# Python Radio 36: Mesh Networking

[Simon Quellen Field](https://medium.com/@simon.field_37276?source=post_page---byline--46d447caaa97---------------------------------------)

[Simon Quellen Field](https://medium.com/@simon.field_37276?source=post_page---byline--46d447caaa97---------------------------------------)

Follow

16 min read

·

Apr 15, 2025

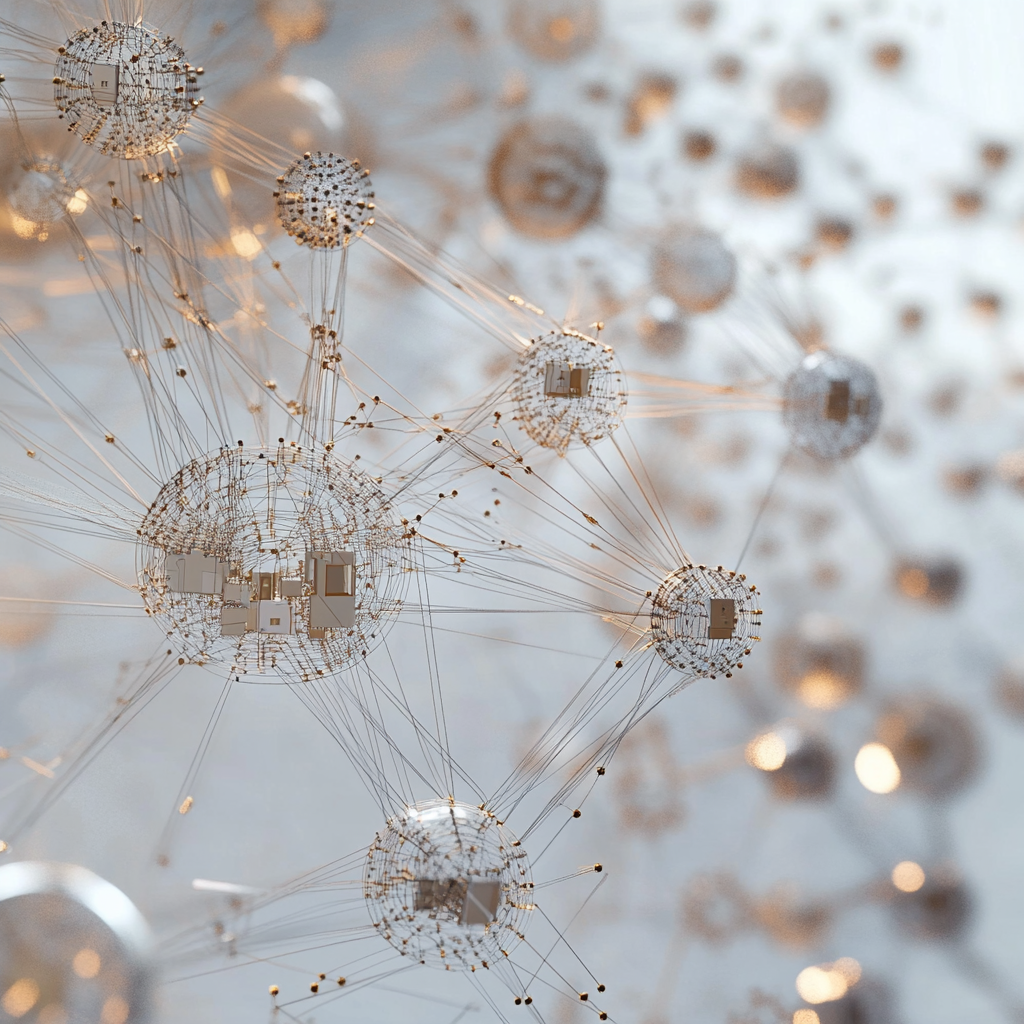
Listen

Share

More

Get the word out to nodes you can’t see.

Press enter or click to view image in full size



MidJourney

Here in California’s San Francisco Bay Area, we have a very active group using the Meshtastic software to form a mesh network of LoRa nodes.

I can send messages to San Francisco 54 miles away using only a few milliwatts of power and a tiny solar-powered computer that cost me about $30 to put together.

The Meshtastic software is very robust and capable, with lots of bells and whistles. I thought it would be good fun to build a much simpler, smaller system using nodes that cost $5 (although I have found two sites on Aliexpress.com that offer them for 99 cents, but only one to a customer, presumably as a marketing program).

By using the built-in WiFi instead of a separate LoRa radio chip, we save cost at the expense of long distance but gain a lot of extra bandwidth.

The board I chose is one of my new favorites, the ESP32S3 Supermini. It has 4 megabytes of flash memory, 2.5 megabytes of RAM, runs at 240 megahertz, has two 32-bit processors (and a third 32-bit low-power processor), WiFi, Bluetooth, 24 GPIO pins, an RGB LED, and a Type-C connector for power and programming.

And the little thing is the size of my thumbnail:

Press enter or click to view image in full size

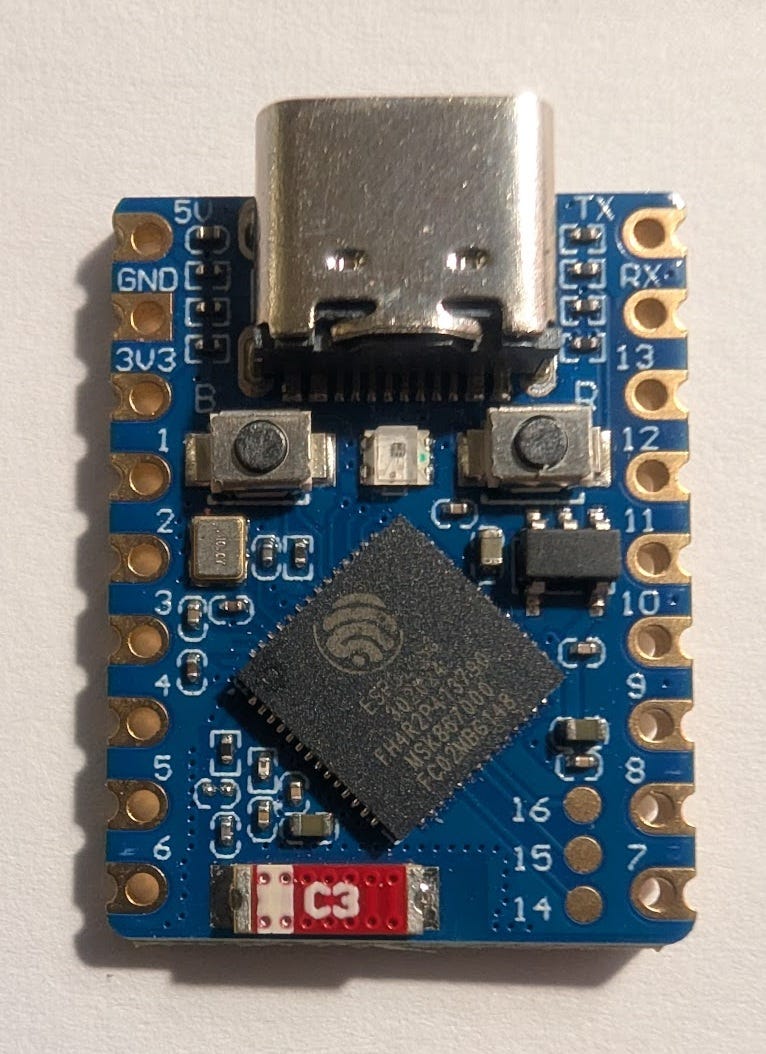


Photo by author

That tiny little surface-mounted red device labeled C3 is the antenna. For most uses, you would probably want to solder a 6-inch wire to that to get better range, but it reaches all over my house through walls, so it is fine for most things.

We use the tiny antenna to make testing our mesh network easier. We want to have some nodes out of reach of at least one node to make sure we are actually networking and not talking to them directly.

### The Design

The goal of the project is to make it simple.

So we will not do any fancy routing. We will broadcast each message to all the neighbors we can reach. They will receive it and broadcast it to everyone they can reach.

This could result in a catastrophic explosion of packets, saturating the network quickly, except for two things:

* Each packet has a time-to-live number that is decremented before retransmitting the packet. If the number gets to zero, the node drops the packet.
* Each packet has a sequence number, and if a node sees a sequence number it has already processed, it drops the packet. We keep the last 20 sequence numbers in a circular list.

That’s the whole design.

Much of the code is just a simple web server.

### The Code

We have two support modules, connect.py and uping.py. We don’t actually need uping.py, but it made debugging the code easier at one point. It ‘ping’s an IP address to verify the connection.

Here is connect.py, the module that set up the WiFi access point and connects to the other nodes’ access points:

class Connect:  
 def \_\_init\_\_( self ):  
 from network import WLAN, STA\_IF, AP\_IF, AUTH\_OPEN  
 from random import randint  
  
 self.ssid = ""  
 self.verbose = False  
 self.network\_list = []  
  
 self.sta = WLAN( STA\_IF )  
 self.sta.active( False )  
 self.sta.active( True )  
  
 self.ap = WLAN( AP\_IF )  
 self.ap.active( False )  
 self.ap.active( True )  
 self.mac = self.ap.config('mac')  
  
 self.mac\_print = ""  
 for m in self.mac:  
 self.mac\_print += hex(m)[2:] + ":"  
 self.mac\_print = self.mac\_print[:-1]  
  
 self.mac\_str = ""  
 for m in self.mac[3:]:  
 self.mac\_str += hex(m)[2:]  
  
 ip1 = 10  
 ip2 = self.mac[-2]  
 ip3 = self.mac[-1]  
 ip4 = 1  
  
 ip\_str = str(ip1) + "." + str(ip2) + "." + str(ip3) + "." + str(ip4)  
 ip\_mask = "255.255.255.0"  
  
 self.ap.ifconfig((ip\_str, ip\_mask, ip\_str, "8.8.8.8"))  
 self.ap\_ip = self.ap.ifconfig()[2]  
 self.sta\_host\_ip = False  
 self.sta\_client\_ip = False  
 self.who\_am\_i = "mesh\_" + self.mac\_str  
  
 self.ap.config( essid=self.who\_am\_i, authmode=AUTH\_OPEN )  
 self.verbose and print(f"Serving as {self.who\_am\_i} on subnet {self.ap\_ip}")  
  
 def connect( self ):  
 from network import WLAN, STA\_IF, AUTH\_OPEN  
 from uping import ping  
 from time import sleep  
  
 self.verbose and print( "Connect" )  
  
 for count in range(5):  
 self.sta.active( True )  
 try:  
 self.network\_list = self.sta.scan()  
 # Sort by RSSI  
 self.network\_list = sorted(self.network\_list, key=lambda x: x[3], reverse=True)  
 break  
 except Exception as e:  
 print( "scan failed:", e )  
 self.sta.disconnect()  
 self.sta.active( False )  
 self.sta = WLAN( STA\_IF )  
 self.sta.active( True )  
  
 try:  
 if not self.sta.isconnected():  
 for net in self.network\_list:  
 self.ssid = net[0].decode("utf-8")  
 if "mesh\_" in self.ssid:  
 try:  
 self.sta.connect( self.ssid, "" )  
 count = 0  
 while not self.sta.isconnected() and count < 10:  
 sleep( 1 )  
 count += 1  
 if not self.sta.isconnected():  
 continue  
 ip = self.sta.ifconfig()[0]  
 self.sta.config(dhcp\_hostname=self.who\_am\_i)  
 self.sta\_host\_ip = self.sta.ifconfig()[2]  
 self.sta\_client\_ip = self.sta.ifconfig()[0]  
 p = ping(str(self.sta\_host\_ip), quiet=True)  
 if p[1] == 0:  
 self.verbose and print(f"Can't ping {self.sta\_host\_ip}")  
 else:  
 # self.verbose and print(f"Pinged: {p}")  
 # self.verbose and print(f"Ifconfig: {self.sta.ifconfig()}")  
 # self.verbose and print(f"Connected to host at subnet {self.sta\_host\_ip}")  
 pass  
 return True  
 except Exception as e:  
 print( "Error in connect():", e )  
 pass  
 else:  
 if self.verbose:  
 print( "Already connected" )  
 return True  
 except Exception as e:  
 print( "connect():", e )  
 self.ssid = ""  
 self.verbose and print(f"No mesh nodes found")  
 return False  
  
 def reconnect( self ):  
 # self.verbose and print( "Reconnecting" )  
 if self.sta.isconnected():  
 self.sta.disconnect()  
 self.sta.active( False )  
 return self.connect()

It sets up an access point, advertising itself as mesh\_XXXXXX, where the Xs are the last 3 bytes of the MAC address, in hexadecimal.

It then searches for other SSIDs that start with “mesh\_” and tries to connect to them.

The first node to power up will not find any. But the second one to power up within reach of the signal will find this node and connect.

The uping.py module implements the ping command:

# µPing (MicroPing) for MicroPython  
# copyright (c) 2018 Shawwwn <shawwwn1@gmail.com>  
# License: MIT  
  
# Internet Checksum Algorithm  
# Author: Olav Morken  
# https://github.com/olavmrk/python-ping/blob/master/ping.py  
# @data: bytes  
def checksum(data):  
 if len(data) & 0x1: # Odd number of bytes  
 data += b'\0'  
 cs = 0  
 for pos in range(0, len(data), 2):  
 b1 = data[pos]  
 b2 = data[pos + 1]  
 cs += (b1 << 8) + b2  
 while cs >= 0x10000:  
 cs = (cs & 0xffff) + (cs >> 16)  
 cs = ~cs & 0xffff  
 return cs  
  
def ping(host, count=4, timeout=5000, interval=10, quiet=False, size=64):  
 import utime  
 import uselect  
 import uctypes  
 import usocket  
 import ustruct  
 import urandom  
  
 # prepare packet  
 assert size >= 16, "pkt size too small"  
 pkt = b'Q'\*size  
 pkt\_desc = {  
 "type": uctypes.UINT8 | 0,  
 "code": uctypes.UINT8 | 1,  
 "checksum": uctypes.UINT16 | 2,  
 "id": uctypes.UINT16 | 4,  
 "seq": uctypes.INT16 | 6,  
 "timestamp": uctypes.UINT64 | 8,  
 } # packet header descriptor  
 h = uctypes.struct(uctypes.addressof(pkt), pkt\_desc, uctypes.BIG\_ENDIAN)  
 h.type = 8 # ICMP\_ECHO\_REQUEST  
 h.code = 0  
 h.checksum = 0  
 h.id = urandom.getrandbits(16)  
 h.seq = 1  
  
 # init socket  
 sock = usocket.socket(usocket.AF\_INET, usocket.SOCK\_RAW, 1)  
 sock.setblocking(0)  
 sock.settimeout(timeout/1000)  
 addr = usocket.getaddrinfo(host, 1)[0][-1][0] # ip address  
 sock.connect((addr, 1))  
 not quiet and print("PING %s (%s): %u data bytes" % (host, addr, len(pkt)))  
  
 seqs = list(range(1, count+1)) # [1,2,...,count]  
 c = 1  
 t = 0  
 n\_trans = 0  
 n\_recv = 0  
 finish = False  
 while t < timeout:  
 if t==interval and c<=count:  
 # send packet  
 h.checksum = 0  
 h.seq = c  
 h.timestamp = utime.ticks\_us()  
 h.checksum = checksum(pkt)  
 if sock.send(pkt) == size:  
 n\_trans += 1  
 t = 0 # reset timeout  
 else:  
 seqs.remove(c)  
 c += 1  
  
 # recv packet  
 while 1:  
 socks, \_, \_ = uselect.select([sock], [], [], 0)  
 if socks:  
 resp = socks[0].recv(4096)  
 resp\_mv = memoryview(resp)  
 h2 = uctypes.struct(uctypes.addressof(resp\_mv[20:]), pkt\_desc, uctypes.BIG\_ENDIAN)  
 # **TODO:** validate checksum (optional)  
 seq = h2.seq  
 if h2.type==0 and h2.id==h.id and (seq in seqs): # 0: ICMP\_ECHO\_REPLY  
 t\_elasped = (utime.ticks\_us()-h2.timestamp) / 1000  
 ttl = ustruct.unpack('!B', resp\_mv[8:9])[0] # time-to-live  
 n\_recv += 1  
 not quiet and print("%u bytes from %s: icmp\_seq=%u, ttl=%u, time=%f ms" % (len(resp), addr, seq, ttl, t\_elasped))  
 seqs.remove(seq)  
 if len(seqs) == 0:  
 finish = True  
 break  
 else:  
 break  
  
 if finish:  
 break  
  
 utime.sleep\_ms(1)  
 t += 1  
  
 # close  
 sock.close()  
 ret = (n\_trans, n\_recv)  
 not quiet and print("%u packets transmitted, %u packets received" % (n\_trans, n\_recv))  
 return (n\_trans, n\_recv)

Unlike the other modules, I didn’t write this one. But it came in quite handy, so I share it here.

### The Main Program

As you might expect, most of the work is done in the main program. Some of it is just a simple web server, something we’ve covered earlier.

The unquote(), handle\_query(), decode\_path(), req\_handler(), and client\_handler() process packets that come in and parse the few HTTP protocols we accept. There is nothing “mesh networky” about them.

The dup\_sequence() method handles the circular list of sequence numbers we’ve seen. When it finds a duplicate, it returns True to drop the packet.

The show\_neighbors() method is for debugging. It prints the neighbor list on the console.

The whodat dictionary helps keep track of the names of the nodes. It is also just for debugging, and it occasionally gets it wrong in non-critical ways.

When a node connects to our node, add\_neighbor() puts it into a list of neighbors. It ignores neighbors we already have in the list. Importantly, it calls greet() to send a message back to the connecting node, telling it who we are, and collecting similar information in the reply.

Before I go any further, here is the code:

from uasyncio import get\_event\_loop  
from connect import Connect  
from time import sleep  
from machine import Pin  
from neopixel import NeoPixel  
from socket import getaddrinfo, socket, AF\_INET, SOCK\_STREAM, SOL\_SOCKET, SO\_REUSEADDR, SOL\_SOCKET  
from whoami import WhoAmI  
from sys import print\_exception  
from random import randint  
  
w = WhoAmI()  
rgb = NeoPixel(Pin(w.neo\_pin(), Pin.OUT), 1)  
  
def unquote(string):  
 if not string:  
 return b""  
  
 if isinstance(string, str):  
 string = string.encode("utf-8")  
  
 bits = string.split(b"%")  
 if len(bits) == 1:  
 return string  
  
 res = bytearray(bits[0])  
 append = res.append  
 extend = res.extend  
  
 for item in bits[1:]:  
 try:  
 append(int(item[:2], 16))  
 extend(item[2:])  
 except KeyError:  
 append(b"%")  
 extend(item)  
  
 return bytes(res).decode("utf-8")  
  
class Mesh:  
 def \_\_init\_\_(self):  
 self.verbose = False  
 self.neighbors = []  
 self.query = ""  
 self.sequence\_number = randint(1, 100\_000\_000)  
 self.sequences = [0] \* 20  
 self.seq\_count = 0  
 self.path = ""  
 self.port = 80  
 self.whodat = {}  
 self.con = None  
 self.name = w.name()  
 self.addr = getaddrinfo("0.0.0.0", self.port)[0][-1]  
 self.srv = socket(AF\_INET, SOCK\_STREAM)  
 self.srv.setsockopt(SOL\_SOCKET, SO\_REUSEADDR, 1)  
 self.srv.bind(self.addr)  
 self.srv.listen(10) # at most 10 clients  
 self.srv.setblocking(False)  
 self.srv.setsockopt(SOL\_SOCKET, 20, self.client\_handler)  
  
 def dup\_sequence(self, seq):  
 for s in self.sequences:  
 if seq == s:  
 return True  
 self.sequences[self.seq\_count] = seq  
 self.seq\_count += 1  
 self.seq\_count %= 20  
 return False  
  
 def show\_neighbors(self):  
 print("Neighbors:")  
 print("----------")  
  
 for n in self.neighbors:  
 ssid, ip, name = n  
 if ip == self.con.sta\_host\_ip:  
 print(f"Neighbor '{ssid}' '{ip}' '{name}' [Our host]")  
 else:  
 print(f"Neighbor '{ssid}' '{ip}' '{name}'")  
  
 for n in self.whodat.values():  
 his\_name, his\_ip = n  
 print(f"Client '{his\_name}' '{his\_ip}'")  
  
 print("----------")  
  
 def who\_name(self, ap\_ip, sta\_ip, name):  
 if name != "unknown":  
 for n in self.whodat.values():  
 his\_name, his\_ip = n  
 if ap\_ip[:-2] == his\_ip[:-2]:  
 self.whodat[his\_ip] = (name, his\_ip)  
 if sta\_ip and sta\_ip[:-2] == his\_ip[:-2]:  
 self.whodat[his\_ip] = (name, his\_ip)  
  
  
 def add\_neighbor(self, ip, ssid="no ssid", name="unknown"):  
 self.who\_name(ip, None, name)  
 # self.verbose and print(f"Adding neighbor: {ip} {ssid} {name}")  
 for i, n in enumerate(self.neighbors):  
 n\_ssid, n\_ip, n\_name = n  
 # self.verbose and print(f"Checking neighbor: {n\_ip} {n\_ssid} {n\_name}")  
 if n\_ip == ip: # We've seen this before  
 # self.verbose and print(f"We already have this neighbor")  
 if n\_name == "unknown":  
 self.neighbors[i][2] = name  
 if n\_ssid == "no ssid":  
 self.neighbors[i][0] = ssid  
 return  
  
 if ip == self.con.ap\_ip: # We are not our own neighbor  
 # self.verbose and print(f"We are not our own neighbor")  
 return  
  
 try:  
 self.greet(ip, name)  
 except Exception as e:  
 self.verbose and print(f"Exception in connecting: {e}")  
 print\_exception(e)  
 pass  
 for n in self.neighbors:  
 n\_ssid, n\_ip, n\_name = n  
 if n\_ssid == ssid and n\_ip = ip and n\_name == name:  
 return  
 self.verbose and print(f"{w.name()}: Added neighbor {ssid} {ip} {name}")  
 self.neighbors.append([ssid, ip, name])  
  
 def execute(self, payload):  
 if payload.get("rgb", False):  
 i = payload["rgb"]  
 r = (i >> 16) & 0xFF  
 g = (i >> 8) & 0xFF  
 b = (i ) & 0xFF  
 rgb[0] = (r, g, b)  
 rgb.write()  
 name = payload["name"]  
 ap\_ip, sta\_ip = payload["origin"]  
 self.who\_name(ap\_ip, sta\_ip, name)  
 elif payload.get("text", False):  
 print()  
 msg = payload["message"]  
 who = payload["to"]  
 name = payload["name"]  
 print(f"Message from {name} to {who}:")  
 print(msg)  
 print()  
 pass  
 elif payload.get("greet", False):  
 name = payload["name"]  
 ap\_ip, sta\_ip = payload["origin"]  
 self.who\_name(ap\_ip, sta\_ip, name)  
 else:  
 self.verbose and print(f"Execute({payload})")  
  
 def colors(self):  
 import json  
 from random import randint  
 r = randint(0, 255)  
 g = randint(0, 255)  
 b = randint(0, 255)  
 for n in self.neighbors:  
 ssid, ip, name = n  
 payload = {  
 "name": w.name(),  
 "to": name,  
 "ssid": self.con.who\_am\_i,  
 "rgb": (r << 16) + (g << 8) + b,  
 "time\_to\_live": 1,  
 "direct": self.con.ap\_ip,  
 "origin": (self.con.ap\_ip, self.con.sta\_client\_ip)  
 }  
 self.direct\_msg(ip, "none", json.dumps(payload))  
  
 def greet(self, ip, name):  
 import json  
 # self.verbose and print(f"Greeting {name} at {ip}")  
 payload = {  
 "name": w.name(),  
 "to": name,  
 "ssid": self.con.who\_am\_i,  
 "greet": True,  
 "time\_to\_live": 0,  
 "direct": self.con.ap\_ip,  
 "origin": (self.con.ap\_ip, self.con.sta\_client\_ip)}  
 self.direct\_msg(ip, "none", json.dumps(payload))  
  
 def direct(self, ip):  
 if ip == self.con.sta\_host\_ip:  
 # self.verbose and print(f"Sending to our host")  
 return True  
 for n in self.whodat.values():  
 his\_name, his\_ip = n  
 if ip == his\_ip:  
 # self.verbose and print(f"Sending to our client {his\_name} at {his\_ip}")  
 return True  
 subnet\_ip = ip[:-2]  
 subnet\_ap = self.con.ap\_ip[:-2]  
 if subnet\_ip == subnet\_ap:  
 # self.verbose and print(f"Sending to same subnet as our ap: {subnet\_ip}")  
 return True  
 return False  
  
 def direct\_msg(self, ip, file, payload\_string):  
 from urequests import get  
 from random import randint  
 import json  
  
 self.sequence\_number += randint(1, 1\_000)  
 payload = json.loads(payload\_string)  
 payload["direct"] = self.con.ap\_ip  
 payload["sequence"] = self.sequence\_number  
 payload\_string = json.dumps(payload)  
  
 # self.verbose and print(f"My ip addresses: {self.con.ap\_ip} {self.con.sta\_client\_ip}")  
 if not self.direct(ip):  
 # self.verbose and print(f"Can't reach {ip} directly")  
 return  
  
 for count in range(5):  
 try:  
 get\_str = f"http://{ip}/{file}?packet={payload\_string}"  
 response = get(get\_str, timeout=randint(1, 10))  
 if response.status\_code != 200:  
 response.close()  
 continue  
 if response.text[0] == "{":  
 p = json.loads(response.text)  
 self.add\_neighbor(p["origin"][0], p["ssid"], p["name"])  
 response.close()  
 return  
 except OSError as e:  
 if e.args[0] == 104: # ECONNRESET  
 # self.verbose and print(f"Connection to {ip} reset")  
 pass  
 elif e.args[0] == 113: # ECONNABORTED  
 # self.verbose and print(f"Connection to {ip} aborted")  
 pass  
 elif e.args[0] == 116: # ETIMEDOUT  
 # self.verbose and print(f"Connection to {ip} timed out")  
 pass  
 else:  
 print(f"Exception in get: {e}")  
 print\_exception(e)  
 except Exception as e:  
 print(f"Exception in get: {e}")  
 print\_exception(e)  
  
 def broadcast(self, payload\_string):  
 from urequests import get  
 self.show\_neighbors()  
 print()  
 print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
 print(f"Broadcast:")  
 for n in self.neighbors:  
 ssid, ip, name = n  
 # self.verbose and print(f"Send to {name} at {ip} ", end="")  
 # self.verbose and print(payload\_string)  
 self.direct\_msg(ip, "none", payload\_string)  
 # for n in self.whodat.values():  
 # his\_name, his\_ip = n  
 # self.verbose and print(f"Send to {his\_name} at {his\_ip} ")  
 # self.direct\_msg(his\_ip, "none", payload\_string)  
 print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
 print()  
  
 def handle\_query(self):  
 import json  
 command, payload\_string = self.query.split("=")  
 payload\_string = unquote(payload\_string)  
 if command == "packet":  
 payload = json.loads(payload\_string)  
 name = payload.get("name", "unknown")  
 ssid = payload.get("ssid", "no ssid")  
 to = payload.get("to", "unknown")  
 seq = payload.get("sequence", False)  
 if self.dup\_sequence(seq):  
 # self.verbose and print("We already saw this message:", payload\_string)  
 return  
 direct = payload.get("direct", "unknown")  
 its\_from\_ap, its\_from\_sta = payload.get("origin", ("unknown", "unknown"))  
 ttl = payload.get("time\_to\_live", 0)  
 # self.verbose and print("Query:", name, ssid, to, its\_from\_ap, its\_from\_sta)  
  
 self.add\_neighbor(its\_from\_ap, ssid, name)  
 self.add\_neighbor(its\_from\_sta, ssid, name)  
  
 if to == w.name(): # It's for me  
 self.execute(payload)  
 elif ttl > 0:  
 ttl -= 1  
 payload["time\_to\_live"] = ttl  
 payload\_string = json.dumps(payload)  
 if to == "all":  
 self.broadcast(payload\_string)  
 else:  
 # Here is where we would put the routing code  
 self.broadcast(payload\_string)  
 elif command == "text":  
 payload = json.loads(payload\_string)  
 text = payload.get("text", "")  
 who = payload.get("to", "")  
 its\_from = payload.get("from", "")  
 message = payload.get("message", "")  
 seq = payload.get("sequence", False)  
 if self.dup\_sequence(seq):  
 # self.verbose and print("We already saw this message:", payload\_string)  
 return  
 print()  
 print("Message:")  
 print(payload\_string)  
 payload = {  
 "text": True,  
 "message": message,  
 "name": its\_from,  
 "to": who,  
 "time\_to\_live": 3,  
 "direct": self.con.ap\_ip,  
 "origin": (self.con.ap\_ip, self.con.sta\_client\_ip)}  
 self.broadcast(json.dumps(payload))  
 print()  
 else:  
 self.verbose and print("Unexpected query:", self.query)  
 return None  
  
 def send\_file(self, client\_socket):  
 from ubinascii import hexlify  
 # self.verbose and print("Send file:", self.path)  
 if self.path:  
 if self.path == "/index.html":  
 style = "<style>\r\n th, td\r\n {\r\n padding: 5px;\r\n spacing: 5px;\r\n }\r\n </style>"  
 my\_info = f"<h2>{w.name()}: ap={self.con.ap\_ip} sta={self.con.sta\_client\_ip} on {self.con.who\_am\_i}</h2><p/>"  
 neighbors = "<table border=1>"  
 for n in self.neighbors:  
 ssid, ip, name = n  
 neighbors += f"\r\n <tr>\r\n <td>{name}</td>\r\n <td>{ip}</td>\r\n <td>{ssid}</td>\r\n </tr>"  
 for n in self.whodat.values():  
 his\_name, his\_ip = n  
 neighbors += f"\r\n <tr>\r\n <td>Child</td>\r\n <td>{his\_ip}</td>\r\n <td>{his\_name}</td>\r\n </tr>"  
 neighbors += "\r\n </table>"  
 station\_str = "<p/>Stations:\r\n"  
 stations = self.con.ap.status("stations")  
 for s in stations:  
 mac = hexlify(s[0]).decode()[4:]  
 station\_str += f" <br/>{mac}\r\n"  
 r, g, b = rgb[0]  
 rgb\_str = f"(r={r}, g={g}, b={b})"  
 script = """  
 <script>  
 sequence\_number = Math.round(Math.random() \* 10000);  
 function sendTextToServer() {  
 sequence\_number++;  
 const who = document.getElementById("who").value;  
 const from = document.getElementById("from").value;  
 const textToSend = document.getElementById("textInput").value;  
 packet = { "message": encodeURIComponent(textToSend), "to": who, "from": from, "sequence": sequence\_number }  
 const url = "/text?text=" + JSON.stringify(packet);  
 fetch(url, { method: 'GET', })  
 .then(response => {  
 if (!response.ok) { throw new Error("HTTP error! status: " + response.status); }  
 return response.text();  
 })  
 .then(data => {  
 document.getElementById("responseFromServer").textContent = "Server Response: " + data;  
 document.getElementById("textInput").value = "";  
 })  
 .catch(error => {  
 console.error("Error sending text:", error);  
 document.getElementById("responseFromServer").textContent = "Error: " + error.message;  
 });  
 }  
 </script>  
"""  
 file = f"""  
<html>  
 <head>  
 <title>Mesh</title>  
 {style}  
 {script}  
 </head>  
 <body>  
 {my\_info}  
 <h3>Mesh: {w.name()}'s neighbors:</h3>  
 {neighbors}  
 {station\_str}  
 <h3>LED:</h3>  
 RGB is {rgb\_str}  
 <p/>  
 <form id="textForm">  
 <div>  
 <label for="who">Send to:</label>  
 <input type="text" id="who" name="text" placeholder="all">  
 <br/>  
 <label for="from">Send from:</label>  
 <input type="text" id="from" name="text" placeholder="Joe">  
 <br/>  
 <label for="textInput">Enter text to send:</label>  
 <input type="text" id="textInput" name="text" required>  
 </div>  
 <button type="button" onclick="sendTextToServer()">Send Text</button>  
 </form>  
 <div id="responseFromServer"></div>  
 </body>  
</html>  
"""  
 client\_socket.write(f"HTTP/1.1 200 OK\r\nContent-Length: {len(file)}\r\nContent-Type: text/html\r\n\r\n")  
 client\_socket.write(file)  
 elif self.path == "/text":  
 file = "Received text message"  
 client\_socket.write(f"HTTP/1.1 200 OK\r\nContent-Length: {len(file)}\r\nContent-Type: text/html\r\n\r\n")  
 client\_socket.write(file)  
 elif self.path == "/none":  
 import json  
 payload = {  
 "name": w.name(),  
 "ssid": self.con.who\_am\_i,  
 "direct": self.con.ap\_ip,  
 "origin": (self.con.ap\_ip, self.con.sta\_client\_ip)}  
 file = json.dumps(payload)  
 client\_socket.write(f"HTTP/1.1 200 OK\r\nContent-Length: {len(file)}\r\nContent-Type: text/html\r\n\r\n")  
 client\_socket.write(file)  
 else:  
 with open(self.path, "rb") as f:  
 file = f.read()  
 client\_socket.write(f"HTTP/1.1 200 OK\r\nContent-Length: {len(file)}\r\nContent-Type: text/html\r\n\r\n")  
 client\_socket.write(file)  
  
 def decode\_path(self, req):  
 if not req:  
 return "/"  
  
 self.path = None  
 cmd, headers = req.split("\r\n", 1)  
 if cmd:  
 s = cmd.split(" HTTP/")  
 cmd = s[0]  
 self.query = ""  
 r = cmd.find("?")  
 if r > 0:  
 self.query = cmd[r+1:]  
 cmd = cmd[:r]  
  
 self.method, self.path = cmd.split(" ")  
  
 if self.path == "/":  
 self.path = "/index.html"  
  
 return self.path  
  
 def req\_handler(self, client\_socket):  
 try:  
 req = client\_socket.read()  
 if req:  
 self.decode\_path(req.decode("utf-8"))  
 # self.verbose and print("Decoded path:", self.path)  
 self.send\_file(client\_socket)  
 client\_socket.close()  
  
 if self.query != "":  
 self.handle\_query()  
 return  
 except OSError as e:  
 if e.args[0] == 128: # ENOTCONN  
 pass  
 except Exception as e:  
 self.verbose and print("Err:", e)  
 print\_exception(e)  
 client\_socket.close()  
  
 def client\_handler(self, srv):  
 try:  
 client\_socket, addr = srv.accept()  
 except OSError as e:  
 if e.args[0] == 23: # Too many open files: we forgot to close a socket somewhere  
 pass  
 # self.verbose and print("Serving:", addr[0])  
 self.whodat[addr[0]] = ("unknown", addr[0])  
 client\_socket.setblocking(False)  
 client\_socket.setsockopt(SOL\_SOCKET, 20, self.req\_handler)  
  
def main():  
 rgb[0] = (64, 0, 0)  
 rgb.write()  
 sleep(10) # During development, so the chip allows file transfers  
 con = Connect()  
 con.verbose = True  
 con.reconnect()  
  
 rgb[0] = (64, 64, 0)  
 rgb.write()  
 mesh = Mesh()  
 mesh.con = con  
 mesh.verbose = True  
 if con.sta\_host\_ip:  
 # mesh.verbose and print(f"{w.name()}: Connected to {con.ssid} at {con.sta\_host\_ip}")  
 mesh.add\_neighbor(con.sta\_host\_ip, con.ssid, "unknown")  
  
 rgb[0] = (0, 64, 0)  
 rgb.write()  
  
 while True:  
 sleep(3 \* 60)  
 print("==============================================================")  
 mesh.show\_neighbors()  
 mesh.colors()  
 if con.sta and con.sta.isconnected() == False:  
 con.reconnect()  
 if con.sta\_host\_ip:  
 # mesh.verbose and print(f"{w.name()}: Connected to {con.ssid} at {con.sta\_host\_ip}")  
 mesh.add\_neighbor(con.sta\_host\_ip, con.ssid, "unknown")  
  
main()

The execute() method handles special message types the node can receive. One is a message that sets the colors and brightness of the RGB LED. This is handy for debugging, as not all nodes will have a computer attached for serial output.

It also handles printing out messages, and the reply to the greet() packet. The latter calls who\_name() to associate the name with the IP addresses we got from the connecting node.

The whole point of a mesh network is to communicate with nodes we can’t connect to directly. That’s where the direct() method comes in. It looks at all the IP addresses we know about and returns False if the address given is unreachable.

We finally get to direct\_msg(), the heart of the code. This assigns sequence numbers and inserts them into the payload, and then tries 5 times to send the packet, stopping at the first success. If it gets a reply, it adds the node as a neighbor (it lets add\_neighbor() handle duplicates).

The broadcast() method is simple. It goes through the neighbor list and sends the packet to each one.

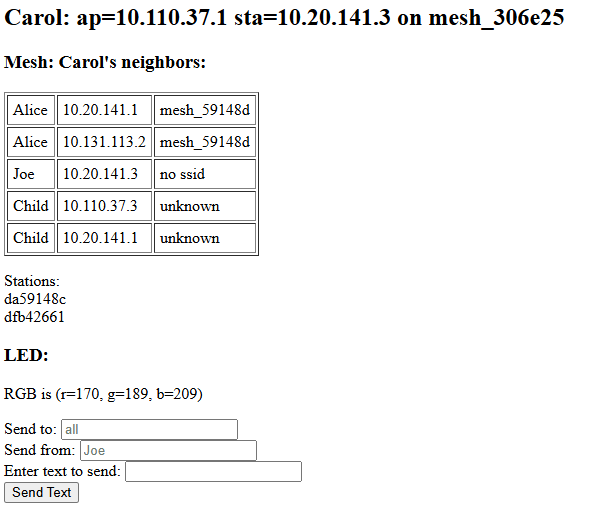
We use the HTTP GET method for packets, which means some packets might have a question mark (a ‘query’) with commands in it. If the command is “packet”, we collect the neighbor information, store it, and then check to see if the packet is for us. If it was, we call execute(). Otherwise we decrement the time\_to\_live number and broadcast the packet to everyone we know.

If the command was “text”, then we got it from a web browser (more on that in a bit). We repackage the data (adding our IP addresses, time\_to\_live, etc.) and broadcast it.

The part of the HTTP packet ahead of the query is the path. This is normally the file the server will read and send to the browser, and indeed we do that if asked. But we also handle “fake” files /text and /none. The /text file just acknowledges that we got a text message, so the web browser that sent it knows we got it. The /none file answers the greet message by returning our name, ssid, and IP addresses.

The bulk of the code in send\_file() creates the /index.html file to send to any web browser that connects with us. This file prints out node information and includes a form for sending messages.

The form looks like this:



I named my nodes Alice, Bob, and Carol (the next one will be Ted).

We can see that Carol has Alice as a neighbor, but Carol can’t see Bob. I sent a message to Bob, telling Bob my browser was Joe, and Carol sent it to Alice who then sent it to Bob. The meshing worked (finally, after three weeks of coding and debugging).

The main() method sets the RGB LED to red, then connects to anyone it can, sets the RGB LED to yellow while it initializes the Mesh class, and then sets it to green before looping to do something every 3 minutes just to show that the mesh is meshing.

The something that it does is just to print out the neighbor list on the console (if there is one) and then broadcast a message to set everyone’s LED to a random color. Again, just to show the system is up and running.

When setting up a node out somewhere here on the farm, it is nice not to need a laptop to see what’s going on. Watching the LED change color tells me everything is working.

It is now easy to add features. You can collect temperature, humidity, and other weather data and send it out to the mesh. You can turn lights on and off, or tell when the well pump is running (you all have those right? Or is it just me?)

Each node has 150 megabits of bandwidth to play with (15 megabytes per second). Our typical packet length is about 100 bytes, so we could theoretically send 150,000 packets per second. We send each of our packets 3 times per node on the assumption all nodes are within 3 hops of one another, but we could easily change this, or make it an option in the web form. But the network would have to get pretty big before we ran out of bandwidth, even though we waste it by using the simple design.

[Programming](https://medium.com/tag/programming?source=post_page-----46d447caaa97---------------------------------------)

[Education](https://medium.com/tag/education?source=post_page-----46d447caaa97---------------------------------------)

[Software Development](https://medium.com/tag/software-development?source=post_page-----46d447caaa97---------------------------------------)

[Python](https://medium.com/tag/python?source=post_page-----46d447caaa97---------------------------------------)