Unidad 11

Redes y Consultas Remotas

Manual del Estudiante

Maria del Carmen Sierra Fernández

Contenido

Nivel III: Aplicaciones Web

Unidad 11: Redes y Consultas Remotas

Como funciona Internet

Usar Internet parece bastante fácil. Escribimos a una dirección web y nos muestra una página. O vamos a nuestro red social u podemos ver fotos de nuestros amigos, familiares y mascotas. Pero se necesita mucho de software y hardware complejos para hacer que Internet parezca tan sencillo. El diseño de las tecnologías que hacen que Internet de hoy, comenzó en la década de 1960, y hubo más de 20 años de investigación sobre cómo construir tecnologías de interconexión de redes antes que la primera **Internet** , que fue construida a finales de la década de 1980. Desde entonces, la investigación y el desarrollo en la mejora de las tecnologías de red ha continuado, las redes se han vuelto más grandes, más rápidas y están distribuidas globalmente en miles de millones de ordenadores.

La innovación más importante que permitió que los mensajes se movieran más rápidamente a través de una red de varios saltos fue romper cada mensaje en pequeños fragmentos y enviar cada fragmento individualmente. En términos de redes, estos mensajes se denominan **paquetes**. La división del mensaje en paquetes redujo en gran medida la cantidad de almacenamiento necesaria en los ordenadores intermedios porque en lugar de necesitar almacenar un mensaje completo durante unas pocas horas, el ordenador intermedio solo necesita almacenar unos pocos paquetes durante unos segundos mientras los paquetes esperan sus turnos en el enlace de salida. Posteriormente estos ordenadores dedicados a las comunicaciones se denominaron **routers** o "Enrutadores" porque su propósito era enrutar los paquetes recibidos hacia su destino final.

Los **routers** o “enrutadores" saben cómo enrutar paquetes a lo largo de una ruta desde una origen a un destino. Cada paquete pasará por múltiples enrutadores durante su viaje desde el ordenador de origen hasta el ordenador destino. Aunque los paquetes pueden ser parte de un mensaje más grande, Los enrutadores envían cada paquete por separado según su origen y direcciones de destino. Diferentes paquetes del mismo mensaje pueden tomar diferentes rutas desde el origen hasta el destino. Y a veces, los paquetes llegan incluso desordenados; un paquete posterior podría llegan antes que un paquete anterior, quizás debido a un **traffic jam**. Cada paquete contiene un **offset** desde el comienzo del mensaje para que el ordenador destino pueda volver a ensamblar los paquetes en el orden correcto y reconstruir el mensaje original.

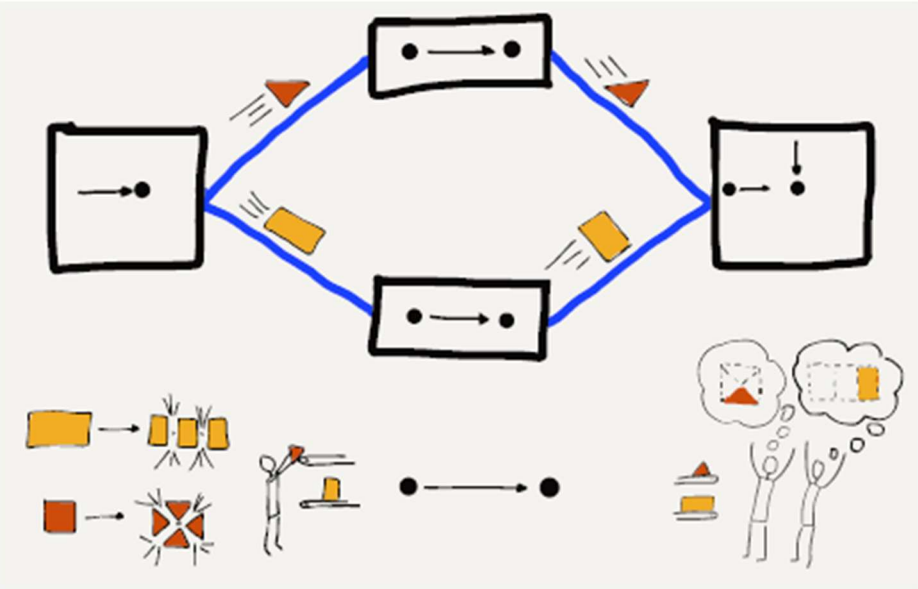


Figura 1: Envío de paquetes

Normalmente, los paquetes encontrarían la ruta más corta entre el origen y el destino, pero si un enlace en esa ruta era un sobrecargado o roto, los enrutadores podrían cooperar y desviar tráfico para tomar rutas un poco más largas que obtendrían paquetes de un origen a un destino lo más rápido posible. El núcleo de Internet es un conjunto de enrutadores cooperantes que se mueven paquetes de muchas fuentes a muchos destinos al mismo tiempo. Cada ordenador o red de área local está conectada a un enrutador que reenvía el tráfico desde su ubicación a los distintos destinos en Internet. Un enrutador puede manejar datos de un solo ordenador como un teléfono inteligente, de varios ordenadores en el mismo edificio, o desde miles de ordenadores conectados a la red de la universidad.

El término **Internet** proviene de la idea de **Internetworking**, que captura la idea de conectar muchas redes. Nuestros ordenadores se conectan a redes locales e Internet conecta las redes locales, por lo que todos nuestros ordenadores pueden hablar entre sí.

Arquitectura de la red

Para diseñar y construir un sistema tan complejo como Internet, los ingenieros intentan dividir un solo problema desafiante en un conjunto de problemas más pequeños que se pueden resolver de forma independiente y luego volver a unirlos para resolver el gran problema original. Los ingenieros que construyeron las primeras redes dividieron el problema general en cuatro subproblemas básicos en los que diferentes grupos podrían trabajar de forma independiente. Les dieron a estas cuatro áreas de la ingeniería los siguientes nombres: **Link o Enlace**, **Internetwork**, **Transport** o Transporte y **Application** o Aplicación.

Figura 2: Capas del modelo TCP/IP

Visualizamos estas áreas como capas apiladas una encima de la otra, con la capa de **Link** o **Enlace** en la parte inferior y la capa de **Aplicación** en la parte superior. La capa de enlace se ocupa de la conexión por cable o inalámbrica desde su computadora a la red de área local y la capa de aplicación es a través de la cual nosotros, como usuarios finales, interactuamos. Un navegador web es un ejemplo de una aplicación en esta arquitectura de Internet.

Informalmente nos referimos a este modelo como el **modelo TCP/IP** en referencia al **Protocolo de Control de Transporte (TCP)** usado para implementar la **capa de Transporte** y el **Protocolo de Internet (IP)** usado para implementar la capa de Internetwork.

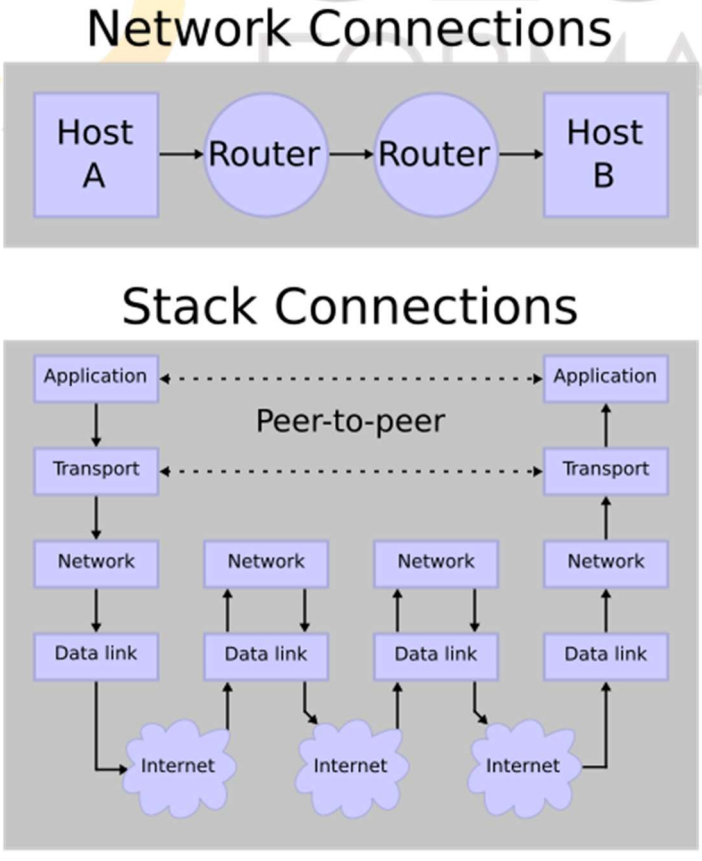


Figura 3: Arquitectura de la red

En la figura 3, vemos esta arquitectura en capas que básicamente representa la conexión a través de Internet. Representa dos ordenadores, donde una aplicación, habla con la capa de transporte, que habla con una capa de Internet, que habla con la capa de enlace y que habla con el **WiFi o Ethernet conexión**. Pasa a través de la toda la red y llega al final del destino que podría ser un servidor web. Sube a través de las capas enlace, transporte y luego ejecuta una aplicación web. Esta aplicación web, luego envía los datos de vuelta, que pasan a través de las capas y llega de nuevo a nuestra aplicación y a nuestra pantalla.

La primera capa se conoce como la **capa de extremo a extremo**, en la arquitectura de red es la **capa de transporte**. En esta Unidad, la idea es, escribir un programa Python que hable a través de la capa de transporte con una aplicación en otro servidor y recupere la información. Por lo tanto, la capa de Internet y la capa de enlace implementan la capa de transporte. Por ahora, vamos a fingir que estas, son un tubo de extremo a extremo que cuando nuestro programa dice algo a través del tubo, el programa en el otro extremo lo escucha, y cuando ese programa envía una respuesta, podemos escucharla.

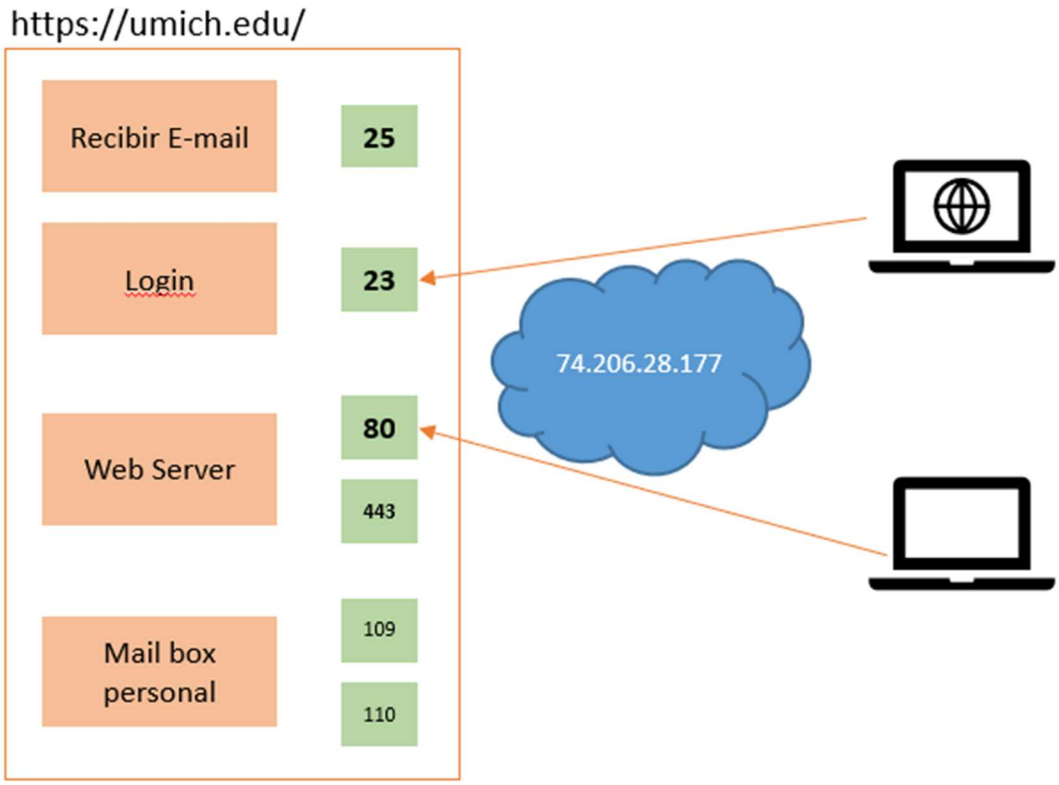
Cuando nuestro programa Python en un extremo se conecta a un servidor de red en el otro extremo, tenemos dos piezas de software ejecutándose en dos ordenadores independientes, donde realmente estas aplicaciones están chateando y/o intercambiando información. A esta conexión le llamamos **socket**. Un **socket** es un canal de comunicación que se construye en Internet el tiempo suficiente para tener esta conversación y luego se elimina.

Otras definiciones más formales de socket son:

- Un **socket**, es un método para la comunicación entre un programa del cliente y un programa del servidor en una red.
- Un **socket** se define como el punto final en una conexión. Los sockets se crean y se utilizan con un sistema de peticiones o de llamadas de función a veces llamados interfaz de programación de aplicación de sockets (**API, Application Programming Interface**).
- Un **socket** es también una dirección de Internet, combinando una **dirección IP** (la dirección numérica única de cuatro partes que identifica a un ordenador particular en Internet) y un **número de puerto** (el número que identifica una aplicación de Internet particular, como FTP, Gopher, o WWW).

La forma en que funcionan los sockets es: cuando un socket quiere establecer una comunicación dice, “Me gustaría hablar con este servidor web en particular”. Por lo tanto, las diferentes aplicaciones en ese servidor están escuchando, en lo que se llaman puertos. Estos puertos TCPIP, son pequeños puertos a los cuales están asociadas las diferentes aplicaciones.

Por ejemplo, el servidor de correo electrónico entrante podría estar en el puerto 25, se despierta y espera que algo suceda.



El puerto que utilizaremos en este Unidad es el puerto 80. El puerto 80, es el puerto web conectado al servidor. El Puerto 443 es el utilizado para sitios HTTPS seguro.

Si queremos conectarnos a un servidor web, utilizamos una URL, pero ¿qué es URL?. **URL** son las siglas en inglés de **Uniform Resource Locator** (en español, localizador uniforme de recursos), que sirve para nombrar recursos en Internet. Esta denominación tiene un formato estándar y su propósito es asignar una dirección única a cada uno de los recursos disponibles en Internet, como por ejemplo páginas, imágenes, vídeos, etc.

Una **URL** es una dirección simple que combina cuatro elementos de información fundamentales:

- el protocolo (por ejemplo, HTTP o HTTPS)
- el servidor o anfitrión con que se establece la comunicación
- el puerto de red en el servidor para conectarse
- la ruta al recurso o archivo que al que se está procurando acceder en el servidor.

Por ejemplo, para la URL `http://ejemplo.com/bio/pedroperez.html`:

- el protocolo es: http
- el servidor es ejemplo.com

- el directorio o la ruta es bio
- el archivo es pedroperez.html, que en este caso es una página web en HTML.

Una **URL** se usa también para identificar direcciones de correo electrónico, localización de archivos a transferir, bases de datos y otros elementos a los que se accede mediante Internet.

En informática utilizamos el término **host** para referirnos al servidor o anfitrión, es decir, para referirnos a los ordenadores u otros dispositivos (tabletas, móviles, portátiles) conectados a una red que proveen y utilizan servicios de ella. Los servidores deben utilizar anfitriones para tener acceso a la red y pueden, a su vez, pedir los mismos servicios a otras máquinas conectadas a la red.

Sockets en Python

Veamos ahora como podemos conectarnos a un servidor desde Python. Lo primero es importar la librería socket:

```
import socket

mysock = socket.socket(socket.AF_INET,socket.SOCK_STREAM )
mysock.connect(('data.pr4e.org', 80))
mysock
```

En estas líneas de código nos estamos conectando al host `data.pr4e.org` utilizando el puerto 80. Pero antes veamos que contiene este sitio, escribimos la URL Index of / (pr4e.org) en el navegador de nuestra preferencia y vemos el contenido de ese sitio web:

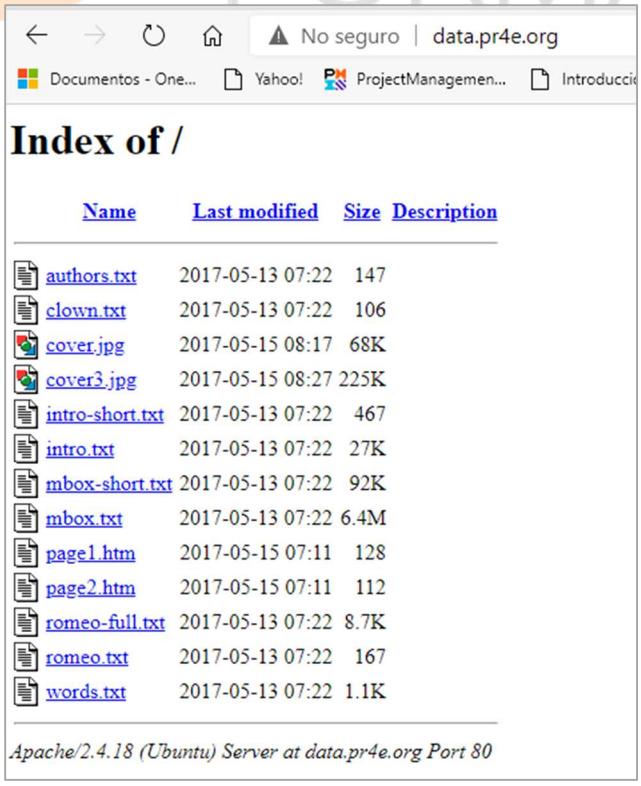


Figura 4: Servidor Apache/2.4.18 (Ubuntu) data.pr4e.org Port 80

Al ejecutar las líneas de código, estas nos retornan los datos de la conexión:

```
<socket.socket fd=1048, family=AddressFamily.AF_INET,
type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.1.73',
55666), raddr=('192.241.136.170', 80)>
```

Sintaxis:

```
socket.socket(family=AF_INET, type=SOCK_STREAM,
              proto=0, fileno=None)
```

La función socket, crea un nuevo socket utilizando la familia de direcciones (family), el tipo de socket y el número de protocolo dados.

- La familia de direcciones y protocolos utilizadas para el primer argumento de socket. Las opciones deben ser AF_INET (la predeterminada), AF_INET6, AF_UNIX, AF_CAN, AF_PACKET o AF_RDS.
- El tipo de conector debe ser SOCK_STREAM (el predeterminado), SOCK_DGRAM, SOCK_RAW o quizás una de las otras constantes SOCK_.
  - SOCK_STREAM (Sockets de flujo) usa como base el protocolo TCP (Transmission Control Protocol). En teoría asegura que los mensajes enviados a destino lleguen en el mismo orden en el que fueron enviados.
  - SOCK_DGRAM (Sockets de datagrama) es especial para trabajar con el protocolo UDP (User Datagram Protocol), a diferencia del anterior los mensajes pueden llegar en distinto orden en el que originalmente fueron enviados.
- El número de protocolo suele ser cero y puede omitirse o, en el caso de que la familia de direcciones sea AF_CAN, el protocolo debe ser uno de CAN_RAW, CAN_BCM, CAN_ISOTP o CAN_J1939.
- Si se especifica fileno, los valores de family, type y proto se detectan automáticamente desde el descriptor de archivo especificado.

Hasta ahora, hemos creado una conexión con un socket y luego lo conectamos a un puerto en particular en ordenador. Es decir, hemos creado la capa de extremo a extremo más baja en el modelo TCPIP. El siguiente paso es enviar y recibir datos.

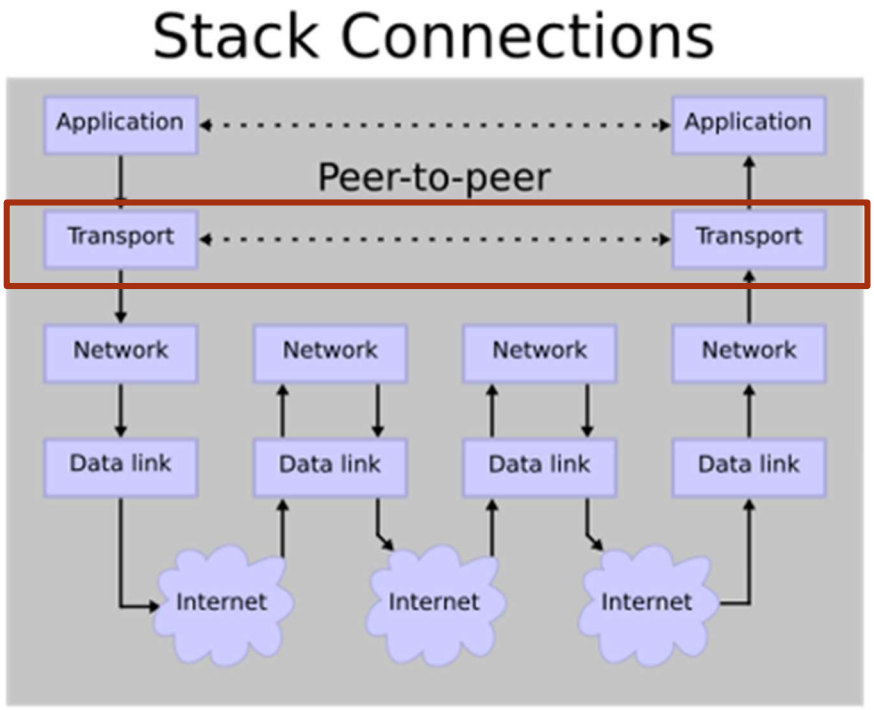


Figura 5: Modelo TCP/IP

Para iniciar la conversación debemos utilizar un protocolo de transporte, en este caso vamos a utilizar el protocolo de transferencia de hipertexto HTTP (Hypertext Transfer Protocol). Pero, ¿qué es HTTP?.

Wikipedia lo define como "El Protocolo de transferencia de hipertexto (en inglés, **Hypertext Transfer Protocol, abreviado HTTP**) es el protocolo de comunicación que permite las transferencias de información a través de archivos (XHML, HTML . . .) en la World Wide Web."

HTTP define la sintaxis y la semántica que utilizan los elementos de software de la arquitectura web (clientes, servidores) para comunicarse. HTTP es un protocolo sin estado, es decir, no guarda ninguna información sobre conexiones anteriores.

En otras palabras, es un conjunto de reglas que nos permiten conocer lo que debemos hacer, conocer la sintaxis que nos retornará, permitiendo así, que los diferentes proveedores puedan trabajar juntos, es decir, son estándares. Una de las cosas que HTTP estandariza, es el protocolo de **Localizadores Uniformes de Recursos o URL**.

Como vimos anteriormente la estructura de una URL puede ser:

```
http://www.dr-chuck.com/page1.htm

protocolo      host          documento
```

Cuando hacemos clic sobre un hipertexto, realmente estamos llamando a una nueva página, el browser o navegador realiza una conexión con el servidor web y ejecuta un **GET request**, esta instrucción trae el contenido de la página especificada en la URL. El servidor retorna un documento en formato HTML, que es lo que vemos en nuestro navegador.

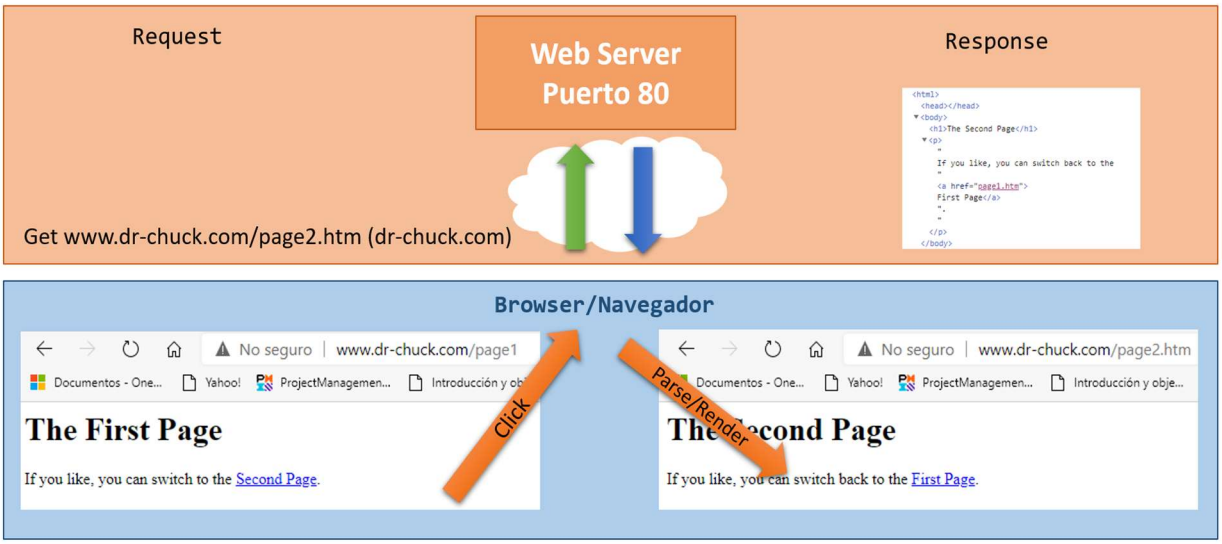
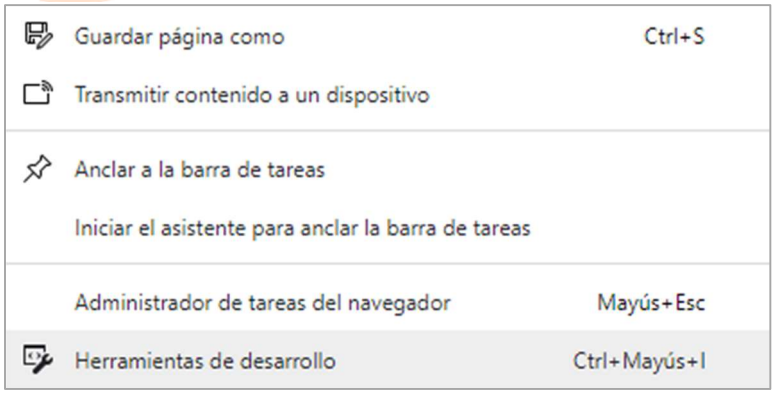


Figura 6: Obtener Datos desde el Servidor

Para ver el código HTML de una página:

1. Seleccionamos los tres **…** en la parte superior derecha del navegador
2. Seleccionamos la opción **Mas herramientas…**
3. Seleccionamos la opción **Herramientas de desarrollo**



Las reglas del protocolo HTTP las podemos consultar en https://www.w3.org/Protocols/rfc2616/rfc2616.txt. En el punto cinco podemos consultar la sintaxis de una solicitud cliente-servidor:

```
Request

A request message from a client to a server includes, within the
first line of that message, the method to be applied to the resource,
the identifier of the resource, and the protocol version in use.
```

```
Request       = Request-Line              ; Section 5.1
                *(( general-header        ; Section 4.5
                 | request-header         ; Section 5.3
                 | entity-header ) CRLF)  ; Section 7.1
                CRLF
                [ message-body ]          ; Section 4.3
```

Para realizar un **HTTP request**, primero nos conectamos al servidor y luego solicitamos un documento utilizando **GET**. Veamos cómo hacer esto con Python:

```
import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('data.pr4e.org', 80))
cmd = 'GET http://data.pr4e.org/romeo.txt
          HTTP/1.0\r\n\r\n'.encode()  # UTF-8
mysock.send(cmd)

while True:
    data = mysock.recv(512) # retorna 512 caracteres
    if len(data) < 1:
        break
    print(data.decode(),end='')

mysock.close()
```

```
HTTP/1.1 200 OK
Date: Fri, 11 Dec 2020 14:03:58 GMT
Server: Apache/2.4.18 (Ubuntu)
Last-Modified: Sat, 13 May 2017 11:22:22 GMT
ETag: "a7-54f6609245537"
Accept-Ranges: bytes
Content-Length: 167
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
Pragma: no-cache
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Connection: close
Content-Type: text/plain

But soft what light through yonder window breaks
It is the east and Juliet is the sun
Arise fair sun and kill the envious moon
Who is already sick and pale with grief
```

El socket es como una puerta fuera del ordenador, la conexión básicamente se extiende fuera de su computadora. Esto podría fallar si el servidor no existe. Si encuentra el servidor, se conecta al puerto 80, y establece el socket. Tenemos un socket conectado a un servidor. Ahora, podemos invocar a los métodos en el objeto socket (`send`, `recv`).

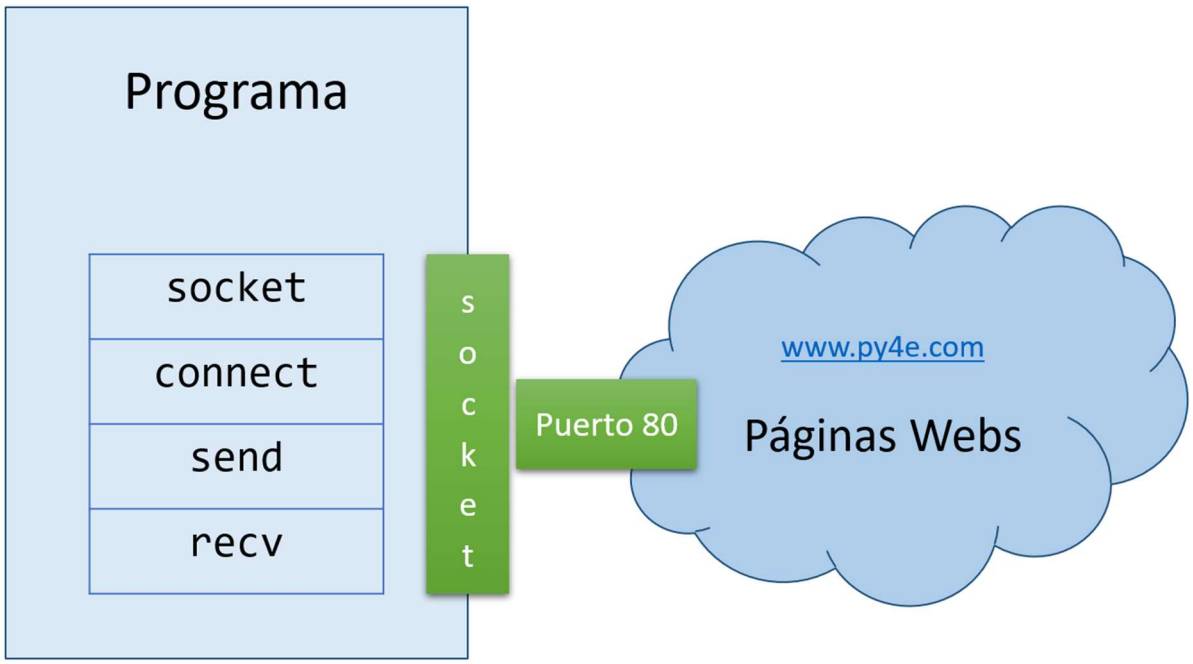
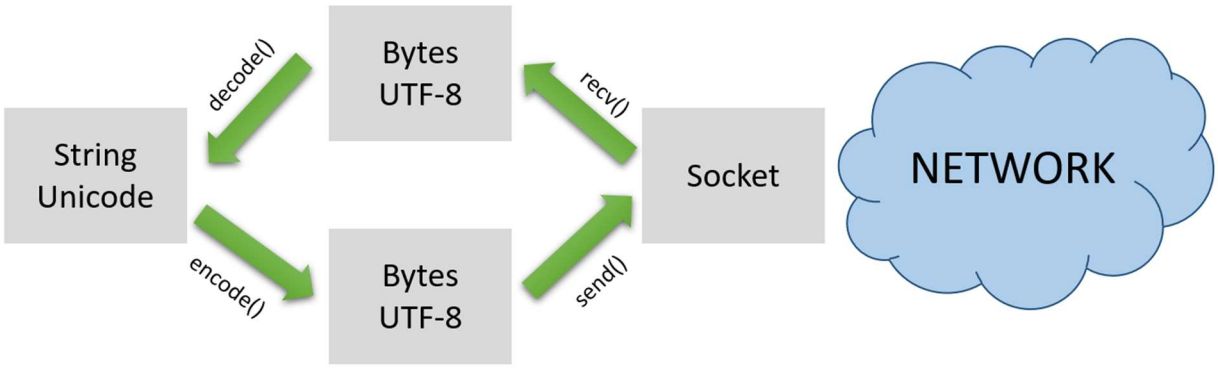


Figura 7: HTTP Request en Python

Lo primero que hace, es realizar una solicitud, el servidor lo recibe, lee algunos archivos y envía los datos de vuelta. Utilizamos un bucle `while` para retener la conexión y con el método `recv()`, es un método en el objeto socket que obtiene todos los datos.

Cuando nosotros establecemos una conversación utilizando sockets o con una BBDD, nosotros tenemos que codificar y decodificar la data (bytes). La función `decode()` devuelve una cadena descodificada de los bytes especificados. La codificación predeterminada es `'utf-8'`. Mientras `encode()` devuelve una versión codificada de la cadena como un objeto bytes. La codificación predeterminada es 'utf-8'.



Ejemplos para obtener datos de una página web

```
# Obtener el contenido del fichero intro.txt
import socket
```

```
mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.connect(('data.pr4e.org', 80))
cmd = 'GET http://data.pr4e.org/intro.txt
HTTP/1.0\r\n\r\n'.encode()  # UTF-8
mysock.send(cmd)

while True:
    data = mysock.recv(512)
    if len(data) < 1:
           break
print(data.decode(),end='')

mysock.close()


# Obtener una imagen de una web

import socket
import time
SERVIDOR = 'data.pr4e.org'
PUERTO = 80

misock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
misock.connect((SERVIDOR, PUERTO))
misock.sendall(b'GET http://data.pr4e.org/cover3.jpg
HTTP/1.0\r\n\r\n')
contador = 0
imagen = b""

while True:
    datos = misock.recv(5120)
    if len(datos) < 1: break
    contador = contador + len(datos)
    imagen = imagen + datos

misock.close()

# Búsqueda del final de la cabecera

pos = imagen.find(b"\r\n\r\n")
print('Header length', pos)
print(imagen[:pos].decode())

# Ignorar la cabecera y guardar los datos de la imagen

imagen = imagen[pos+4:]
fichero = open("portada.jpg", "wb")
fichero.write(imagen)
fichero.close()
```

Establecer comunicación entre un Cliente y un Servidor

Los siguientes son ejemplos que utilizan el **protocolo TCP/IP**: un servidor que hace eco de todos los datos que recibe de nuevo (dando servicio a un solo cliente) y un cliente que lo utiliza. Debemos tener en cuenta que un servidor debe realizar la secuencia `socket()`, `bind()`, `listen()`, `accept()`, mientras que un cliente solo necesita el `socket()`, `connect()`.

```
# Echo server program
     import socket

     HOST = ''                 # Symbolic name meaning all available
     interfaces
     PORT = 50007              # Arbitrary non-privileged port
     with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
         s.bind((HOST, PORT))
         s.listen(1)
         conn, addr = s.accept()
         with conn:
             print('Connected by', addr)
             while True:
                 data = conn.recv(1024)
                 if not data: break
                 conn.sendall(data)

# Echo client program
     import socket

     HOST = 'daring.cwi.nl'    # The remote host
     PORT = 50007              # The same port as used by the server
     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))
```

Crear una conexión «cliente – servidor» usando sockets en Python

```
“””
CÓDIGO DEL CLIENTE
“””

# importamos el módulo para trabajar con sockets

import socket

# Creamos un objeto socket para el servidor. Podemos dejarlo sin
# parámetros pero si quieren pueden pasarlos de la manera
```

```
# server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s = socket.socket()

# Nos conectamos al servidor con el método connect.
# Tiene dos parámetros: el primero es la IP del servidor y el
# segundo el puerto de conexión

s.connect((SERVIDOR, PUERTO))

#Creamos un bucle para retener la conexión

while True:

    # Instanciamos una entrada de datos para que
    # el cliente pueda enviar mensajes

    mensaje = raw_input("Mensaje a enviar >> ")

    # Con la instancia del objeto servidor (s) y el metodo send,
    # enviamos el mensaje introducido
    s.send(mensaje)

    # Si por alguna razón el mensaje es close cerramos la conexión

    if mensaje == "close":
        break

# Imprimimos la palabra Adiós para cuando se cierre la conexión

print("Adios.")

# Cerramos la instancia del objeto servidor
s.close()


“””
CÓDIGO DEL SERVIDOR
“””

# importamos el módulo socket

import socket

# instanciamos un objeto para trabajar con el socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Con el método bind le indicamos que puerto debe escuchar
```

```
# y de que servidor esperar conexiones. Es mejor dejarlo en blanco
# para recibir conexiones externas si es nuestro caso

s.bind((SERVIDOR, PUERTO))

# Aceptamos conexiones entrantes con el método listen, y además
# aplicamos como parámetro
# El número de conexiones entrantes que vamos a aceptar
s.listen(1)

# Instanciamos un objeto sc (socket cliente) para recibir datos,
# al recibir datos este devolverá también un objeto que representa
# una tupla con los datos de conexión: IP y puerto

sc, addr = s.accept()

while True:

    # Recibimos el mensaje, con el método recv recibimos datos
    # y como parámetro la cantidad de bytes para recibir

    recibido = sc.recv(1024)

    # Si el mensaje recibido es la palabra close
    # se cierra la aplicación

    if recibido == "close":
        break

    # Si se reciben datos nos muestra la IP y el mensaje recibido

    print(str(addr[0]) + " dice: ", recibido)

    # Devolvemos el mensaje al cliente

    sc.send(recibido)

print("Adiós!")

# Cerramos la instancia del socket cliente y servidor

sc.close()
s.close()
```

Módulo de manejo de URL

El módulo `urllib` en Python es una colección de módulos que podemos usar para trabajar con URLs. Este paquete cuenta con 4 módulos, que facilitan el manejo de URLs, definen

errores para ser capturados en una sentencia try/except y contienen funciones que permiten descargar archivos con Python.

- `urllib.request` para abrir y leer URL
- `urllib.error` que contiene las excepciones generadas por `urllib.request`
- `urllib.parse` para analizar URL

El módulo `urllib.request` define funciones y clases que ayudan a abrir una **URL** (principalmente HTTP: autenticación básica y resumida, redirecciones, cookies y más. El módulo `urllib.request` define las siguientes funciones:

```
urllib.request.urlopen (url, data = None)
```

- ***url:*** Abre la URL, que puede ser una cadena o un objeto de solicitud.
- ***data:*** debe ser un objeto que especifica datos adicionales que se enviarán al servidor, o None si no se necesitan dichos datos.

El módulo `urllib.request` es usado principalmente para buscar y abrir URLs. Una de sus funciones principales es `urllib.request.urlopen()`. La función recibe una URL, ya sea como cadena de texto o como un objeto `Request` y su objeto de retorno depende del tipo de URL que se le pase como argumento. Por ejemplo:

- Ingresando como argumento una `URL HTTP`, obtenemos un objeto `HTTPResponse` ligeramente modificado ya que cuenta con un par de métodos extra.
- Ingresando como argumento una `URL FTP` y `URLs` de **archivos y datos** esta función retorna un objeto `urllib.response.addinfourl`.

Veamos un ejemplo de varias funciones útiles que poseen los objetos `HTTPResponse` devueltos por la función `urllib.request.urlopen()`:

```
from urllib.request import urlopen

# Retorna un objeto httpresponse
httpresponse = urlopen('http://www.google.com')

httpresponse.geturl() # retorna la URL que fue solicitada.
'http://www.google.com'

header = httpresponse.info()
header.as_string()
```

```
'Date: Tue, 05 Jan 2021 15:44:31 GMT\nExpires: -1\nCache-Control: private,
max-age=0\nContent-Type: text/html; charset=ISO-8859-1\nP3P: CP="This is not
a P3P policy! See g.co/p3phelp for more info."\nServer: gws\nX-XSS-
Protection: 0\nX-Frame-Options: SAMEORIGIN\nSet-Cookie:
NID=206=rlNHu5U8uxX7LXqR4c8czyl9FeXZvm5sWDDJEnSQxSNHnpQaqO9c4WBi-
Dxm7wxXTtJzgZunyZ2HUWb4jXAXf0mUFMvfDsREUGJGsPsDJEzAKO4zUnByZvBR2rtzp-
q0f1OUTWE9OkYcOGstfpepo90UAeF7dIu3OgmVwQZhn8o; expires=Wed, 07-Jul-2021
```

```
15:44:31 GMT; path=/; domain=.google.com; HttpOnly\nAccept-Ranges:
none\nVary: Accept-Encoding\nConnection: close\nTransfer-Encoding:
chunked\n\n'
```

Como podemos ver en el ejemplo creamos un objeto `HTTPResponse` al pasarle una URL tipo HTTP como argumento a la función `urlopen()`. Un objeto `Response` contiene información sobre la respuesta del servidor a nuestra solicitud. Algunas de las funciones que podemos observar en el ejemplo son:

`response.geturl()`: retorna la URL que fue solicitada.

`response.info()`: al ejecutar solo este código nos muestra el tipo de objeto que retorna, un objeto `HTTPMessage`. Este contiene toda la información guardada en el objeto HTTPResponse, para leerla debemos llamar al método `HTTPMessage.as_string()`.

`response.getcode()`: retorna el código de la solicitud HTTP, en nuestro caso, 200 quiere decir que la solicitud fue exitosa, sin errores.

Descargar un Archivo

Un caso de uso común del módulo `urllib` es la descarga de un archivo. Veamos un par de maneras en las que podemos lograr esta tarea:

```
from urllib.request import urlopen

url = 'http://data.pr4e.org/romeo.txt'

response =  urlopen(url)
data = response.read()

with open('prueba.txt', 'wb') as archivo:
    archivo.write(data)

with open('prueba.txt', 'r') as archivo:
    print(archivo.read())
```

Aquí solo abrimos una URL que nos lleva a un archivo TXT guardado en esta página web. Luego leemos los datos y los escribimos en el disco. Al imprimir, vemos que hemos guardado un archivo TXT. Un enfoque alternativo, sería usar la función `urlretrieve`:

```
from urllib.request import urlretrieve

url = 'http://data.pr4e.org/romeo.txt'

archivo_tmp, header = urlretrieve(url)

with open('prueba2.txt', 'wb') as archivo:
    with open(archivo_tmp, 'rb') as tmp:
        archivo.write(tmp.read())
```

El método `urlretrieve` copia un objeto de red a un archivo local. El archivo que se copia obtiene un nombre aleatorio y va a un directorio temporal a menos que usemos el segundo parámetro del método `urlretrieve` donde podemos especificar la ubicación donde queremos que el archivo sea guardado.

El módulo `urllib.parse` define una interfaz estándar para dividir las cadenas de la URL en componentes (esquema de direccionamiento, ubicación de red, ruta, etc.), para combinar los componentes en una cadena de URL y convertir una "URL relativa" en una URL absoluta dada una "URL base".

Admite los siguientes esquemas de URL: archivo, ftp, gopher, hdl, http, https, imap, mailto, mms, news, nntp, prospero, rsync, rtsp, rtspu, sftp, shttp, sip, sips, snews, svn, svn + ssh, telnet, wais, ws, wss.

```
From urllib.parse import urlparse

result =
urlparse('https://duckduckgo.com/?q=20ython&t=ffab&ia=web')

result

ParseResult(scheme='https', netloc='duckduckgo.com', path='/',
params='', query='q=python&t=ffab&ia=web', fragment='')

result.netloc
'duckduckgo.com'

result.geturl()
'https://duckduckgo.com/?q=20ython&t=ffab&ia=web'

result.port
None
```

Aquí importamos la función `urlparse` y pasamos como argumento una consulta de búsqueda en el buscador DuckDuckGo. La consulta que hicimos fue simplemente la palabra "python". Como podemos ver, la función `urlparse()` devolvió un objeto `ParseResult` el cual podemos usar para aprender más sobre la URL. Por ejemplo podemos obtener la ubicación de la red, su ruta, la información relacionada al puerto (ninguna en este caso) y mucho más.

FUNCIONES	USO
urllib.parse.urlparse	Separa diferentes componentes de URL
urllib.parse.urlunparse	Unir diferentes componentes de URL
urllib.parse.urlsplit	Es similar a urlparse () pero no divide los parámetros
urllib.parse.urlunsplit	Combina el elemento tupla devuelto por urlsplit () para formar la URL

urllib.parse.urldeflag	Si la URL contiene un fragmento, devuelve una URL que elimina el fragmento.

```
from urllib.parse import *

parse_url = urlparse('https://www.geeksforgeeks.org/python-
langtons-ant/')
print(parse_url)  # Separa diferentes componentes de URL
print("----------------")

unparse_url = urlunparse(parse_url)
print(unparse_url) # Une diferentes componentes de URL
print("----------------")
```

```
ParseResult(scheme='https', netloc='www.geeksforgeeks.org',
path='/python-langtons-ant/', params='', query='', fragment='')
----------------
https://www.geeksforgeeks.org/python-langtons-ant/
----------------
```

```
from urllib.parse import *

url='https://tutorialspoint.com/python'

urlsplit_url = urlsplit(url) # Es similar a urlparse () pero no
divide los parámetros
print(urlsplit_url )
print("----------------")

urlunsplit_url  = urlunsplit(urlsplit_url)
print(urlunsplit_url) # Combina el elemento tupla devuelto por
urlsplit () para formar la URL
print("----------------")
```

```
SplitResult(scheme='https', netloc='tutorialspoint.com',
path='/python', query='', fragment='')
----------------
https://tutorialspoint.com/python
```

Enviando un formulario

Este módulo también posee el método `urlencode`, el cual es de gran ayuda para enviar datos a una URL. Un caso de uso típico para la librería `urllib.parse` es rellenar y enviar un formulario web. Veamos cómo podemos hacer esto haciendo que el buscador DuckDuckGo haga una búsqueda de "python".

```
from urllib.request import urlopen
from urllib.parse import urlencode

data = urlencode({'q':'python', 'ia':'web'}) # q=python&ia=web
```

```
url = 'http://www.duckduckgo.com/?{}'.format(data)
response = urlopen(url)

with open('results.html', 'wb') as file:
    file.write(response.read())

with open('results.html', 'r') as file:
    print(file.read())
```

El módulo `urllib.error` define las clases de excepción para las excepciones generadas por `urllib.request`. La clase de excepción básica es `URLError`.

- URLError: se genera por los errores en las URL o errores durante la obtención de la URL debido a la conectividad, y tiene una propiedad "razón" que le dice al usuario la razón del error.
- HTTPError: se genera para los errores HTTP exóticos, como los errores de solicitud de autenticación. Es una subclase o URLError. Los errores típicos incluyen "404" (página no encontrada), "403" (solicitud prohibida) y "401" (se requiere autenticación).

```
# URLError

from urllib.request import urlopen

# Realizar esta prueba sin conexión a internet

try:
    x = urlopen('https://www.google.com')
    print(x.read())
except Exception as e :
    print("Ha ocurrido un error", str(e))
```

```
Ha ocurrido un error <urlopen error [Errno 11001] getaddrinfo
failed>
```

```
# HTTPError

from urllib.request import urlopen

try:
    x = urlopen('https://www.google.com / search?q = test')
    print(x.read())

except Exception as e :
    print("Ha ocurrido un error...", str(e))
```

```
Ha ocurrido un error... URL can't contain control characters.
'www.google.com ' (found at least ' ')
```

Módulo urllib3

El urllib3 es una librería de Python para trabajar con un servidor HTTP y manejar varios tipos de solicitudes HTTP en el programa Python. El urllib3 es simplemente un módulo en Python para manejar la URL en el programa Python. Por ejemplo, podemos acceder a una dirección URL HTTP en Internet y descargar los datos de esa dirección URL. Esto hace que el desarrollo del programa Python que llama a la URL de una manera mucho más fácil. Puede utilizar la función urlopen() de esta biblioteca para capturar los datos de una URL en el programa Python. El urllib3 es una versión mejorada del módulo urllib. Podemos descargarlo e instalarlo usando pip.

```
pip install urllib3
```

Lo primero es importar la librería

```
import urllib3
```

Necesitaremos una instancia de `PoolManager` para realizar solicitudes. Este objeto controla todos los detalles del pool de conexiones y la seguridad de subprocesos.

```
url = 'https://www.python.org/'

http = urllib3.PoolManager()

r = http.request('GET', url)
```

Sintaxis:

```
request(method, url, fields=None, headers=None)
```

`request()` devuelve un objeto `HTTPResponse`. El objeto `HTTPResponse` proporciona atributos de status, data y headers:

```
r.data
b'<!doctype html>\n<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8
lt-ie9">   <![endif]-->\n<!--[if IE 7]>      <html class="no-js ie7 lt-ie8
lt-ie9">          <![endif]-->\n<!--[if IE 8]>      <html class="no-js ie8
lt-ie9">…

r.status
200

r.headers
HTTPHeaderDict({'Connection': 'keep-alive', 'Content-Length': '49558', 'Server':
'nginx', 'Content-Type': 'text/html; charset=utf-8', 'X-Frame-Options': 'DENY',
'Via': '1.1 vegur, 1.1 varnish, 1.1 varnish', 'Accept-Ranges': 'bytes', 'Date':
'Tue, 05 Jan 2021 18:05:52 GMT', 'Age': '481', 'X-Served-By': 'cache-bwi5144-BWI,
cache-mad22021-MAD', 'X-Cache': 'HIT, HIT', 'X-Cache-Hits': '1, 1', 'X-Timer':
'S1609869953.648132,VS0,VE1', 'Vary': 'Cookie', 'Strict-Transport-Security': 'max-
age=63072000; includeSubDomains'})
```

Buscaremos una página web y la almacenaremos en un archivo de texto usando `urllib3`. El módulo shutil se usa cuando se trabaja con archivos. Ahora inicialice la variable de cadena como esta:

Luego, usamos el `PoolManager` de `urllib3` que realiza un seguimiento del pool de conexiones necesarios, se crea el archivo `filename` y luego utilizamos `GET` para descargar el archivo.

```
import urllib3, shutil

url = 'https://www.python.org/'

c = urllib3.PoolManager()

filename = "python.txt"

with c.request('GET', url, preload_content=False) as res,
open(filename, 'wb') as out_file:
    shutil.copyfileobj(res, out_file)

url =
"https://cflvdg.avoz.es/sc/JJ11XK4rpXWDpjzaTALyPz9bhnQ=/x/2020/
08/27/00121598518381871168995/Foto/foto1.jpg"

c = urllib3.PoolManager()

filename = "imagen.jpg"

with c.request('GET', url, preload_content=False) as res,
open(filename, 'wb') as out_file:
    shutil.copyfileobj(res, out_file)


import io

url = "https://www.lavozdegalicia.es/noticia/coruna/para-
comer/2020/12/10/nova-lua-chea-gastrobar-raices-
presentes/00031607613509913282287.htm"

http  = urllib3.PoolManager()

res = http.request('GET', url, preload_content=False)
res.auto_close = False

for line in io.TextIOWrapper(res):
    encontrado = re.findall(r'<p>(.*?)</p>',line)
    if len(encontrado) > 0:
        print(encontrado)
```

Módulo requests

`requests` es una librería Python que facilita el trabajo con peticiones HTTP. Es útil, cuando necesitemos realizar peticiones web, ya sea para consumir un API, extraer información de una página o enviar el contenido de un formulario de manera automatizada.

Al tratarse de una librería de terceros, lo primero que debemos hacer es instalarla.

```
pip install requests
```

Una de las operaciones que podemos hacer con la librería `requests` es hacer una petición `GET`, ya sea para obtener el contenido de una web o para realizar una petición a un API.

Para ello, simplemente tenemos que invocar a la función `get()` indicando la URL a la que hacer la petición.

```
import requests
resp = requests.get('https://www.google.com/')
```

La función devuelve un objeto `Response`, que en este caso se ha asignado a la variable resp, con toda la información de la respuesta.

Cuando la respuesta que devuelve un servidor es de tipo texto, por ejemplo html o xml, el contenido se encuentra en el atributo text del objeto Response.

```
resp.text
'<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="es"><head><meta
content="Google.es permite acceder a la información mundial en castellano, catalán, gallego,
euskara e inglés." name="description"><meta content="noodp" name="robots"><meta
content="text/html; charset=UTF-8" http-equiv="Content-Type"><meta
content="/images/branding/googleg/1x/googleg_standard_color_128dp.png"
itemprop="image"><title>Google…
```

`requests` automáticamente decodifica el contenido devuelto por el servidor, adivinando la codificación a emplear a partir de las cabeceras de la respuesta. Para conocer la codificación empleada puedes acceder al atributo `encoding`.

Para obtener el código de estado de la respuesta, debemos acceder al atributo `status_code` de la misma.

```
resp.status_code
200
```

Las cabeceras de la respuesta están accesibles a través del atributo `headers`. Este atributo es un diccionario especial que contiene cada una de las cabeceras devueltas como claves del diccionario.

```
resp.headers
```

```
{'Date': 'Tue, 05 Jan 2021 13:00:57 GMT', 'Content-Type':
'text/html;charset=ISO-8859-1', 'Vary': 'Accept-Encoding', 'X-UA-
Compatible': 'IE=Edge', 'X-Esi': '1', 'Access-Control-Allow-Credentials':
'true', 'Content-Security-Policy': "default-src https: data: 'unsafe-inline'
'unsafe-eval'; child-src https: data: blob:; connect-src https: data: blob:;
font-src https: data:; img-src https: data:; media-src https: blob:; object-
src https:; script-src https: data: blob: 'unsafe-inline' 'unsafe-eval';
style-src https: 'unsafe-inline'; block-all-mixed-content; upgrade-insecure-
requests; report-uri https://lavoz.report-uri.io/r/default/csp/enforce",
'Cache-Control': 's-maxage=43200', 'vha6-origin': 'vcp03', 'Content-
Encoding': 'gzip', 'Age': '27597', 'Set-Cookie':
'apw_aac_web=1612137600.4.1.0003160761350991328228 7.3yBo96vsQKbuvfkQIBD9yKPR
eKVstlMzcnGxF_wO9TQ; domain=.lavozdegalicia.es; path=/; expires=Mon, 01 Feb
2021 00:00:00 GMT; SameSite=None; Secure, apw_cache=0edd58ab-1ff8-4b65-9d10-
ec9e503d7819.1609879255.NJ75BA.ES.0...0.0..0.wDgCHDQlVoYz0aGowgS5_LpR4_TZUbM
X_dF5miBpLmw; domain=.lavozdegalicia.es; path=/; expires=Wed, 05 Jan 2022
20:40:55 GMT; httpOnly; SameSite=None; Secure', 'Accept-Ranges': 'bytes',
'Transfer-Encoding': 'chunked', 'Connection': 'keep-alive'}
```

Si queremos consultar las cookies devueltas por el servidor, lo podemos hacer accediendo al atributo cookies de la respuesta. Este atributo es de tipo `RequestsCookieJar`, que actúa como un diccionario con mejoras, para indicar el dominio y/o el path de una cookie, entre otras cosas:

```
resp.cookies
<RequestsCookieJar[Cookie(version=0, name='apw_aac_web',
value='1612137600.4.1.0003160761350991328228 7.3yBo96vsQKbuvfkQIBD9yKPReKVstl
MzcnGxF_wO9TQ', port=None, port_specified=False,
domain='.lavozdegalicia.es', domain_specified=True, domain_initial_dot=True,
path='/', path_specified=True, secure=True, expires=1612137600,
discard=False, comment=None, comment_url=None, rest={'SameSite': 'None'},
rfc2109=False), Cookie(version=0, name='apw_cache', value='0edd58ab-1ff8-
4b65-9d10-
ec9e503d7819.1609879255.NJ75BA.ES.0...0.0..0.wDgCHDQlVoYz0aGowgS5_LpR4_TZUbM
X_dF5miBpLmw', port=None, port_specified=False, domain='.lavozdegalicia.es',
domain_specified=True, domain_initial_dot=True, path='/',
path_specified=True, secure=True, expires=1641415255, discard=False,
comment=None, comment_url=None, rest={'httpOnly': None, 'SameSite': 'None'},
rfc2109=False)]>
```

Para aquellos casos en los que la respuesta no es texto, como por ejemplo una imagen o un pdf, entonces se debe acceder al atributo `content`, ya que este devuelve el contenido como una secuencia de bytes.

```
import requests
import shutil

url =
"https://cflvdg.avoz.es/sc/JJ11XK4rpXWDpjzaTALyPz9bhnQ=/x/2020/
08/27/00121598518381871168995/Foto/foto1.jpg"

r = requests.get(url, stream=True)

with open('image.jpg', 'wb') as f:
```

```
        r.raw.decode_content = True
        shutil.copyfileobj(r.raw, f)
```

Por último, hay un caso especial que permite acceder al socket que devuelve la respuesta del servidor. Es a través del atributo `raw`. Sin embargo, en lugar de acceder al atributo raw directamente, es preferible llamar a la función `iter_content` usando el siguiente patrón, especialmente cuando se quiere hacer streaming en crudo de la descarga.

Si necesitamos hacer una petición POST para enviar los datos de un formulario. En este caso, debemos llamar a la función `post()` e indicar en el parámetro `data` un diccionario con los datos del cuerpo de la petición. Al pasar los datos en el parámetro `data`, `requests` se encarga de codificarlos correctamente antes de realizar la petición.

Para el caso en que un formulario tenga uno o más campos multivaluados, se pueden especificar los diferentes valores de dos maneras distintas. En un diccionario, indicando una lista de valores para una clave:

```
import requests

form_data = {'color': ['blanco', 'verde'], 'idioma': 'es'}
resp = requests.post('http://mipagina.xyz/formulario/',
data=form_data)
```

O como una lista de tuplas:

```
import requests

form_data = [('color', 'blanco'), ('color', 'verde'),
('idioma', 'es')]
resp = requests.post('http://mipagina.xyz/formulario/',
data=form_data)
```

Del mismo modo que existen las funciones `get()` y `post()`, `requests` tiene funciones para los siguientes métodos de petición: `PATCH`, `PUT`, `DELETE`, `HEAD` y `OPTIONS`. Son `patch()`, `put()`, `delete()`, `head()` y `options()`.

Peticiones y respuestas JSON

Uno de los usos más habituales de la librería `requests` es hacer peticiones a un API desde una aplicación. Una de las principales características de consumir un API es que, generalmente, los datos se envían y se obtienen en formato JSON.

Python GET requests

Para hacer una petición `GET`, simplemente hay que llamar a la función `get()`.

Si la respuesta es un JSON, que es lo más común, podemos llamar al método `json()` de la respuesta para que decodifique los datos y los devuelva como un diccionario con los campos de dicho JSON.

```
import requests

r= requests.get('https://api.github.com')

try:
    data = r.json()
    print(data)
except ValueError:
    print("No es un JSON válido…")

{'current_user_url': 'https://api.github.com/user',
'current_user_authorizations_html_url':
'https://github.com/settings/connections/applications{/client_i
d}', 'authorizations_url':
'https://api.github.com/authorizations', 'code_search_url':
'https://api.github.com/search/code?q={query}{&page,per_page,so
rt,order}'…
```

Para enviar datos en formato JSON a un API empleando los métodos `POST`, `PUT` o `PATCH`, simplemente pasa un diccionario a través del parámetro `json.requests` ya se encarga de especificar la cabecera `Content-Type` por ti y de serializar los datos de forma correcta.

Bibliografía

- Introduction to Networking by Charles Severance
- Wikipedia: Puertos de red - Wikipedia, la enciclopedia libre
- Crear una conexión "cliente - servidor" usando sockets en Python (developeando.net)
- Request Methods - urllib3 2.0.0.dev0 documentation