Enterprise Network Design Report

Cover Page

Title: Enterprise Network Design Simulation Using Cisco Packet Tracer For Citywide Street Lamp coverage and issues
Course: IT Platform Project
Professor: Prof. Dr. Rand Kouatly
University: University of Europe
Semester: Winter 2024/25

Table of Contents

---

1. Introduction

This report presents the design, configuration, and testing of an enterprise computer network using Cisco Packet Tracer that is mainly focused around the citywide usage of electricity and possible causes of errors, helping dispatching teams and being able to understand issues regarding city planning with the usage of street lamps. The project aims to simulate a network infrastructure suitable for a business environment with workstations, servers, mobile devices, laptops, and network components.

2. Network Design Overview

The network consists of multiple sub-networks connected through routers and switches. The WAN network uses a Class A IP address, while the local network operates on Class C IP addresses. Devices such as workstations, laptops, and mobile phones are configured via DHCP, while servers have static IP addresses. Two web servers, Two DNS server, and

DHCP server in the form of a firewall are included to provide essential services.

To mainly explain their usages, and what they do in simple terms:

- **WEB SERVER**
  web-servers are used to serve websites/functionalities that are generally used for delivering web content to requesting clients (computers), since the usage of Cisco packet tracer without “RealHTTP” (which is a way to make packet tracer communicate with your physical actual computer, allowing you to code freely without many limitations) was not our first approach as we wanted to be able to maintain everything inside of the packet tracer files, we have chosen http-server and http-client as templates to be able to serve Website. Kindly check CISCO PACKET TRACER PYTHON API in order to see allowed external packages for this project.

- **DNS SERVER**
  they are used to be able to communicate domain names (ex:google.com) into ip addresses (ex: IP class C 192.168.1.1). Workstations generally use domain names instead of IP addresses in order to visit websites and they are easier to remember. A DNS server can have multiple “websites” configured to it, A workstation (will be referenced as computer from now on for ease of explanations) needs to have a DNS server address configured to it to access its translation properties.

- **DHCP SERVER**
  main point of a DHCP server is assigning IP addresses to interfaces and devices. an actual device configured specifically for DHCP will be able to assign ip addresses in a range and subnet the person has specified. But in the case of ASA5506-X it is mainly unusable because of ASA5506-X not being able to assign IP addresses outside of the range of the subnetwork the interface (or VLAN for ease of use) is in.

  Note: As you will see further ahead, the Option 6 (assigning dns addresses) of DHCP in ASA5506-X is disabled within cisco packet tracer, limiting the ability to assign multiple DNS addresses to the same VLAN or interface.

Sub-networks:

- **Office Sub-network 1:** Workstations, laptops, and a wireless access point; The main point of usage of this subnetwork is to be able to access reports/create reports about the subject of citywide streetlamps, making sure that the access to light is not limited and can be fixed if there are any errors

- **Office Sub-network 2:** Workstations, laptops and a wireless access point; the main usage of the subnetwork given is to understand the city wide structure and make use of Motion sensors on the street lamps, in order to understand bottlenecks inside the city which might cause issues later on

- **Sub-network 3:** Mainly used for IOT devices, Each device has its own DHCP address, and communicates with the WEB server and gives information about the sensors, then this information gets parsed into parts and viewed in a webpage.

- **DMZ Zone :** Hosting the web server and DNS server for outside network. It allows citizens to be able to make issue reports, which will then be checked by the offices and will find any issues regarding the area

- **Server Zone :** Although this might seem similar to a DMZ zone, it is not, by definition a DMZ zone is a demilitarized zone, The purpose of a DMZ zone is to provide extra security while this zone is configured to help ease of use inside the private network, Assigning DNS and a WebServer which is created in the purpose of collecting data from the private network of IoT devices

---

3. Equipment Selection

The following network devices and components were selected based on compatibility and project requirements:

- **Workstations:** 10
- **Laptops:** 10
- **Mobile Devices:** 2
- **Switches:** 4 (2960-4TT)
- **Routers:** 5 (5x 2911 each with custom module for supporting fiber)
- **Servers:** 4 (Web, DHCP, DNS)
- **Wireless Access Points:** 2
- **Cables:** Fiber optic for router connections, Ethernet for device-to-switch connections.

---

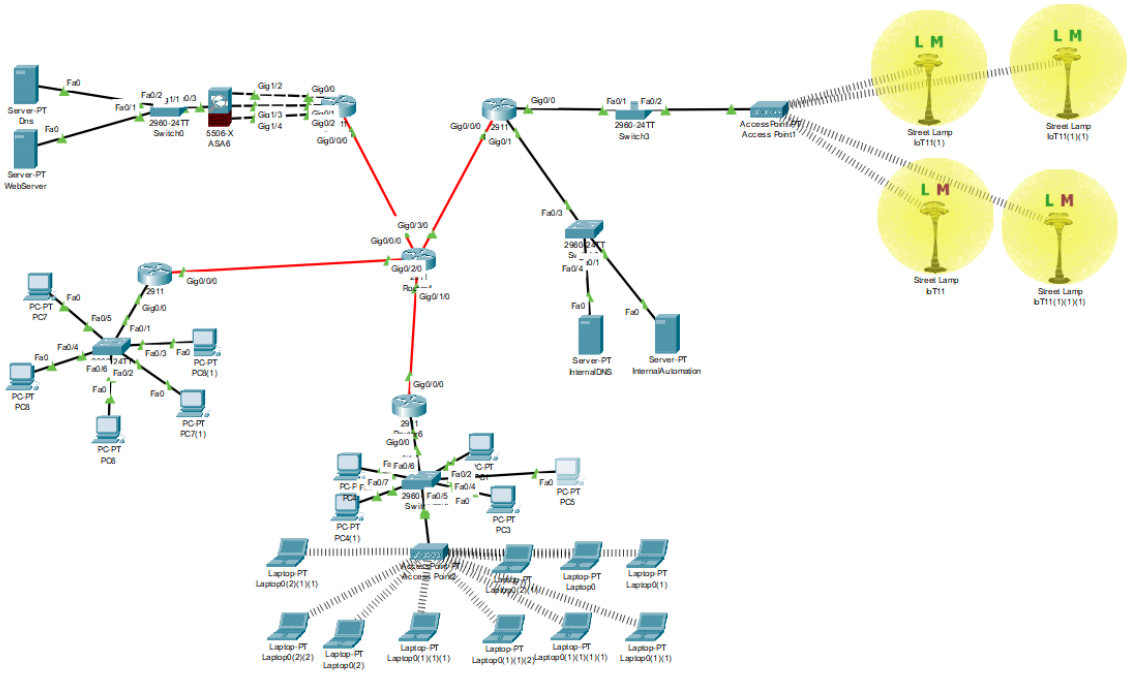
4. Network Topology Design

Topology Overview

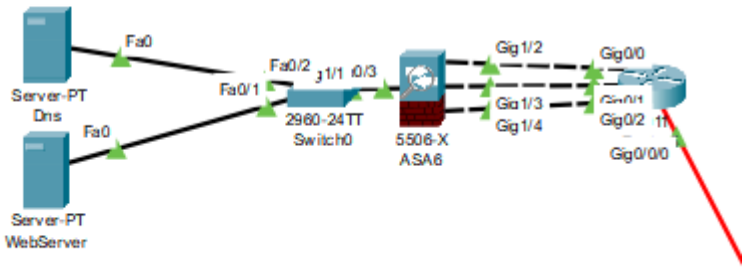
The network topology includes:

- Star topology for local networks.
- Router interconnections using fiber optic cables.

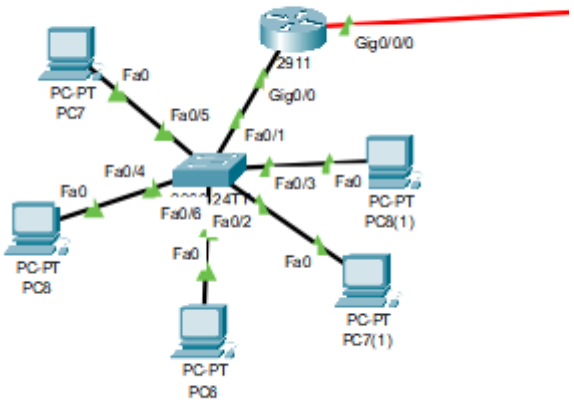
Complete Diagram:



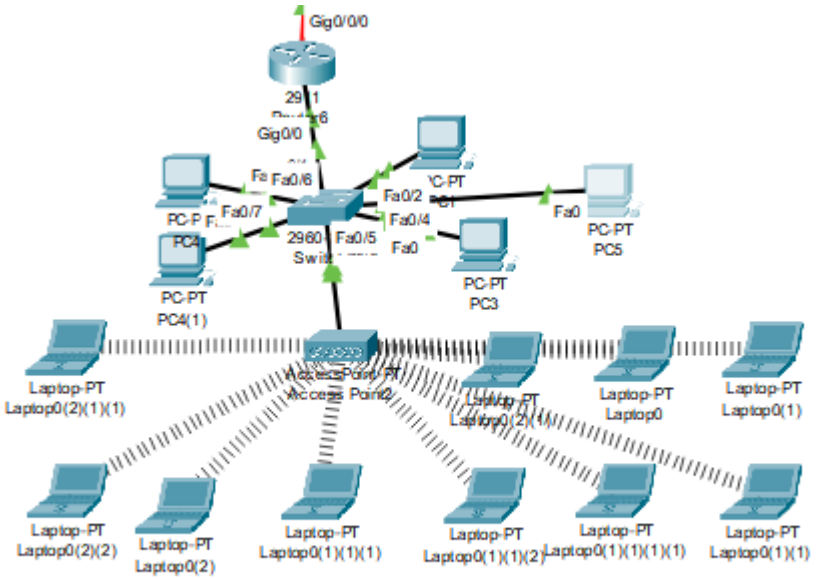
DMZ Zone



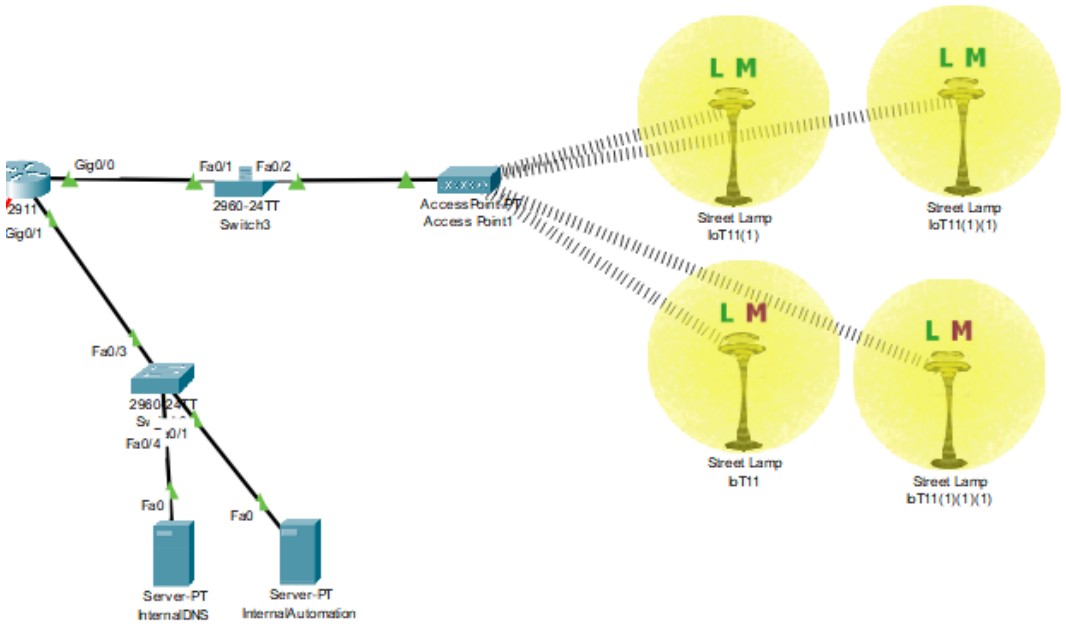
Office Zone2:



Office zone 1:



IoT and Server Zone



Network Flow

Network is connected Via OSPF, and in the following section the configurations will be explained for each device:

Router WAN:

```
!
interface GigabitEthernet0/0/0
 ip address 10.0.3.2 255.255.255.0 (Assigns an ip address to the interface)
!
interface GigabitEthernet0/1/0
 ip address 10.0.1.2 255.255.255.0 (Assigns IP to Interface)
!
interface GigabitEthernet0/2/0
 ip address 10.0.2.2 255.255.255.0 (Assign IP to Interface)
!
interface GigabitEthernet0/3/0
 ip address 10.0.4.2 255.255.255.0 (Assign IP to Interface)
!
router ospf 1 (Creates an OSPF “job”)
 log-adjacency-changes (make it check the adjacent)
 network 10.0.1.0 0.0.0.255 area 0 (add ip address range from 10.0.1.0 to 10.0.1.255)
 network 10.0.2.0 0.0.0.255 area 0
 network 10.0.3.0 0.0.0.255 area 0
 network 10.0.0.0 0.255.255.255 area 0
```

Router 6:

```
!
interface GigabitEthernet0/0
ip address 192.168.1.129 255.255.255.128
ip helper-address 192.168.1.2
duplex auto
 speed auto
!
```

```
!
interface GigabitEthernet0/0/0
 ip address 10.0.1.1 255.255.255.0
!
router ospf 1
 log-adjacency-changes
 network 10.0.1.0 0.0.0.255 area 0
 network 192.168.1.128 0.0.0.127 area 0
```

ROUTER OFFICE 2:

```
!
interface GigabitEthernet0/0
 ip address 192.168.2.129 255.255.255.128
 ip helper-address 192.168.2.2
 duplex auto
 speed auto
!
interface GigabitEthernet0/0/0
 ip address 10.0.2.1 255.255.255.0
!
router ospf 1
 log-adjacency-changes
 network 10.0.2.0 0.0.0.255 area 0
 network 192.168.2.128 0.0.0.127 area 0
```

ROUTER IOT:

```
!

interface GigabitEthernet0/0

 ip address 192.168.3.129 255.255.255.128
```

```
 ip helper-address 192.168.3.2
 duplex auto
 speed auto
!
interface GigabitEthernet0/1
 ip address 192.168.10.1 255.255.255.0
 duplex auto
 speed auto

interface GigabitEthernet0/0/0
 ip address 10.0.4.1 255.255.255.0
!

router ospf 1
 log-adjacency-changes
 network 10.0.4.0 0.0.0.255 area 0
 network 192.168.10.0 0.0.0.255 area 0
 network 192.168.3.128 0.0.0.127 area 0
```

FIREWALL:

```
interface GigabitEthernet1/1
 nameif inside
 security-level 0
 ip address 192.168.4.1 255.255.255.0

interface GigabitEthernet1/2
```

```
 nameif outside
 security-level 100
 ip address 192.168.1.2 255.255.255.0
!
interface GigabitEthernet1/3
 nameif vlan3
 security-level 100
 ip address 192.168.2.2 255.255.255.0
!
interface GigabitEthernet1/4
 nameif vlan4
 security-level 100
 ip address 192.168.3.2 255.255.255.0
!


access-list inside-traffic extended permit icmp any 192.168.4.0 255.255.255.0
access-list inside-traffic extended permit icmp 192.168.4.0 255.255.255.0 any
access-list inside-traffic extended permit tcp any 192.168.4.0 255.255.255.0 eq www
access-list inside-traffic extended permit tcp 192.168.4.0 255.255.255.0 any eq www
access-list inside-traffic extended permit icmp 192.168.4.0 255.255.255.0 192.168.2.0
255.255.255.0
access-list inside-traffic extended permit tcp 192.168.4.0 255.255.255.0 192.168.2.0
255.255.255.0 eq www
access-list inside-traffic extended permit icmp 192.168.4.0 255.255.255.0 192.168.3.0
255.255.255.0
access-list inside-traffic extended permit tcp 192.168.4.0 255.255.255.0 192.168.3.0
255.255.255.0 eq www
```

```
access-list inside-traffic extended permit icmp 192.168.2.0 255.255.255.0 192.168.4.0
255.255.255.0

access-list inside-traffic extended permit tcp 192.168.2.0 255.255.255.0 192.168.4.0
255.255.255.0 eq www

access-list inside-traffic extended permit icmp 192.168.3.0 255.255.255.0 192.168.4.0
255.255.255.0

access-list inside-traffic extended permit tcp 192.168.3.0 255.255.255.0 192.168.4.0
255.255.255.0 eq www

!


!


access-group inside-traffic in interface inside

!

dhcpd dns 192.168.10.3

!

dhcpd address 192.168.1.150-192.168.1.200 outside

dhcpd dns 192.168.10.3 interface outside

dhcpd enable outside

dhcpd address 192.168.2.150-192.168.2.200 vlan3

dhcpd dns 192.168.10.4 interface vlan3

dhcpd option 3 ip 192.168.2.129 interface vlan3

dhcpd enable vlan3

!

dhcpd address 192.168.3.150-192.168.3.200 vlan4

dhcpd option 3 ip 192.168.3.129 interface vlan4

dhcpd enable vlan4

!
```

```
!
router ospf 1
 router-id 1.1.1.1
 log-adjacency-changes
 network 192.168.1.0 255.255.255.0 area 0
 network 192.168.2.0 255.255.255.0 area 0
 network 192.168.3.0 255.255.255.0 area 0
 network 192.168.4.0 255.255.255.0 area 0
!
```

---

5. IP Addressing Scheme

WAN Network:

- Class A IP Range: 10.x.x.x

  10.0.1.0 (OfficeRouter1-WAN)

  10.2.2.0 (OfficeRouter2-WAN)

  10.3.3.0 (OfficeRouter3-WAN)

  10.4.4.0 (routerDMZ-WAN)

Local Networks (Class C):

- **Office 1:** 192.168.3.x/24
- **Office 2:** 192.168.2.x/24
- **Office 2:** 192.168.1.x/24
- **DMZ Zone:** 192.168.4.x/24

Static IPs for Servers:

- Web Server: 192.168.10.3

- DNS Server: 192.168.10.4

---

6. Device Configurations

- **Routers:** Configured with appropriate IPs, routing tables (RIP protocol used for ease of routing), and static routes.
- **Servers:** Static IPs assigned and services installed.
- **Workstations and Laptops:** Configured via DHCP.

Configurations Details: Router Configurations:

- Router WAN: IP Address 10.0.1.2, OSPF routing enabled.
- Router WAN: IP Address 10.0.2.2, OSPF routing enabled.
- Router WAN: IP Address 10.0.3.2, OSPF routing enabled.
- Router WAN: IP Address 10.0.4.2, OSPF routing enabled.
- Router DMZ: IP Address 192.168.4.1, configured with OSPF.
- Router Office 1: IP Address 192.168.1.129, configured with OSPF.
- Router Office 1/ WAN router: IP Address 10.0.1.1, configured with OSPF.
- Router Office 2: IP Address 192.168.2.129, configured with OSPF.
- Router Office 2/ WAN router: IP Address 10.0.2.1, configured with OSPF.
- Router IoT:IP address 192.168.3.129, Configured via OSPF
- Router IoT/WAN: IP Address 10.0.4.1 Configured with OSPF

Switch Configurations:

- All switches use default VLAN 1.
- Trunk ports enabled between switches and routers.

Server Configurations:

- DHCP Server:Firewall IP 192.168.1.1, configured with DHCP pool
  192.168.2.1 configured with DHCP pool.

  (1)

  serverPool: 192.168.1.129 Default Gateway.

  Start IP address: 192.168.1.150-200

  (2)

  serverPool1: 192.168.2.1 Default Gateway.

  Start IP address: 192.168.2.150-200

  (3)

serverPool2: 192.168.3.1 Default Gateway.

Start IP address: 192.168.3.150-200

- DNS Server: IP 192.168.10.4, domain name resolution enabled.
    - domain name:autolocal.com
- Web Server: IP 192.168.10.3, web service running.
    - uses configured python scriptfor loading maing page.

Workstations and Laptops:

- IP assignment via DHCP.
    - uses routing OSPF to get access to the DHCP server
- Connectivity tested successfully with ping and browser access.
    - No problems found.

Wireless Access Points:

- SSIDs configured for internal network access.
- WPA2 security enabled.

---

7. Testing and Validation

- **Connectivity Tests:** Ping between all devices.
- **Web Server Access:** Browser access using the configured domain name.
- **DNS Tests:** Lookup tests for name resolution.
- **DHCP Tests:** IP assignment validation.

---

8. Project Portfolio

Hardware Recommendations

- **Servers:** High-performance rack servers.
- **Switches and Routers:** Managed and enterprise-grade devices.
- **Firewalls:** Hardware firewalls for DMZ protection.

Software Recommendations

- Cisco IOS for routers and switches.
- Ubuntu OS for server management.

Support Services

- Service Catalog: Network maintenance and upgrades.
    - Recommended to use one 2911 and switch Routers every few years and make controls on them to update to the current tech stack
    - Recommended to update OS of servers and devices to make sure there are no security concerns.
    - Recommended to use Fiber between routers
    - Recommended to use Modules to extend the Routers capabilities for Fiber
    - Recommended to use Switches connected between each router and workstation/laptop/computer/mobile

- Service Pipeline: Planned expansions.
    - In future in case more IOT devices are introduced, should make balance arrangements around the amount of loads received
- Retired Services: Legacy network equipment replacement.
    - Every few years we get changes

---

9. Conclusion

9. Conclusion (Expanded)

The enterprise network simulation successfully meets the specified design and operational requirements, ensuring that the network is both efficient and secure for business use. Throughout the project, careful consideration was given to the structure, configuration, and scalability of the network, ensuring that it is adaptable for future growth while supporting current business operations.

One of the key achievements of this simulation was the successful design of a network topology that incorporates both wired and wireless connections, making it versatile for different device types, including workstations, laptops, and mobile devices. The use of routers, switches, and wireless access points in strategic locations guarantees that all devices are interconnected with high availability and minimal latency, contributing to an efficient workflow within the organization. The network's fault tolerance features, such as RIP routing and redundant connections, ensure that critical services remain uninterrupted even in the event of a failure or increased load.

The IP addressing scheme was carefully planned, using a mix of static and dynamic IP addresses to effectively manage the network. The Class A IP range for the WAN network allows for the efficient allocation of addresses across different subnets, while the Class C subnetting for the local networks ensures proper segmentation for security and performance.

The DHCP server configuration on firewall ensures that devices are automatically assigned IP addresses, streamlining the process of network device management and reducing human error.

In terms of security, the network design incorporates essential protective measures, such as the use of WPA2 encryption on wireless access points and strict access control policies on routers and switches. The separation of the DMZ zone from the internal network helps mitigate the risk of external attacks reaching sensitive internal services, and the use of static IP addresses for critical services such as the web, DNS, and DHCP servers further enhances the security by ensuring consistent access.

Testing and validation of the network were essential to ensure that all configurations were functioning as expected. Connectivity tests, including pinging between devices and validating web server access, confirmed that the network was stable and responsive. DNS and DHCP functionality was also thoroughly tested to ensure seamless name resolution and IP address assignment. These tests proved the reliability of the network infrastructure, and any minor issues encountered were swiftly addressed, improving the overall performance.

This simulation has provided valuable insights into the complexities of designing and configuring an enterprise network. It highlighted the importance of scalability, security, and ease of maintenance. The lessons learned during this process, such as the configuration of RIP routing and the management of IP address pools, are directly applicable to real-world networking scenarios. The use of Cisco Packet Tracer as a simulation tool allowed for hands-on practice and experimentation, giving a deeper understanding of networking protocols and configurations without the need for physical equipment.

In conclusion, this project has demonstrated the ability to design and simulate a robust and efficient enterprise network capable of supporting a variety of business functions. The network is prepared to handle the needs of an organization both now and in the future, with scalability built into its design. It has also provided valuable experience in troubleshooting, configuring, and validating network devices, which will be essential in future real-world projects. As businesses continue to grow and adopt more advanced technologies, the principles and methodologies used in this simulation will remain relevant, ensuring that organizations can maintain secure and reliable network infrastructures.

---

10. References

- Cisco Packet Tracer Documentation

Appendices

- Network Configurations
- Packet Tracer Simulation File (attached)

```
# First, we need to get some tools that will help us work with messages and data
from udp import UDPSocket  # This is a tool to send and receive messages over the internet.
from http import HTTPServer  # This tool helps us create a server that can send web pages.
from time import sleep, time  # These tools help us pause the program and get the current time.

# This is a list that will hold any problems or issues reported by the street lights.
reported_issues = []  # It's like a list where we write down things that are not working right.

# This is a special place where we keep data for each street light. Each light will have its own data.
ip_data = {}  # Think of it like a big notebook where we write down info for each street light.

# This function runs when a message is received from a street light.
def onUDPReceive(ip, port, data):
    formatted_data = process_street_lamp_data(ip, data)  # We process the data to make it easier to understand.
    if formatted_data:
        store_data(ip, formatted_data)  # If the data is good, we store it in our notebook.

# This function is for when we get messages about problems from the street lights.
def onUDPReceiveIssue(ip, port, data):
    print("Received from " + ip + ":" + str(port) + ": " + data)  # We print out the message to see what's happening.

    try:
        # Try to change the message into something the program can understand (JSON).
        issues_data = json.loads(data)  # This turns the message into a list of problems.

        # If it works, we update our list of reported issues.
        global reported_issues
        reported_issues = issues_data  # Replace the old list with the new list of problems.

        print("Updated issues: " + str(reported_issues))  # We print the new list to see what changed.

    except Exception as e:
        print("Error parsing received data: " + str(e))  # If there's an error, we say what went wrong.

# This function takes the data from a street light and changes it into something we can read and use.
def process_street_lamp_data(ip, data):
    try:
        # Split the data into smaller pieces using "^" as the separator.
        data_parts = data.split("^")
        formatted_data = {}  # We create a new place to store the clean data.
```

```
        formatted_data["ip"] = ip  # We save the IP address of the street light.

        light_level = 0  # We start by assuming the light is off.
        motion_level = 0  # We start by assuming there is no movement.

        # We go through each piece of the data.
        for part in data_parts:
            key_value = part.split(",")  # Split each piece into a key and a value.
            if key_value[0] == "light":  # If the data is about the light...
                light_level = float(key_value[1])  # Turn the light level into a number.
                formatted_data["light_level"] = "off" if light_level > 0 else "on"  # If the light level is more than 0, the light is off, otherwise it's on.
                formatted_data["light_time"] = 0 if light_level > 0 else calculate_light_on_time(ip, motion_level, light_level)  # How long has the light been on?
            elif key_value[0] == "motion":  # If the data is about motion...
                motion_level = float(key_value[1])  # Turn the motion level into a number.
                formatted_data["motion_state"] = "up" if motion_level > 0 else "down"  # If there's movement, we say "up," if not, we say "down."

        return formatted_data  # After cleaning up the data, we return it.
    except Exception as e:
        print("Error processing data: " + str(e))  # If something went wrong, we say what the error was.
        return None  # Return nothing if there's an error.

# This function figures out how long the light has been on.
def calculate_light_on_time(ip, motion_level, light_level):
    global ip_data  # We will use the big notebook where we store the street light info.

    current_time = time()  # Get the current time (like looking at a clock).

    # If we don't have data for this street light, we create new data for it.
    if ip not in ip_data:
        ip_data[ip] = {
            "last_updated": current_time,
            "total_time": 0,
            "is_on": False,  # At the beginning, the light is off.
        }

    # Check if the light should be on or off.
    light_is_on = light_level <= 0  # If the light level is less than or equal to 0, the light is on.

    if light_is_on:
        # If the light is on and it wasn't on before, we reset the timer.
        if not ip_data[ip]["is_on"]:
            ip_data[ip]["last_updated"] = current_time  # Set the start time.
            ip_data[ip]["is_on"] = True  # Mark that the light is on now.
        # Calculate how long the light has been on.
```

```
        elapsed_time = current_time - ip_data[ip]["last_updated"]
        ip_data[ip]["total_time"] += elapsed_time  # Add the elapsed time to the total time the
light has been on.
        ip_data[ip]["last_updated"] = current_time  # Update the time the light was last checked.
    else:
        # If the light is off, we reset the time it was on.
        if ip_data[ip]["is_on"]:
            ip_data[ip]["is_on"] = False  # Mark that the light is off now.
            ip_data[ip]["total_time"] = 0  # Reset the time the light was on.
        ip_data[ip]["last_updated"] = current_time  # Update the time the light was last checked.

    # Return how long the light has been on, rounded to two decimal places.
    return round(ip_data[ip]["total_time"], 2)

# This function stores the data we processed for a street light.
def store_data(ip, data):
    if ip in ip_data:
        ip_data[ip].update(data)  # If we already have data for this street light, we update it.
    else:
        ip_data[ip] = data  # If we don't have data for this street light, we add it to the notebook.

# This function is called when someone visits the "/issues" page on the website.
def onRouteIssues(url, response):
    print("Request for /issues")

    # We create the HTML page to show the list of reported issues.
    issues_html = """
    <html>
        <head><title>Reported Issues</title></head>
        <body>
            <h1>Reported Issues</h1>
            <table border="1">
                <tr>
                    <th>Issue</th>
                </tr>
    """
    # We go through each issue and add it to the table.
    for issue in reported_issues:
        issues_html += "<tr><td>" + issue + "</td></tr>"

    issues_html += """
            </table>
        </body>
    </html>
    """
    response.setContentType("text/html")  # We tell the server that this is an HTML page.
    response.send(issues_html)  # We send the HTML page as the response.
```

```
# This function is called when someone visits the "/data" page on the website.
def onRouteData(url, response):
    print("Request for /data")

    # We create the HTML page to show the street light data in a table.
    table_html = """
    <html>
        <head>
            <title>Street Lamp Data</title>
            <style>
                body {
                    font-family: Arial, sans-serif;
                    background-color: #1e1e1e;
                    color: #f0f0f0;
                    margin: 0;
                    padding: 0;
                    display: flex;
                    justify-content: center;
                    align-items: center;
                    height: 100vh;
                }
                h1 {
                    text-align: center;
                    color: #e67e22;
                    font-size: 2.5em;
                    margin-bottom: 20px;
                }
                table {
                    width: 80%;
                    margin: 20px auto;
                    border-collapse: collapse;
                    border-radius: 10px;
                    overflow: hidden;
                }
                th, td {
                    padding: 10px;
                    text-align: center;
                    border: 1px solid #444;
                }
                th {
                    background-color: #2c3e50;
                    color: #ecf0f1;
                }
                td {
                    background-color: #34495e;
                    color: #ecf0f1;
                }
                tr:hover {
```

```
            background-color: #2980b9;
        }
        .container {
            width: 90%;
            max-width: 1200px;
            margin: 0 auto;
        }
        .footer {
            position: fixed;
            bottom: 0;
            width: 100%;
            background-color: #2c3e50;
            color: #ecf0f1;
            text-align: center;
            padding: 10px;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1>Street Lamp Data</h1>
        <table>
            <tr>
                <th>IP Address</th>
                <th>Motion Sensor State</th>
                <th>Total Time the Light Has Been On For (s)</th>
            </tr>
"""
# We go through each street light's data and add it to the table.
for ip, data in ip_data.items():
    table_html += "<tr>"
    table_html += "<td>" + ip + "</td>"
    table_html += "<td>" + data.get("motion_state", "") + "</td>"
    table_html += "<td>" + str(data.get("light_time", "")) + "</td>"
    table_html += "</tr>"

table_html += """
        </table>
    </div>
    <div class="footer">
        <p>Street Lamp Monitoring System</p>
    </div>
</body>
</html>
"""
response.setContentType("text/html")  # Tell the server we are sending an HTML page.
response.send(table_html)  # Send the HTML page.
```

WEBSERVER 2

```
import json  # Import a module that helps us work with text in a special format called JSON
from udp import UDPSocket  # Import a tool that helps us send messages over a network using UDP
from http import HTTPServer  # Import a tool to create a website that can show pages and accept messages

# Global variable to store reported issues
reported_issues = []  # A list that will hold all the problems people report

# Function to send reported issues to another server using UDP
def send_issues_to_server(issues):
    server_ip = "192.168.10.3"  # The special address where we will send the issues (like an email address for computers)
    server_port = 2121  # The special number used to send messages to the right place on the server

    # Convert the list of issues to a JSON string so it can be sent over the internet
    issues_json = json.dumps(issues)  # Turning our list into a format computers can understand

    # Create a UDP socket, which is like a digital envelope to send messages
    udp = UDPSocket()

    # Send the message to the server using UDP
    udp.send(server_ip, server_port, issues_json)  # Sends the message with issues to the server
    print("Reported issues sent to the destination server.")  # Let us know the message was sent
    print(issues_json)  # Show us what we sent

# HTTP route handler for /report - displays the issue submission form
def onRouteReport(url, response):
    print("Request for /report")  # Prints out whenever someone asks for the report page

    # HTML form to submit issues with a modern, dark-themed design
    form_html = """
<html>
    <head>
        <title>Report an Issue</title>
        <style>
            body {
                font-family: Arial, sans-serif;  # Makes the text look nice
```

```
    background-color: #1e1e1e;  # Dark background color
    color: #f0f0f0;  # Light color for the text
    margin: 0;  # No extra space on the sides
    padding: 0;  # No extra space on the top or bottom
    display: flex;  # Makes the content nicely organized in rows or columns
    justify-content: center;  # Centers the content
    align-items: center;  # Centers the content vertically
    height: 100vh;  # Takes up the whole screen
    text-align: center;  # Centers the text
    flex-direction: column;  # Stack items on top of each other
}
h1 {
    color: #e67e22;  # Bright orange color for the title
    font-size: 2.5em;  # Big text for the title
    margin-bottom: 20px;  # Space below the title
    text-align: center;  # Centers the title text
}
form {
    background-color: #34495e;  # Dark blue background for the form
    padding: 20px;  # Space inside the form
    border-radius: 10px;  # Round corners for the form
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.3);  # Makes the form look like it's floating
    width: 100%;  # Full width of the screen
    max-width: 600px;  # But never bigger than 600px
}
label {
    font-size: 1.2em;  # Makes the text a little bigger
    margin-bottom: 10px;  # Space below the label
    display: block;  # Makes each label appear on its own line
    text-align: left;  # Aligns the text to the left
}
textarea {
    width: 100%;  # Makes the box take up all the space inside the form
    height: 150px;  # Makes the box tall enough to write a message
    padding: 10px;  # Space inside the box around the text
    border: 1px solid #444;  # A border around the box
    border-radius: 5px;  # Round corners for the text box
    background-color: #2c3e50;  # Dark background for the text box
    color: #ecf0f1;  # Light color for the text inside the box
    resize: none;  # Disables resizing the box
}
input[type="submit"] {
    background-color: #e67e22;  # Bright orange color for the button
    color: white;  # White text on the button
    padding: 12px 24px;  # Bigger button with more space inside
    border: none;  # No border around the button
    border-radius: 5px;  # Rounded corners for the button
    font-size: 1.1em;  # Slightly bigger text for the button
```

```
            cursor: pointer;  # Shows a hand when you hover over the button
            transition: background-color 0.3s ease;  # Smooth color change when hovering
            margin-top: 10px;  # Space above the button
        }
        input[type="submit"]:hover {
            background-color: #d35400;  # Darker orange when hovering over the button
        }
    </style>
</head>
<body>
    <h1>Report an Issue</h1>
    <form action="/submit_report" method="get">
        <label for="issue">Describe your issue:</label>
        <textarea name="issue" placeholder="Enter your issue here..."></textarea>
        <input type="submit" value="Submit Report">
    </form>
</body>
</html>
    """
    response.setContentType("text/html")  # We are sending back an HTML page
    response.send(form_html)  # Send the HTML page with the form

# HTTP route handler for /submit_report - handles the issue submission
def onRouteSubmitReport(url, response):
    print("Received form submission")  # Prints out when someone submits an issue

    # Extract form data manually from the URL
    if '?' in url:
        query_string = url.split('?', 1)[1]  # Get everything after the '?' symbol
        params = {}  # Empty dictionary to hold the form data
        for pair in query_string.split('&'):  # Split the data into separate pieces
            if '=' in pair:
                key, value = pair.split('=', 1)  # Separate the key and value
                params[key] = value  # Store them in our dictionary

        issue = params.get('issue', '')  # Get the issue from the form, if there is one

        if issue:
            # Store the issue in the reported_issues list
            reported_issues.append(issue)

            # Send the updated issues list to the destination server using UDP
            send_issues_to_server(reported_issues)

            confirmation_html = """
            <html>
<head>
    <title>Issue Submitted</title>
```

```
    <style>
        body {
            font-family: Arial, sans-serif;
            background-color: #000000;
            color: #f0f0f0;
            display: flex;
            justify-content: center;
            align-items: center;
            height: 100vh;
            flex-direction: column;
        }
        h1 {
            color: #e67e22;
            font-size: 3em;
            margin-bottom: 20px;
        }
        p {
            font-size: 1.2em;
            margin-bottom: 30px;
        }
        a {
            font-size: 1.3em;
            color: #2980b9;
            text-decoration: none;
            background-color: #e67e22;
            padding: 10px 20px;
            border-radius: 5px;
            transition: background-color 0.3s ease;
        }
        a:hover {
            background-color: #2980b9;
            color: #fff;
        }
        .footer {
            position: fixed;
            bottom: 0;
            width: 100%;
            background-color: #2c3e50;
            color: #ecf0f1;
            text-align: center;
            padding: 15px;
            font-size: 0.9em;
        }
    </style>
</head>
<body>
    <h1>Thank you for your report!</h1>
    <p>Your issue has been successfully recorded.</p>
```

```
        <a href="/report">Go back to the report page</a>
        <div class="footer">
            <p>&copy; 2025 Issue Reporting System</p>
        </div>
    </body>
</html>
            """
            response.setContentType("text/html")
            response.send(confirmation_html)  # Show the confirmation page
        else:
            error_html = """
            <html>
                <head><title>Submission Error</title></head>
                <body>
                    <h1>Error: No issue provided!</h1>
                    <p>Please describe your issue before submitting.</p>
                    <a href="/report">Go back to the report page</a>
                </body>
            </html>
            """
            response.setContentType("text/html")
            response.send(error_html)  # Show the error page if no issue was provided
    else:
        error_html = """
        <html>
            <head><title>Submission Error</title></head>
            <body>
                <h1>Error: No form data received!</h1>
                <p>Please use the form to submit an issue.</p>
                <a href="/report">Go back to the report page</a>
            </body>
        </html>
        """
        response.setContentType("text/html")
        response.send(error_html)  # Show error if something went wrong

# HTTP route handler for /issues - displays all reported issues in a table
def onRouteIssues(url, response):
    print("Request for /issues")  # Prints out when someone asks to see the issues

    issues_html = """
    <html>
        <head>
            <title>Reported Issues</title>
            <style>
                body {
                    font-family: Arial, sans-serif;
                    background-color: #1e1e1e;
```

```
            color: #f0f0f0;
            display: flex;
            justify-content: center;
            align-items: center;
            height: 100vh;
            flex-direction: column;
        }
        h1 {
            color: #e67e22;
            font-size: 2.5em;
            margin-bottom: 20px;
        }
        ul {
            background-color: #34495e;
            padding: 20px;
            border-radius: 10px;
            width: 100%;
            max-width: 600px;
            box-shadow: 0 4px 8px rgba(0, 0, 0, 0.3);
            margin-bottom: 20px;
        }
        li {
            padding: 10px;
            border-bottom: 1px solid #444;
            font-size: 1.2em;
        }
        li:last-child {
            border-bottom: none;
        }
        a {
            font-size: 1.3em;
            color: #2980b9;
            text-decoration: none;
            background-color: #e67e22;
            padding: 10px 20px;
            border-radius: 5px;
            transition: background-color 0.3s ease;
        }
        a:hover {
            background-color: #2980b9;
            color: #fff;
        }
    </style>
</head>
<body>
    <h1>Reported Issues</h1>
    <ul>
"""
```

```
    for issue in reported_issues:
        # Decode URL-encoded data (replace '+' with spaces)
        decoded_issue = issue.replace('+', ' ')  # Decode '+' into spaces
        decoded_issue = decoded_issue.replace('%20', ' ')  # Decode '%20' into spaces
        issues_html += "<li>" + decoded_issue + "</li>"  # Add each issue to the list
    issues_html += """
            </ul>
            <a href="/report">Go back to the report page</a>
        </body>
    </html>
    """
    response.setContentType("text/html")
    response.send(issues_html)  # Send the page with the list of issues

# HTTP route handler for wildcard (any other URL)
def onRouteWildcard(url, response):
    print("Request for " + url)  # Prints out when any other page is requested
    response.setContentType("text/plain")
    response.send("Wildcard route")  # Sends a simple message for any other page

# Main function to set up the HTTP server and handle requests
def main():
    # Set up HTTP server routes
    HTTPServer.route("/report", onRouteReport)  # This handles requests for the report form
    HTTPServer.route("/submit_report", onRouteSubmitReport)  # This handles form
submissions
    HTTPServer.route("/issues", onRouteIssues)  # This shows the reported issues
    HTTPServer.route("/*", onRouteWildcard)  # This catches any unknown pages

    # Start HTTP server
    server_started = HTTPServer.start(80)  # Start the server and listen on port 80
    if server_started:
        print("HTTP server started: " + str(server_started))  # Let us know the server started

    # Keep the program running to handle requests
    try:
        while True:
            pass  # Infinite loop to keep the server running
    except KeyboardInterrupt:
        print("Shutting down the server...")  # Stop the server when we tell it to

# Start the program
if __name__ == "__main__":
    main()
```

1. Bridge Network:

- In **Bridge Network** mode, the VM is connected to the physical network as if it were another physical machine on the same network.

  (remember our metaphors? Hope you do, Bridge network is basically the same as

  how the computers are attached to the thing called “switch” which basically allows devices access to whatever it is connected to, The switches actually use layer 2 method, Which is finding where you live by asking your name instead of addresses only)

- The VM gets its own IP address from the same DHCP server as your physical machine or can have a static IP configured within the network.

  (Our Best friend DHCP is back!!! Since this virtual machines act like computers connected via switch, they can also use DHCP like the ones in our network!)

- The VM can communicate with other devices on the same network, including the host machine and other VMs, as though it were a physical computer.

  (well pretty much self explanatory, host machine is the physical machine you use)

- This setup is typically used when you want the VM to be fully accessible over the network, such as hosting a web server or using the VM as a remote device.

  (again self explanatory, to create a webserver and as a remote device)

2. NAT Network:

- **NAT (Network Address Translation) Network** allows the VM to access external networks (like the internet), but it is not directly accessible from the outside world.

  (Well, I dont remember whether or not we talked about this but Network address translation is basically disguising your address with another address, It sometimes makes communications way easier, and again for the interested people, If you use double NAT on the same network for access, it means you made a shit network! keep that in mind)

- The VM shares the host's IP address for outgoing traffic, but it has a private IP address within the VM's network.
  (well, basically when we try to access internet, or something like that, our ip gets disguised as our physical computers address)

- Incoming traffic is not automatically forwarded to the VM unless you configure port forwarding.
  (Ports are like the doors in your home, you give something the address and the

doors, they can access inside, or the inside people can access the outside)

- This setup is useful when you need internet access on the VM but don't need the VM to be directly accessible from other machines on the network.

  (As said above, use this when you dont need other devices to access the VM, this is the default setting for Virtual Box normally)

Key Differences:

- **Bridge Network**: VM behaves like a separate physical machine on the network.
- **NAT Network**: VM shares the host's network interface for outbound traffic, and is isolated from incoming traffic.