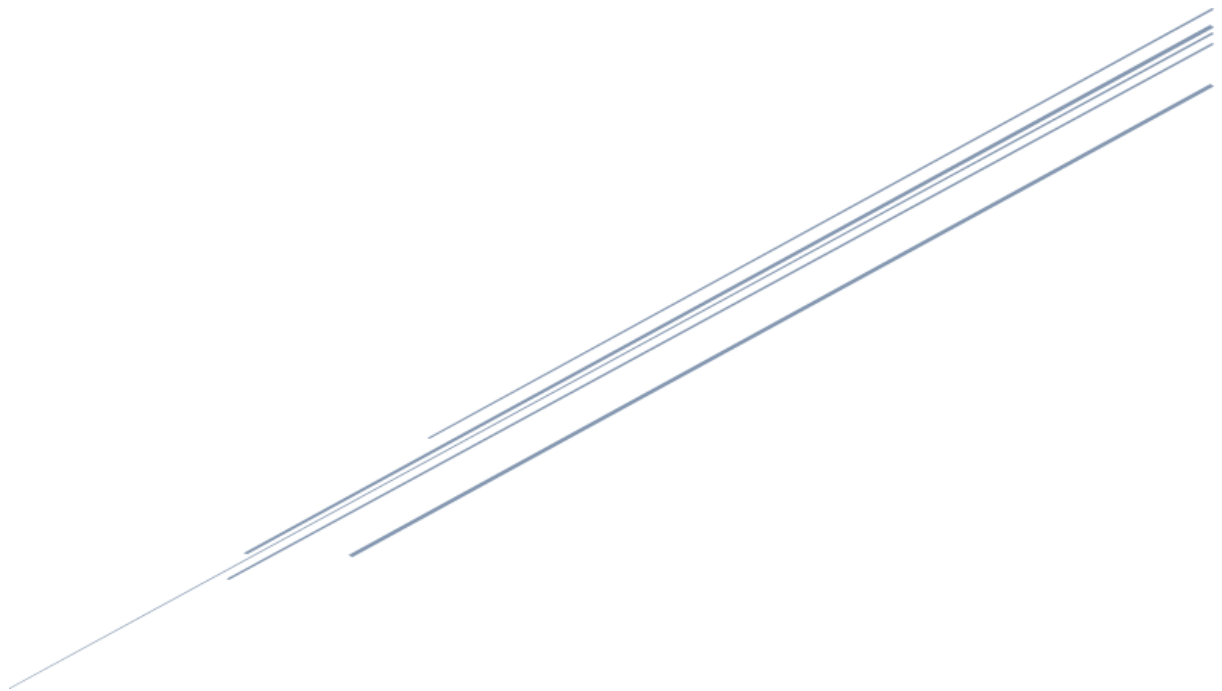


# PRÀCTICA 1



Rems Nalivaiko

Pol Rubio

18 d'octubre de 2023

103311 - XARXES I SERVEIS

Fundació TecnoCampus Mataró-Maresme

# Index

<b>Introduction</b>	<b>3</b>
<b>Program Execution</b>	<b>4</b>
Script Overview	4
Locator Class:	4
Mapper Class:	4
Jumper Class:	4
Execution Flow	5
Argument Parsing:	5
Network Traversal:	5
Geolocation Retrieval:	5
Map Rendering:	5
Output:	5
Prerequisites	5
Functionality Explanation	6
Locator Class	6
Mapper Class	6
Jumper Class	6
Execution Entry: if <code>__name__=="__main__"</code> :	7
Wireshark	8
<b>Example of execution</b>	<b>10</b>
<b>Funcionament ICMP</b>	<b>11</b>
<b>Development Challenges</b>	<b>13</b>
Handling Timeouts and Repeated IPs:	13
Geolocation Data Accuracy:	13
Library Dependencies:	13
Map Rendering Performance:	13
<b>Conclusions</b>	<b>13</b>
<b>Bibliography</b>	<b>14</b>

# Introduction

This project is all about showing how we can trace data packets as they travel across networks to get where they're going. We also want to map out that path on an actual map, so people can really see the route the data takes across the whole internet infrastructure around the world.

The main goal here is to build a tool that combines network diagnostics and geolocation in one smooth package and that way, folks get a visual, easy-to-understand view of the network path and how it connects to different geographical places. We're hoping this will help make network routing and the geography side of things more clear.

We're using some Python libraries to make this happen - ipinfo for the geolocation piece matplotlib and basemap to draw the map, socket for network stuff, scapy to build and send the packets, and argparse so we can parse command line arguments. With these our script can do the network hops to trace the path, figure out the IP addresses, collect the geographic data, and finally draw the map that shows how the packets got from point A to point B.

In the following sections we'll dig into the details of how the program runs, explaining what each part and function does to deliver on the goals of the project. That should provide a good understanding of how we designed it, built it, and how to use it. The documentation will walk through the key classes and methods so it's clear how everything comes together.

# Program Execution

The Python script developed for this project combines network traversal with geolocation services to trace and visualise the path data the packets take to reach a designated destination. This section explains the script's functionality, the working of each class, and the prerequisites for successful execution.

## Script Overview

The script is structured around three primary classes: Locator, Mapper, and Jumper, each having a distinct role in the overall program execution.

### Locator Class:

The Locator class is responsible for obtaining geolocation data of IP addresses. It leverages the `ipinfo` library to fetch latitude and longitude information, which is crucial for mapping the network path geographically.

### Mapper Class:

The Mapper class utilizes the `mpl_toolkits.basemap` and `matplotlib` libraries to render a geographical map and plot the network path based on the geolocation data obtained. It draws the map, the network nodes, and the lines connecting these nodes to represent the path.

### Jumper Class:

The Jumper class embodies the core of network traversal. Using the `scapy` library, it crafts and sends ICMP packets with incrementing TTL values to trace the route to the destination. It also handles timeouts and repeated IPs to ensure a precise and efficient path tracing.

## Execution Flow

### Argument Parsing:

At the outset, the script parses command-line arguments to obtain user inputs like the destination IP, network interface, packet protocol, timeout value, and output image filename.

### Network Traversal:

An instance of the Jumper class is created and the `path_finder` method is invoked to commence the network traversal and collect IPs of the intermediate nodes.

### Geolocation Retrieval:

With the IP addresses collected, an instance of the Locator class is created and invoked to fetch the geolocation data of these IPs.

### Map Rendering:

Finally, an instance of the Mapper class is created and invoked to render the geographical map and plot the network path.

### Output:

The map is either displayed to the user or saved as an image file based on the user input.

### Prerequisites

- Python 3.8 or above.
- Required libraries: `ipinfo`, `mpl_toolkits.basemap`, `matplotlib`, `socket`, `scapy`, and `argparse`.
- Network access to send ICMP packets and fetch geolocation data.
- (Optional) A `.env` file containing the `ACCESS_TOKEN` for the `ipinfo` service if no token is provided through the script.

## Functionality Explanation

The script is meticulously crafted to ensure a smooth execution flow while encapsulating the core functionalities into distinct classes and methods. Here's a detailed breakdown of the key functions and classes in the script:

### Locator Class

```
__init__(self, token:str|None=None, env_file:str=".env") -> None:
```

This constructor initialises the Locator class, handling the ipinfo access token either from an argument or from an environment file.

```
run(self, ips:list[str]):
```

This method takes a list of IP addresses, fetches their geolocation data, and returns a dictionary mapping IPs to their respective latitude and longitude coordinates.

### Mapper Class

```
mapit(self, coordinates:list[tuple[float, float]], timeout:bool=False,  
image_filename:str|None="world_map.png"):
```

This method takes a list of coordinates, and an optional timeout flag and image filename, to render and display (or save) the geographical map depicting the network path.

### Jumper Class

```
__init__(self, destination:str, interface:str, timeout:int=2, verbose:int=0,  
max_requests:int=5):
```

The constructor initialises the Jumper class with essential parameters like destination IP, network interface, timeout value, verbosity level, and maximum request attempts.

```
single_jump(self, ttl:int=1):
```

This method performs a single network hop by sending an ICMP or UDP (depending on passed parameters to the constructor) packet with a specified TTL value, and returns the time taken and the received packet (if any).

```
path_finder(self, max_ttl:int=255):
```

This method orchestrates the network traversal by invoking `single_jump` with incrementing TTL values, collecting IPs of intermediate nodes, handling timeouts, and repeated IPs.

`resolve_ip(self, ip:str):`

This method attempts to resolve a given IP address to its hostname and returns the IP itself if resolution fails.

Execution Entry: `if __name__=="__main__":`

The script execution begins here, where command-line arguments are parsed, and instances of Jumper, Locator, and Mapper classes are created to be invoked in sequence to perform the network traversal, geolocation retrieval, and map rendering, respectively.

Argument Parsing:

Utilizes the argparse library to define, parse, and handle command-line arguments, enabling user input for various parameters :

1. **destination:** This first argument is required, and it's used to specify the destination IP address that the program will ping.
2. **-i or --interface:** This is an optional argument for specifying the network interface to use for sending ping requests. If no value is provided, it will be set to None.
3. **-t or --timeout:** An optional argument to set the timeout for ping responses. It's expected to be an integer. If no value is provided, it defaults to 2 seconds.
4. **-v or --verbose:** An optional argument with no value (just a flag). If specified, it increases the verbosity of the program. It's initially set to False.
5. **-m or --max-requests:** An optional argument to set the maximum number of requests the program will send to the same destination. It's expected to be an integer, it defaults to 5.
6. **-p or --packet-size:** An optional argument to set the packet sent in bytes. It defaults to 60B. The minimum number of bytes is 28B.
7. **-o or --output:** An optional argument to specify the filename of the output image. It defaults to "world\_map.png".
8. **-u or --udp:** An optional argument with no value (just a flag). If specified, it changes the protocol to UDP. It's initially set to False.

Users can run the script from the command line and provide these arguments to configure the behaviour of the program. For example, they can specify the destination IP address with -d, set the timeout with -t, or request more verbose output with -v. This allows the script to be more versatile and adaptable for different use cases.

icmp					
No.	Time	Source	Destination	Protocol	Length Info
283	46.987990778	172.20.10.2	1.1.1.1	ICMP	74 Echo (ping) request id=0x0000, seq=0/0, ttl=1 (no response found!)
284	46.990478920	172.20.10.1	172.20.10.2	ICMP	70 Time-to-live exceeded (Time to live exceeded in transit)
285	47.017314165	172.20.10.2	1.1.1.1	ICMP	74 Echo (ping) request id=0x0000, seq=0/0, ttl=2 (no response found!)
297	49.054690718	172.20.10.2	1.1.1.1	ICMP	74 Echo (ping) request id=0x0000, seq=0/0, ttl=3 (no response found!)
302	51.071358910	172.20.10.2	1.1.1.1	ICMP	74 Echo (ping) request id=0x0000, seq=0/0, ttl=4 (no response found!)
303	51.115028678	10.243.213.41	172.20.10.2	ICMP	70 Time-to-live exceeded (Time to live exceeded in transit)
309	51.272776266	172.20.10.1	172.20.10.2	ICMP	70 Time-to-live exceeded (Time to live exceeded in transit)
318	51.305622096	172.20.10.2	1.1.1.1	ICMP	74 Echo (ping) request id=0x0000, seq=0/0, ttl=5 (no response found!)
311	51.335464358	5.205.25.61	172.20.10.2	ICMP	106 Time-to-live exceeded (Time to live exceeded in transit)
314	51.440827407	172.20.10.2	1.1.1.1	ICMP	74 Echo (ping) request id=0x0000, seq=0/0, ttl=6 (no response found!)
317	53.461714094	172.20.10.2	1.1.1.1	ICMP	74 Echo (ping) request id=0x0000, seq=0/0, ttl=7 (no response found!)

[illegible]

```

▶ Frame 284: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface wlp89s0, id 0
▶ Ethernet II, Src: 9a:60:ca:1e:e4:64 (9a:60:ca:1e:e4:64), Dst: IntelCor_a0:84:f8 (84:5c:f3:a0:84:f8)
▶ Internet Protocol Version 4, Src: 172.20.10.1, Dst: 172.20.10.2
▼ Internet Control Message Protocol
    Type: 11 (Time-to-live exceeded)
    Code: 0 (Time to live exceeded in transit)
    Checksum: 0x7a85 [correct]
    [Checksum Status: Good]
    Unused: 00000000
▶ Internet Protocol Version 4, Src: 172.20.10.2, Dst: 1.1.1.1
▼ Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x727a [unverified] [in ICMP error packet]
    [Checksum Status: Unverified]
    Identifier (BE): 0 (0x0000)
    Identifier (LE): 0 (0x0000)
    Sequence Number (BE): 0 (0x0000)
    Sequence Number (LE): 0 (0x0000)

```



■ ■ ■

TTL 5

## Packet

[illegible]

## Response

[illegible]

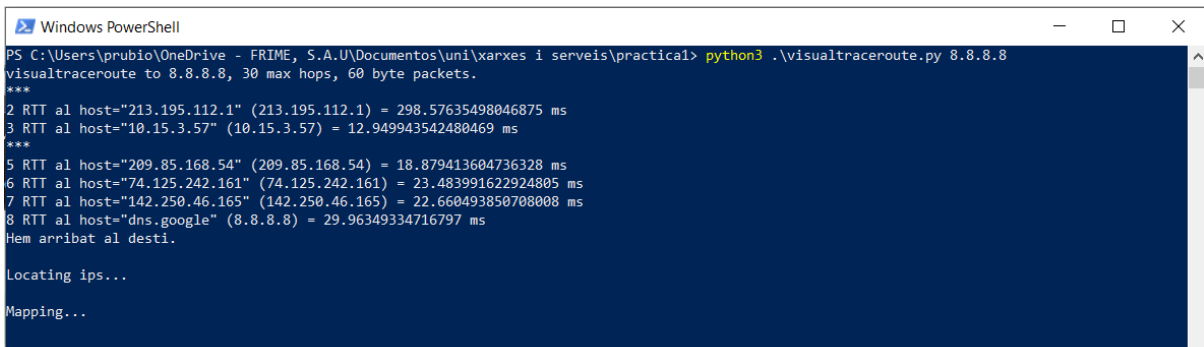
## Example of execution

1. First, we need to install the requirement package list with the command:

```
pip install -r requirements.txt
```

2. An "access\_token.txt" file must be created in which the IpInfo API token must be stored.
3. Run the code of our program with the following command, in this example we will trace the route to google dns (8.8.8.8) and leave all the arguments in default:

```
python3 visualtraceroute.py 8.8.8.8
```



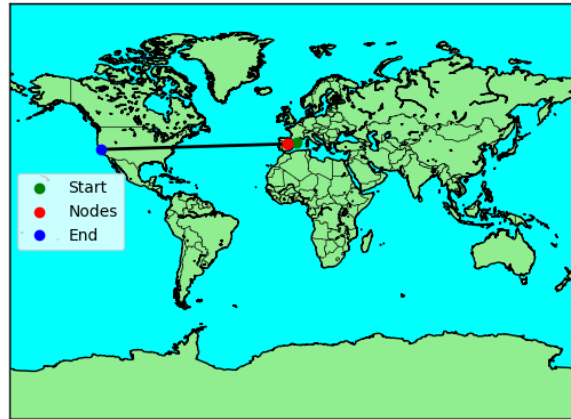
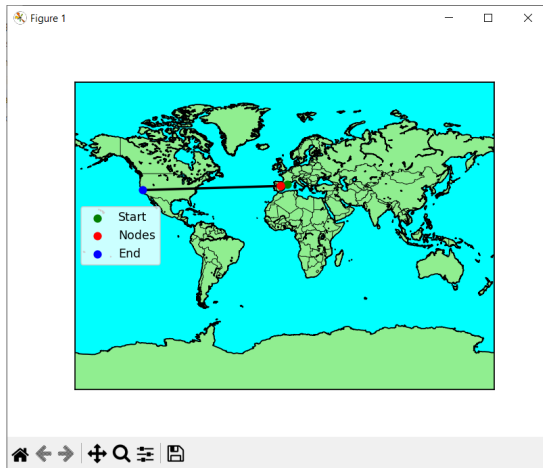
```
Windows PowerShell
PS C:\Users\prubio\OneDrive - FRIME, S.A.U\Documentos\uni\arxes i serveis\practical> python3 .\visualtraceroute.py 8.8.8.8
visualtraceroute to 8.8.8.8, 30 max hops, 60 byte packets.
***
 2 RTT al host="213.195.112.1" (213.195.112.1) = 298.57635498046875 ms
 3 RTT al host="10.15.3.57" (10.15.3.57) = 12.949943542480469 ms
***
 5 RTT al host="209.85.168.54" (209.85.168.54) = 18.879413604736328 ms
 6 RTT al host="74.125.242.161" (74.125.242.161) = 23.483991622924805 ms
 7 RTT al host="142.250.46.165" (142.250.46.165) = 22.660493850708008 ms
 8 RTT al host="dns.google" (8.8.8.8) = 29.96349334716797 ms
Hem arribat al desti.

Locating ips...
Mapping...
```

4. Once the execution of the code has been completed, a Matplotlib tab will open where we can see the map with the route. In addition, an image of the map with the route will be saved to us with the name we specified, by default "world\_map.png".

Matplotlib:

Saved foto ("world\_map.png"):



# Funcionament ICMP

## 1. Quina és la mida i el format de la capçalera d'un paquet ICMP?

Mida: 28 Bytes

Format:

En el format de paquet ICMP, els primers 32 bits del paquet contenen tres camps:

- Type (8 bits): Els 8 bits inicials del paquet són per al tipus de missatge, proporcionen una breu descripció del missatge perquè la xarxa receptora sàpiga quin tipus de missatge està rebent i com respondre. Alguns tipus de missatges comuns són els següents:
  - Type 0 – Resposta d'eco
  - Type 3 – Destí inassolible
  - Type 5 – Missatge de readreçament
  - Type 8 – Sol·licitud d'eco
  - Type 11 – Temps excedit
  - Type 12 – Problema de paràmetres
- Code (8 bits): el codi són els següents 8 bits del format del paquet ICMP. Aquest camp conté informació addicional sobre el missatge d'error i el tipus.
- Checksum (16 bits): els últims 16 bits són per al camp de suma de comprovació en l'encapçalat del paquet ICMP. La suma de verificació s'utilitza per a verificar la quantitat de bits del missatge complet i habilitar l'eina ICMP per a garantir que es lliurin les dades completes.

Els següents 32 bits de l'encapçalat ICMP són encapçalats estesos que tenen la funció d'assenyalar el problema en el missatge IP. Les ubicacions dels bytes s'identifiquen mitjançant el punter que genera el missatge de problema i el dispositiu receptor cerca aquí per a assenyalar el problema.

L'última part del paquet ICMP són les dades o la càrrega útil de longitud variable. Els bytes màxims en IPv4 són 576 bytes i en IPv6, 1280 bytes.

## 2. Quin és el valor que identifica el protocol ICMP dins la capçalera IP?

▶ Frame 311: 186 bytes on wire (1488 bits), 186 bytes captured (1488 bits) on interface 0	0000 84 5c f3 a0 84 f8 9a 60
▶ Ethernet II, Src: 9a:60:ca:1e:e4:64 (9a:60:ca:1e:e4:64), Dst: 02:00:00:00:00:00	0010 00 ac 00 00 00 00 f8 01
▶ Internet Protocol Version 4, Src: 5.205.25.61, Destination: 5.205.25.61	0020 0a 02 0b 00 f5 72 00 00
0100 .... = Version: 4	0030 00 00 01 01 01 a9 ac 14
.... 0101 = Header Length: 20 bytes (5)	0040 72 07 00 00 00 00 58 58
▶ Differentiated Services Field: 0x00 (DSCP: CS0)	0050 58 58 58 58 58 58 58 58
Total Length: 172	0060 58 58 58 58 58 58 00 00
Identification: 0x0000 (0)	0070 00 00 00 00 00 00 00 00
▶ 000. .... = Flags: 0x0	0080 00 00 00 00 00 00 00 00
...0 0000 0000 0000 = Fragment Offset: 0	0090 00 00 00 00 00 00 00 00
Time to Live: 248	00a0 00 00 00 00 00 00 00 00
Protocol: ICMP (1)	00b0 01 01 01 66 d0 01 00 09
Header Checksum: 0xed30 [validation disabled]	
[Header checksum status: Unverified]	
Source Address: 5.205.25.61	

0x01

## 3. Quin és el port d'origen i destí (capa de transport) que utilitza el protocol ICMP?

Els missatges ICMP no tenen números de port d'origen i destinació associats, però fan servir els camps "Type" i "Code" dins de l'encapçalat ICMP per a especificar el propòsit i els detalls del missatge.

## 4. Quin és el tipus/codi que s'utilitza per identificar un paquet ICMP de tipus Echo Reply?

En la capa del ICMP hi ha un camp de "type", 8 (0x08) sent resposta tipus Echo Reply.

## 5. Quin és el tipus/codi que s'utilitza per identificar un paquet ICMP de tipus Time Exceeded?

Valor 11 (0x0b)

## Development Challenges

Throughout the development process, we encountered several challenges that required thoughtful consideration and resolution. Some of these challenges included:

### Handling Timeouts and Repeated IPs:

Ensuring a precise and efficient network traversal required implementing a robust mechanism to handle timeouts and repeated IPs. This was particularly crucial to avoid infinite loops and to manage scenarios where the network path could not be fully resolved.

### Geolocation Data Accuracy:

The accuracy of geolocation data is contingent on the reliability of the ipinfo service. There were instances where the geolocation data was not precise, which affected the accuracy of the mapped network path.

### Library Dependencies:

The project relies on multiple external libraries. Ensuring compatibility and managing dependencies was a task that required careful attention to avoid conflicts and ensure smooth operation.

### Map Rendering Performance:

Rendering a detailed geographical map while maintaining a responsive user experience was a challenge. Optimizing the map rendering process was essential to achieve a balance between visual appeal and performance.

## Conclusions

The project successfully merged network diagnostics with geolocation services, providing a visually intuitive representation of network paths. Through the meticulous design of the script, we have demonstrated how Python, coupled with various libraries, can be employed to trace network paths and render geographical maps.

# Bibliography

## **Python Standard Library:**

Official documentation: <https://docs.python.org/3/library/index.html>

## **ipinfo:**

Official website: <https://ipinfo.io/>

Python library documentation: <https://github.com/ipinfo/python>

## **mpl\_toolkits.basemap and matplotlib:**

Basemap Toolkit documentation: <https://matplotlib.org/basemap/>

Matplotlib documentation: <https://matplotlib.org/>

## **scapy:**

Official documentation: <https://scapy.net/>

## **argparse:**

Official documentation: <https://docs.python.org/3/library/argparse.html>

## **socket:**

Official documentation: <https://docs.python.org/3/library/socket.html>