NETWORK FUNDAMENTALS AND SOCKET PROGRAMMING (TCP/UDP)

CS 3030: Python

Instructor: Damià Fuentes Escoté

UCCS University of Colorado Colorado Springs

TCP (Transmission Control Protocol)

- TCP stands for Transmission Control Protocol. It is the most commonly used protocol on the Internet.
- Data through internet is transmitted in packets. TCP guarantees the recipient will receive the packets in order by numbering them. The recipient sends messages back to the sender saying it received the messages. If the sender does not get a correct response, it will resend the packets to ensure the recipient received them. Packets are also checked for errors.
- TCP is all about this reliability — packets sent with TCP are tracked so no data is lost or corrupted in transit. This is why file downloads do not become corrupted even if there are network hiccups. Of course, if the recipient is completely offline, your computer will give up and you will see an error message saying it can not communicate with the remote host.

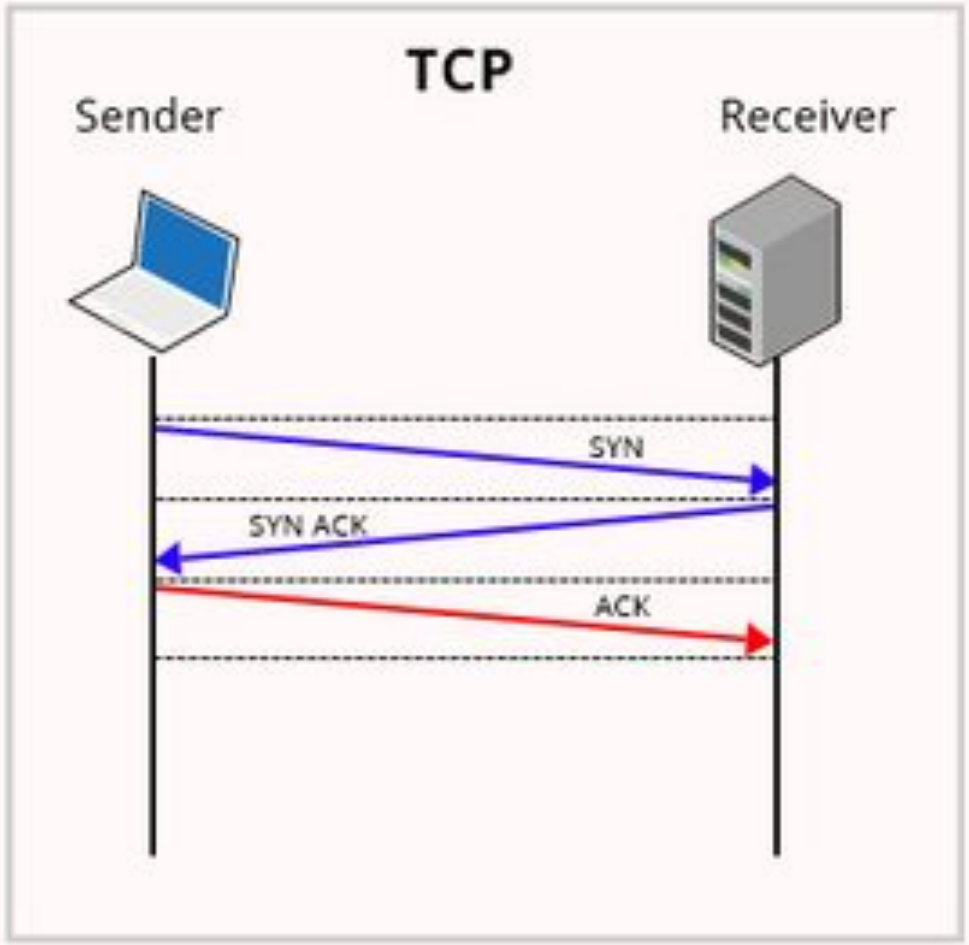
TCP (Transmission Control Protocol)

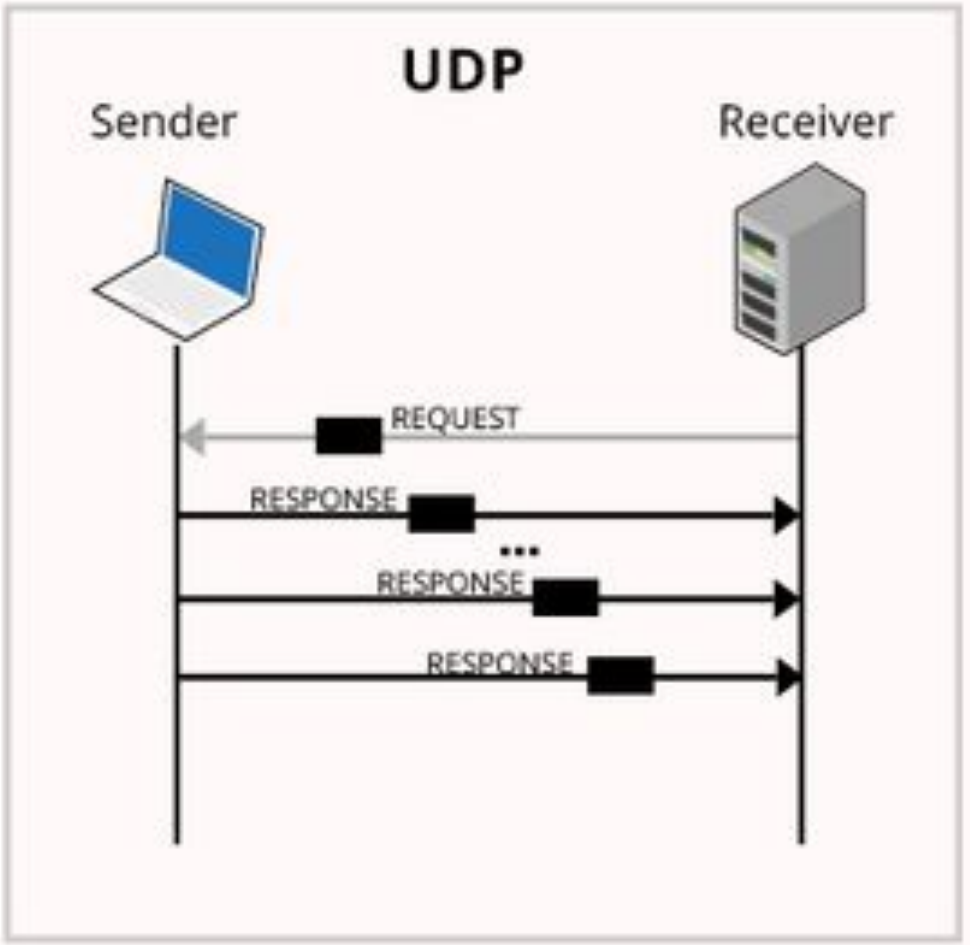
- **Is reliable**
  - *Packets dropped in the network are detected and retransmitted by the sender.*
- **Has in-order data delivery**
  - *Data is read by your application in the order it was written by the sender.*

UDP (User Datagram Protocol)

- UDP stands for User Datagram Protocol — a datagram is the same thing as a packet of information. The UDP protocol works similarly to TCP, but it throws all the error-checking stuff out. All the back-and-forth communication and deliverability guarantees slow things down.
- When using UDP, packets are just sent to the recipient. The sender will not wait to make sure the recipient received the packet — it will just continue sending the next packets. If you are the recipient and you miss some UDP packets, too bad — you can not ask for those packets again. There is no guarantee you are getting all the packets and there is no way to ask for a packet again if you miss it, **but losing all this overhead means the computers can communicate more quickly.**
- UDP is used when speed is desirable and error correction is not necessary. For example, UDP is frequently used for live broadcasts and online games.

TCP Vs UDP Communication





TCP	UDP
Reliable	Unreliable
Connection-oriented	Connectionless
Segment retransmission and flow control through windowing	No windowing or retransmission
Segment sequencing	No sequencing
Acknowledge segments	No acknowledgement

Socket background

- Sockets have a long history. Their use originated with ARPANET in 1971. Today, although the underlying protocols used by the socket API have evolved over the years, and we've seen new ones, the low-level API has remained the same.

What is a socket?

- Sockets allow communication between two different processes on the same or different machines.
  - *To be more precise, it's a way to talk to other computers using standard Unix file descriptors.*
  - *To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as read() and write() work with sockets in the same way they do with files.*

Where is sockets used?

- A Unix Socket is used in a client-server application framework. A server is a process that performs some functions on request from a client. **Most of the application-level** protocols like FTP, SMTP, and POP3 **make use of sockets to establish connection between client and server and then for exchanging data**.
  - This is the type of application that we will be covering.

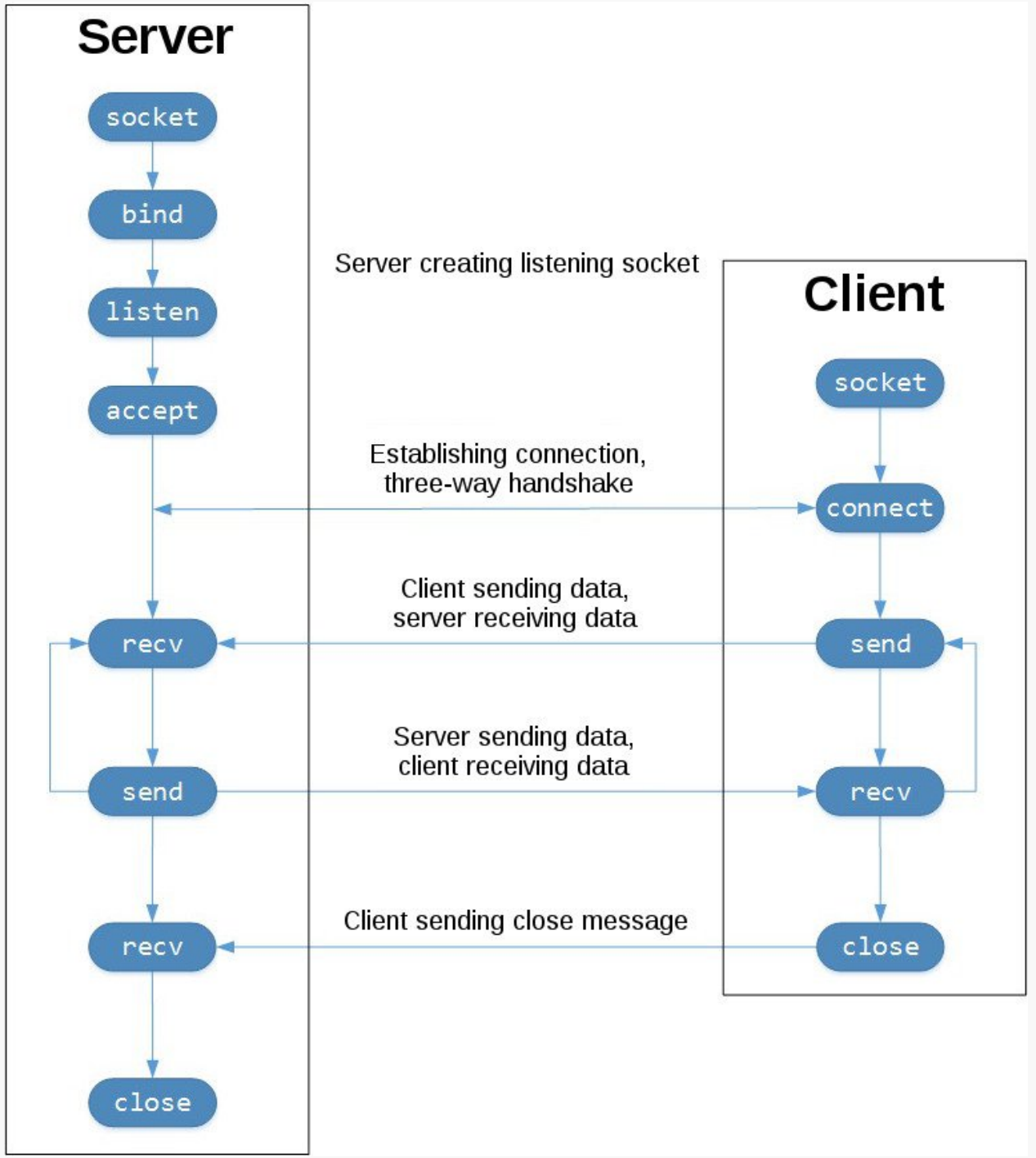
Socket API Overview

- socket()
- bind()
- listen()
- accept()
- connect()
- connect_ex()
- send()
- recv()
- close()

Socket types

- **Stream Sockets** − Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order − "A, B, C". These sockets use TCP for data transmission. If delivery is impossible, the sender receives an error indicator.
  - socket.SOCK_STREAM
- **Datagram Sockets** − Delivery in a networked environment is not guaranteed. They're connectionless because you don't need to have an open connection as in Stream Sockets − you build a packet with the destination information and send it out. They use UDP.
  - socket.SOCK_DGRAM
- **Raw Sockets**
- **Sequenced Packet Sockets**

TCP Socket flow



TCP Socket flow

- The left-hand column represents the server. On the right-hand side is the client.
- Starting in the top left-hand column, note the API calls the server makes to setup a “listening” socket:
  - *socket()*
  - *bind()*
  - *listen()*
  - *accept()*
- A listening socket does just what it sounds like. It listens for connections from clients. When a client connects, the server calls accept() to accept, or complete, the connection.
- The client calls connect() to establish a connection to the server and initiate the three-way handshake. The handshake step is important since it ensures that each side of the connection is reachable in the network, in other words that the client can reach the server and vice-versa. It may be that only one host, client or server, can reach the other.
- In the middle is the round-trip section, where data is exchanged between the client and server using calls to send() and recv().
- At the bottom, the client and server close() their respective sockets.

TCP Echo Client and Server

- Let’s create our first client and server.
- We’ll begin with a simple implementation. The server will simply echo whatever it receives back to the client.

TCP Echo server

```
import socket

print('Starting echo-server’)

HOST = '127.0.0.1’
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    client_sock, addr = s.accept()
    with client_sock :
        print('Connected by', addr)
        while True:
            data = client_sock.recv(1024)
            if not data:
                break
            client_sock.sendall(data)
```

TCP Echo server

```
import socket

print('Starting echo-server’)

HOST = '127.0.0.1’
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    client_sock, addr = s.accept()
    with client_sock :
        print('Connected by', addr)
        while True:
            data = client_sock.recv(1024)
            if not data:
                break
            client_sock.sendall(data)
```

The server's hostname or IP address

TCP Echo server

```
import socket

print('Starting echo-server’)

HOST = '127.0.0.1’
PORT = 65432
```

The port used by the server

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    client_sock, addr = s.accept()
    with client_sock :
        print('Connected by', addr)
        while True:
            data = client_sock.recv(1024)
            if not data:
                break
            client_sock.sendall(data)
```

TCP Echo server

```
import socket

print('Starting echo-server’)

HOST = '127.0.0.1’
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    client_sock, addr = s.accept()
    with client_sock :
        print('Connected by', addr)
        while True:
            data = client_sock.recv(1024)
            if not data:
                break
            client_sock.sendall(data)
```

socket.socket() creates a socket object that supports the context manager type, so you can use it in a with statement. There’s no need to call s.close()

TCP Echo server

```
import socket

print('Starting echo-server’)

HOST = '127.0.0.1’
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    client_sock, addr = s.accept()
    with client_sock :
        print('Connected by', addr)
        while True:
            data = client_sock.recv(1024)
            if not data:
                break
            client_sock.sendall(data)
```

The arguments passed to socket() specify the address family and socket type. AF_INET is the Internet address family for IPv4.

SOCK_STREAM is the socket type for TCP, the protocol that will be used to transport our messages in the network.

TCP Echo server

```
import socket

print('Starting echo-server')

HOST = '127.0.0.1'
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    client_sock, addr = s.accept()
    with client_sock :
        print('Connected by', addr)
        while True:
            data = client_sock.recv(1024)
            if not data:
                break
            client_sock.sendall(data)
```

bind() is used to associate the socket with a specific network interface and port number:

- host can be a hostname or an IP address
- port should be an integer from 1-65535 (0 is reserved). It's the TCP port number to accept connections on from clients. Some systems may require superuser privileges if the port is < 1024.

TCP Echo server

```
import socket

print('Starting echo-server')

HOST = '127.0.0.1'
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    client_sock, addr = s.accept()
    with client_sock :
        print('Connected by', addr)
        while True:
            data = client_sock.recv(1024)
            if not data:
                break
            client_sock.sendall(data)
```

listen() enables a server to accept() connections. It makes it a "listening" socket.

listen() has a backlog parameter. It specifies the number of unaccepted connections that the system will allow before refusing new connections. Starting in Python 3.5, it's optional. If not specified, a default reasonable backlog value is chosen.

TCP Echo ser

```
import socket

print('Starting echo-server’)

HOST = '127.0.0.1’
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    client_sock, addr = s.accept()
    with client_sock :
        print('Connected by', addr)
        while True:
            data = client_sock.recv(1024)
            if not data:
                break
            client_sock.sendall(data)
```

accept() blocks and waits for an incoming connection. When a client connects, it returns a new socket object representing the connection and a tuple holding the address of the client. The tuple will contain (host, port) for IPv4 connections.

One thing that’s imperative to understand is that we now have a new socket object from accept(). This is important since it’s the socket that you’ll use to communicate with the client. It’s distinct from the listening socket that the server is using to accept new connections.

TCP Echo ser

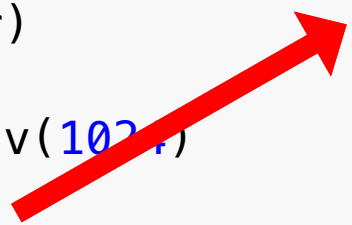
```
import socket

print('Starting echo-server’)

HOST = '127.0.0.1’
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    client_sock, addr = s.accept()
    with client_sock :
        print('Connected by', addr)
        while True:
            data = client_sock.recv(1024)
            if not data:
                break
            client_sock.sendall(data)
```

After getting the client socket object conn from accept(), an infinite while loop is used to loop over blocking calls to conn.recv(). This reads whatever data the client sends and echoes it back using conn.sendall().

If conn.recv() returns an empty bytes object, b' ', then the client closed the connection and the loop is terminated. The with statement is used with conn to automatically close the socket at the end of the block.

TCP Echo server

```
import socket

print('Starting echo-server’)

HOST = '127.0.0.1’
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    client_sock, addr = s.accept()
    with client_sock :
        print('Connected by', addr)
        while True:
            data = client_sock.recv(1024)
            if not data:
                break
            client_sock.sendall(data)
```



For example, if we want to develop a server without using any framework, we would process the received data and return something else that the client was asking for.

TCP Echo client

```
import socket

print('Starting echo-client’)

HOST = '127.0.0.1'
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world’)
    data = s.recv(1024)

print('Received', repr(data))
```

TCP Echo client

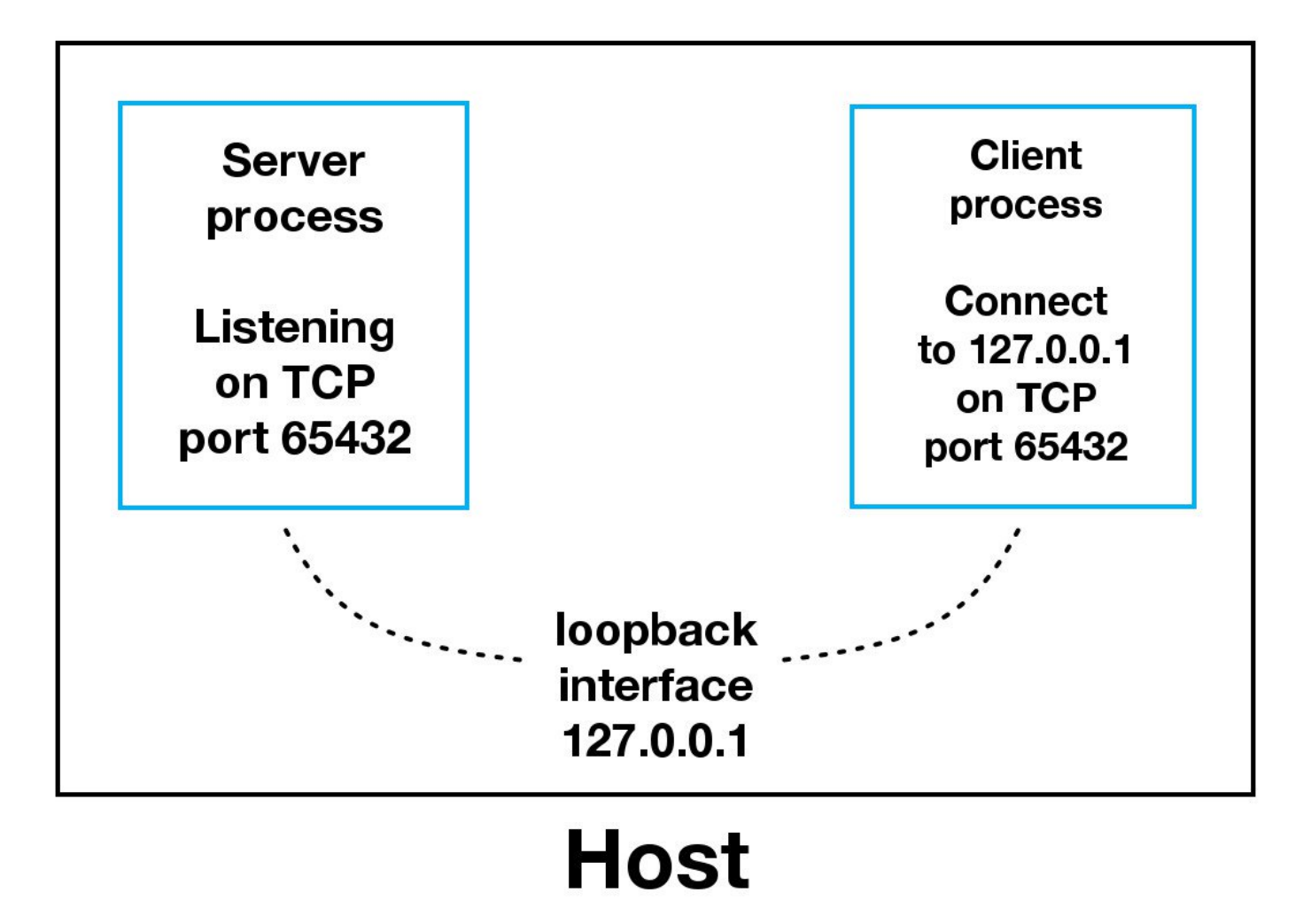
```
import socket

print('Starting echo-client’)

HOST = '127.0.0.1'
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world’)
    data = s.recv(1024)

print('Received', repr(data))
```

In comparison to the server, the client is pretty simple. It creates a socket object, connects to the server and calls s.sendall() to send its message. Lastly, it calls s.recv() to read the server’s reply and then prints it.

Running the TCP Echo Client and Server

- In the terminal:
  - *python3 echo-server.py*
  - *Your terminal will appear to hang. That's because the server is blocked (suspended) in the call* `conn, addr = s.accept()`
- In another terminal:
  - *python3 echo-client.py*

Communication Breakdown



Localhost, private IPs and public IPs

- Till now we have run our server on localhost (127.0.0.1), but we can also run our server over other IPs.
- To know our **local IP** (also named as **private IP**):
  - *Mac: In terminal we can run ifconfig*
  - *Windows: ipconfig*
  - *Python:* `socket.gethostbyname(socket.gethostname())`
- Then, we change our HOST variables in our echo-server.py and our echo-client.py to our IP.
  - *Now, a client could access the server from another computer in the same network.*

Localhost, private IPs and public IPs

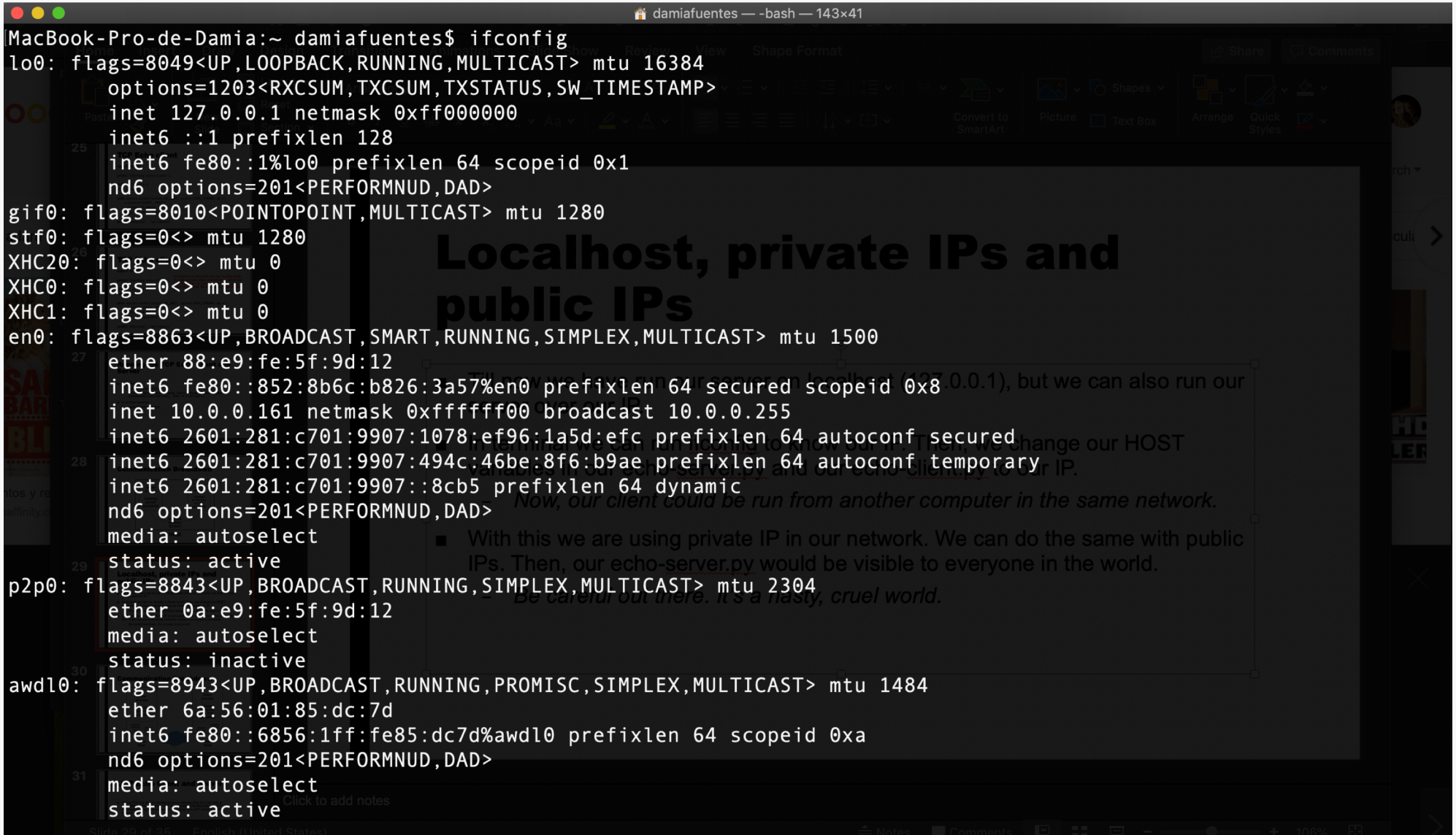
Be careful when you retrieve your local IPs. Some networks (like UCCS-Wireless) uses directly public IPs.

PRIVATE IP ADDRESS

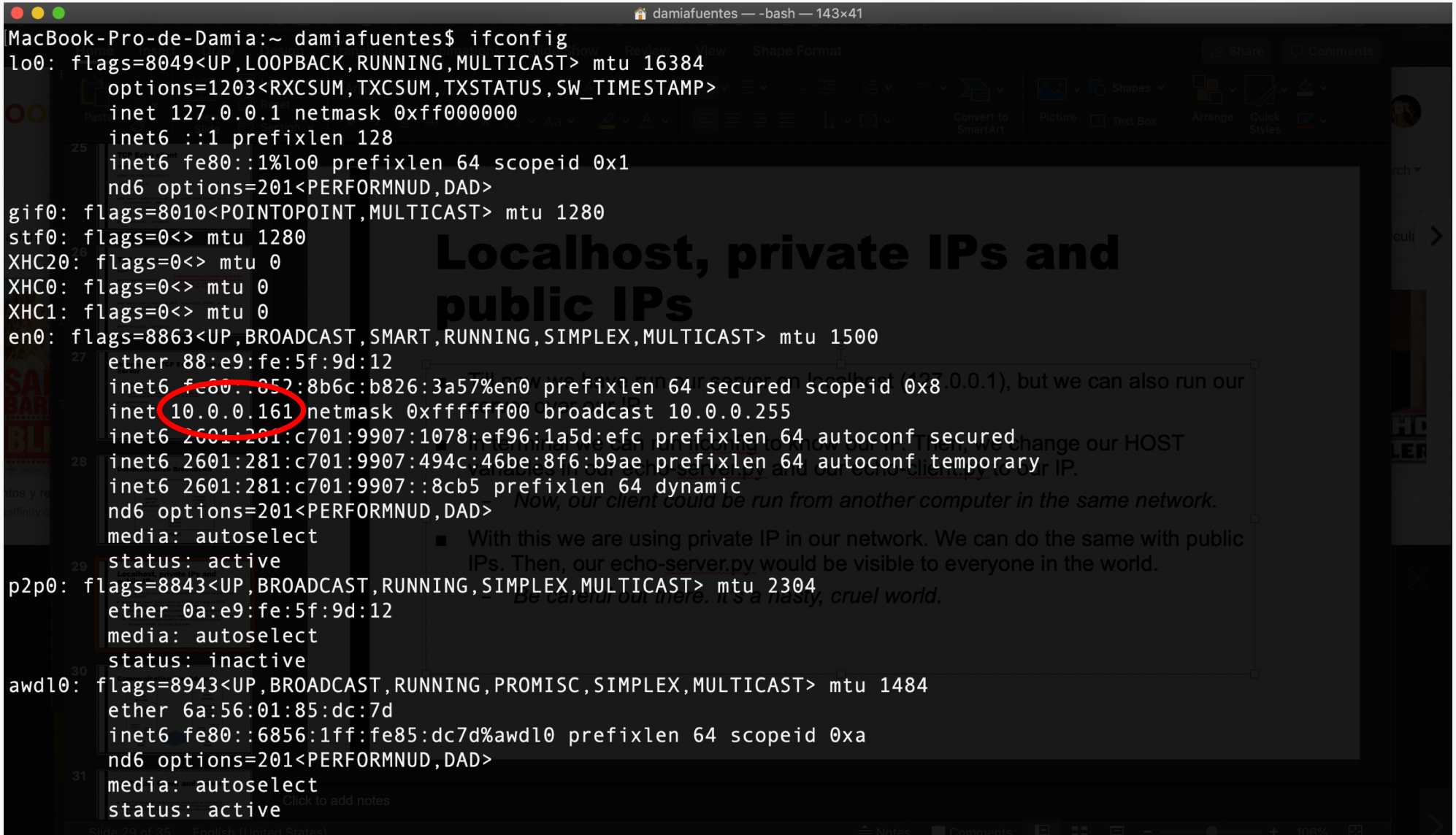
- The Internet Assigned Numbers Authority (IANA) has reserved the following three blocks of the IP address space for private internets (local networks):

10.0.0.0	-	10.255.255.255
172.16.0.0	-	172.31.255.255
192.168.0.0	-	192.168.255.255

Local IP


damiafuentes — -bash — 143×41
MacBook-Pro-de-Damia:~ damiafuentes$ ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
options=1203<RXCSUM,TXCSUM,TXSTATUS,SW_TIMESTAMP>
inet 127.0.0.1 netmask 0xff000000
inet6 ::1 prefixlen 128
inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
nd6 options=201<PERFORMNUD,DAD>
gif0: flags=8010<POINTOPOINT,MULTICAST> mtu 1280
stf0: flags=0<> mtu 1280
XHC20: flags=0<> mtu 0
XHC0: flags=0<> mtu 0
XHC1: flags=0<> mtu 0
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
ether 88:e9:fe:5f:9d:12
inet6 fe80::852:8b6c:b826:3a57%en0 prefixlen 64 secured scopeid 0x8
inet 10.0.0.161 netmask 0xffffff00 broadcast 10.0.0.255
inet6 2601:281:c701:9907:1078:ef96:1a5d:efc prefixlen 64 autoconf secured
inet6 2601:281:c701:9907:494c:46be:8f6:b9ae prefixlen 64 autoconf temporary
inet6 2601:281:c701:9907::8cb5 prefixlen 64 dynamic
nd6 options=201<PERFORMNUD,DAD>
media: autoselect
status: active
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
ether 0a:e9:fe:5f:9d:12
media: autoselect
status: inactive
awdl0: flags=8943<UP,BROADCAST,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1484
ether 6a:56:01:85:dc:7d
inet6 fe80::6856:1ff:fe85:dc7d%awdl0 prefixlen 64 scopeid 0xa
nd6 options=201<PERFORMNUD,DAD>
media: autoselect
status: active
Localhost, private IPs and public IPs
Now, our client could be run from another computer in the same network.
With this we are using private IP in our network. We can do the same with public IPs. Then, our echo-server.py would be visible to everyone in the world.
Be careful out there. It's a nasty, cruel world.

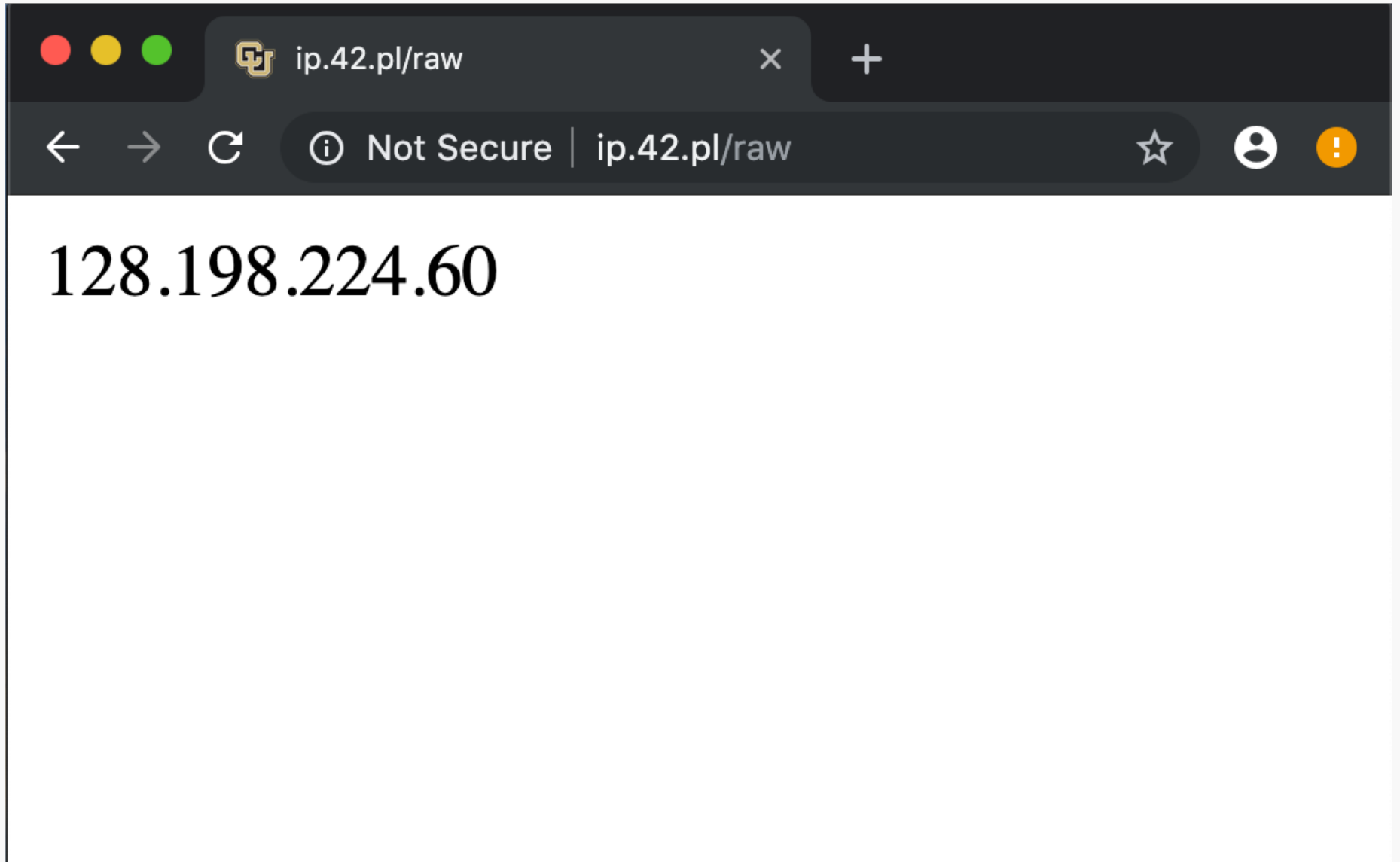
Local IP


damiafuentes — -bash — 143×41
MacBook-Pro-de-Damia:~ damiafuentes$ ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
options=1203<RXCSUM,TXCSUM,TXSTATUS,SW_TIMESTAMP>
inet 127.0.0.1 netmask 0xff000000
inet6 ::1 prefixlen 128
inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
nd6 options=201<PERFORMNUD,DAD>
gif0: flags=8010<POINTOPOINT,MULTICAST> mtu 1280
stf0: flags=0<> mtu 1280
XHC20: flags=0<> mtu 0
XHC0: flags=0<> mtu 0
XHC1: flags=0<> mtu 0
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
ether 88:e9:fe:5f:9d:12
inet6 fe80::852:8b6c:b826:3a57%en0 prefixlen 64 secured scopeid 0x8
inet 10.0.0.161 netmask 0xffffff00 broadcast 10.0.0.255
inet6 2601:281:c701:9907:1078:ef96:1a5d:efc prefixlen 64 autoconf secured
inet6 2601:281:c701:9907:494c:46be:8f6:b9ae prefixlen 64 autoconf temporary
inet6 2601:281:c701:9907::8cb5 prefixlen 64 dynamic
nd6 options=201<PERFORMNUD,DAD>
media: autoselect
status: active
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
ether 0a:e9:fe:5f:9d:12
media: autoselect
status: inactive
awdl0: flags=8943<UP,BROADCAST,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1484
ether 6a:56:01:85:dc:7d
inet6 fe80::6856:1ff:fe85:dc7d%awdl0 prefixlen 64 scopeid 0xa
nd6 options=201<PERFORMNUD,DAD>
media: autoselect
status: active
Localhost, private IPs and public IPs
Now, our client could be run from another computer in the same network.
With this we are using private IP in our network. We can do the same with public IPs. Then, our echo-server.py would be visible to everyone in the world.
Be careful out there. It's a nasty, cruel world.

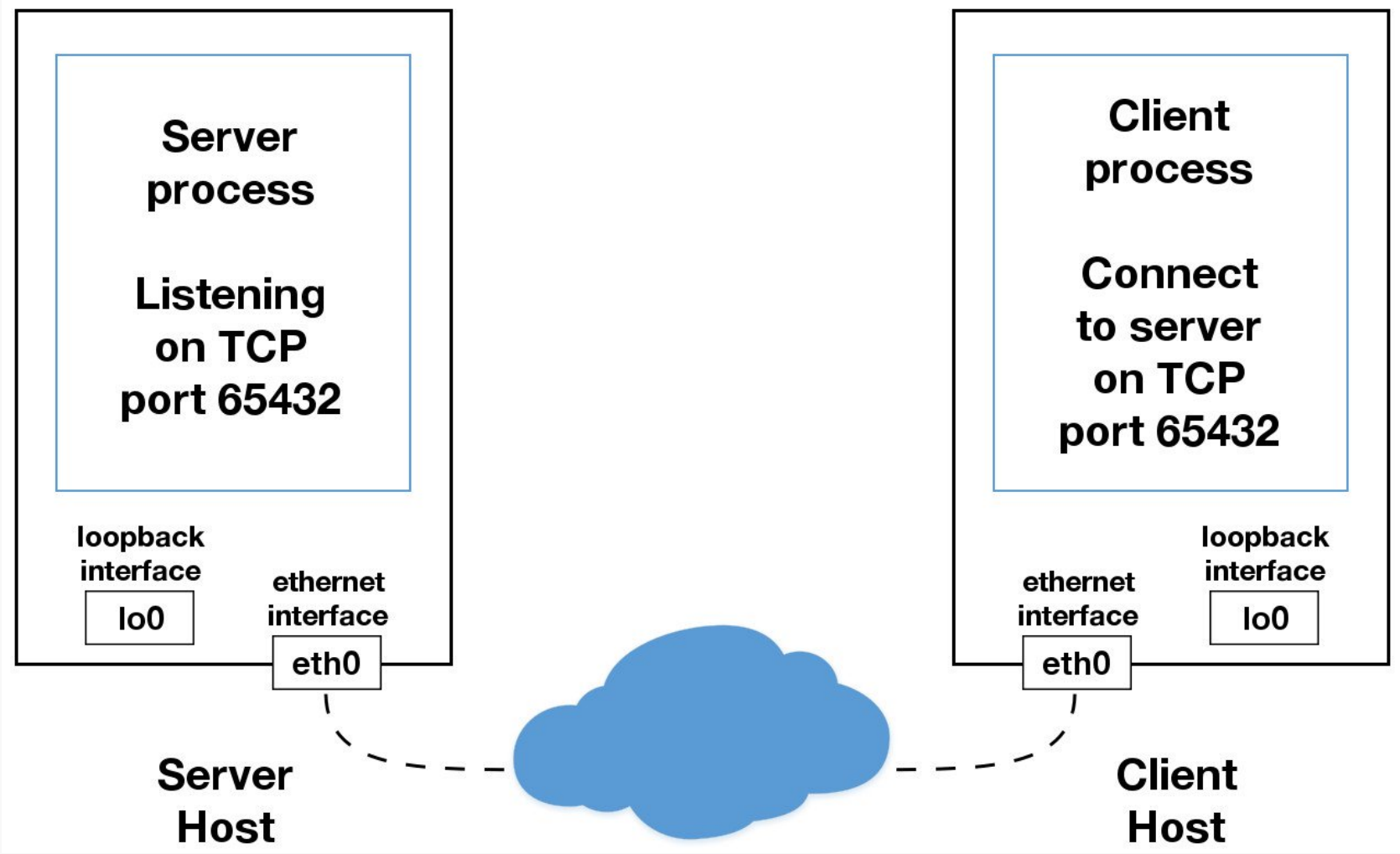
Localhost, local IPs and public IPs

- We can do the same with **public IPs**. Then, our echo-server.py would be visible to everyone in the world.
  - *Python:* `requests.get("http://ip.42.pl/raw").text`
  - *Be careful out there. It's a nasty, cruel world.*

Public IP



Communication Breakdown



UDP Echo Client and Server

- Where TCP is a *stream oriented* protocol, ensuring that all of the data is transmitted in the right order, UDP is a *message oriented* protocol. UDP does not require a long-lived connection, so setting up a UDP socket is a little simpler.

UDP Echo Server

```
print('Starting udp echo-server’)

HOST = '127.0.0.1’
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.bind((HOST, PORT))
    while True:
        data, addr = s.recvfrom(4096)
        if data:
            print("Data received, sending it back.", data)
            sent = s.sendto(data, addr)
```

UDP Echo Server

```
print('Starting udp echo-server’)

HOST = '127.0.0.1’
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.bind((HOST, PORT))
    while True:
        data, addr = s.recvfrom(4096)
        if data:
            print("Data received, sending it back.", data)
            sent = s.sendto(data, addr)
```

Similar to the TCP Echo Server, but it doesn’t listen and accept connections.

This reads whatever data the client sends and echoes it back to the same client.

UDP Echo Client

```
import socket

print('Starting udp echo-client’)

HOST = '127.0.0.1’
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    message = b'This is the message.  It will be repeated to me.’
    sent = s.sendto(message, (HOST, PORT))
    data, server = s.recvfrom(4096)

print('Received', repr(data))
```

Further reading

- We just have covered the basics of socket programing. There is a lot more like Multi-Connection Client and Server. If you want to learn more:
  - *https://realpython.com/python-sockets/#tcp-sockets*

What you have to understand from this class

- Two protocols:
  - *UDP/TCP*
  - *Which one we should use depending on what we what.*
- High-level libraries like Django, Flask, Google cloud, Amazon Web Services, etc. are based on what we have done today. That will be better seen on the homework.

Time to code – Homework 9 Exercise 3

- Write a TCP Server and a TCP Client. The server will have a string variable that will be changed with values sent by the client. The client will ask the user for the following options:
  1. *Retrieve the text in the server*
  2. *Update the text in the server*
  3. *Close*
- On option 1 the client will ask the server for the text and the server will send the text to the client.
- On option 2 the client will ask the user for some input text and that text will be sent to the server, which will update the text variable.
- On option 3 the client will tell the server to close the socket and both client and server will close the socket connection.

Next two classes

- **11/4/18 -> Quiz 4**
- Develop a Django RESTful Server
  - *Slides*
  - *Homework/tutorial -> Extensive task that can take some hours*
    -> We are going to start it in class
    -> The second class is meant to work on it
  - *The topic can be very useful for most of you final projects, your own ideas, etc*
  - *We are going to develop a simple Instagram server with users and posts.*