Report Submission Information (must be completed before submitting report!)

- Student 1 Full Name and Number : Xiufu Sun 1372750
- Student 2 Full Name and Number : Xiang Xu 1454161
- Workshop day : Wednesday
- Workshop time : 1pm

Workshop 1 Part 2 – Network Addressing

Objectives:

- Gain hands-on experience on network addressing.
- Practical experience with IPv4 and MAC addresses, prefixes, and CIDR using python.

> **Common objectives of all workshops:** Gain hands-on experience and learn by doing! Understand how theoretical knowledge discussed in lectures relates to practice. Develop motivation for gaining further theoretical and practical knowledge beyond the subject material.

Overview:

In this workshop, you will gain practical experience on networking basics we have covered in the classroom focusing on network addressing. Firstly, you will learn how to represent and manipulate network addresses using the *netaddr* python library. Specifically, you will experiment with IPv4 and MAC addresses, subnets, masks, prefixes, and CIDR.

Workshop Preparation: [before you arrive to the lab]

You can come to the workshops as you are or you can prepare beforehand to learn much more! We will give you a lot of time to finish the tasks but those are the bare minimums. Just like in the lectures, the topics we cover in the workshops are quite deep and we can only do so much in two hours. There is much more to learn and coming prepared to the workshop is one of the best ways to gain more knowledge! For example, there are a few questions in each workshop which you can answer beforehand.

> **Self-learning** is one of the most important skills that you should acquire as a student. Today, self-learning is much easier than it used to be thanks to a plethora of online resources. For this workshop, start by exploring the resource mentioned in the preparation steps below.

Workshop Preparation Steps:

1. Common step for all workshops: read the Workshop Manual.
2. Review relevant lecture slides on addressing, graph theory, and routing.
3. Read netaddr and networkx documentation.
4. Practice your Python skills.

Tasks and Questions:

Follow the procedures described below, perform the given tasks and answer the workshop questions **preferably** on the Python notebook itself! If you prefer, however, you can still provide answers as separate scripts and submit as a pdf file in addition to the Python notebook (code can be zipped). Ensure that your code is clean and appropriately commented.

The resulting notebook will be (part of) your Workshop Report!

> **The goal is to learn**, NOT blindly follow the procedures in the fastest possible way! **Do not simply copy-paste answers (from Internet, friends, etc.). You can and should use all available resources but only to develop your own understanding. If you copy-paste, you will pay the price in the final exam!**

Section 1: Network Addresses

Refer to the netaddr library documentation for the following exercises. First, however, we need to install relevant packages.

If you are using Anaconda, *networkx* should be already installed. **If not,** you can install simply by executing the following cell.

In [1]:
```
!pip install networkx
```

```
Collecting networkx
  Downloading networkx-3.5-py3-none-any.whl.metadata (6.3 kB)
Downloading networkx-3.5-py3-none-any.whl (2.0 MB)
   ---------------------------------------- 0.0/2.0 MB ? eta -:--:--
   ---------- ----------------------------- 0.5/2.0 MB 2.4 MB/s eta 0:00:01
   ------------------------------ --------- 1.6/2.0 MB 4.4 MB/s eta 0:00:01
   ---------------------------------------- 2.0/2.0 MB 4.7 MB/s eta 0:00:00
Installing collected packages: networkx
Successfully installed networkx-3.5
```

The package *netaddr* is probably not installed. You can install it similarly by executing

In [2]:
```
!pip install netaddr
```

```
Collecting netaddr
  Downloading netaddr-1.3.0-py3-none-any.whl.metadata (5.0 kB)
Downloading netaddr-1.3.0-py3-none-any.whl (2.3 MB)
   ---------------------------------------- 0.0/2.3 MB ? eta -:--:--
   ---------------------------------------- 2.3/2.3 MB 18.2 MB/s eta 0:00:00
Installing collected packages: netaddr
Successfully installed netaddr-1.3.0
```

Ask for help from your demonstrator in case you need it. Alternatively, you can install the needed packages by opening the terminal in Anaconda environment and typing `pip install networkx` and `pip install netaddr`

In [1]:
```
%matplotlib notebook
from IPython import display
import numpy as np
import random
import matplotlib.pyplot as plt

from netaddr import *
import pprint  #this is optional

# Utils
def binary_to_ipv4(binary_str):
    octets = [binary_str[i:i+8] for i in range(0, len(binary_str), 8)]
    decimal_octets = [str(int(octet, 2)) for octet in octets]
    ipv4_address = '.'.join(decimal_octets)
    return ipv4_address
```

Do not forget to change the variables below for full credit!

In [2]:
```
# Fixing random seed ensures that you always get the same pseudorandom numbers when using random package.
# We do this to ensure reproducibility but each group should have a different one. Here is a suggestion!
#
# You can use the hash of the ratio of your student numbers to obtain a unique, group-specific seed.

studentid1 = 1372750
studentid2 = 1454161
a = hash(studentid1/studentid2) # use a hash of a function of your student id numbers for random seed
random.seed(a) # needs to be called in each cell below
```

In netaddr library, *IPAddress* object represents a single IP address whereas IPNetwork objects are used to represent subnets, networks or VLANs that accept CIDR prefixes and netmasks.

Question 1.1.

Represent your computer’s IPv4 address using the IPAddress object, e.g. myip=IPAddress('10.100.144.45'). Now, referring to the library documentation, find its version, binary representation, and ipv6 mapping. Discuss your findings briefly.

> Hint: you can find your computer's IP address from command line using ipconfig in windows and ifconfig in linux.

```
In [3]: from netaddr import IPAddress

        myip = IPAddress('10.13.122.222')

        print(f"IP Address: {myip}")
        print(f"IP Version: {myip.version}")

        print(f"Binary representation: {myip.bin}")
        print(f"Binary with separators: {myip.bin[0:8]}.{myip.bin[8:16]}.{myip.bin[16:24]}.{myip.bin[24:32]}")

        print(f"IPv6 mapping: {myip.ipv6()}")
```

```
IP Address: 10.13.122.222
IP Version: 4
Binary representation: 0b1010000011010111101011011110
Binary with separators: 0b101000.00110101.11101011.011110
IPv6 mapping: ::ffff:10.13.122.222
```

Answer:
My IP address is `10.13.122.222`, which is an IPv4 address, so the version is 4.
In binary it is written as `00001010.00001101.01111010.11011110`.
The netaddr library also gives an IPv6 mapping, which is `::ffff:10.13.122.222`.
This mapping is called an IPv4-mapped IPv6 address, used for compatibility when IPv6 hosts need to communicate with IPv4 hosts.
In summary, the object shows the version, binary form, and IPv6 compatibility of an IPv4 address.

Question 1.2.

Represent the following randomly generated **subnetwork** below, using *IPNetwork* object. Find, using netaddr library functions, its beginning IP address, prefix-length, size, broadcast, netmask, and hostmask. Find the broadcast, netmask, and hostmask bits in binary format. Discuss your findings briefly.

```
In [4]: random.seed(a) # each group gets a unique pseudorandom number
        bin_addr = "{:b}".format(random.randint(0, 2**32)<<9)
        subnetwork = binary_to_ipv4(bin_addr[-32:]) + '/23'
        subnetwork
```

```
Out[4]: '241.211.74.0/23'
```

```
In [5]: from netaddr import IPNetwork, IPAddress

        network = IPNetwork(subnetwork)
        print(f"Subnetwork: {network}")

        print(f"Beginning IP address (Network address): {network.network}")
        print(f"Prefix length: {network.prefixlen}")
        print(f"Network size (total addresses): {network.size}")
        print(f"Broadcast address: {network.broadcast}")
        print(f"Netmask: {network.netmask}")
        print(f"Hostmask: {network.hostmask}")

        print(f"\nBinary representations:")
        print(f"Network address binary: {IPAddress(network.network).bin}")
        print(f"Broadcast binary: {IPAddress(network.broadcast).bin}")
        print(f"Netmask binary: {IPAddress(network.netmask).bin}")
        print(f"Hostmask binary: {IPAddress(network.hostmask).bin}")
```

```
Subnetwork: 241.211.74.0/23
Beginning IP address (Network address): 241.211.74.0
Prefix length: 23
Network size (total addresses): 512
Broadcast address: 241.211.75.255
Netmask: 255.255.254.0
Hostmask: 0.0.1.255

Binary representations:
Network address binary: 0b11110001110100110100101000000000
Broadcast binary: 0b11110001110100110100101111111111
Netmask binary: 0b11111111111111111111111000000000
Hostmask binary: 0b111111111
```

Answer:
The subnetwork is `241.211.74.0/23`.
Its network address is `241.211.74.0`, the broadcast address is `241.211.75.255`, and the prefix length is 23.
This gives a total of 512 addresses.
The netmask is `255.255.254.0` and the hostmask is `0.0.1.255`.
In binary, the netmask has the first 23 bits as 1, and the hostmask shows the 9 host bits as 1.
This means 23 bits are fixed for the network, and the remaining 9 bits are available for hosts.

Next, we will investigate subnets, supernets, arbitrary IP address ranges and how to represent those using CIDR notation. You can apply these tools to example problems we have done in the lectures as well!

Question 1.3.

Create and list subnets and supernets with prefix-lengths of 26 and 20, that are contained within and contain the randomly generated **subnetwork2** below, respectively, using *subnet* and *supernet* functions.
Explain/comment briefly.

In [6]:
```python
random.seed(a+1) # each group gets a unique pseudorandom number
bin_addr2 = "{:b}".format(random.randint(0, 2**32)<<9)
subnetwork2 = binary_to_ipv4(bin_addr2[-32:]) + '/23'
subnetwork2
```

Out[6]:
```
'174.191.82.0/23'
```

In [7]:
```python
from netaddr import IPNetwork


network2 = IPNetwork(subnetwork2)
print(f"Original subnetwork2: {network2}")
print(f"Original prefix length: {network2.prefixlen}")
print(f"Original network size: {network2.size}")

# Create subnets
print(f"\nSUBNETS (prefix-length 26):")

subnets_26 = list(network2.subnet(26))
print(f"Number of 26 subnets: {len(subnets_26)}")
print(f"Each subnet size: {subnets_26[0].size} addresses")

print(f"List of all 26 subnets:")
for i, subnet in enumerate(subnets_26):
    print(f"  Subnet {i+1}: {subnet}")
    print(f"    Range: {subnet.network} - {subnet.broadcast}")
    print(f"    Usable hosts: {subnet.network + 1} - {subnet.broadcast - 1}")

# Create supernets
print(f"\nSUPERNETS (prefix-length 20):")

supernets_20 = list(network2.supernet(20))
print(f"Number of 20 supernets: {len(supernets_20)}")

print(f"List of all 20 supernets:")
for i, supernet in enumerate(supernets_20):
    print(f"  Supernet {i+1}: {supernet}")
    print(f"    Range: {supernet.network} - {supernet.broadcast}")
    print(f"    Size: {supernet.size} addresses")
    print(f"    Contains original network: {network2 in supernet}")
```

```
Original subnetwork2: 174.191.82.0/23
Original prefix length: 23
Original network size: 512

SUBNETS (prefix-length 26):
Number of 26 subnets: 8
Each subnet size: 64 addresses
List of all 26 subnets:
  Subnet 1: 174.191.82.0/26
    Range: 174.191.82.0 - 174.191.82.63
    Usable hosts: 174.191.82.1 - 174.191.82.62
  Subnet 2: 174.191.82.64/26
    Range: 174.191.82.64 - 174.191.82.127
    Usable hosts: 174.191.82.65 - 174.191.82.126
  Subnet 3: 174.191.82.128/26
    Range: 174.191.82.128 - 174.191.82.191
    Usable hosts: 174.191.82.129 - 174.191.82.190
  Subnet 4: 174.191.82.192/26
    Range: 174.191.82.192 - 174.191.82.255
    Usable hosts: 174.191.82.193 - 174.191.82.254
  Subnet 5: 174.191.83.0/26
    Range: 174.191.83.0 - 174.191.83.63
    Usable hosts: 174.191.83.1 - 174.191.83.62
  Subnet 6: 174.191.83.64/26
    Range: 174.191.83.64 - 174.191.83.127
    Usable hosts: 174.191.83.65 - 174.191.83.126
  Subnet 7: 174.191.83.128/26
    Range: 174.191.83.128 - 174.191.83.191
    Usable hosts: 174.191.83.129 - 174.191.83.190
  Subnet 8: 174.191.83.192/26
    Range: 174.191.83.192 - 174.191.83.255
    Usable hosts: 174.191.83.193 - 174.191.83.254

SUPERNETS (prefix-length 20):
Number of 20 supernets: 3
List of all 20 supernets:
  Supernet 1: 174.191.80.0/20
    Range: 174.191.80.0 - 174.191.95.255
    Size: 4096 addresses
    Contains original network: True
  Supernet 2: 174.191.80.0/21
    Range: 174.191.80.0 - 174.191.87.255
    Size: 2048 addresses
    Contains original network: True
  Supernet 3: 174.191.80.0/22
    Range: 174.191.80.0 - 174.191.83.255
    Size: 1024 addresses
    Contains original network: True
```

Answer:

Given the network `174.191.82.0/23`:

- **/26 subnets :**
  1. `174.191.82.0/26` → 174.191.82.0–174.191.82.63
  2. `174.191.82.64/26` → 174.191.82.64–174.191.82.127
  3. `174.191.82.128/26` → 174.191.82.128–174.191.82.191
  4. `174.191.82.192/26` → 174.191.82.192–174.191.82.255
  5. `174.191.83.0/26` → 174.191.83.0–174.191.83.63
  6. `174.191.83.64/26` → 174.191.83.64–174.191.83.127
  7. `174.191.83.128/26` → 174.191.83.128–174.191.83.191
  8. `174.191.83.192/26` → 174.191.83.192–174.191.83.255
- **/20 supernet :**
  - `174.191.80.0/20` covering `174.191.80.0–174.191.95.255` (4096 addresses), which contains the original `/23`.

In the output, `/21` and `/22` supernets also appear because the tool lists all possible parent networks larger than `/23`.

Question 1.4.

Create a subnet for the range of addresses between 192.0.2.1 and 192.0.2.33 using *IPRange* function. What are the CIDR subnetworks that cover this range of addresses? Can you cover this range with a superset of two CIDR subnetworks that differs from the original set by only one IP address? What is that extra IP address?

In [8]:
```python
from netaddr import IPRange, IPAddress, IPNetwork

start_ip = IPAddress('192.0.2.1')
end_ip = IPAddress('192.0.2.33')
ip_range = IPRange(start_ip, end_ip)

print(f"IP Range: {ip_range}")
print(f"Start IP: {start_ip}")
print(f"End IP: {end_ip}")
print(f"Total addresses in range: {ip_range.size}")

# Find CIDR subnetworks that cover this range
cidrs = ip_range.cidrs()
print(f"\nCIDR subnetworks that cover this range:")
print(f"Number of CIDR blocks: {len(cidrs)}")

total_addresses = 0
for i, cidr in enumerate(cidrs):
    print(f"  CIDR {i+1}: {cidr}")
    print(f"    Range: {cidr.network} - {cidr.broadcast}")
    print(f"    Size: {cidr.size} addresses")
    total_addresses += cidr.size

print(f"Total addresses in all CIDR blocks: {total_addresses}")
```

```
IP Range: 192.0.2.1-192.0.2.33
Start IP: 192.0.2.1
End IP: 192.0.2.33
Total addresses in range: 33

CIDR subnetworks that cover this range:
Number of CIDR blocks: 6
  CIDR 1: 192.0.2.1/32
    Range: 192.0.2.1 - None
    Size: 1 addresses
  CIDR 2: 192.0.2.2/31
    Range: 192.0.2.2 - None
    Size: 2 addresses
  CIDR 3: 192.0.2.4/30
    Range: 192.0.2.4 - 192.0.2.7
    Size: 4 addresses
  CIDR 4: 192.0.2.8/29
    Range: 192.0.2.8 - 192.0.2.15
    Size: 8 addresses
  CIDR 5: 192.0.2.16/28
    Range: 192.0.2.16 - 192.0.2.31
    Size: 16 addresses
  CIDR 6: 192.0.2.32/31
    Range: 192.0.2.32 - None
    Size: 2 addresses
Total addresses in all CIDR blocks: 33
```

Answer:
Yes, the range can be covered with two CIDR blocks:

- `192.0.2.0/27` (covers 192.0.2.0 – 192.0.2.31)
- `192.0.2.32/31` (covers 192.0.2.32 – 192.0.2.33)

This exactly covers 192.0.2.0 – 192.0.2.33, which differs from the original range by only one extra address: **192.0.2.0**.

Finally, we have a look at MAC addresses. The netaddr library EUI objects provide an interface to the **OUI** (Organisationally Unique Identifier) and **IAB** (Individual Address Block) registration databases available from the IEEE. Explain your findings and/or comment briefly.

Question 1.5.

Find the MAC address of your computer or mobile device (e.g. using ipconfig or device settings). Create an EUI object and find OUI (or IAB) details as well as registration information of your network device.

> Note: the database installed with netaddr can be over one year old, so if your device is newer, it cannot find its registration!

In [9]:
```
from netaddr import EUI, mac_unix_expanded

# replace with your MAC address
mac = EUI('8C-17-59-52-2C-1D')

# print in standard format
print("MAC:", mac)

# show OUI details
print("OUI:", mac.oui)

# registration info
print("Registry:", mac.oui.registration())
```

```
MAC: 8C-17-59-52-2C-1D
OUI: 8C-17-59
Registry: {'address': ['Lot 8, Jalan Hi-Tech 2/3', 'Kulim  Kedah  09000', 'MY'],
 'idx': 9181017,
 'offset': 1257529,
 'org': 'Intel Corporate',
 'oui': '8C-17-59',
 'size': 141}
```

Answer:
The MAC address is `8C-17-59-52-2C-1D`.
Its OUI is `8C-17-59`, which belongs to **Intel Corporate**.
The registration information shows Intel Corporate, Lot 8, Jalan Hi-Tech 2/3, Kulim, Kedah 09000, Malaysia.

Workshop Report Submission Instructions

You should ideally complete the workshop tasks and answer the questions within the respective session! Submission deadlines will be announced on *Canvas/LMS*.

It is **mandatory to follow all of the submissions guidelines** given below. *Don't forget the Report submission information on top of this notebook!*

1. The completed Jupyter notebook and its pdf version (you can simply print-preview and then print as pdf from within your browser) should be uploaded to the right place in *Canvas* by the announced deadline. You should NOT zip all your files into one. You MUST submit your report as a PDF file SEPARATELY to pass the plagiarism check.
2. *It is your responsibility to follow the announcements!* **Late submissions will be penalised!**
3. Filename should be "ELEN90061 Workshop **W: StudentID1-StudentID2** of session **Day-Time**", where **W** refers to the workshop number, **StudentID1-StudentID2** are your student numbers, **Day-Time** is your session day and time, e.g. *Tue-14*.
4. Answers to questions, simulation results and diagrams should be included in the Jupyter notebook as text, code, plots. *If you don't know latex, you can write formulas/text to a paper by hand, scan it and then include as image within Markdown cells.*
5. One report submission per group.

Additional guidelines for your programs:

- Write modular code using functions.
- Properly indent your code. But Python forces you do that anyway ;)
- Heavily comment the code to describe your implementation and to show your understanding. No comments, no credit!
- Make the code your own! It is encouraged to find and get inspired by online examples but you should exactly understand, modify as needed, and explain your code via comments. There will be no credit for blind copy/paste even if it somehow works (and it is easier to detect it than you might think)!

In [ ]:
```
```