STANDALONE WI-FI BASED IOT SYSTEMS USING THE RASPBERRY PI PICO-W

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by
Christopher Chan
MEng Field Advisor: Hunter Adams
Degree Date: January 2024

Abstract

Master of Engineering Program
School of Electrical and Computer Engineering
Cornell University
Design Project Report

Project Title: **Standalone Wi-Fi Based IoT Systems Using The Raspberry Pi Pico-W**

Author: **Christopher Chan**

Abstract

In June 2022, Raspberry Pi released a new RP2040 microcontroller, the Raspberry Pi Pico-W. The Pico-W is an augmentation of the Raspberry Pi Pico, taking the vast features of the Pico and adding Wi-Fi and Bluetooth through the inclusion of an Infineon CYW43439 chip. The objective of this project was to explore the Pico-W's new features and lay a foundation for its use in large-scale networking applications. Exploration began with a deep dive into the new CYW43 driver in the Pico-SDK, as well as the Lightweight IP Stack (LwIP). Using this knowledge, we created a UDP demo that future ECE 4760 students can reference to incorporate wireless communication into their final projects without prior experience with the Pico-W or LwIP. The project culminated in the creation of a standalone Pico-W mesh network that identifies its own topology and optimizes its routing patterns accordingly.

Executive Summary

The objective of this design project was twofold. First, we wanted to learn about the Raspberry Pi Pico-W and create reference material that could be of service to ECE 4760 students seeking to incorporate wireless features into their final projects. Second, we wanted to create a framework for data transfer across networks of Pico-Ws. Naturally, this divided the project into two phases: research and implementation.

The project started with an investigation of Raspberry Pi's Pico C/C++ SDK and Pico SDK Examples repository. The Pico C/C++ SDK is the programming interface provided by Raspberry Pi for developing software for the Raspberry Pi Pico, as well as its IoT-enabled successor, the Pico-W. The Raspberry Pi Pico SDK Examples repository is a growing collection of code examples demonstrating the basic functions of each peripheral on the Pico. During this research phase, we familiarized ourselves with the Pico-W's build system, the CYW43 driver in the Pico SDK, and the Lightweight IP Stack (LwIP). At the end of phase 1, we created a UDP demo that showcases all of the functionality we had learned about the Pico-W thus far.

After concluding research on the Pico-W, we devised plans to build a framework for efficient data flow across a network of Pico-Ws. This started by augmenting the UDP demo created at the end of phase 1. We then created a pair of algorithms that handled network configuration and optimization. First, our neighbor finding algorithm initializes the whole network with ID numbers and prepares each node to perform a routing algorithm. Then, we use a modified distance vector routing algorithm to optimize the flow of data in the network by finding the shortest path between any pair of nodes.

With everything in place, we developed a testing framework to simulate different network topologies. Using this framework we tested multiple layouts and verified that the network was able to find the optimal routing configurations regardless of the shape or size of the network. Our final design demonstrates that Pico-W networks are a prime candidate for constructing large-scale networks.

Table of Contents

Phase 1

Design Problem 6
Pico-W vs. Pico 6
Configuring the build environment for the Pico-W 6
Pinout differences between the Pico and Pico-W 7
The CYW43 Driver 8
Connecting to Wi-Fi for the first time 8
Scanning for Wi-Fi networks 9
The Lightweight IP Stack (LwIP) 11
LwIP raw UDP API 11
Sending data from a Pico-W to a laptop 13
Two-way UDP Communication using LwIP 16
High-level overview 16
Implementation 17

Phase 2

Design Problem 19
Augmented UDP program 19
High-level overview 20
New thread: protothread_udp_ack 20
Enhanced UDP packets 20
Automatically finding the access point 21
Sending ACK packets 21
Calculating round trip time (RTT) 22
Neighbor Finding 22
High-level overview 23
New thread: protothread_connect 24
Adding ID numbers to packets 24
Distinguishing between initialized and uninitialized neighbors 25
Distance Vector Routing 25
High-level overview 26
Node and Neighbor Structs 27

Augmenting protothread_connect with a new type of scan....28
Updating a Distance Vector....28
Poisoned reverse and the count-to-infinity problem....29
Routing packets....30
Testing Methodology....31
Physical IDs....31
Simulating non-adjacency between nodes....31
Results and Conclusions....32
Performance assessment....32
Time to convergence....32
Flexibility and scalability....33
What's next?....33
Acknowledgments....35
References....36
Raspberry Pi....36
Bruce Land....36
Lightweight IP Stack....36
Research on UDP, TCP, and routing algorithms....36
My own work....36

Appendix

Appendix A: Neighbor Finding Walkthrough....37
Appendix B: Distance Vector Routing Walkthrough....40

Phase 1

Investigating the Pico-W

Design Problem

The first objective of this project was to gain familiarity with the Pico-W's new features. This meant experimenting with functions from the SDK's new CYW43 driver, as well as experimenting with the Lightweight IP Stack (the open-source framework that gives the Pico-W access to the UDP and TCP protocols).

Demo code for the Pico-W was only added to the pico-examples repository shortly before the conception of this project. At the time, Raspberry Pi's documentation of those features was understandably lacking. Thus, experimentation with the new material was a necessary precursor to the second half of this project. Though I will detail my findings below, I suggest that any user looking to work with the Raspberry Pi Pico-W should now reference Raspberry Pi's official documentation. Note that the code snippets in this section originate from both Raspberry Pi's official examples and my own (products of my tinkering with the official examples).

Pico-W vs. Pico

Immediately upon starting work with the Pico-W, I found two major differences between the Pico-W and the original Pico. The first difference was in configuring the build environment using CMake. The second difference was in the pinout of the board itself. Even though these differences do not affect the regular workflow of programming with the Pico-W, I believe that they are crucial to recognize before working with the Pico-W.

Configuring the build environment for the Pico-W

The process for configuring the build environment for the Pico-W is similar to the setup process for the original Pico. The Pico-W uses all of the same toolchains as the Pico, and the pico-sdk and pico-examples repositories are both compatible with the Pico-W.

This setup process remains identical until we begin compiling code using CMake. When initializing a build directory for the first time with the original Pico, we use the following set of commands:

```
$ mkdir build
$ cd build
$ cmake -G "NMake Makefiles" ..
$ nmake
```

When configuring the build system for the Pico-W, the flag *-DPICO_BOARD=pico_w* must be included to tell CMake that the files are being compiled for the Pico-W specifically. For those following Hunter Adams' guide, the following set of commands is the most compatible method of configuring the environment for the Pico-W:

```
$ mkdir build
$ cd build
$ cmake -DPICO_BOARD=pico_w -G "NMake Makefiles" ..
$ nmake
```

Note that the *-DPICO_BOARD=pico_w* flag will specify to the compiler that *all* of the demos in this directory are to be built for the Pico-W. If configuring the build system for a directory which contains both Pico and Pico-W demos, it may be better to configure this option on a per project basis by using the following CMake function:

```
# Set PICO_BOARD environment variable to "pico_w"
set(PICO_BOARD pico_w)
```

In Raspberry Pi's provided examples for the Pico-W, the SSID and password for the network are specified at compile time rather than in the code itself. Therefore, when configuring the build system for the pico-examples repository, two *additional* flags must be included that specify the network name and password. Raspberry Pi recommends the following set of commands in their documentation:

```
$ mkdir build
$ cd build
$ cmake -DPICO_BOARD=pico_w -DWIFI_SSID="Your Network"
      -DWIFI_PASSWORD="Your Password" ..
$ nmake
```

These flags specify values for the *WIFI_SSID* and *WIFI_PASSWORD* macros found in the Raspberry Pi examples for the Pico-W.

Pinout differences between the Pico and Pico-W

Most of the Pico-W's pinout is inherited directly from the original Pico, but there are some GPIO pins that have been rewired to accommodate the cyw43. The only relevant difference to this project was the LED pin, which is now wired through the cyw43 instead. Interfacing with the onboard LED now requires the following *#include* directives:

```
#include "boards/pico_w.h"
#include "pico/cyw43_arch.h"
```

On the Pico-W, blinking the LED now looks like this:

```
while (true) {
    cyw43_arch_gpio_put(CYW43_WL_GPIO_LED_PIN, 1); // Turn on
    sleep_ms(500);
    cyw43_arch_gpio_put(CYW43_WL_GPIO_LED_PIN, 0); // Turn off
    sleep_ms(500);
}
```

The CYW43 Driver

The Pico-W's expansive networking capabilities are all made possible by the CYW43439 wireless chip. The Pico SDK provides a high-level API for interfacing with the CYW43 called *pico_cyw43_arch*. This library provides an abstraction over the lower-level *cyw43_driver* library. The *pico_cyw43_arch* API can be included with the following directive:

```
#include "pico/cyw43_arch.h"
```

Connecting to Wi-Fi for the first time

After configuring the build system and compiling the blink demo to verify that it was working, the logical next step was to get the Pico-W online. All projects using the onboard wireless chip must begin by initializing the cyw43 itself.

```
// Initialize Wifi chip
printf("Initializing cyw43...");
if (cyw43_arch_init()) {
    printf("failed to initialize.\n");
    return 1;
} else {
    printf("initialized!\n");
}
```

The *cyw43_arch_init()* function returns 0 on success. Although most applications will never need to, the cyw43 can be de-initialized by calling *cyw43_arch_deinit()*.

Once initialized, the CYW43 is capable of behaving as either a station or an access point. Station mode allows the Pico-W to connect to Wi-Fi, while access point mode allows the Pico-W to host its own Wi-Fi network. We will discuss Pico-W access points and their relevance to this project in great detail later. For this example, station mode is the obvious choice. To configure the Pico-W as a station:

```
// Enable station mode
cyw43_arch_enable_sta_mode();
```

After configuring the Pico-W as a station, we can now connect to Wi-Fi. The Pico SDK provides numerous functions for connecting the Pico-W to Wi-Fi. The official documentation explains the exact use cases for each one, but the one that I found most applicable was *cyw43_arch_wifi_connect_timeout_ms()*.

```
// Connect to Wi-Fi
cyw43_arch_wifi_connect_timeout_ms(WIFI_SSID, WIFI_PASSWORD,
                                   CYW43_AUTH_WPA2_AES_PSK, 30000);
```

The *cyw43_arch_wifi_connect_timeout_ms()* function takes 4 arguments. The first two arguments specify the SSID and password of the network, while the last argument specifies the maximum amount of time (in milliseconds) to spend trying to connect. This function is blocking and returns 0 on success.

Scanning for Wi-Fi networks

Another feature of the Pico-W is the ability to scan for Wi-Fi networks using the CYW43. In my testing, the Pico-W could be in either station mode or AP mode during a scan, however Raspberry Pi's demo puts the Pico-W in station mode before scanning, and the vast majority of my scans were also conducted from station mode. Scanning for Wi-Fi networks using the Pico-W takes a callback-based approach. The following function initiates a scan:

```
int err = cyw43_wifi_scan(&cyw43_state, &scan_options, NULL,
                          print_result);
```

The last argument passed to *cyw43_wifi_scan()* specifies the callback function to be invoked every time a scan result is found. The second to last argument specifies options for the scan. According to the Pico-W documentation, values passed here are currently ignored. An empty set of options for passing to the function can be constructed as follows:

```
// Scan options don't matter
cyw43_wifi_scan_options_t scan_options = {0};
```

The *cyw43_wifi_scan()* function returns 0 if a scan was successfully started.

An incredibly important feature to note is that scans execute in the background. They are *not* blocking. This can be extremely problematic if continued execution depends on the result of the scan. Therefore if the user wants to block until the scan finishes, something like the following control structure must be used:

```
while (cyw43_wifi_scan_active(&cyw43_state)) {
    // Block until scan is complete
}
```

The function *cyw43_wifi_scan_active()* returns *true* if a scan is currently running in the background. According to cyw43 driver documentation, the argument passed to this function must always be *&cyw43_state*. The most basic callback function for a scan takes the result and prints out its contents:

```
// Wifi scan callback function, prints cyw43_ev_scan_result_t as a string
static int print_result(void* env, const cyw43_ev_scan_result_t* result)
{
    if (result) {
        // Compose MAC address
        char bssid_str[40];
        sprintf(bssid_str, "%02x:%02x:%02x:%02x:%02x:%02x", result->bssid[0],
                result->bssid[1], result->bssid[2], result->bssid[3],
                result->bssid[4], result->bssid[5]);

        printf("ssid: %-32s rssi: %4d chan: %3d mac: %s sec: %u\n",
               result->ssid, result->rssi, result->channel, bssid_str,
               result->auth_mode);
    }

    return 0;
}
```

The most valuable result here (in my opinion) is the received signal strength indicator, returned above as *result–>rssi*. RSSI is a decibel (dB) measure of the received signal strength. In most of my implementations I reduce the printed output to just the SSID and RSSI of the network. The output of a Wi-Fi scan looks like this:

```
Performing wifi scan
ssid: [range]_E30AJT7113776F        rssi:  -76 chan:   3 mac: 40:ca:63:11:d6:2c sec: 5
ssid: [range]_E30AJT7113776F        rssi:  -76 chan:   3 mac: 40:ca:63:11:d6:2c sec: 5
ssid: [range]_E30AJT7113776F        rssi:  -78 chan:   3 mac: 40:ca:63:11:d6:2c sec: 5
ssid: 618-G                         rssi:  -87 chan:   1 mac: 86:2a:a8:54:9c:47 sec: 5
ssid: 618-A                         rssi:  -86 chan:   1 mac: 8a:2a:a8:54:9c:47 sec: 5
ssid: 618-B                         rssi:  -88 chan:   1 mac: 8e:2a:a8:54:9c:47 sec: 5
ssid: [range]_E30AJT7113895E        rssi:  -84 chan:   3 mac: 88:57:1d:de:65:bf sec: 5
ssid: [range]_E30AJT7113935N        rssi:  -79 chan:   4 mac: 88:57:1d:19:63:ed sec: 5
ssid: [range]_E30AJT7113935N        rssi:  -72 chan:   4 mac: 88:57:1d:19:63:ed sec: 5
ssid: [range]_E30AJT7113935N        rssi:  -75 chan:   4 mac: 88:57:1d:19:63:ed sec: 5
ssid: [range]_E30AJT7113935N        rssi:  -74 chan:   4 mac: 88:57:1d:19:63:ed sec: 5
ssid: [range]_E30AJT7113935N        rssi:  -76 chan:   4 mac: 88:57:1d:19:63:ed sec: 5
ssid: SpectrumSetup-E4              rssi:  -86 chan:   6 mac: 74:93:da:05:90:e2 sec: 5
ssid: NETGEAR73                     rssi:  -92 chan:   5 mac: a0:04:60:8a:29:5a sec: 5
ssid: buff.liv                      rssi:  -90 chan:  11 mac: 34:53:d2:e6:05:92 sec: 5
ssid: SpectrumSetup-71              rssi:  -51 chan:  11 mac: 94:18:65:50:15:ab sec: 5
ssid: Tina                          rssi:  -81 chan:  11 mac: 14:7d:05:4c:08:83 sec: 5
ssid: SpectrumSetup-97              rssi:  -59 chan:  11 mac: 2c:ea:dc:b1:10:95 sec: 5
ssid: SpectrumSetup-71              rssi:  -59 chan:  11 mac: 94:18:65:50:15:ab sec: 5
ssid: SpectrumSetup-71              rssi:  -77 chan:  11 mac: 74:37:5f:a1:a7:73 sec: 5
ssid: SpectrumSetup-71              rssi:  -49 chan:  11 mac: 94:18:65:50:15:ab sec: 5
ssid: buff.liv                      rssi:  -89 chan:  11 mac: 34:53:d2:e6:05:92 sec: 5
ssid: SpectrumSetup-71              rssi:  -64 chan:  11 mac: 94:18:65:50:15:ab sec: 5
```

Note that the same Wi-Fi network can appear multiple times in the same scan. In my later scan callback functions, I implement a check to only print each network's information once.

The Lightweight IP Stack (LwIP)

The Lightweight IP Stack is an implementation of the TCP/IP protocol originally authored by Adam Dunkels (the creator of Protothreads). The "lightweight" nature of LwIP makes it perfect for use on embedded systems. In particular, LwIP provides access to "raw" APIs for UDP/TCP, which are described as "an event-driven API designed to be used without an operating system."[1] Since the Pico-W does not natively support an operating system, the "raw" APIs are our only method of using LwIP.

LwIP raw UDP API

The "raw" UDP API makes use of a callback function for handling the reception of UDP datagrams. The user must define their own callback function, which can later be specified to the LwIP UDP API. The following code snippet depicts the UDP callback function that I used in all of my examples:

```
// UDP recv callback function
void udp_recv_callback(void* arg, struct udp_pcb* upcb, struct pbuf* p,
                       const ip_addr_t* addr, u16_t port)
```

[1] https://www.nongnu.org/lwip/2_1_x/group__callbackstyle__api.html

```
{
    // Prevent "unused argument" compiler warning
    LWIP_UNUSED_ARG(arg);

    printf("You've got mail! (received a packet)\n");

    if (p != NULL) {
        // Copy the payload into the recv buffer
        memcpy(recv_data, p->payload, UDP_MSG_LEN_MAX);

        // Free the packet buffer
        pbuf_free(p);

        // Signal waiting threads
        PT_SEM_SAFE_SIGNAL(pt, &new_udp_recv_s);
    } else {
        printf("ERROR: NULL pt in callback\n");
    }
}
```

This callback function simply checks that the contents of the *pbuf* are non-null, then it copies the contents of the payload to a *char[]* called *recv_data*. Keeping this function short minimizes the amount of time spent in the ISR, which is generally a good practice. Note that after copying the contents, the *pbuf* must be freed to avoid a memory leak. In my implementation, the contents of the *recv_data* buffer are then later parsed by a thread that gets signaled by the callback function.

To initialize the UDP receive functionality, the user must perform three actions. Just like the *udp_beacon* demo, the user must first allocate a new PCB for the Pico-W's end of the communication. The following function initializes a new UDP PCB, with the added specification that we would like to listen for both IPv4 and IPv6 packets.

```
// Create a new UDP PCB
udp_recv_pcb = udp_new_ip_type(IPADDR_TYPE_ANY);
```

If the PCB allocation is successful (non-null), we can bind the PCB to the socket IP address and port number:

```
// Bind the UDP PCB to the socket
// - netif_ip4_addr returns the pico's ip address
err = udp_bind(udp_recv_pcb, netif_ip4_addr(netif_list),
               UDP_PORT); // DHCP assigned address
```

Finally we can specify the callback function to the lower level API.

```
// Assign the callback function for when a UDP packet is received
udp_recv(udp_recv_pcb, udp_recv_callback, NULL);
```

Sending data from a Pico-W to a laptop

After connecting the Pico-W to Wi-Fi, the next goal was to successfully communicate with it. Luckily, Raspberry Pi provides a simple demo called *udp_beacon* that uses the LwIP API to broadcast UDP packets to a target. First, the Pico-W is placed in station mode and connected to an existing Wi-Fi network using the functions discussed above. For this demo, in addition to specifying a network SSID and password, the user must also specify a destination IPv4 address and port number.

```
#define BEACON_TARGET "192.168.1.1"
#define UDP_PORT      4444
```

Two steps of setup must be completed before sending anything. First, the user must allocate a new protocol control block (PCB) for the UDP socket. The PCB stores all of the state information associated with the socket on the Pico-W. Once initialized, the user does not need to interact with this data structure (outside of potentially deallocating it). Second, the IP address specified in the *BEACON_TARGET* macro must be converted to a form that is understandable by the LwIP API.

```
// Allocate new UDP PCB
struct udp_pcb* pcb = udp_new();

// Convert IP address
ip_addr_t addr;
ipaddr_aton(BEACON_TARGET, &addr);
```

Packets in LwIP are constructed using the *pbuf* data structure. The *pbuf* struct contains a void pointer that points to a dynamically allocated payload, along with fields specifying the length of the packet and other IP constructs. Therefore, to send a UDP packet using LwIP, we must first allocate a *pbuf* using the *pbuf_alloc()* function.

```
// Allocate pbuf
struct pbuf* p =
    pbuf_alloc(PBUF_TRANSPORT, BEACON_MSG_LEN_MAX + 1, PBUF_RAM);
```

After allocating the packet buffer, we can populate its payload using C standard library functions. In this basic UDP beacon demo, we send the value of a counter. First store a pointer to the *pbuf*

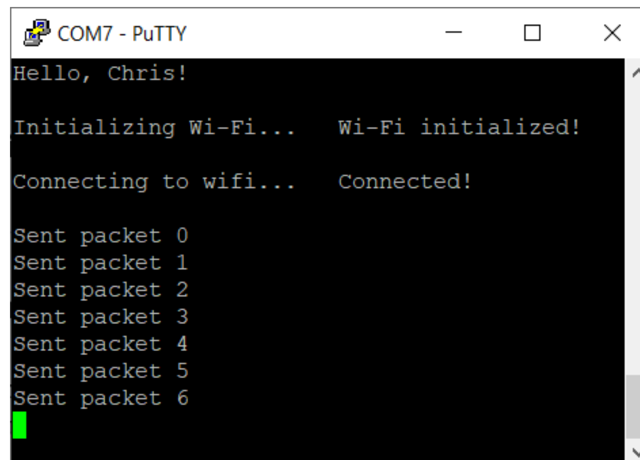
payload. Then we zero the entire payload using *memset()*. Finally we use *snprintf()* to write the contents of the counter to the payload.

```
// Compose payload
char* req = (char*) p->payload;
memset(req, 0, BEACON_MSG_LEN_MAX + 1);
snprintf(req, BEACON_MSG_LEN_MAX, "%d", counter);
```

After composing the packet, we can send it using a call to the LwIP *udp_sendto()* function.

```
// Send packet
err_t er = udp_sendto(pcb, p, &addr, UDP_PORT);
```

The first two arguments are the PCB and *pbuf*. The last two arguments specify the packet's destination. Once connected, the Pico-W will send UDP datagrams over the specified Wi-Fi network. Note that the Pico-W will *not* receive confirmation that the packet has arrived at its destination. Acknowledgements are not required by the UDP protocol. Here is the demo code's output to the serial terminal:



To receive packets from the beacon, I set up a simple python script running on my laptop that receives and decodes the incoming datagrams. The following code was adapted from [this example](#) of a python UDP server. First we import python's socket library. Then we specify the local IP address, port number, and the size of the incoming packets. Note that these must match the values that were provided to the Pico-W in our C code.

```
localIP = "192.168.1.1"
localPort = 4444
bufferSize = 1024
```

We can then create a UDP socket on the receiving end and bind that socket to the specified IP address and port number:

```
# Create a datagram socket
UDPServerSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

# Bind to address and ip
UDPServerSocket.bind((localIP, localPort))
```

The following loop will now listen for incoming datagrams and print out their contents:

```
# Listen for incoming datagrams
while (True):
    bytesAddressPair = UDPServerSocket.recvfrom(bufferSize)

    message, address = bytesAddressPair[0].decode(), bytesAddressPair[1]

    print("Client IP Address:{}".format(address))
    print("Message from Client:{}".format(message))
    print()
```

Here is a screenshot of the script receiving the 6 packets sent by the Pico-W from the *udp_beacon* example:

```
Select Developer PowerShell for VS 2022
PS C:\Users\Chris\Documents\Pico\PicoW-Demos\debug\udp_beacon> make
python udp_server.py
UDP server up and listening!
Client IP Address:('192.168.1.77', 56016)
Message from Client:0


Client IP Address:('192.168.1.77', 56016)
Message from Client:1


Client IP Address:('192.168.1.77', 56016)
Message from Client:2


Client IP Address:('192.168.1.77', 56016)
Message from Client:3


Client IP Address:('192.168.1.77', 56016)
Message from Client:4


Client IP Address:('192.168.1.77', 56016)
Message from Client:5


Client IP Address:('192.168.1.77', 56016)
Message from Client:6
```

Two-way UDP Communication using LwIP

With the knowledge of the Pico-W and the LwIP library, we now had the requisite toolset to begin constructing our first demo. The objective I chose for this was to demonstrate two-way UDP communication between a Pico-W access point and a Pico-W station. Despite its simplicity, I chose this objective for two reasons. First, it combines all of the functionality we have explored so far: hosting an access point, connecting to Wi-Fi, and sending/receiving UDP datagrams using LwIP. Second, this demo serves as the ideal starting point for ECE 4760 students who are looking to incorporate wireless communication into their final projects.

High-level overview

The following is an overview of the two-way UDP demo's behavior. Each node only has two responsibilities: listen for user input and listen for incoming packets. The resulting control flow is therefore quite simple.

Initialization:

1. Pico A hosts its access point under the name "picow_test"
2. Pico B enables station mode and connects to Pico A's access point

Loop:

1. Each Pico waits for user input to the serial monitor. Upon receiving input the Pico creates a UDP packet with the input as the payload and sends that packet to the other Pico.
2. Upon receiving a packet, the Pico parses the contents of the packet and prints out the payload to the serial monitor.

Implementation

The demo runs on three threads: *protothread_udp_send*, *protothread_udp_recv*, and *protothread_serial*. In addition it uses the UDP callback function and UDP callback initialization discussed previously. Here is a breakdown of each thread's responsibilities:

Thread: *protothread_serial*

- Spawn a thread for non-blocking read (yield until user input is detected)
- Upon user input, write the contents of *pt_serial_in_buffer* to the send buffer

Thread: *protothread_udp_send*

- Yield until the send buffer is written
- Initialize a new *pbuf*
- Copy the contents of the send buffer to the *pbuf* payload
- Send the packet using *udp_sendto()*

Thread: *protothread_udp_recv*

- Yield until the receive buffer is written
- Print out the contents of the receive buffer

Main:

- Initialize the cyw43
- If this Pico is an access point
    - Allocate the TCP server state
    - Enable AP mode
    - Initialize DHCP
- Otherwise, if this Pico is a station
    - Connect to the network "picow_test"
- Initialize UDP receive
- Start Protothreads

The *udp_send_recv* demo leverages compile-time definitions using CMake to compile into two different programs, one for the access point (*udp_ap.uf2*) and one for the station (*udp_station.uf2*).

Phase 2

Building a Standalone IoT System

Design Problem

Having gained familiarity with the Pico-W, the CYW43 API, and the LwIP stack, the next objective for the project was to create a final deliverable. In particular, we wanted to showcase the features of the Pico-W that we found most exciting with regards to its potential for IoT applications. We decided that those features were the following:

1. The ability for a Pico-W to host its own access point.
2. The ability to use Pico-W access points to communicate between two Picos without an external Wi-Fi network.

In brainstorming ideas for a final deliverable, we quickly gravitated towards the use of Pico-Ws as nodes in a larger network. We thought that the ability for Pico-Ws to connect together and behave as a single distributed system offered a new kind of versatility that distinguished the Pico-W from its predecessor. Multiple applications in this category were proposed, such as the use of Pico-Ws for geofencing or area-wide monitoring systems. In theorizing how these systems might work, a recurring component was figuring out how to efficiently route data across the network. The absence of a strict routing scheme is inefficient and leads to computational overhead, while pre-programming each node with a routing scheme inhibits the flexibility and scalability of a Pico-W network. Rather than focusing on a single use-case, we decided to target the underlying infrastructure that would make these large-scale networks possible.

We decided that our final objective was to implement a standalone IoT network using Pico-Ws as the nodes. In particular, we wanted to leverage the Pico-W's ability to host an access point to remove the network's reliance on an external Wi-Fi network. This feature has implications for how the network could be used, since Wi-Fi networks are not always present where IoT devices are meant to be deployed. We planned to implement a system for on-the-fly recognition of the network topology. This enables the devices to self-organize after being deployed. This system broke down into two components: a neighbor finding algorithm to organize the nodes and a routing algorithm to determine what paths packets should take through the network.

Augmented UDP program

Using my initial two-way UDP communication demo as a starting point, I added automatic Wi-Fi connection, packet acknowledgement, and round trip time (RTT) calculation. Automatic connection and packet acknowledgement became crucial features of the final design, which relies heavily on a revolving cycle of connecting and disconnecting to access points. Measuring RTT enabled me to quantize the performance of the system.

High-level overview

This example retains most of the control flow from the original UDP demo. A new thread is introduced called *protothread_udp_ack* which has a similar behavior to the existing *protothread_udp_send* thread, but for ACK packets. Here is an overview of the program's behavior:

Initialization:

1. Pico A hosts its access point using a name that starts with "*picow*"
2. Pico B enables station mode and scans for networks starting with "*picow*"
3. If it finds Pico A during this scan, it connects to Pico A's access point

Loop:

1. Each Pico waits for user input to the serial monitor. Upon receiving input the Pico creates a UDP packet with the input as the payload and sends that packet to the other Pico.
2. Upon receiving a packet, the Pico parses the contents of the packet and prints out the payload to the serial monitor. If the received packet contains data, return an acknowledgement packet back to the sender.

New thread: protothread_udp_ack

This program adds a new thread to the previous UDP program, meaning it now runs on four threads: *protothread_udp_send*, *protothread_udp_recv*, *protothread_udp_ack,* and *protothread_serial*. It still uses the same UDP callback infrastructure, but adds the code from the Wi-Fi scan example.

Enhanced UDP packets

To implement these new features, it was clear that each packet would have to contain more information. To accomplish this, I changed the contents of each packet to include five fields, delimited by semicolons:

TYPE	SRC. IP ADDR.	ACK #	TIMESTAMP	MSG.

1. The packet type (either "data" or "ack")
2. The IP address of the sender
3. An ACK number
4. The system time when the packet was sent
5. The contents of the packet

Automatically finding the access point

To remove the reliance on a hard-coded network SSID, I incorporated features from the Wi-Fi scan example to let the Pico-W station automatically seek the access point. First, the station assumes that the access point is hosting a network whose SSID starts with "*picow*" as the first 5 characters. Then the device scans for Wi-Fi networks with a callback function that explicitly searches for networks that start with "*picow*". If the scan encounters such a network, the SSID is saved in a global variable. The following code snippet excludes a significant portion of the callback function, but illustrates the general logic behind this process:

```
// Get first 5 characters of the SSID
char header[10] = "";
snprintf(header, 6, "%s", result->ssid);

if (strcmp(header, "picow") == 0) {
    snprintf(target_ssid, SSID_LEN, "%s", result->ssid); // Store the result
}
```

A W-Fi scan is then initiated from the *main()* function. If a result is found, the SSID is passed as an argument to *cyw43_arch_wifi_connect_timeout_ms()*.

Sending ACK packets

Acknowledgement packets (ACK packets for short) are a common concept in computer networking, most known for its use in TCP/IP. TCP specifies that received packets must be responded to by sending an ACK packet back to the sender. Since I didn't need any of the other features that TCP/IP provides, I decided to add my own simple version of acknowledgements onto my existing UDP implementation.

To avoid contention over buffers and threads used for sending data, I created a new thread called *protothread_udp_ack* that replicates the functionality of *protothread_udp_send*. When a packet is received, *protothread_udp_recv* checks if the incoming packet is data or an ACK. If the incoming packet is data, then an ACK packet is composed and placed in the buffer for *protothread_udp_ack*. The return address for this packet is then assigned using the *SRC. IP ADDR.* field from the received packet:

```c
// If data was received, respond with ACK
if (strcmp(packet_type, "data") == 0) {
    // Assign return address and ACK number
    strcpy(return_addr_str, src_addr);
    strcpy(return_timestamp, timestamp_str);
    return_ack_number = atoi(packet_num);

    // Signal ACK thread
    PT_SEM_SIGNAL(pt, &new_udp_ack_s);
}
```

Like the send thread, *protothread_udp_ack* also uses a semaphore to signal when a packet has been added to the queue.

Calculating round trip time (RTT)

Using the timestamp inserted into the packet, it is trivial to compute the packet's RTT by subtracting the timestamp from the current system time.

```c
// If this packet is an ack, calculate the RTT
if (strcmp(packet_type, "ack") == 0) {
    rtt_ms = (time_us_64() - timestamp) / 1000.0f;
}
```

Here the RTT is being computed in units of milliseconds. In my experimentation, RTT was typically around 30ms for small packets (around 100 bytes or less). Occasionally, a packet sent by an access point to a station would have an RTT around 120ms. I'm not sure why this was, but I'd hypothesize that it's related to some amount of background work regularly being performed to keep the access point running.

Neighbor Finding

The distance vector routing algorithm operates under the assumption that all nodes are aware of their neighbors. Therefore, before I could implement the routing algorithm itself, I first had to design a protocol for the Pico-Ws to figure out who their neighbors are. In addition, I needed to design a way for the Pico-Ws to distinguish between different neighbors. The neighbor finding algorithm that I created accomplishes both of these tasks:

1. It assigns every Pico a unique ID number. This gives each Pico a way of distinguishing itself and its neighbors from the rest of the network.
2. It provides every Pico knowledge of who its neighbors are.

The entire procedure is carried out automatically. There is no user input required except for an initial prompt to the root node.

High-level overview

The neighbor finding (NF) algorithm is similar to a depth-first search (DFS) algorithm. The head of the search is represented by a "token" packet that begins at the root node. The contents of the token packet indicate the ID number of the last node to be initialized by the algorithm. The initial state of the network is as follows:

- The token begins at the root node, its value is 0.
- The root node begins in station mode, initialized with ID #0, and it doesn't know who its neighbors are.
- Every other node begins in AP mode, with no ID number, and no sense of who its neighbors are.

The NF algorithm begins by typing the word "token" into the serial terminal on the root node. Every node waits until it receives the token packet. Once the token is received, it undergoes the following procedure:

1. If I have no ID number, assign myself an ID number and increment the token value. Record the ID of the Pico-W who gave me the token as my "parent" node.
2. If I already have an ID number, or if I just assigned myself an ID number, perform a Wi-Fi scan. If I find an uninitialized neighbor, pass them the token.
3. If no uninitialized neighbors are found, record the scan results as my list of neighbors, then return the token back to my parent node.

The algorithm terminates when the token returns to the root node and the root node has no uninitialized neighbors. When this happens, it is guaranteed that all nodes have ID numbers and know who their neighbors are.

The inability to maintain global information among all of the nodes forces us to make a key adaptation over a traditional DFS algorithm. Notice that when the NF algorithm reaches a maximum depth, the token packet must backtrack until it finds a node with an uninitialized neighbor. This is unlike a traditional DFS algorithm, which typically maintains a stack of unvisited nodes and sprouts a new branch when a maximum depth is reached. In the case of the NF algorithm, a queue of nodes is impossible to maintain, since the token can only be passed between two Pico-Ws that are within range of each other's access points.

New thread: protothread_connect

The neighbor finding program makes large additions to the augmented UDP program. The most important addition is a new thread called *protothread_connect* that handles toggling the access point on and off, scanning for Wi-Fi networks, and connecting to other networks. Other threads interact with *protothread_connect* by specifying an ID to connect to using the *target_ID* variable. Depending on the value of *target_ID*, the thread then takes one of three actions:

target_ID	**Action taken by *protothread_connect***
-2	Conduct a scan for any uninitialized neighbors. If one is found, store the result.
-1	Enable the access point.
≥ 0	Attempt to connect to the access point with the name "*picow_<target_ID>*"

Adding ID numbers to packets

With the introduction of ID numbers, I needed to add two new fields to each packet, one for the source node's ID and one for the destination node's ID. The new packet format has seven fields, five of which are directly inherited from the augmented UDP program:

TYPE	*DST. ID*	*SRC. IP*	*SRC. IP ADDR.*	*ACK #*	*TIMESTAMP*	*MSG.*

With seven fields per packet, I found it appropriate to create a new struct that aggregates all of the information contained in each packet into a single data structure.

```
// Structure that stores a packet
typedef struct packet {
    char packet_type[TOK_LEN];
    int dest_id;
    int src_id;
    char ip_addr[TOK_LEN];
    unsigned int ack_num;
    uint64_t timestamp;
    char msg[UDP_MSG_LEN_MAX];
} packet_t;
```

This structural change enabled the abstraction of packet-related functions into their own header file (*packet.h*). This file contained functions for creating packets, converting packets into strings, converting strings into packets, and printing the contents of packets in a formatted manner.

Distinguishing between initialized and uninitialized neighbors

To make the algorithm work, I had to devise a way to assign each Pico-W access point a unique SSID. For initialized nodes (nodes with an ID number) this is trivial. Any node with an ID number hosts a network using the name *picow_<ID>*, where *<ID>* is the ID number assigned by the neighbor finding algorithm. This problem becomes non-trivial when generating SSIDs for uninitialized nodes. To solve this, I use the *pico_get_unique_board_id()* function, which returns a 64-bit board ID number that is unique to each Pico-W. Every uninitialized node then hosts a network with the name *pidog_<unique ID>*, where *<unique ID>* is the 64-bit number returned by *pico_get_unique_board_id()*. The following image shows the result of a Wi-Fi scan that found both *picow* and *pidog* networks:

```
Starting neighbor finding scan...success!
        ssid: pidog_E6614864D388AD21        rssi:  -24 dB  <-- New node
        ssid: picow_0                       rssi:  -42 dB  <-- ID = 0
            ...
        scan result: pidog_E6614864D388AD21
Connecting to pidog_E6614864D388AD21...connected!
```

Distance Vector Routing

Distance-vector (DV) routing algorithms are a class of routing algorithms that optimize routing patterns in a network by iteratively exchanging information about the local topology between nodes. The DV routing algorithm has multiple characteristics that make it ideal for our application:

1. The DV algorithm is *decentralized* → No node is required to maintain complete information about the network topology.
2. The DV algorithm is *asynchronous* → No overarching clock is required to keep nodes synchronized.
3. The DV algorithm is *self-terminating* → When the network has reached the optimal state, nodes will automatically stop exchanging information.

As the network grows bigger, the task of maintaining the entire network topology and keeping nodes in-synch becomes more and more difficult. The use of a decentralized, asynchronous routing algorithm is what makes the network more flexible and more scalable than a network operating with pre-programmed routing patterns.

High-level overview

In my implementation of the DV routing algorithm, each node in the network maintains the following information about itself and its neighbors:

1. The distance from itself to every other node in the network (called a *distance vector*).
2. A routing table that specifies the next-hop node for a packet traveling to any ID number.
3. An approximation of each of its neighbors' distance vectors.
4. The list of neighbors that are not up-to-date on its distance vector.

The routing algorithm begins when the root node first sends its distance vector to a neighbor. This can be configured to happen automatically at the end of the neighbor finding phase.

When a node x receives a DV from one of its neighbors n, it uses that new information to update its own DV according to the following equation:

$$d_x(y) = min \{ d_x(y), cost(x \rightarrow n) + d_n(y) \}$$

In this equation, $d_x(y)$ is x's distance to y, $d_n(y)$ is n's distance to y, and $cost(x \rightarrow n)$ is the link cost from x to n. If a node x finds that its distance vector has been updated as a result of the new information, it flags all of its neighbors as "*not up-to-date*" indicating that every neighbor n needs an updated copy of x's distance vector. In my implementation, I assumed a link cost of 1 between any pair of adjacent nodes, meaning my implementation optimized for the number of hops. Alternative metrics for link cost could be functions of RTT (optimizing for speed) or RSSI (optimizing for reliability).

Like the NF algorithm, nodes are in access point mode by default. If a node has neighbors who are not up-to-date on its DV, it first waits until it has not received a distance vector within the past 15 seconds. After 15 seconds it drops its access point and scans for neighbors who are not up-to-date on its distance vector. If no unupdated neighbors are found, the node re-enables its access point for a random duration of time between 15 and 30 seconds. The assumption here is that if no unupdated neighbors are found, those neighbors are likely also in station mode conducting a scan.

If the node does find a neighbor, it connects to that neighbor and sends them an updated copy of its distance vector. Upon receiving an ACK from that neighbor, it runs another scan and looks for another unupdated neighbor. The node continues distributing its distance vector until it no longer finds any unupdated neighbors during its scan, after which it re-enables its access point for a random duration of time.

In a traditional implementation of the DV algorithm on wired routers, a node would broadcast its distance vector to all of its neighbors simultaneously. Since each Pico-W can only connect to one access point at a time, my implementation tries to update as many neighbors as possible, but settles for the fact that not every neighbor might be available (in access point mode)

at any given time. If there are still unupdated neighbors after a set of scans, the node will automatically conduct another set of scans after a short "cooldown" period.

Node and Neighbor Structs

In order to aggregate each node's metadata, I created a *node* struct. This stores identification information, data about the node's neighbors, its distance vector, and its routing table. Only one instance of this *struct* is ever initialized in the program. This node is called *self*, and its declaration can be found in the *node.h* file.

```
// Node struct
typedef struct node {

    int ID;         // ID number
    int parent_ID; // Parent node ID number

    unsigned int counter; // Count number of packets sent

    char wifi_ssid[SSID_LEN];  // My SSID when hosting an access point
    char ip_addr[IP_ADDR_LEN]; // IPv4 address

    int knows_nbrs; // Has the node been assigned an ID and found its neighbors

    int ID_is_nbr[MAX_NODES]; // Hashmap (<ID>, <bool>), true if ID is neighbor
    nbr_t* nbrs[MAX_NODES];   // Neighbor data, indexed by ID number
    int num_nbrs;             // Number of neighbors

    int dist_vector[MAX_NODES];   // My distance vector
    int routing_table[MAX_NODES]; // My routing table

} node_t;
```

The most important field in this struct is *nbrs[]*, which stores a list of pointers to each of the node's neighbors. These neighbors are dynamically allocated at the end of the neighbor finding phase. Each neighbor is represented using the following struct:

```
// Neighbor struct
typedef struct nbr {
    int ID;                     // ID number
    int cost;                   // Cost of sending a packet to this neighbor
    int dist_vector[MAX_NODES]; // Estimate of nbr's distance vector

    bool up_to_date;       // Is this nbr up-to-date on my DV?
    uint64_t last_contact; // Last time I tried/succeeded talking to this nbr
```

```
    bool new_dv; // New DV for this node that I haven't read yet?
} nbr_t;
```

This structure stores each neighbor's ID, distance vector, and time of last contact. It also keeps track of whether the neighbor is waiting on an updated copy of our distance vector, or if it has sent a distance vector to us that we have not accounted for yet.

Augmenting protothread_connect with a new type of scan

To scan for unupdated neighbors I implemented a new scan callback function that searches for unupdated neighbors. The callback function parses each neighbor's SSID for their ID number and checks if that neighbor has been updated with the current version of the distance vector. Here is an excerpt of the code from the new callback function:

```
nbr_t* nb = self.nbrs[id];

if (nb->up_to_date == true) {
    printf("\tssid: %-*s Last contact: %4.1fs\n", SSID_LEN,
           result->ssid, (nb->last_contact) / 1e6);
} else {
    printf("\tssid: %-*s Last contact: %4.1fs  <-- Needs my DV\n",
           SSID_LEN, result->ssid, (nb->last_contact) / 1e6);

    // Update "neediest" neighbor
    if (nb->last_contact < routing_scan_result->last_contact) {
        routing_scan_result = nb;
    }
}
```

The above code snippet introduces the concept of the "neediest neighbor," which refers to the neighbor that has had the longest time since being updated with a distance vector. The motivation here is that a node should try to keep all of its neighbors as up-to-date as possible, which means prioritizing the least recently updated neighbors first.

Updating a Distance Vector

As previously discussed, each time a node receives a distance vector from one of its neighbors it uses the new data to update its own routing information. For a node to update its distance vector, the following code is invoked:

```c
nbr_t* nb  = n->nbrs[nbr_ID];
nb->new_dv = false;
bool my_dv_updated = false;

// Check for a new shortest path to each node
for (int id = 0; id < MAX_NODES; id++) {
    int curr_dist = n->dist_vector[id];
    int new_dist  = nb->cost + nb->dist_vector[id];

    if (new_dist < curr_dist) {
        my_dv_updated = true;

        n->dist_vector[id]   = new_dist;
        n->routing_table[id] = nb->ID;

        printf("New dist to node %d through %d:\n", id, nbr_ID);
        printf("\tself.dist_vector[%d]: %d --> %d\n", id, curr_dist,
               new_dist);
    }
}
```

The node iterates through each ID number. If a new shortest path is found through neighbor *nb*, the node's distance vector and routing table are updated to reflect the new route.

Poisoned reverse and the count-to-infinity problem

A common edge-case in many dynamic routing algorithms (those that adapt the routing patterns to reflect changes in network topology) is the count-to-infinity problem. The count-to-infinity problem is the phenomenon by which a change in the network topology prompts a reconfiguration of the routing patterns that results in a routing loop (i.e. A routes through B, B routes through A). This stems from the fact that nodes in a decentralized algorithm have an incomplete view of the network topology, and thus cannot recognize when a routing loop is occurring. Thankfully, there are many methods of resolving the count-to-infinity problem, poisoned reverse being one of them.

Poisoned reverse is implemented as follows. Suppose a network where node A and node B are determining their optimal paths to node C. Further suppose that node A is routing through node B to get to node C. When node A sends its distance vector to node B, it will *lie* to B by claiming that the distance from A → C is infinite. This guarantees that B cannot find a shortest path to C through A, preventing B from routing packets that are destined for node C back to node A.

The following code snippet shows the implementation of my *dv_to_str()* function, which converts a distance vector to a string. This function is invoked whenever a node wants to

compose a packet payload containing a distance vector. Each value of the distance vector is delimited with a hyphen.

```
void dv_to_str(char* buf, node_t* n, int recv_ID, int dv[], bool poison)
{
    char dv_str[3 * MAX_NODES];
    int index = 0; // Index in the buffer where we are writing
    int value;     // Value of the distance vector to insert

    // Check for poison reverse on the first entry, then write the first entry
    // without a delimiter before it
    value = (poison && n->routing_table[0] == recv_ID) ? POISON_DIST : dv[0];
    index += snprintf(&dv_str[index], DV_MAX_LEN - index, "%d", value);

    for (int id = 1; id < MAX_NODES; id++) {
        // If poisoned reverse is true and I route through [recv_ID] to get
        // to this node, report distance as infinite (poison distance).
        value =
            (poison && n->routing_table[id] == recv_ID) ? POISON_DIST : dv[id];

        // Write the rest of the entries with a '-' delimiter
        index += snprintf(&dv_str[index], DV_MAX_LEN - index, "-%d", value);
    }

    // Write to the buffer
    snprintf(buf, 3 * MAX_NODES, "%s", dv_str);
}
```

For each entry in the distance vector, the node checks to see if the next-hop node associated with that destination is in fact the recipient of this distance vector. If so, it writes a special value (denoted by *POISON_DIST*) to that entry in the vector.

Routing packets

With the network fully set up, routing packets is trivial. When a node receives a packet, it checks that packet's destination (specified by the packet's *dest_id* field). If this node is the intended destination, no action is required. Otherwise, the node checks its routing table for the next-hop node for packets with that destination. It then forwards the packet downstream to the next node.

Testing Methodology

During our early experiments with the Pico-W, we found that (in an unobstructed environment) the effective range of a Pico-W access point was around 10 meters. This has excellent implications for the Pico-W's potential applications, but makes it difficult to properly test a network of Pico-Ws without access to a large area where Pico-Ws can be placed out-of-range of each other's access points. Therefore, I needed to implement a testing framework that allowed me to simulate network layouts without physically placing nodes out-of-range of each other.

Physical IDs

I implemented a system of physical IDs, independent from the IDs assigned by the neighbor finding phase. Each node's physical ID is derived from its 64-bit unique ID by indexing into the following array:

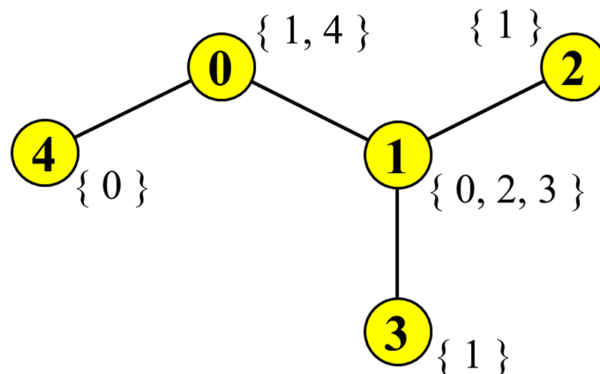
```
char board_IDs[NUM_BOARDS][20] = {
    "E6614864D32F7622", // 0
    "E6614864D36FAF21", // 1
    "E6614864D388AD21", // 2
    "E6614864D3138C21", // 3
    "E661410403492722"  // 4
};
```

I inserted an extra field into each SSID specifying the Pico-W's physical ID number. Each uninitialized node hosts a network with the SSID *pidog_<phys ID>_<unique ID>*, while an initialized node uses the SSID *picow_<phys ID>_<ID>*.

Simulating non-adjacency between nodes

The network layout is specified using an adjacency matrix. The following matrix is one of my testing layouts:

```
// Five nodes
int adj_list[MAX_NODES][MAX_NODES] = {
    {1, 4, EOL},    // 0
    {0, 2, 3, EOL}, // 1
    {1, EOL},       // 2
    {1, EOL},       // 3
    {0, EOL}        // 4
};
```



I then augmented each Wi-Fi scan callback function with an additional section that checks the adjacency matrix for connectivity. The scan callback function parses the physical ID out of the SSID and exits if the node isn't adjacent.

Results and Conclusions

The following section covers my experimentation with the completed design. In particular, I make an overall evaluation of the design's performance, followed by my findings on the routing algorithm's time to convergence, the network's flexibility, and a discussion of future areas of interest.

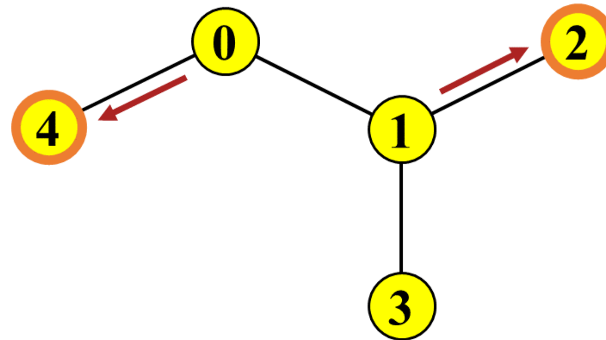
Performance assessment

The objective of this phase was to create an underlying framework to handle data communication in Pico-W networks. The performance of the design will therefore be evaluated on its ability to successfully move packets across different network topologies. Using simulated layouts, I was able to test a number of different layouts to verify the correct functionality of the network. In every case the network successfully initialized itself with ID numbers and optimized its routing patterns. Once the network was fully set up, I was consistently able to route packets between any two nodes in the network.

Time to convergence

In a network with five nodes, the neighbor finding algorithm takes about 30 seconds to resolve. Once each node has an assigned ID number, the distance vector routing phase takes a similar amount of time (30 to 45 seconds) to optimize the network routing configuration. One characteristic about the distance vector routing protocol is that optimizations are done in parallel across the entire network. The only restriction on parallelism is that no node may connect to two

access points at once. The following diagram illustrates an example of two communications that could occur in parallel:



The orange outlined nodes (IDs #2 and #4) are currently hosting their access points, while nodes #0 and #1 are sending their distance vectors to nodes #4 and #2 (respectively).

Flexibility and scalability

The ability for the network to self-identify its own topology makes it incredibly flexible. In brainstorming use cases for a Pico-W network, applications requiring multiple topologies were theorized: rings, chains, sparsely and tightly connected networks. Through experimentation with different network layouts using my simulation framework, I am convinced that the network would be able to handle any of those topologies.

The parallel nature of the distance vector routing algorithm has excellent implications for scalability. Namely, it implies that as the network scales in size, the time to convergence remains manageable. The self-optimizing nature of the network means that no extra effort is required on behalf of the user to configure a larger network.

What's next?

The goal of this design project was to demonstrate the viability of the Pico-W as the basis for nodes in large network-based IoT applications. The flexibility and scalability of the network leads me to believe that a Pico-W network is more than capable of serving that purpose. In addition, this design is just a baseline for what can be achieved with Pico-W networks. The following examples are refinements that could be made to this design:

1. Increase the size of the send and receive queues → Larger queues would allow the multiple packets to traverse the network at the same time and improve network throughput.
2. Decrease the scan cooldown period → If tuned properly, shorter delays could significantly decrease convergence time during the routing phase without increasing the rate of connection failures.
3. Use two Pico-Ws per node instead of one → Using a pair of microcontrollers would enable each node to simultaneously behave as an access point and a station at the same time. This would eliminate the need to toggle between modes, decreasing overhead and significantly increasing the network throughput.

Alongside software optimizations, augmentations could be made to the nodes themselves. In a recent conversation with Professor Joe Skovira, we talked about the implications of interfacing Pico-Ws with LoRa modules: low-power and long range radio transmission modules built for embedded IoT devices. The use of LoRa modules could potentially extend the range of a single node from 10 meters to a few kilometers, greatly increasing the versatility of these networks.

Acknowledgments

I would like to thank my advisor Hunter Adams for his support throughout my design project. We developed the entire concept for this project together, so this project wouldn't exist without him. Without his teaching I would have neither the skillset nor the passion for working with these incredible devices.

I would also like to thank Bruce Land for his guidance. Discussions with him contributed heavily towards my understanding of the Raspberry Pi Pico and the Lightweight IP library. His investigation with the Pico-W has also been incredibly interesting, and I suggest that anyone looking to work with the Pico-W check out his work as well.

References

Raspberry Pi

Raspberry Pi Documentation:
https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html
Pico SDK Documentation:
https://cec-code-lab.aps.edu/robotics/resources/pico-c-api/index.html
The Pico C/C++ SDK:
https://github.com/raspberrypi/pico-sdk
Pico SDK Examples:
https://github.com/raspberrypi/pico-examples

Bruce Land

Bruce Land's Website:
https://people.ece.cornell.edu/land/courses/projects/index_projects.html

Lightweight IP Stack

LwIP Overview:
https://www.nongnu.org/lwip/2_1_x/index.html
LwIP Raw UDP:
https://lwip.fandom.com/wiki/Raw/UDP

Research on UDP, TCP, and routing algorithms

Computer Networking: A Top-Down Approach, 8th Edition, James F Kurose & Keith W. Ross

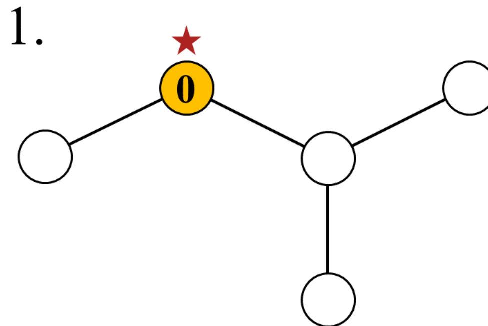
My own work

My GitHub Repository:
https://github.com/cc2698/PicoW-Demos

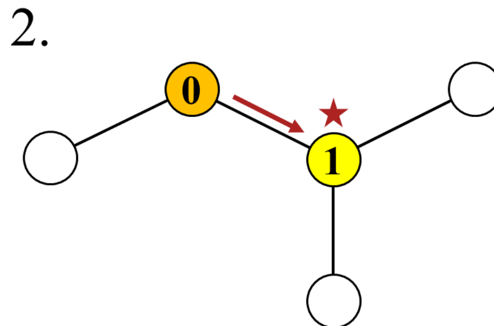
Appendix A: Neighbor Finding Walkthrough

The following is a full walkthrough of the neighbor finding algorithm. It shows step-by-step how each node is assigned an ID number.

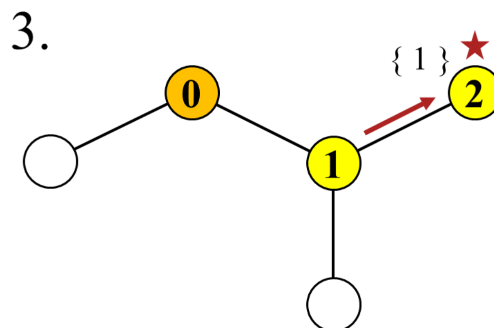
1. All nodes except for the root node begin uninitialized. The token (indicated by a star) begins at the root node, which has an ID number of 0.



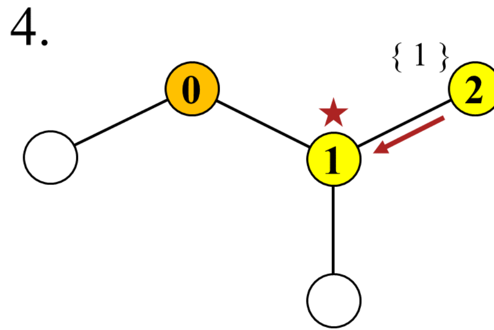
2. Node 0 performs a Wi-Fi scan and sees two uninitialized neighbors. It passes the token to one of those neighbors. That neighbor initializes itself with an ID number of 1.



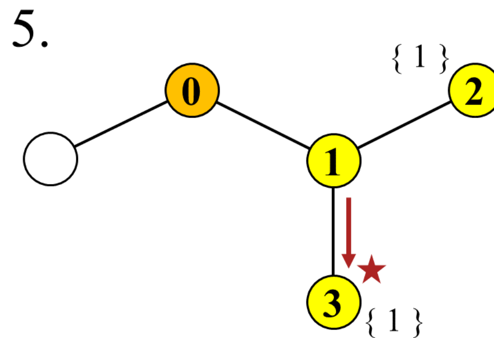
3. Node 1 performs a Wi-Fi scan and sees two uninitialized neighbors. It passes the token to one of those neighbors. That neighbor initializes itself with an ID number of 2. Having no uninitialized neighbors, Node 2 records its list of neighbors { 1 }.



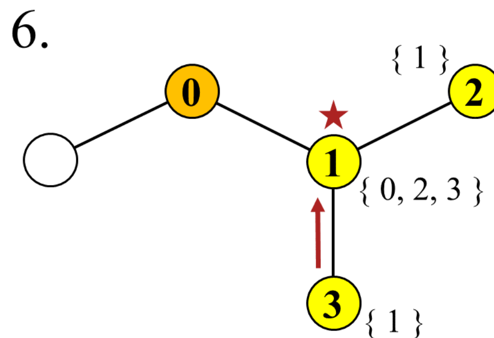
4. Node 2 passes the token back to its parent, node 1.



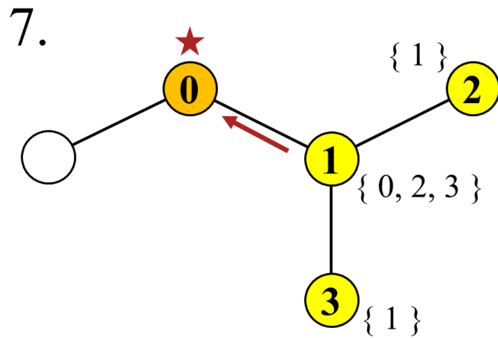
5. Node 1 performs a Wi-Fi scan and sees one uninitialized neighbor. It passes the token to one of those neighbors. That neighbor initializes itself with an ID number of 3. Having no uninitialized neighbors, Node 3 records its list of neighbors { 1 }.



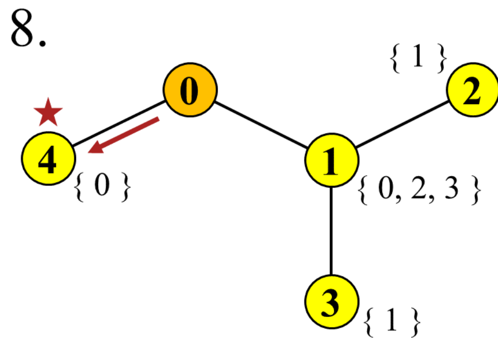
6. Node 3 passes the token back to its parent, node 1. Having no uninitialized neighbors, Node 1 records its list of neighbors { 0 2 3 }.



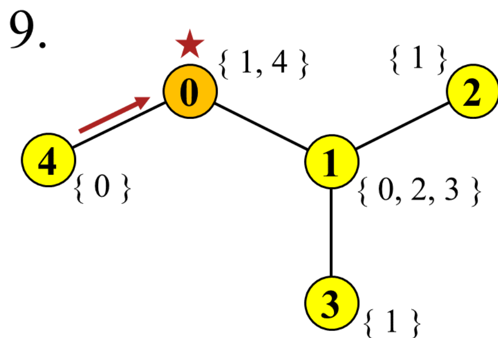
7. Node 1 passes the token back to its parent, node 0.



8. Node 0 performs a Wi-Fi scan and sees one uninitialized neighbor. It passes the token to one of those neighbors. That neighbor initializes itself with an ID number of 4. Having no uninitialized neighbors, Node 4 records its list of neighbors { 0 }.



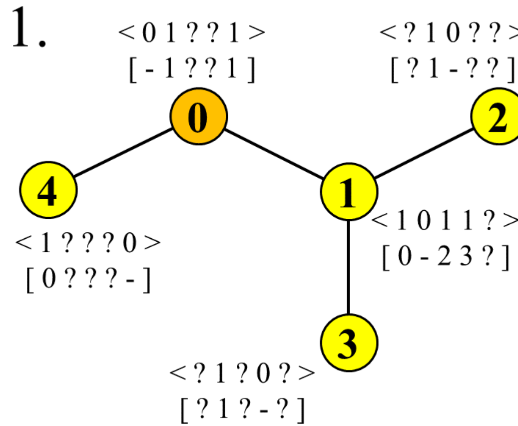
9. Node 4 passes the token back to its parent, node 0. The token has returned to the root node, and the root node has no uninitialized neighbors, indicating that the neighbor finding algorithm has finished.



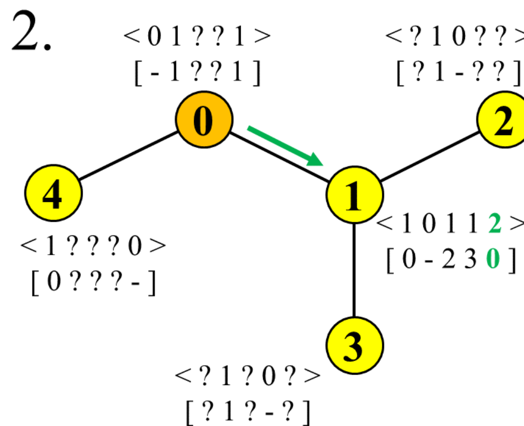
Appendix B: Distance Vector Routing Walkthrough

The following is a partial walkthrough of the distance vector algorithm. It shows how the exchange of information between nodes leads to an optimized routing scheme. Transmissions that do not change any routing tables are omitted.

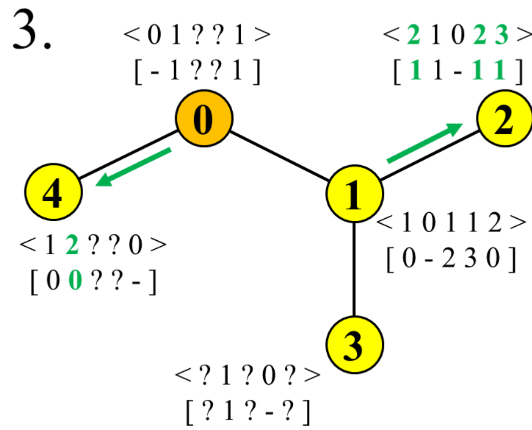
1. Each node begins with an incomplete distance vector and routing table.



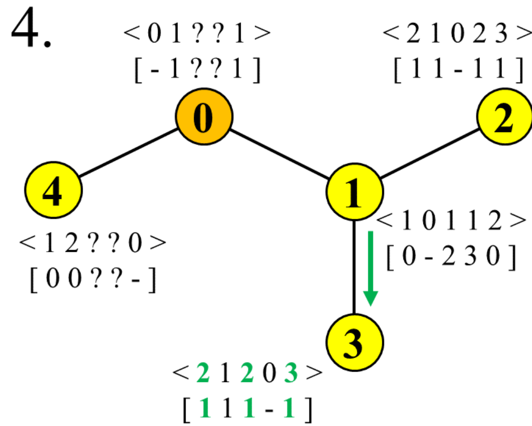
2. Node 0 scans its neighbors and finds that node 1 is not up-to-date. It passes node 1 a copy of its distance vector. Node 1 updates accordingly.



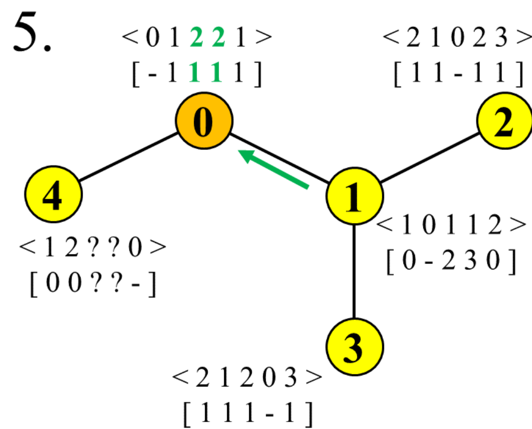
3. Node 0 scans its neighbors and finds that node 4 is not up-to-date. It passes node 4 a copy of its distance vector. Node 4 updates accordingly. At the same time, node 1 scans its neighbors and finds that node 2 is not up-to-date. It passes node 2 a copy of its distance vector. Node 2 updates accordingly.



4. Node 1 scans its neighbors and finds that node 3 is not up-to-date. It passes node 3 a copy of its distance vector. Node 3 updates accordingly.



5. Node 1 scans its neighbors and finds that node 0 is not up-to-date. It passes node 0 a copy of its distance vector. Node 0 updates accordingly. Note that node 4's copy of node 0's distance vector is now out-of-date.



6. Node 0 scans its neighbors and finds that node 4 is not up-to-date. It passes node 4 a copy of its distance vector.