

REDES Y SISTEMAS DISTRIBUIDOS 2025

Lab 2: Aplicación del servidor

GRUPO 16:

Franicevich Gabriel
Flores Rodrigo
Loyola Antonella
Yut Carolina



Objetivos de desarrollar un protocolo de transferencias de datos

- Aplicar la comunicación cliente/servidor por medio de la programación de sockets, desde la perspectiva del servidor.
- Familiarizarse con un protocolo de aplicación diseñado en casa.
- Comprender, diseñar e implementar un programa servidor de archivos en Python.



CLIENTE-SERVIDOR

SERVIDOR:

- °Se crea en una dirección IP y un puerto conocido.

- °En este caso escucha y responde a los siguientes pedidos:

- Listado de archivos (get_file_listing)
 - Información sobre archivos (get_metadata)
 - Fragmentos de archivos (get_slice)

- °Codifica y decodifica datos usando Base64.

CLIENTE:

- °Envía comandos al servidor para solicitar recursos.

- °Se conecta a un servidor remoto o local

- °No necesariamente tiene una IP fija

- °Se comunica a través de un puerto TCP

- °Envía comandos de forma secuencial al servidor

- °Espera y procesa las respuestas antes de enviar el siguiente comando

- °Puede pedir archivos completos, partes de archivos, o metadatos

- ° Interpreta las respuestas del servidor según el protocolo HFTP.



PROTOCOLO HFTP: Home-made File Transfer Protocol

- El cliente establece una conexión TCP con el servidor en cualquier puerto.
- El cliente envía un pedido (comando HFTP) en texto ASCII, finalizado con `\r\n`.
- El servidor recibe el pedido, lo interpreta y valida sus argumentos.
- El servidor responde con un código (por ejemplo, 0 OK).
- El cliente puede seguir enviando más comandos mientras la conexión siga activa.
- Cuando el cliente envía el comando quit, el servidor responde y cierra la conexión.



PARTES DEL PROYECTO

1. Servidor principal que acepta conexiones entrantes – **server.py**
2. Manejador de comandos y lógica de comunicación – **connection.py**
3. Cliente que se conecta al servidor y solicita recursos – **client.py**
4. Pruebas automáticas para validar respuestas y errores – **server-test.py**

server.py

```
class Server(object):
    """
    El servidor, que crea y atiende el socket en la dirección y puerto
    especificados donde se reciben nuevas conexiones de clientes.
    """

    def __init__(self, addr=DEFAULT_ADDR, port=DEFAULT_PORT,
                  directory=DEFAULT_DIR):
        print("Serving %s on %s:%s." % (directory, addr, port))
        # Crear socket del servidor, configurarlo, asignarlo
        # a una dirección y puerto, etc.

        # Creamos un socket TCP
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        # Reutilizamos la dirección si esta en TIME_WAIT
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

        # Lo asociamos a la dirección y el puerto
        self.sock.bind((addr, port))

        # lo ponemos a escuchar (hasta 5 conexiones en cola)
        self.sock.listen(5)

        # guardamos el directorio para usar en Connection
        self.directory = directory
```

```
def serve(self):
    """
    Loop principal del servidor. Se acepta una conexión a la vez
    y se espera a que concluya antes de seguir.
    """
    while True:
        try:
            # Aceptar una conexión al server, crear una
            # Connection para la conexión y atenderla hasta que termine.
            # espera y acepta una nueva conexión entrante
            client_socket, addr = self.sock.accept()
            print("Connected by:", addr)

            # crea una instancia de connection con el socket del cliente
            conn = connection.Connection(client_socket, self.directory)

            # crea y lanza un hilo para manejar la conexión
            thread = threading.Thread(target=conn.handle)
            thread.start()

        except KeyboardInterrupt:
            print("\n[INFO] Servidor detenido por el usuario.")
            break

        except Exception as e:
            print(f"[ERROR] Error atendiendo la conexión: {e}")
```

connection.py: Comandos

```
def handle_get_slice(self, args):
    #validacion de cantidad de argumentos
    if self.invalid_args(args, 3):
        return

    #separa los 3 argumentos
    filename, offset_str, size_str = args

    #validacion de que offset y size son numeros
    if not offset_str.isdigit() or not size_str.isdigit():
        response = f"{INVALID_ARGUMENTS} {error_messages[INVALID_ARGUMENTS]}{EOL}"
        self.socket.send(response.encode('ascii'))
        return

    #convertir offset y size a enteros (porq los necesito como num para leer el archivo)
    offset = int(offset_str)
    size = int(size_str)

    #concatenar el dir base del servidor con el nombre del archivo pedido
    filepath = os.path.join(self.directory, filename)

    #verif q el archivo existe
    if not os.path.isfile(filepath):
        response = f"{FILE_NOT_FOUND} {error_messages[FILE_NOT_FOUND]}{EOL}"
        self.socket.send(response.encode('ascii'))
        return

    #CASO BORDE
    #ver que el offset no sea mayor que el tamaño del archivo
    file_size = os.path.getsize(filepath)
    if offset + size > file_size:
        response = f"{BAD_OFFSET} {error_messages[BAD_OFFSET]}{EOL}"
        self.socket.send(response.encode('ascii'))
        return

    #abro el archivo en modo binario
    with open(filepath, 'rb') as f:
        f.seek(offset) #va al byte indicado
        chunk = f.read(size) #lee la cantidad pedida de bytes

    #codificar en base64
    encoded = b64encode(chunk).decode('ascii')
    #enviar el 0.OK\r\n si todo salió bien
    response = f"{CODE_OK} {error_messages[CODE_OK]}{EOL}"
    self.socket.send(response.encode('ascii'))

    #finalmente se mandan los datos codificados
    self.socket.send((f"{encoded}{EOL}").encode('ascii'))
```

```
def handle_get_file_listing(self, args):
    """
    os.listdir(dir)      Devuelve archivos y carpetas en dir
    os.path.isfile()      Comprueba si es un archivo
                        (No queremos devolver carpetas)

    os.path.join(dir)    Coloca / \ dependiendo del SO
    """

    if self.invalid_args(args, 0):
        return

    filenames = [
        f for f in os.listdir(self.directory)
        if os.path.isfile(os.path.join(self.directory, f))
    ]

    response = f"{CODE_OK} {error_messages[CODE_OK]}{EOL}"
    self.socket.send(response.encode('ascii'))

    for name in filenames:
        response = f"{name}{EOL}"
        self.socket.send(response.encode('ascii'))

    self.socket.send(EOL.encode('ascii'))

def handle_get_metadata(self, args):
    if self.invalid_args(args, 1):
        return

    filename = args[0]

    file_path = os.path.join(self.directory, filename)

    if not os.path.isfile(file_path):
        response = f"{FILE_NOT_FOUND} {error_messages[FILE_NOT_FOUND]}{EOL}"
        self.socket.send(response.encode('ascii'))
        return
    file_size = os.path.getsize(file_path) #devuelve el tam de un archivo en bytes
    response = f"{CODE_OK} {error_messages[CODE_OK]}{EOL}{file_size}{EOL}"
    self.socket.send(response.encode('ascii'))

def handle_quit(self, args):
    if self.invalid_args(args, 0):
        return
    response = f"{CODE_OK} {error_messages[CODE_OK]}{EOL}"
    self.socket.send(response.encode('ascii'))
    self.connect = False

def invalid_args(self, args, size):
    if len(args) != size:
        response = f"{INVALID_ARGUMENTS} {error_messages[INVALID_ARGUMENTS]}{EOL}"
        self.socket.send(response.encode('ascii'))
        return True
    return False
```




Estrategias para atender múltiples clientes simultáneamente

- Threads

Por cada cliente se lanza un nuevo hilo que se encarga de atender su conexión.

- Multiprocessing

Por cada cliente se crea un nuevo proceso, puede usar múltiples núcleos y si un proceso falla no afecta al resto

- I/O multiplexing

permite manejar muchos sockets simultáneamente desde un solo hilo o proceso, esperando actividad en cada conexión.



Diferencias entre localhost y 0.0.0.0

localhost y **127.0.0.1** se refieren a la interfaz del equipo local, al iniciar un servidor con esta dirección solo aceptara conexiones provenientes del mismo equipo. En cambio, **0.0.0.0** nos permite que el servidor reciba conexiones en las redes disponibles y esto permite que las conexiones sean tanto locales como remotas (otro dispositivo conectado a la misma red local).



¿Cómo funciona el paradigma cliente/servidor? ¿Cómo se ve esto en la programación con sockets?

Cómo funciona el paradigma cliente/servidor:

- El cliente inicia la comunicación con el servidor.
- El cliente solicita recursos al servidor.
- El servidor responde a la solicitud.
- El cliente y el servidor se comunican mediante protocolos de red.

Cómo funciona la programación con sockets:

- Los sockets establecen una conexión entre el cliente y el servidor.
- El servidor espera las solicitudes de los clientes.
- El cliente y el servidor intercambian información.
- El cliente y el servidor se desconectan.



¿Cual es la diferencia entre Stream (TCP) y Datagram (UDP), desde la perspectiva del socket?

Son pilares fundamentales de internet, permitiendo diferentes tipos de transmisión de datos desde un origen de red hasta su destino. TCP es un protocolo orientado a la conexión, lo que significa que se establece una conexión entre dos dispositivos antes de que comience la comunicación, mientras que UDP es un protocolo sin conexión, se utiliza en situaciones en las que la velocidad es más importante que la confiabilidad de datos. prioriza la velocidad y la eficiencia. Por ejemplo TCP se utiliza para transferencia de archivos, correo electrónico, navegación por la web, transacciones bancarias en línea y UDP se utiliza por ejemplo para transmisión en tiempo real, videojuegos, sistemas de monitoreo.



¿Qué es el protocolo FTP? ¿Para qué sirve?

Las siglas FTP significa protocolo de transferencia de archivos, su nombre indica que es un protocolo que permite transferir archivos directamente de un dispositivo a otro.

El funcionamiento básico de FTP:

FTP opera estableciendo dos conexiones entre el cliente y el servidor: Conexión de control: Se utiliza para enviar comandos y recibir respuestas entre el cliente y el servidor. Conexión de datos: Se encarga de la transferencia real de archivos.

Base64




¿Qué es?

>> Es un método de codificación que convierte datos binarios en texto ASCII, permitiendo que se transmitan de forma segura por protocolos que solo aceptan texto.

¿Para qué lo usamos en este lab?

>> Usamos Base64 para poder enviar archivos binarios convertidos a texto por la conexión TCP.

Esto también evita que secuencias como `\r\n` dentro del contenido del archivo se confundan con los finales de línea del protocolo.



¿Qué pasa si queremos enviar un archivo contiene los caracteres `\r\n`? ¿Cómo lo soluciona esto su código?

Si un archivo contiene los caracteres `\r\n`, podrían interpretarse como el final de una línea o comando, lo cual rompería el protocolo y generaría errores en la lectura o interpretación de los datos.

Nuestro código soluciona esto codificando el contenido del archivo en **Base64** antes de enviarlo. De esta forma:

- Todo se convierte en texto ASCII seguro.
- Los `\r\n` originales del archivo ya no están presentes como tales, sino como parte de una secuencia codificada.
- El receptor puede decodificar el contenido sin confusión ni errores en el protocolo.

Así, evitamos ambigüedades al transmitir archivos que contienen cualquier tipo de datos, incluso binarios o texto con saltos de línea.

Conclusiones

Este trabajo nos permitió entender cómo un protocolo de aplicación como HFTP puede abstraernos de los detalles complejos de transporte (como TCP), permitiéndonos enfocarnos en diseñar comandos claros y respuestas organizadas.

En comparación con labs anteriores, trabajar sobre un protocolo definido por nosotros facilitó la implementación y el testing, ya que el flujo de mensajes es predecible y estructurado.

¡Gracias! Nos vemos en el lab 3.

