardware that the Pico W utilizes), does not support the ability to send raw 802.11 frames. The ability to craft a raw frame and send it out with a spoofed MAC address would require modifying/adding on to the MicroPython low level libraries. This defeats the purpose of switching to MicroPython in the first place, which is for easier implementation.

There are several implementations using the ESP8266 Wi-Fi chip to create a deauthentication device which was found and referenced. It was identified that the function “wifi_send_pkt_freedom()” function is a key component is successfully conducting the attack as it provides the ability to send raw 802.11 frames. This function appears to come from the ESP8266WiFi.h library. This function does not exist for the Pico W. Further research revealed that the CYW43439 Wi-Fi chip automatically does the processing (adding and removal) of the 802.11 headers in the chipset itself, independent of the CPU. Additionally, the datasheet for the CYW43439 Wi-Fi chip mentions supporting the ability for the CYW43439 Wi-Fi chip to process Control and Data Frames but has no mention of its ability to support processing Management Frames (Section 11.2). The deauthentication attack requires the crafting of custom 802.11 frames to be sent as the deauthentication frame. This led to the conclusion that it is not possible to conduct a deauthentication attack with the Pico W as it is a hardware limitation of the CYW43439 Wi-Fi chip.

Therefore, the team came to the decision to acquire the ESP8266 Wi-Fi chip (ESP01) for the implementation of the deauthentication feature. The Pico W communicates with the ESP01 through UART. However, the only commands that the Pico W is able to send to the ESP01 are “AT” (Attention) commands, which provide high-level APIs for the Pico W to interact with the ESP01. In order to achieve frame injection, which is required for performing a deauthentication attack, low level functions must be available. This requires the flashing of custom firmware to the ESP01. Unfortunately, additional hardware (ESP8266 Flasher and Programmer) is required to achieve this. The team was not able to acquire the necessary hardware in time before the project submission, preventing the deauthentication feature from being completed.

The next technical challenge faced was regarding DHCP Flooding. The team was able to implement DHCP Flooding working on the Pico W but is unable to test if the attack will successfully work due to DHCP Flooding Protection measures being enabled by default on the Aps that the team has available to them. Even using existing tools such as Yersinia yields no successful results when conducting a DHCP Flood attack.

Another significant hurdle is mastering the nuances of writing efficient C code for the Pico W. This was imperative as our project is predominantly C-based, with a little usage of MicroPython. This language restriction meant that we had to navigate the steep learning curve of C’s syntax and memory management without the luxury of fallbacks provided by more forgiving languages. This is especially so with the limited memory resources of the Pico W, giving us further challenges with regards to memory management. We had to ensure that every line of code was as lean and efficient as possible.

The last hurdle we faced is concerning the device’s networking capabilities. The Pico W hardware is not inherently designed to function as a wireless access point (AP) while concurrently managing connection to the internet, This limitation is critical when the project requires the Pico W to route traffic between client devices connected to it and the internet, to allow clients to access the internet when they are connected to our Access Point. This is a common functionality in network bridges. The challenge lies in the hardware and firmware capability of the Pico W, it is unable to handle dual modes of Wifi operation. Typically, a device that serves as an AP must manage client connections, which includes handling their TCP requests and handling DNS queries, and channeling these packets to the internet. Simultaneously, it needs to maintain its client mode connection to another AP for internet access. The Pico W, unfortunately, does not support such concurrent operations natively. This limitation required us to explore alternative solutions. And we came up with a solution using two Pico Ws to facilitate the routing of traffic from client to the internet and vice versa, with the help of the I2C protocol between the two Picos.

Lessons Learned

In the course of developing this project, several valuable lessons were gleaned, shedding light on the significance of resource management, adaptability in programming, flexibility in problem-solving, the importance of maintaining code quality, documentation and code modularity. These lessons serve as a testament to the growth and maturation of our team and it offers insights that can be used in future projects.

Resource Management

Learning to work within the constraints of limited memory and processing power has been a challenge for our team, but it is instrumental in the enhancement of efficiency in our code and resource utilization. This experience reinforced the importance of optimizing code for resource-constrained environments, allowing significant improvements in performance.

Adaptability in Programming

The necessity to adapt programming strategies when restricted to a specific language, highlighted the importance of possessing a versatile coding skillset. It became evident that being proficient in a variety of programming languages is an asset, as it enables developers like us to choose the most suitable tools for a given task. This adaptability proved crucial in navigating the project's challenges and constraints

Flexibility in Problem-solving

In the face of technical limitations, being flexible and creative with solutions emerged as an indispensable trait. This project served as a compelling example of the need to think outside the box and explore unconventional approaches. Like the decision to employ two Pico Ws instead of using one. This showcases our team's ability to adapt and find innovative solutions to complex problems. This flexibility in problem-solving enabled us to overcome obstacles and achieve our project goals effectively.

Coding Quality and Documentation

Ensuring code readability and maintainability, as well as providing informative comments on GitHub, became paramount in our development process. This lesson emphasized the significance of maintaining clean, well-documented code and conducting thorough testing before pushing changes to the repository. By adhering to these best practices, we not only facilitated collaboration within our team, but also ensured the sustainability of the project in the long term.

Making Code Dynamic for Seamless Integration

The last significant lesson learned during the project was the importance of making our code dynamic and modular to facilitate fast and easy integration between the various functions and components. As many of us worked on different parts of the project, we have to ensure that the code we produce could be easily integrated with others. By designing our code with modularity in mind, we are able to create independent and reusable functions that seamlessly fit together.

In conclusion, the lessons learned throughout the scope of this project has enriched our collective knowledge and skills as developers. These experiences have reinforced the importance of resource management, adaptability, flexibility, code quality and code modularity, all of which are crucial elements for success in the dynamic and ever-evolving field of software development. Armed with these lessons, we are better equipped with the skills to tackle future hurdles.