Control 1
CC4303-2 Redes

José Espina
joseguillermoespina@gmail.com

1. Contenido del presente control

El control comienza con las instrucciones dadas por el cuerpo docente de la asigntura, donde se identifican los requerimientos funcionales implícitos y explícitos. Luego se explica la propuesta de solución, y el detalle de su prueba de concepto (prototipo funcional) a través del *framework* "socketserver"de Python 3[1], inspirado en los ejercicios del libro "Python Network Programming"[3]. Finalmente, se responde el cuestionario que viene junto al problema

2. Contexto, problema y requerimientos

Uno de los grandes problemas que ha aparecido con la movilidad de usuarios es la pérdida de conexión entre cliente y servidor ya sea por cambio de antena, paso de internet móvil a wi-fi, de wi-fi a cable, y viceversa. Esto lleva a que se sobrecargue la inteligencia por el lado del servidor (si se usa UDP) o se cierren/abran conexiones con el consiguiente overhead en handshaking (si se usa TCP). Algunos ingenieros señalan que la capa de transporte no tiene por qué preocuparse de eso y debiesen ser las capas superiores las encargadas. En esta actividad se le pedirá diseñar o modificar un protocolo de comunicaciones para este nuevo escenario

A partir de las instrucciones, se identifican los siguientes requerimientos:

1. Construir un servidor http que implemente GET, recibiendo peticiones de un cliente "móvil" (puede construir una función UniqID() para identificar cada dispositivo) y le entregue a cada dispositivo su respuesta, sin repetir ni equivocarse

2. El cliente debiese poder conectarse al servidor, pedir un archivo, recibir parte de él, desconectarse, conectarse de nuevo, pedir un archivo y recibir el resto

3. Su solución debe escribirse en python usando sockets, sin otras librerías para redes y NO DEBE caerse cuando el servidor le responde a un cliente "inexistente"(que se está moviendo)

4. La entrega serán dos archivos: un .zip con su solución en python de su cliente y servidor (código fuente), y un PDF donde estará un pequeño informe con sus decisiones de diseño, supuestos seguidos (que sean razonable, no pueden suponer que sólo se conectarán una vez, por ejemplo)

5. Responda justificadamente a las siguientes preguntas:

   a) ¿Qué pasa cuando el cliente termina de recibir un archivo pero igual lo sigue pidiendo? ¿Por qué?

   b) ¿Cómo funciona su protocolo si hay múltiples clientes? digamos que su servidor tiene la misma tasa de consultas que Google.

   c) A partir de la pregunta anterior ¿Cambia su respuesta si en vez de texto responde con multimedia (tamaño $>$ 1 MB)? ¿Por qué?

   d) ¿Su solución sigue siendo válida si se usan proxies http? ¿Por qué?

3. Propuesta de solución: un protocolo

Se propone la creación de un pequeño protocolo que no implementa nuevos comandos, si no que reutiliza la sintaxis básica de HTTP 1.1. Funcionará en las capas de Aplicación, ya que se limitara a interpretar las solicitudes del cliente y las correspondientes respuestas con "chunks"de datos del archivo solicitado. Se reutilizará TCP y HTTP en las capas Transporte, Sesión y Presentación

3.1. ¿Porqué TCP y HTTP?

Las razones de la elección del uso de TCP y HTTP son las siguientes:

1. El servidor enviará archivos texto plano y binarios. Es importante la integridad de los paquetes, la que UDP no lo asegura

2. No se requiere hacer *broadcast* de paquetes, la que es una característica de UDP

3. La solución usa parte de HTTP para confeccionar la respuesta, y en general HTTP se usa sobre TCP por ser "un protocolo confiable"[2]

4. La solución propuesta necesita de conocer el tamaño (en bytes) y el tipo de dato que se enviará al cliente (texto plano o binario), por lo que se usará HTTP, en específico la versión 1.1, ya que este protocolo implementa esos datos a través de sus cabeceras *Content-length* y *Content-type*

A continuación, se describe el funcionamiento básico del software mediante el siguiente diagrama de flujo y algoritmo

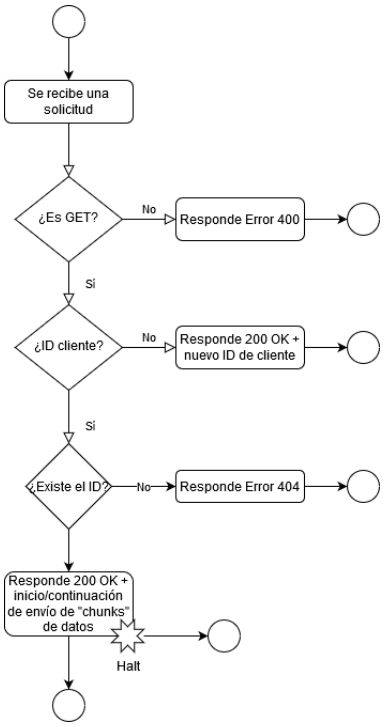

Figura 1: Diagrama de flujo de la solución propuesta. Notar que el "halt"se produce cuando el cliente se está moviendo, lo que cortaría la comunicación

3.2. Diagrama de flujo de la solución

3.3. Algoritmo de la solución

Algorithm 1: Prototipo mini-protocolo

```
Input: Solicitud de un cliente mediante socket abierto
Output: Nuevo identificador de cliente, ó "chunks"de datos de archivo
        (texto o binario) al cliente
inicializacion variables y constantes;
request = socket.recv(SIZE);
if type(request) != GET then
    socket.send(ERROR_400);
else
    if id_cliente in request then
        file.seek(progresos[id_cliente]);
        data = file.read(CHUNK_SIZE);
        socket.send(OK_200);
        while data do
            socket.send(data);
            progresos[id_cliente] += len(data);
            data = file.read(CHUNK_SIZE);
        end
    else
        id_cliente=uuid.uuid4();
        progresos[id_cliente] = 0;
        socket.send(OK_200);
        socket.send(id_cliente);
    end
end
```

La idea general de la solución es la implementación de un servidor mediante uso de sockets, que pueda:

1. Recibir una solicitud de cliente de tipo GET. Para cualquier otro tipo de solicitud el servidor responderá error 403 "Prohibido"

2. Si en la solicitud viene un código único de cliente, iniciar o retomar el envío de un archivo solicitado según lo registrado en un diccionario llave - valor, donde la llave es un código único generado para el cliente, y el valor es el número del último byte del archivo enviado al cliente

3. Si en la solicitud no viene un código único de cliente, se asume que es un nuevo cliente, se genera un nuevo código de cliente único de largo de 34 caracteres, y se envía al cliente

4. Si viene un código de cliente y éste no existe en el diccionario de progresos, se responde con error 404 "No encontrado"

3.4. Supuestos y limitaciones

1. El prototipo sólo demuestra la posibilidad de retomar una descarga interrumpida ante una desconexión del cliente. Se probó con curl. No se usó netem

2. No se programó un cliente, ya que curl cumple lo necesario para hacer las pruebas. Tampoco se implementó alguna programa para fusionar los pedazos de datos transferidos para reconstruir el archivo original *hosteado* por el servidor, ya que existe el comando "cat"(en sistemas basados en Unix) que hace ese trabajo

3. La solución se probó sólo en el sistema operativo Windows, con Python 3 de Anaconda y curl del WSL(*Windows Subsystem for Linux*). Debiese funcionar en sistemas basados en Unix, debido a que se usaron sólo librerías estándar de Python 3.x

4. La versión probada y enviada, es de un sólo hilo (*single-thread*), por lo que puede atender sólo un cliente a la vez. Sin embargo, no es muy complejo mejorar el servidor a multi-hilo (*multi-thread*). Se dejó comentado el código necesario para hacer esa implementación. Cabe mencionar que se hizo una prueba y funcionó. Como evidencia de ésto último, se deja un *screenshot* en el apéndice del documento

5. Un cliente puede descargar y dejar "en pausa"sólo 1 archivo. Deberá terminar la descarga para poder solicitar otro

6. Se asume que parámetros y archivos solicitados están correctos

7. No se optimizó el código para un mejor rendimiento, ni eficaz uso de recursos

8. No se hizo pruebas de borde ni de saturación

4. Detalles de la implementación

Se implementó un *handler* del *framework* socketserver para cumplir con los requerimientos nombrados en la primera sección El código completo se adjunta con la tarea y, también, está disponible en un repositorio privado en GitHub (`https://github.com/joseespinadote/control1-redes`, solicitar previamente acceso al autor)

No se programó un cliente, ya que el servidor, al responder con cabeceras compatibles con HTTP 1.1, puede interactuar con clientes que usen el mismo protocolo, como el programa "curl", que permite enviar peticiones GET al servidor, y almacenar los *chunks* de datos en un archivo. Tampoco se implementó la concatenación de los pedazos del archivo descargado, ya que se utilizó "cat"

Los partes fundamentales de la implementación se explican a continuación:

4.1. Definición de variables y constantes globales

Por tratarse de un prototipo, se definieron las siguientes variables y constantes globales sin pensar mucho en optimizar el código para un mejor rendimiento y eficaz uso de recursos

```
BUFFER_ENTRADA = 256
TAMANO_PEDAZOS = 512
TAMANO_UUID = 36
DICT_RESPUESTAS_HTTP = {200: "HTTP/1.1 200 OK\n",
                        400: "HTTP/1.1 400 Bad Request\n",
                        403: "HTTP/1.1 403 Forbidden\n",
                        404: "HTTP/1.1 404 Not Found\n"}
dict_cliente_progeso = dict()
```

Listing 1: Definición de variables y constantes globales

1. BUFFER_ENTRADA: Constante que corresponde a la cantidad de bytes que lee para procesar la solicitud del cliente conectado. Es un valor arbitrario

2. TAMANO_PEDAZOS: Constante que corresponde a la cantidad de bytes que conforman los *chunks* (pedazos) de datos que se envían a través del socket entre el servidor y el cliente. Por ejemplo, si se envía un archivo de 1000 bytes, entonces se enviará completamente en $1000/TAMANO\_PEDAZOS$ *chunks*. El valor es arbitrario, y se escogió un tamaño pequeño con fines de aprendizaje y experimentación

3. TAMANO_UUID: Constante que representa la cantidad de caracteres de los identificadores únicos de cliente

4. DICT_RESPUESTAS_HTTP Constante que almacena, en un diccionario, las cadenas de texto que se deben responder en las cabeceras de la comunicación HTTP 1.1

4.2. La función principal (main)

La función principal se ejecuta al llamar al script desde un terminal. Se encarga de inciar un servidor de socket (propio del *framework* socketserver) localmente (*localhost*) y con un puerto entregado por el sistema operativo

```
if __name__ == '__main__':
    address = ('localhost', 0)
    servidor = socketserver.TCPServer(address, Control_uno_handler)
    ip, port = servidor.server_address
    print('Servidor funcionando en {ip}:{puerto}'.format(ip=ip,
    puerto=port))
    try:
        servidor.serve_forever()
    except KeyboardInterrupt:
        pass
    finally:
        servidor.shutdown()
```

```
12          servidor.socket.close()
```

Listing 2: La función principal

El detalle es el siguiente:

1. La línea 2 y 3 se encargan de crear la conexión del servidor. El valor "0", en el segundo parámetro de la tupla, indica se le pedirá al sistema operativo que asigne un puerto. Se podría asignar uno manualmente

2. Control_uno_handler es una clase que hereda de *socketserver.BaseRequestHandler*, e implementa el ciclo de operación de un servidor socket. Más adelante se verá que sólo es necesario sobrecargar el método "handler"de todos los métodos disponibles

3. El bloque *try-except-finally* permite cerrar el servidor correctamente al recibir un "control-c"desde el teclado, sin dejar el puerto "secuestrado"por el sistema operativo

4.3. La función *genera_cabeceras* de la clase *Control_uno_handler*

La función prepara las cabeceras para un cliente que interprete HTTP 1.1

```
1  def genera_cabeceras(self, codigo, tamanio_respuesta_bytes,
       content_type="text/plain"):
2          now = datetime.now()
3          stamp = mktime(now.timetuple())
4          str_headers = DICT_RESPUESTAS_HTTP[codigo]
5          str_headers += "Date: " + str(format_date_time(stamp)) + "\
       n"
6          str_headers += "Server: ~el joServer 'the espinator'~ v0.1\
       n"
7          str_headers += "Last-Modified: " + str(format_date_time(
       stamp)) + "\n"
8          str_headers += "Accept-Ranges: bytes\n"
9          str_headers += "Content-Length: "+str(
       tamanio_respuesta_bytes)+"\n"
10         str_headers += "Cache-Control: max-age=0\n"
11         str_headers += "Expires: " + str(format_date_time(stamp)) +
        "\n"
12         str_headers += "Content-Type: " + content_type + "\n"
13         str_headers += "\n"
14         return str_headers
```

Listing 3: El método del handler que permite generar las cabeceras HTTP

La función se encarga de armar las cabeceras en el *string* "str_headers", a partir de los parámetros código HTTP (que pueden ser los valores del diccionario de códigos que se encuentran en las constantes globales explicadas previamente), tamaño en bytes de repuesta y el tipo de contenido de la respuesta

4.4. Retorno de error 403 “Prohibido”, en caso que la solicitud no sea de tipo GET

En caso de recibir una solicitud que no sea de tipo GET, se responde con un error 403 “Prohibido”

```
        if not str_data.startswith('GET /'):
            headers = self.genera_cabeceras(403, 0)
            self.request.send(str.encode(headers))
            return
```

Listing 4: Error 403

4.5. Respuesta al cliente con un nuevo ID de cliente único

La siguiente sección de código se encarga de generar un nuevo ID de cliente gracias al paquete UUID de python. Esto sucede cuando el cliente hace una llamada GET sin parámetros (ver el diagrama de flujo o el algoritmo)

```
    if str_data[5] == ' ':
        str_cliente_id = str(uuid.uuid4())
        cliente_id = str.encode(str_cliente_id)
        dict_cliente_progeso[str_cliente_id] = 0
        headers = self.genera_cabeceras(200, TAMANO_UUID)
        self.request.send(str.encode(headers))
        self.request.send(cliente_id)
        return
```

Listing 5: La creación y envío de un nuevo código único de cliente

En caso de venir un identificador de cliente en la solicitud, y este no exista en el diccionario de clientes y progreso, se responde con un error 404 “No encontrado”

```
    if cliente_id not in dict_cliente_progeso.keys():
            headers = self.genera_cabeceras(404, 0)
            self.request.send(str.encode(headers))
            return
```

Listing 6: Caso del error 404

4.6. Perapración para el envío del archivo

En esta sección de código se extrae la metadata para crear cabeceras apropiadas para el envío del archivo

```
    parametros_GET = str_data.split(" ")[1].split("/")
        cliente_id = parametros_GET[1]
        archivo = parametros_GET[2]
        tipo_contenido = mimetypes.guess_type(archivo)[0]
        total = os.path.getsize(archivo)
```

Listing 7: Extracción de parámetros GET y metadata del archivo

El paquete *mimetypes* de Python permite extraer el tipo de contenido a partir de la ruta de un archivo

4.7. Envío del archivo y seguimiento del progreso para el cliente

Ésta es la sección de código más importante

Se procede a rescatar el progreso del envío del archivo del cliente, el que será 0 si es el primer intento. En el caso que el cliente ya haya descargado parte de un archivo en una instancia previa, el método *seek* del objeto que tiene la referencia al archivo permite posicionar el lector en el byte donde el cliente necesita continuar su descarga

El envío de "chunks"de datos va precedido por la cabecera "200 OK"que avisa que la conexión fue exitosa. El bloque *Try-Except* permite enviar los datos y almacenar la cantidad de bytes transmitidos en caso de que la comunicación se corte

```
            progreso = dict_cliente_progeso[cliente_id]
            total_enviado = progreso
            with open(archivo, 'rb') as output:
                output.seek(progreso)
                str_headers = self.genera_cabeceras(
                    200, total-progreso, tipo_contenido)
                self.request.send(str.encode(str_headers))
                while True:
                    data = output.read(TAMANO_PEDAZOS)
                    if not data:
                        del dict_cliente_progeso[cliente_id]
                        break
                    try:
                        self.request.send(data)
                    except ConnectionResetError:
                        print('El cliente {} alcanzo a descargar {}
 bytes'.format(
                            cliente_id, dict_cliente_progeso[
cliente_id]))
                        return
                    total_enviado += len(data)
                    dict_cliente_progeso[cliente_id] =
total_enviado
                    print('{} de {} bytes enviados'.format(
total_enviado, total),
                          sep=' ', end='\r', flush=True)
                    time.sleep(0.05)
```

Listing 8: Envío de datos al cliente

En caso que el envío del archivo termine, se borra el identificador único del cliente en el diccionario. Si el cliente desea volver a pedir otro archivo, debe volver a pedir un nuevo código

Nota importante: El comando *sleep* permite ralentizar la comunicación sólo con objetivos de experimentación

5. Demostración del prototipo

Se presenta, a continuación, indicaciones para ejecutar el prototipo

1. Ejecutar el script del prototipo (control1.py) usando Python 3.6 o superior). El servidor indicará el puerto que le asignó el sistema operativo. Ejemplo:

```
$ python control1.py
Servidor funcionando en 127.0.0.1:52285
```

Listing 9: En consola #1: Se inicia el servidor

2. En otra terminal, hacer una solicitud GET con curl al host incluyendo su puerto, sin parámetros, para solicitar un nuevo id de cliente. Ejemplo:

```
$ curl 127.0.0.1:52285
2011f981-98e1-42f2-b06c-b26867918ec9
```

Listing 10: En consola #2: curl solicita al servidor un nuevo código de cliente

3. Se ejecuta nuevamente curl, esta vez incluyendo el identificador único retornado en el paso anterior, más un nombre de archivo que exista en el directorio del servidor. Se debe añadir, a curl, el parámtro "-o nombre_de_archivo"para que almacene en un archivo los *chunks* de datos enviados por el servidor. Si no se hace esto último, curl intentará mostrar el archivo binario por pantalla (salida estándar) y ocurrirá un error, tanto en el cliente como en el servidor. Una vez haya iniciado la descarga, esperar unos segundos e interrumpir mediante teclado (control-c) a curl, para simular que el cliente "se está moviendo", y, por ende, un corte en la comunicación

   Para ejemplificar, se adjunta en la tarea un documento "Word"para ser enviado entre servidor y cliente:

```
    curl 127.0.0.1:52285/2011f981-98e1-42f2-b06c-b26867918ec9/
    word.doc -o descarga.parte1
    %    Total    % Received  % Xferd  Average    Speed
    Time        Time        Time    Current

    Dload       Upload       Total       Spent         Left
    Speed
   47      98k   47        47616  0         0    8334
     0          0:00:12   0:00:05     0:00:0    8425
```

Listing 11: En consola #2: curl llama al servidor

   En el ejemplo, se cortó con control-c la ejecución curl en el momento en que se habían transferido 47616 bytes de 98 KB (aprox), correspondiente al 47 % del total de la transferencia. Mientras, en la primera consola, el servidor mostrará el próximo byte que queda pendiente

```
python control1.py
Servidor funcionando en 127.0.0.1:52285
El cliente 2011f981-98e1-42f2-b06c-b26867918ec9 alcanzo a
    descargar 51712 bytes
```

Listing 12: En consola #1: El servidor envía el archivo al cliente

Nota importante: Curl, al ser detenido abruptamente con control-c, no muestra refleja el total bytes transferidos. El valor exacto son 51712 bytes. En el próximo paso, donde no se interrumpirá a curl, se podrá comprobar que la cantidad de bytes transferidos calzan tanto en el cliente como en el servidor

4. Se procede volver a usar curl con los mismos comandos que en el paso 3, esta vez con un nuevo nombre de archivo, con el fin de no perder el progreso de lo ya descargado. Esta vez, permitir que curl termine de descargar el archivo

```
curl 127.0.0.1:52285/2011f981-98e1-42f2-b06c-b26867918ec9/word
    .doc -o descarga.parte2
  %     Total     % Received % Xferd  Average   Speed    Time
       Time      Time    Current

    Dload      Upload    Total      Spent      Left    Speed
100 48640  100       48640    0      0      8524           0
     0:00:05  0:00:05    --:--:--     8422
```

Listing 13: En consola #2: El cliente curl pide el resto del archivo

Opcionalmente, se puede agregar el parámetro "-D nombre_de_archivo"para almacenar las cabeceras enviadas por el servidor

Ahora en el servidor se muestra que se transfirieron los 100352 bytes correspondientes al total del peso del archivo

```
$ python control1.py
Servidor funcionando en 127.0.0.1:52285
El cliente 2011f981-98e1-42f2-b06c-b26867918ec9 alcanzo a
    descargar 51712 bytes
100352 de 100352 bytes enviados
```

Listing 14: En consola #1: El servidor refleja que envió el total del archivo

5. Terminada la descarga, concatenar ambos archivos con el comando "cat"para poder disfrutar del documento "Word"transferido. Ejemplo:

```
cat descarga.parte1 descarga.parte2 > merge.doc
```

Listing 15: En consola #2: Se fusionan las 2 partes del archivo

6. Se puede observar que el peso (o mejor aún, un *checksum*) del archivo concatenado es equivalente al original. Incluso se puede abrir con LibreOffice, OpenOffice, Google Drive, entre otros, y revisar que su contenido es el mismo y que no hay ruido. Ejemplo:

```
$ ls -l
-rw-r--r-- 1 josee josee  51712 Jun  4 17:30 descarga.parte1
-rw-r--r-- 1 josee josee  48640 Jun  4 17:47 descarga.parte2
-rw-r--r-- 1 josee josee    261 May 30 01:54 headers.txt
drwxr-xr-x 1 josee josee      0 May 31 00:53 imagenes
-rw-r--r-- 1 josee josee 100352 Jun  4 17:47 merge.doc
-rw-r--r-- 1 josee josee 100352 May 30 00:55 word.doc
```

Listing 16: En consola #2: Se fusionan las 2 partes del archivo

6. Cuestionario

6.1. ¿Qué pasa cuando el cliente termina de recibir un archivo pero igual lo sigue pidiendo? ¿Por qué?

En el caso del prototipo, si el cliente vuelve a pedir el archivo se obtendrá un error 404 "No encontrado"por parte del servidor, debido a que este quita del diccionario de progresos a los clientes que ya terminaron de enviar el archivo solicitado. Para inciar una nueva descarga, el cliente debe volver a pedir un nuevo código único haciendo una llamada GET, sin parámetros

6.2. ¿Cómo funciona su protocolo si hay múltiples clientes? digamos que su servidor tiene la misma tasa de consultas que Google

La solución entregada es de un hilo (single thread), por lo que puede atender sólo a un cliente. En caso de habilitar multi hilo (multi thread, cuyo código se dejó comentado) el límite de la cantidad de clientes estará acotado por la cantidad que puedan ser almacenados en el diccionario de progreso. Es decir, por la cantidad de progresos que puedan asignarse en memoria RAM.

El largo del código único de cliente es de 36 caracteres, por ende, son 36 bytes. El progreso es de tipo entero. No existe una definición única del tamaño de un entero en Python, debido a que depende de la arquitectura. En el caso del computador del autor, un entero de Python es de 24 bytes (se puede averiguar con la sentencia sys.getsizeof(int()) del paquete sys)

Sumando, cada cliente, almacenado en memoria RAM, tendrá un peso de 60 bytes

Si, por ejemplo, se utilziara un servidor físico que tuviese 16 GB libres a disposición del script (sin contar sistema operativo y otros procesos), este podría registrar en memoria RAM hasta 286.331.153 clientes.

Ahora bien, tomando en cuenta otros aspecto como la red, se toma como ejemplo la conexión de internet del autor, que es de 200 MB del ISP Claro, que registra 10 MB por segundo de subida (`https://www.speedtest.net/result/9549991722`). Entonces, considerando que a cada cliente se le envían paquetes de datos TCP de 512 bytes, el servidor podrá responder sólo a 20.480 clientes por segundo

6.3. A partir de la pregunta anterior ¿Cambia su respuesta si en vez de texto responde con multimedia (tamaño ¿1 MB)? ¿Por qué?

No cambiaría porque, por cliente, se almacena un identificador y su progreso, más el tamaño del "chunk". Por ende, el tamaño del archivo no afecta la cantidad de datos almacenados en la memoria RAM del servidor. Además, el servidor carga el archivo en memoria, en "chunks"de tamaño arbitrario, en binario (se

puede apreciar el parámetro “b”de la apertura de archivo en el código Python) y siempre se envía en bytes, sin importar el tipo de dato del archivo

Diferente sería el caso que fuese un *streaming* multimedia, lo cual esta solución no sería la apropiada, ya que TCP no está diseñado para ese tipo de situaciones

6.4. ¿Su solución sigue siendo válida si se usan proxies http? ¿Por qué?

Si se trata de un proxy transparente, es válida, porque las respuestas que emite el servidor cuentan con cabeceras que cumplen con HTTP 1.1

Si el servidor proxy hiciera algo más sofisticado, como intervenir el orden de los parámetros de la petición GET, o solicitar una archivo con un *offset*, el servidor podría dejar de funcionar

7. Apéndice: evidencia de múltiples clientes

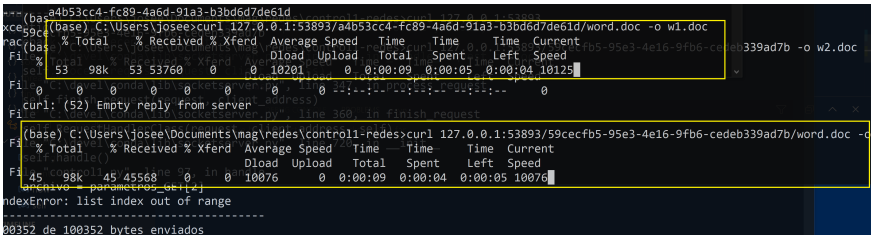

Figura 2: *Screenshot* de 2 clientes simultáneos usando la librería *threading* de Python

Referencias

[1] Documentación oficial del *framework* socketserver de Python 3 `https://docs.python.org/3/library/socketserver.html`, visitado durante Mayo del 2020

[2] RFC 2616 `https://www.ietf.org/rfc/rfc2616.txt`

[3] Rhodes, B., & Goerzen, J. (2010). Foundations of Python network programming: the comprehensive guide to building network applications with Python. New York: Apress.