Infrastructure as Code-Walkthrough

Python Network Utilities

Contents

Last Updated: 19 October 2022

A **walkthrough** is intended to bring you through a technical exercise. A walkthrough shows you how I completed a task in a particular context, on specific systems, at a point in time. The document is a good basic guide, but you should always confirm that the information is current and if a more recent best practice exists, you should make yourself aware of it.

Introduction

This walkthrough is provided to demonstrate how to get started with network protocols in Python. I assume you have no knowledge of networking, but I do not explain terms in great detail. It a term has you confused, do some background research. Otherwise, identify what you do not know, this will be covered in the networking module.

I'm also going to do things the simplest way possible. I'm going to use FTP instead of the secure version SFTP. In later sessions, I am going to use Telnet instead of SSH. Please note that in a real project you will always use the secure versions of these protocols. There is a real chicken-and-egg problem for me here, I do not plan to cover cryptography until the third quarter of the networking module.

In these notes, I introduce UDP, TCP and multicast, without exhaustive explanation, this will be covered in the networking module.

You will find very good tutorials openly available, look at

https://realpython.com/python-sockets/

Deliverables

You will be required to keep a list of commands with notes as to why you are using them as well as results of any testing you have done. You may be asked to present these notes as evidence of work completed or as part of a lab book. Check with your lecturer!

Prerequisites

1. I will carry out my exercises using a Windows 11 desktop. You can carry out this work on almost any modern platform.
2. You should have already installed Python as per my notes.
3. You should have already installed VSC as per my notes.
4. You have created a directory to keep example files in and you are ready to code.
    a. This could be on your OneDrive, in these examples, I am using **OneDrive\Python\Exercises_11**

FTP

First Test

The simplest (and most insecure) way we can transfer files will be to use file transfer protocol (FTP).

I can install the most modern FTP library using

pip install pyftpdlib

I can run a simple server to share my working directory from the command prompt using

python -m pyftpdlib -w --user=*username* --password=*password*

Anonymous FTP (no username or password) works using

python -m pyftpdlib -w

I can then use WinSCP to make the connection and upload/download files.

Test this and see if you can make it work!

Some Code

Anonymous FTP is still used to distribute open-source software, we refer to these locations as mirror sites.

Look at ftp.heanet.ie using a web browser. I want to get the latest security keys from

/mirrors/ubuntu-cdimage/releases/22.04/release

The file is called **SHA256SUMS** and you should be able to find it via a web browser.

I wrote the following simple code using the original Python **ftplib**.

```
import ftplib

# Set the path
path = '/mirrors/ubuntu-cdimage/releases/22.04/release'
# What file to download
filename = 'SHA256SUMS'
# Make the connection
ftp = ftplib.FTP("ftp.heanet.ie")
# Anonymous login
ftp.login()
# Change to the correct directory
ftp.cwd(path)
# Retrieve the file
ftp.retrbinary("RETR " + filename, open(filename, 'wb').write)
# Cleanly exit
ftp.quit()
```

This is the simplest possible usable and useful script!

Having fixed paths, URLs, and file names does seem very unprofessional.

To make this a little more usable, I could put all my settings in a dictionary (we covered these in Walkthrough 3).

```
import ftplib

FTP = {
    "PATH": '/mirrors/ubuntu-cdimage/releases/22.04/release',
    "FILENAME": 'SHA256SUMS',
    "URL": 'ftp.heanet.ie'
}

# Make the connection
ftp = ftplib.FTP(FTP['URL'])
# Anonymous login
ftp.login()
# Change to the correct directory
ftp.cwd(FTP["PATH"])
# Retrieve the file
ftp.retrbinary("RETR " + FTP["FILENAME"], open(FTP["FILENAME"], 'wb').write)
ftp.quit()
```

This is shorter, cleaner code. But there is a better way.

I create a directory called **Exercises_11\settings** and create a new file, **ftp.py** in it.

```
FTP = {
    "PATH": '/mirrors/ubuntu-cdimage/releases/22.04/release',
    "FILENAME": 'SHA256SUMS',
    "URL": 'ftp.heanet.ie'
}
```

My final code is economical, and it is clear what the settings are and where they come from. To download another file, I just need a different settings file.

```
"""
FTP file downloader
Tested with Python >=3.6
By: JOR
    v0.1    20AUG20
"""
import ftplib
import settings.ftp as settings

# Make the connection
ftp = ftplib.FTP(settings.FTP['URL'])
# Anonymous login
ftp.login()
# Change to the correct directory
ftp.cwd(settings.FTP["PATH"])
# Retrieve the file
ftp.retrbinary("RETR " + settings.FTP["FILENAME"], open(settings.FTP["FILENAME"], 'wb').write)
ftp.quit()
```

Telemetry with UDP

For some things in networking, we use User Datagram Protocol (UDP). We send packets of data to a target, but we have no idea if they are received or not. Strangely, this is normal for many network protocols, especially those that carry real time data.

First, I create **Exercises_11\settings\udp.py**

```
UDP = {
    "SERVER_UDP_IPv4": '127.0.0.1',
    "CLIENT_UDP_IPv4": '127.0.0.0',
    "SERVER_PORT": 23
}
```

Then I create **Exercises_11\udp_client.py**

```
'''
UDPClient by: JOR
Send UDP packets to a particular address and port.
Alpha: 13FEB22
'''

import socket
import time
from datetime import datetime
import settings.udp as settings

UDP_IP = settings.UDP["SERVER_UDP_IPv4"]
UDP_PORT = settings.UDP["SERVER_PORT"]

print(f'This is the UDP client, it will try to connect to a server at {UDP_IP}:{UDP_PORT} in the settings file.')
print('This script has no error handling, by design')

while True:
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
        s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
        message_text = f"ATU {datetime.now()}"
        message = message_text.encode('utf-8')
        s.sendto(message, (UDP_IP, UDP_PORT))
        print(f'Sent {message_text}')
        time.sleep(1)
```

In network terminology, a socket is an endpoint on a node, which can take part in two-way communication. I figured out the Python socket syntax and parameters from [1].

This program constructs a test line, for example

ATU 2022-10-13 12:38:01.285021

This is sent to a server defined by the settings file, once per second, and to the screen as feedback to the user. I tested this on the VDI and because of firewall restrictions, I used the loopback address 127.0.0.1.

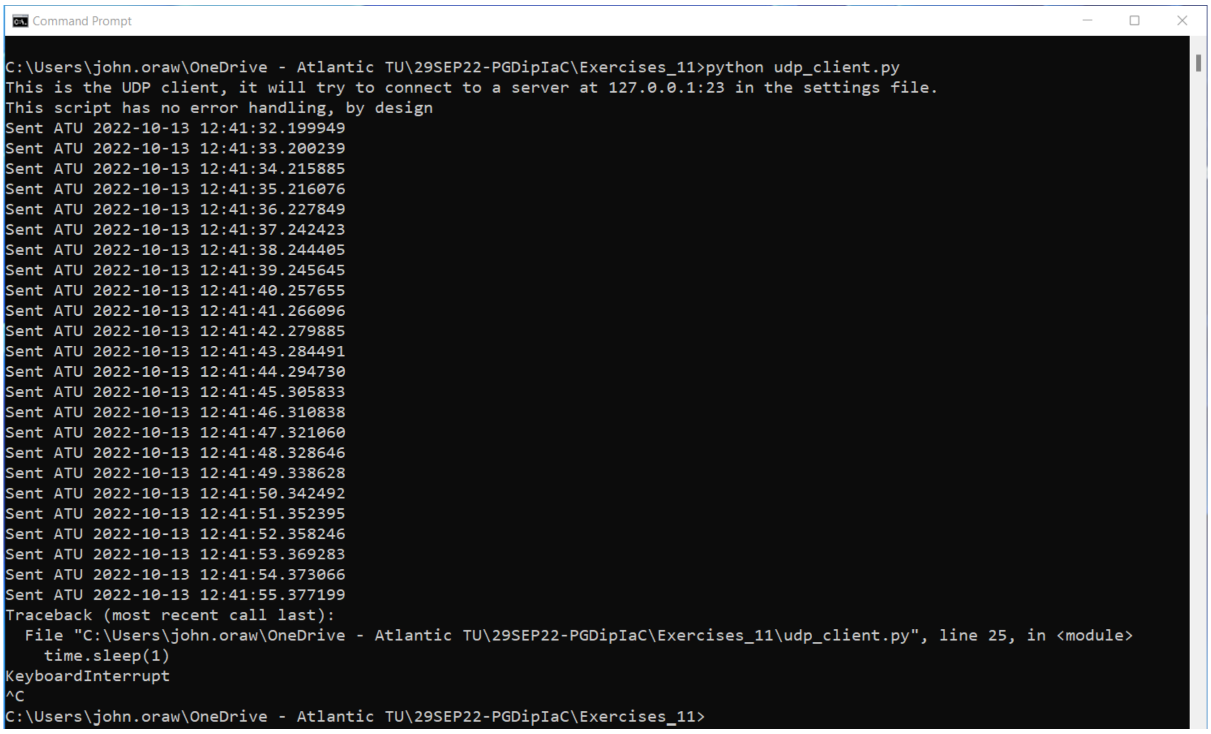
This worked for me.

Then I create **Exercises_11\udp_server.py**

```
'''
UDPServer by: JOR
Listens for packets on a particular address and port.
Alpha: 13FEB22
'''

import socket
import settings.udp as settings

UDP_IP = settings.UDP["SERVER_UDP_IPv4"]
UDP_PORT = settings.UDP["SERVER_PORT"]
BUFFER_SIZE = 1024

print(f'This is the UDP server, it will open a port at {UDP_IP}:{UDP_PORT} and being listening')
print(f'Make sure the actual server IP address matches {UDP_IP} in the settings file')
print('This script has no error handling, by design')

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
    s.bind( (UDP_IP, UDP_PORT) )
    print('Listening on', UDP_IP)
    while True:
        data, addr = s.recvfrom(BUFFER_SIZE)
        data = data.decode()
        print(addr, data)
```

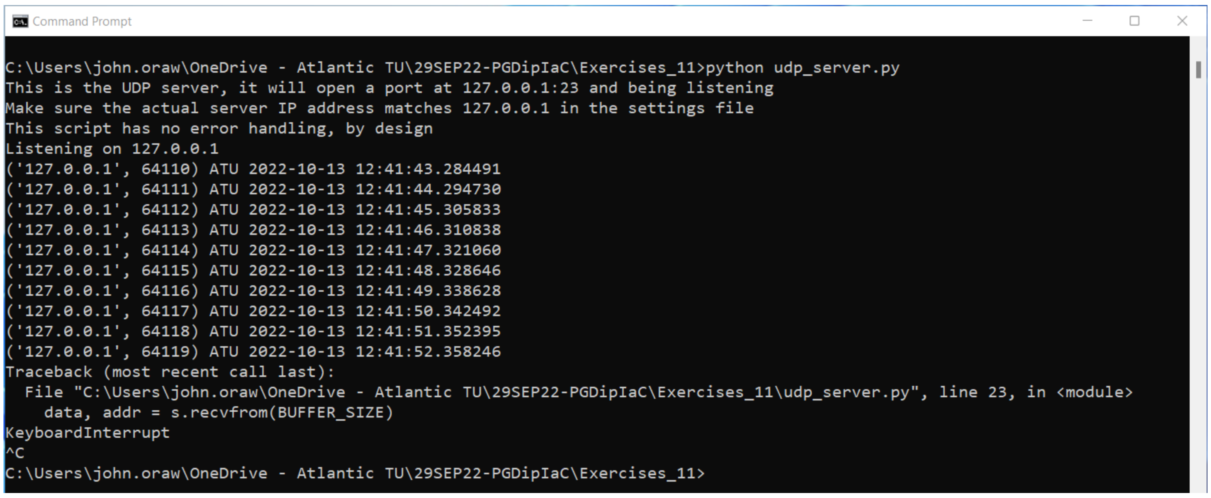
Testing

To make this work, I run the programs in two separate console sessions. I must press **[ctrl][c]** to exit. I start the client sending packets.



```
Command Prompt

C:\Users\john.oraw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_11>python udp_client.py
This is the UDP client, it will try to connect to a server at 127.0.0.1:23 in the settings file.
This script has no error handling, by design
Sent ATU 2022-10-13 12:41:32.199949
Sent ATU 2022-10-13 12:41:33.200239
Sent ATU 2022-10-13 12:41:34.215885
Sent ATU 2022-10-13 12:41:35.216076
Sent ATU 2022-10-13 12:41:36.227849
Sent ATU 2022-10-13 12:41:37.242423
Sent ATU 2022-10-13 12:41:38.244405
Sent ATU 2022-10-13 12:41:39.245645
Sent ATU 2022-10-13 12:41:40.257655
Sent ATU 2022-10-13 12:41:41.266096
Sent ATU 2022-10-13 12:41:42.279885
Sent ATU 2022-10-13 12:41:43.284491
Sent ATU 2022-10-13 12:41:44.294730
Sent ATU 2022-10-13 12:41:45.305833
Sent ATU 2022-10-13 12:41:46.310838
Sent ATU 2022-10-13 12:41:47.321060
Sent ATU 2022-10-13 12:41:48.328646
Sent ATU 2022-10-13 12:41:49.338628
Sent ATU 2022-10-13 12:41:50.342492
Sent ATU 2022-10-13 12:41:51.352395
Sent ATU 2022-10-13 12:41:52.358246
Sent ATU 2022-10-13 12:41:53.369283
Sent ATU 2022-10-13 12:41:54.373066
Sent ATU 2022-10-13 12:41:55.377199
Traceback (most recent call last):
  File "C:\Users\john.oraw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_11\udp_client.py", line 25, in <module>
    time.sleep(1)
KeyboardInterrupt
^C
C:\Users\john.oraw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_11>
```

Then I start the server and make sure it can receive packets.



```
Command Prompt

C:\Users\john.oraw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_11>python udp_server.py
This is the UDP server, it will open a port at 127.0.0.1:23 and being listening
Make sure the actual server IP address matches 127.0.0.1 in the settings file
This script has no error handling, by design
Listening on 127.0.0.1
('127.0.0.1', 64110) ATU 2022-10-13 12:41:43.284491
('127.0.0.1', 64111) ATU 2022-10-13 12:41:44.294730
('127.0.0.1', 64112) ATU 2022-10-13 12:41:45.305833
('127.0.0.1', 64113) ATU 2022-10-13 12:41:46.310838
('127.0.0.1', 64114) ATU 2022-10-13 12:41:47.321060
('127.0.0.1', 64115) ATU 2022-10-13 12:41:48.328646
('127.0.0.1', 64116) ATU 2022-10-13 12:41:49.338628
('127.0.0.1', 64117) ATU 2022-10-13 12:41:50.342492
('127.0.0.1', 64118) ATU 2022-10-13 12:41:51.352395
('127.0.0.1', 64119) ATU 2022-10-13 12:41:52.358246
Traceback (most recent call last):
  File "C:\Users\john.oraw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_11\udp_server.py", line 23, in <module>
    data, addr = s.recvfrom(BUFFER_SIZE)
KeyboardInterrupt
^C
C:\Users\john.oraw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_11>
```

Telemetry with TCP

When we need reliable data communications, we use Transmit Control Protocol (TCP). The client and server in this communication have a complex network stack. Every communication is acknowledged, and functionality exists within the protocol for it to auto-tune to the characteristics of the communication medium.

First, I create **Exercises_11\settings\tcp.py**

```
TCP = {
    "SERVER_TCP_IPv4": '127.0.0.1',
    "SERVER_PORT": 23
}
```

Then I create **Exercises_11\tcp_client.py**

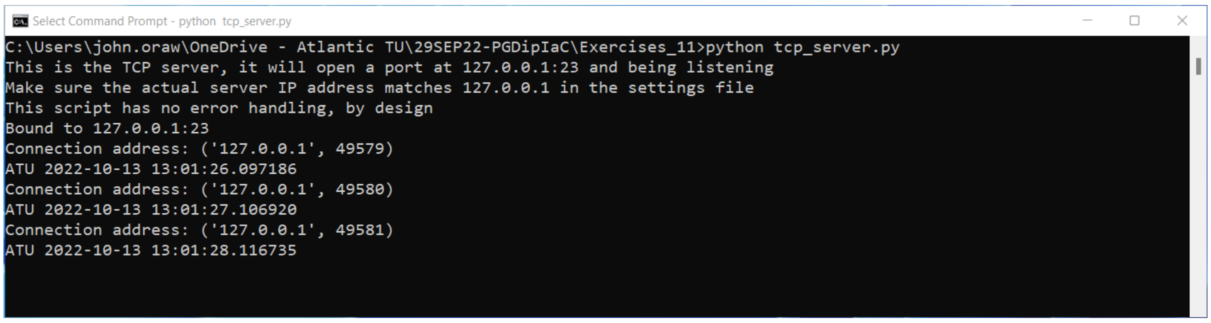
```
'''
TCPClient by: JOR
Send TCP packets to a particular address and port.
Alpha: 13FEB22
'''

import socket
import time
from datetime import datetime
import settings.tcp as settings

TCP_IP = settings.TCP["SERVER_TCP_IPv4"]
TCP_PORT = settings.TCP["SERVER_PORT"]

print(f'This is the TCP client, it will try to connect to a server at {TCP_IP}:{TCP_PORT} in the settings file.')
print('This script has no error handling, by design')

BUFFER_SIZE = 1024
while True:
    print(f'Trying to open a socket to {TCP_IP}:{TCP_PORT}')
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        message_text = f"ATU {datetime.now()}"
        message = message_text.encode('utf-8')
        s.connect((TCP_IP, TCP_PORT))
        s.send(message)
        print(f'Sent {message_text} to {TCP_IP}:{TCP_PORT}')
        data = s.recv(BUFFER_SIZE)
        print('Server echoed:', data)
        time.sleep(1)
```

Then I create **Exercises_11\tcp_server.py**

```
'''
TCPServer by: JOR
Listens for packets on a particular address and port.
Alpha: 13FEB22
'''

import socket
import settings.tcp as settings

TCP_IP = settings.TCP["SERVER_TCP_IPv4"]
TCP_PORT = settings.TCP["SERVER_PORT"]
BUFFER_SIZE = 1024

print(f'This is the TCP server, it will open a port at {TCP_IP}:{TCP_PORT} and being listening')
print(f'Make sure the actual server IP address matches {TCP_IP} in the settings file')
print('This script has no error handling, by design')

try:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((TCP_IP, TCP_PORT))
        print(f'Bound to {TCP_IP}:{TCP_PORT}')
        while True:
            s.listen(1)
            conn, addr = s.accept()
            print(f'Connection address: {addr}')
            data = conn.recv(BUFFER_SIZE).decode()
            print(data)
            conn.send(data.encode())
except socket.error as e:
    print(f'Error: {e}')
```

Testing

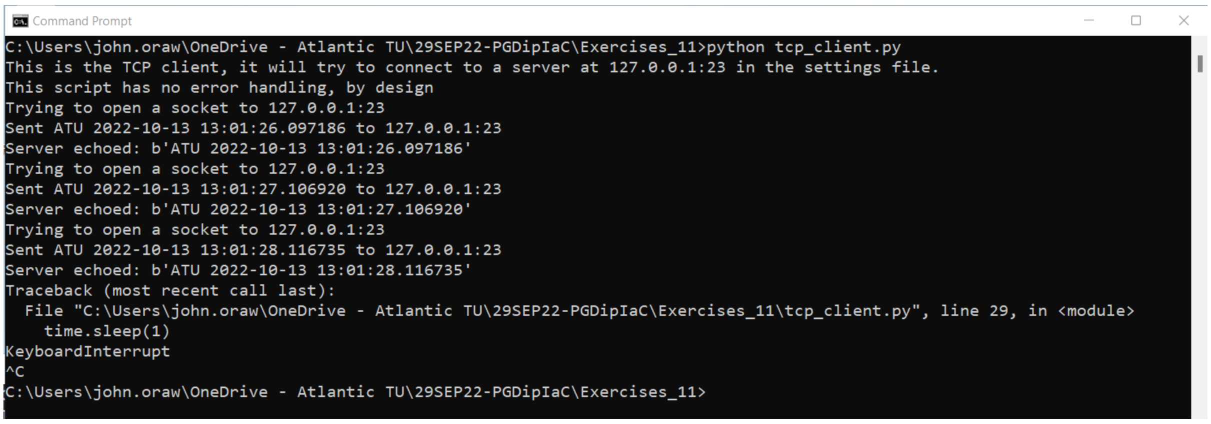
To make this work, I run the programs in two separate console sessions. I must press **[ctrl][c]** to exit. I start the server first; it will listen indefinitely.



Select Command Prompt - python tcp_server.py

```
C:\Users\john.oraw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_11>python tcp_server.py
This is the TCP server, it will open a port at 127.0.0.1:23 and being listening
Make sure the actual server IP address matches 127.0.0.1 in the settings file
This script has no error handling, by design
Bound to 127.0.0.1:23
Connection address: ('127.0.0.1', 49579)
ATU 2022-10-13 13:01:26.097186
Connection address: ('127.0.0.1', 49580)
ATU 2022-10-13 13:01:27.106920
Connection address: ('127.0.0.1', 49581)
ATU 2022-10-13 13:01:28.116735
```

I start the client next. I need to do it in this order. The client tries to make a connection to a definite end point. There is no error handling, so if the connection is not available, it times out and exits.



Command Prompt

```
C:\Users\john.oraw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_11>python tcp_client.py
This is the TCP client, it will try to connect to a server at 127.0.0.1:23 in the settings file.
This script has no error handling, by design
Trying to open a socket to 127.0.0.1:23
Sent ATU 2022-10-13 13:01:26.097186 to 127.0.0.1:23
Server echoed: b'ATU 2022-10-13 13:01:26.097186'
Trying to open a socket to 127.0.0.1:23
Sent ATU 2022-10-13 13:01:27.106920 to 127.0.0.1:23
Server echoed: b'ATU 2022-10-13 13:01:27.106920'
Trying to open a socket to 127.0.0.1:23
Sent ATU 2022-10-13 13:01:28.116735 to 127.0.0.1:23
Server echoed: b'ATU 2022-10-13 13:01:28.116735'
Traceback (most recent call last):
  File "C:\Users\john.oraw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_11\tcp_client.py", line 29, in <module>
    time.sleep(1)
KeyboardInterrupt
^C
C:\Users\john.oraw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_11>
```

I could not exit using **[ctrl][c]**, just close the command window if this is the case.

Whenever you see **b'something'** this indicates that Python is handling <u>bytes</u>. This is normal in communications, but you may have to do some conversions to and from strings.

Telemetry with Multicast

In simple terms, I can do a broadcast on a network. This results in every client on the network receiving and processing the packet. That's a great way to cause congestion on a network! There are cases where the output of process needs to be seen by many other processes on many other nodes. Imagine I have power monitoring equipment in my data center, and I have distributed control to take action based on the power state. For example, if the power is out, and the temperature is OK, I might shut down ventilation fans to reduce the load on the Uninterruptible Power Supply (UPS). One way to do this is to use multicast.

First, I create **Exercises_11\settings\mc.py**

```
MCSERVER = {
    "MCAST_GROUP": '239.1.1.1',
    "IP_ADDRESS": '127.0.0.1',
    "PORT": 5001
}

MCCLIENT = {
    "MCAST_GROUP": '239.1.1.1',
    "IP_ADDRESS": '127.0.0.1',
    "PORT": 5001
}
```

Then I create **Exercises_11\mc_client.py**

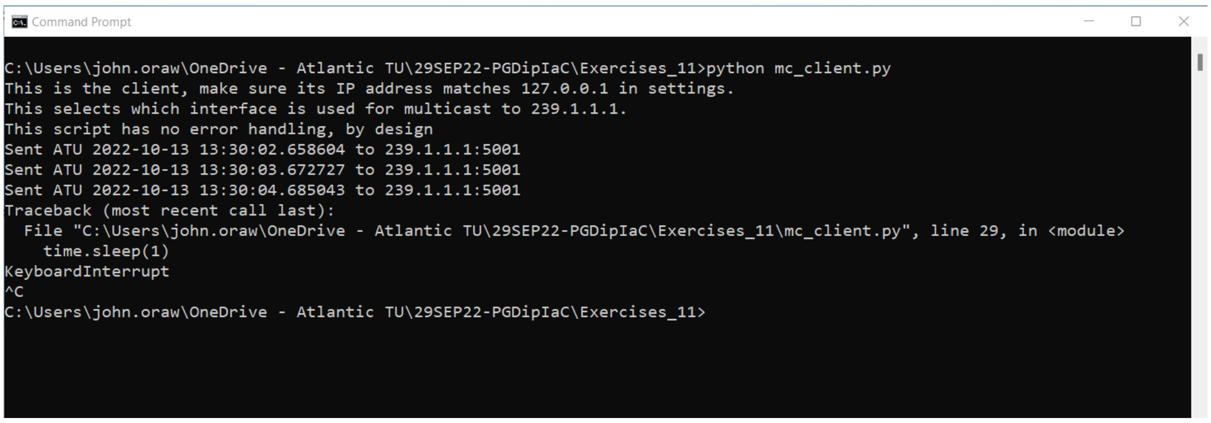
```
'''
Multicast client by: JOR
Sends multicast packets to a particular address and port.
Alpha: 13FEB22
'''
import socket
import time
from datetime import datetime
import settings.mc as settings

# Set multicast information
MCAST_GRP = settings.MCCLIENT["MCAST_GROUP"]
MCAST_PORT = settings.MCCLIENT["PORT"]
MCAST_IF_IP = settings.MCCLIENT["IP_ADDRESS"]

print(f'This is the client, make sure its IP address matches {MCAST_IF_IP} in settings.')
print(f'This selects which interface is used for multicast to {MCAST_GRP}.')
print('This script has no error handling, by design')

while True:
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP) as s:
        # inet_aton converts IPv4 from the a dotted decimal string to 32 bit packed binary format
        s.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, socket.inet_aton(MCAST_GRP) +
socket.inet_aton(MCAST_IF_IP))
        message_text = f"ATU {datetime.now()}"
        message = message_text.encode('utf-8')
        s.sendto(message, (MCAST_GRP, MCAST_PORT))
        print(f'Sent {message_text} to {MCAST_GRP}:{MCAST_PORT}')
        time.sleep(1)
```

Then I create **Exercises_11\mc_server.py**

```
'''
Multicast Server by: JOR
Reads multicast packets from a particular address and port.
Alpha: 13FEB22
'''
import socket
import struct
import settings.mc as settings

# Set multicast information
MCAST_GRP = settings.MCSERVER["MCAST_GROUP"]
SERVER_ADDRESS = ('', settings.MCSERVER["PORT"])
MCAST_IF_IP = settings.MCSERVER["IP_ADDRESS"]

print('This is the server.')
print(f'Make sure its IP address matches {MCAST_IF_IP} in settings.')
print(f'This selects which interface is used to listen for multicast as {MCAST_GRP}.')
print('This script has no error handling, by design.')

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.bind(SERVER_ADDRESS)
    # inet_aton converts IPv4 from the a dotted decimal string to 32 bit packed binary format
    s.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, socket.inet_aton(MCAST_GRP) +
socket.inet_aton(MCAST_IF_IP))

    while True:
        print('Waiting to receive message')
        data, address = s.recvfrom(1024)
        print(f'received {len(data)} bytes from {address}')
        print(data)
```

Testing

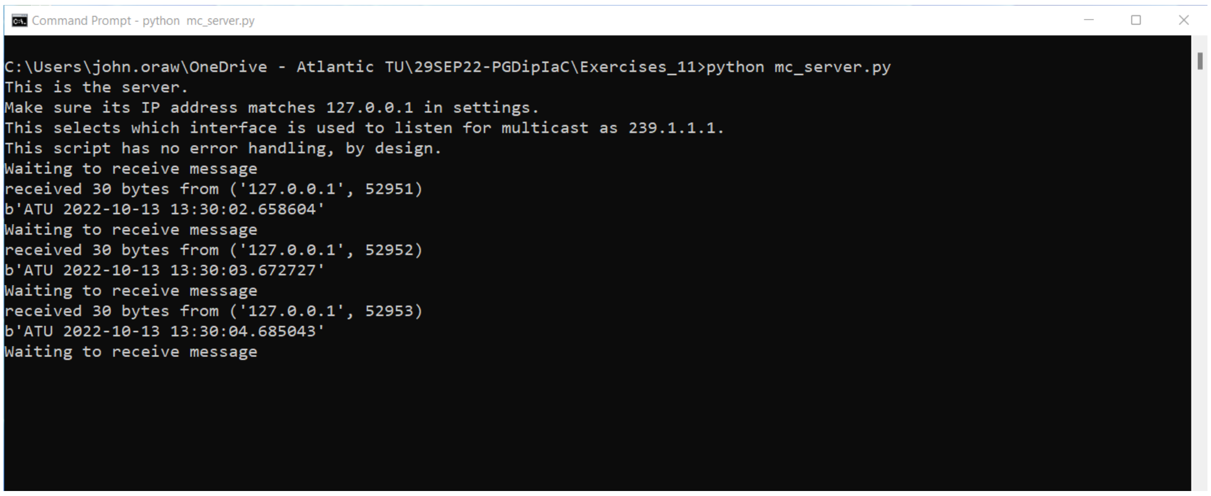
To make this work, I run the programs in two separate console sessions. I must press **[ctrl][c]** to exit. I start the client sending packets.



Command Prompt

```
C:\Users\john.oraw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_11>python mc_client.py
This is the client, make sure its IP address matches 127.0.0.1 in settings.
This selects which interface is used for multicast to 239.1.1.1.
This script has no error handling, by design
Sent ATU 2022-10-13 13:30:02.658604 to 239.1.1.1:5001
Sent ATU 2022-10-13 13:30:03.672727 to 239.1.1.1:5001
Sent ATU 2022-10-13 13:30:04.685043 to 239.1.1.1:5001
Traceback (most recent call last):
  File "C:\Users\john.oraw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_11\mc_client.py", line 29, in <module>
    time.sleep(1)
KeyboardInterrupt
^C
C:\Users\john.oraw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_11>
```

I run the server next.



Command Prompt - python mc_server.py

```
C:\Users\john.oraw\OneDrive - Atlantic TU\29SEP22-PGDipIaC\Exercises_11>python mc_server.py
This is the server.
Make sure its IP address matches 127.0.0.1 in settings.
This selects which interface is used to listen for multicast as 239.1.1.1.
This script has no error handling, by design.
Waiting to receive message
received 30 bytes from ('127.0.0.1', 52951)
b'ATU 2022-10-13 13:30:02.658604'
Waiting to receive message
received 30 bytes from ('127.0.0.1', 52952)
b'ATU 2022-10-13 13:30:03.672727'
Waiting to receive message
received 30 bytes from ('127.0.0.1', 52953)
b'ATU 2022-10-13 13:30:04.685043'
Waiting to receive message
```

I could not exit using **[ctrl][c]**, just close the command window if this is the case.

Finally

Sorry for dumping network terminology on you when we may not yet have covered networking!

In my examples, I used the loopback address 127.0.0.1 to get around Windows Firewall and the security restrictions in the VDI. If you can run these programs on two or more virtual machines, it is a much better way to test things. I am a big fan of Raspberry Pi, and I will often use a rack of these for testing. Many of my production instrumentation systems end up on a Raspberry Pi or similar.

This code is based on functions I have used for years for testing client-server applications. Around 2013 I started writing code to control automation infrastructure; think about all the things that are required in the electrical and cooling parts of a data center.

This code has been incrementally improved, but always with the goal of keeping it as simple and stable as possible. I made all sorts of excursions into multithreaded code and that all worked. I also did versions based on object-oriented code.

I want to keep things as simple and reliable as possible, the code here works fine. Versions of this code can be found operating on many instrumentation systems in many countries.