




REDES Y SISTEMAS DISTRIBUIDOS 2024

PROYECTO: APLICACIÓN SERVIDOR – PROTOCOLO HFTP.
UNC – FAMAF

Integrantes:

- Ávalos Franco Ezequiel.
 - Rodriguez Braian Nicolas.
 - Sosa Joaquin.
- 

CONTENIDO E IMPORTANCIA DEL PROYECTO

- **OBJETIVO:** Diseño de un protocolo de transferencia de archivos seguro, eficiente y robusto.
- **IMPORTANCIA:**
 - Transmisión de datos/información mediante una comunicación.
 - Comunicación segura y eficiente.
 - Garantizar disponibilidad y confidencialidad de información durante una transmisión.
 - Esencial para el funcionamiento adecuado de redes e internet.

CLIENTE/SERVIDOR

- El paradigma cliente-servidor funciona de la siguiente manera:
- CLIENTE:

El cliente envía solicitudes a un servidor para acceder a datos, archivos y/o servicios.

Luego de que el servidor procese las solicitudes, este recibe las respuestas.
- SERVIDOR:

El servidor recibe solicitudes, procesa las solicitudes y envía la respuesta al cliente acorde a lo pedido.

En caso de error, envía un mensaje detallando lo sucedido.

CLIENTE/SERVIDOR – CON SOCKETS

- CLIENTE:

- 1) El cliente crea un socket y establece conexión con el servidor en el puerto con dirección IP al que desea conectarse.
- 2) Una vez establecida la conexión, envía solicitudes.
- 3) Recibe respuestas del servidor.
- 4) Luego de completar la comunicación, se cierra la conexión.

- SERVIDOR:

- 1) El servidor crea un socket en un puerto específico.
- 2) Espera las solicitudes de los clientes.
- 3) Cuando se establece la conexión, crea un nuevo socket para esa conexión.
- 4) Utiliza el socket nuevo para recibir solicitudes, procesarlas y enviar respuestas.

TCP vs UDP – Perspectiva del socket

TCP:

- 1) Proporciona un flujo de datos orientado a la conexión confiable entre 2 puntos en la red.
- 2) Los sockets TCP están asociados con un flujo de bytes continuo y bidireccional.
- 3) Proporciona control de flujo y congestión para garantizar una transmisión confiable y eficiente.
- 4) Esencial para quien requiera una transferencia confiable.

UDP:

- 1) Proporciona un servicio no orientado a la conexión.
- 2) NO garantiza la entrega de datos ni en orden.
- 3) Proporciona una transmisión de datos rápida y eficiente ya que no proporciona control de flujo y de errores.
- 4) Esencial para quien requiera velocidad y latencia antes que confiabilidad.

PROTOCOLO FTP

El protocolo FTP es un protocolo de transferencia de archivos.

UTILIDAD:

- Usado para transferir archivos hacia/desde host remoto.
- Cada archivo tiene restricciones de acceso y posesión.
- FTP permite inspeccionar carpetas
- FTP permite mensajes de control textuales.
- Utiliza el modelo cliente/servidor.

3 tipos de mensajes son intercambiados:

- Comandos enviados al servidor FTP
- Mensajes de respuesta a comandos del servidor FTP
- Mensajes con datos enviados.

BASE64

- Base64 es un método de codificación de datos que convierte datos binarios en una representación ASCII legible.
- En nuestro caso lo utilizamos para:
 - Entregar el contenido de un archivo de manera legible.
 - En caso de encontrar el string `\r\n` que indica el final de la línea, esto provocaría que deje de leer... pero, ¿qué pasa si es parte del contenido del texto del archivo?

Pues, en ese caso, usamos `read` ya que esta función lee los EOF. Entonces leeríamos los EOL como si fuera un carácter normal ya que los ignora.

FUNCIONES PRINCIPALES DEL SERVIDOR

- Las funciones principales son las siguientes: `COMAND_HANDLER`.

```
def command_handler(self, command: str):  
    cmd = command.split()  
  
    if cmd[0] == 'get_file_listing':  
        self.get_file_listing()  
  
    elif cmd[0] == 'get_metadata':  
        if len(cmd) != 2:  
            self.send(string_code_message(INVALID_ARGUMENTS))  
  
            elif (isinstance(cmd[1],str)): # Ver que el offset o el size sea valido para cmd[1]  
                self.get_metadata(cmd[1])  
            else:  
                self.send(string_code_message(INVALID_ARGUMENTS))  
  
    elif cmd[0] == 'get_slice':  
        if len(cmd) != 4:  
            self.send(string_code_message(INVALID_ARGUMENTS))  
  
            elif(isinstance(cmd[1],str)) and cmd[2].isdecimal() and cmd[3].isdecimal():  
                print(cmd)  
                self.get_slice(cmd[1], int(cmd[2]), int(cmd[3])) #MUY IMPORTANTE, mandar el offset y el size como int, sino toma error  
            else:  
                self.send(string_code_message(INVALID_ARGUMENTS)) #Este ELSE no cambia nada (CREO). Lo dejo por las dudas.  
  
    elif cmd[0] == 'quit':  
        if len(cmd) == 1:  
            self.quit()  
        else:  
            self.send(string_code_message(INVALID_ARGUMENTS))  
  
    else:  
        self.send(string_code_message(INVALID_COMMAND))
```


FUNCIONES PRINCIPALES DEL SERVIDOR

- Las funciones principales son las siguientes: HANDLE.

```
def handle(self, conexion):  
    self.threading.acquire() #permitir al hilo que use el socket  
    def handler():  
        try:  
            conexion.handle()  
        finally:  
            self.threading.release()  
  
    thread = threading.Thread(target = handler) #mandamos el hilo a connection.py  
    thread.start()
```

FUNCIONES PRINCIPALES DEL SERVIDOR

- Las funciones principales son las siguientes: GET_SLICE.

```
def get_slice(self, filename, offset:int, size:int):
    #
    # El servidor responde con el fragmento de archivo pedido codificado en base64 y un \r\n.
    #
    print(os.listdir(self.directory))
    if not (os.path.isfile(os.path.join(self.directory, filename))): #Si el archivo no existe...
        respuesta = string_code_message(FILE_NOT_FOUND)
        self.send(respuesta)
    else:
        realsize = os.path.getsize(os.path.join(self.directory, filename))
        print(offset)
        print(size)
        print(realsize)
        print(offset + size)
        if offset < 0 or offset + size > realsize:
            respuesta = string_code_message(BAD_OFFSET)
            self.send(respuesta)
        else:
            archivoname = os.path.join(self.directory, filename)
            respuesta = string_code_message(CODE_OK)
            self.send(respuesta)
            fd = open(archivoname, 'rb')          #Abrimos el archivo}
            fd.seek(offset)
            resto = size
            while (resto > 0):
                bts = fd.read(resto)
                resto -= 1
                #bts = os.pread(fd, size, offset)          # Toma un fd, size y offset, con eso leemos desde el offset la
                                                         # cantidad de 'offset' bytes. Esto devuelve un bytestring
                bts = b64encode(bts)                    # SSComo b64encode toma un string ascii, por eso lo pasamos antes. Devolvemos el
                self.send(bts)
            respuesta = ''
            self.send(respuesta)
```

FUNCIONES PRINCIPALES DEL SERVIDOR

- Las funciones principales son las siguientes: GET_FILE_LISTING.

```
def get_file_listing(self):  
    #  
    # El servidor responde con una secuencia de líneas terminadas en \r\n,  
    # cada una con el nombre de los archivos disponibles.  
    #  
  
    respuesta = string_code_message(CODE_OK) + EOL #Respuesta = Código OK! + \r\n  
  
    for directorio in os.listdir(self.directory): # Para directorio en la lista de directorios del self.directory,  
        respuesta = respuesta + f"{directorio} {EOL}" # lo agregamos a la respuesta.  
    self.send(respuesta)
```

FUNCIONES PRINCIPALES DEL SERVIDOR

- Las funciones principales son las siguientes: GET_METADATA.

```
def get_metadata(self, filename):  
    #  
    # El servidor responde con una cadena indicando su valor en bytes.  
    #  
  
    if not (os.path.isfile(os.path.join(self.directory, filename))): #isfile -> ¿Es un archivo? #Join -> Entra al archivo  
        respuesta = string_code_message(FILE_NOT_FOUND)  
    else:  
        respuesta = string_code_message(CODE_OK) + EOL # code_ok + /r/n  
        bts = os.path.getsize(os.path.join(self.directory, filename)) #bytes + /r/n  
        respuesta = respuesta + f"{bts}" #code ok + /r/n ESPACIO bytes /r/n  
    self.send(respuesta)
```

ERRORES Y DIFICULTADES - RESOLUCIÓN

- A la hora de correr los tests y de probar nuestra implementación nos encontramos con lo siguiente:
 - El test se colgaba y nunca salía.
 - Tests fallidos.
 - Tests incompletos (algunos no se evaluaban).

Para resolver todo lo dicho, lo que hicimos fue ver en que test estaba el error y tratar de comprender de qué se trataba.

Para ello, tuvimos que re-leer documentación, implementar casos para el manejo de errores, etc.

- Uno de ellos fue (por e.g) en la función `GET_FILE_LISTING()` donde nuestra función no devolvía un EOL al final luego de cada archivo brindado.

RELACIÓN Y CONCLUSIÓN

- La principal diferencia con el otro laboratorio es que en este trabajamos del lado del servidor y no del cliente.

La relación con el laboratorio anterior es que seguimos manejando una comunicación CLIENTE/SERVIDOR.

La abstracción nos permitía no preocuparnos por cómo estaba implementado, pero en este laboratorio si. - Las APIs están en una capa superior y es mas fácil abstraerse de como se comunican exactamente el servidor y el cliente -.

- En conclusión, seguimos aprendiendo sobre esta comunicación que se efectúa entre cliente/servidor. Además seguimos aprendiendo Python, que fue nuestro lenguaje de programación para este y el anterior laboratorio.

FIN DEL PROYECTO Y PRESENTACIÓN

- Agradecemos a todos por tomarse el tiempo de ver nuestra presentación sobre el proyecto sobre: Comunicación cliente/servidor y el diseño de protocolo para la comunicación. Esperamos haber expresado bien nuestros conocimientos sobre este proyecto.

REDES Y SISTEMAS DISTRIBUIDOS 2024

UNC – FAMAF