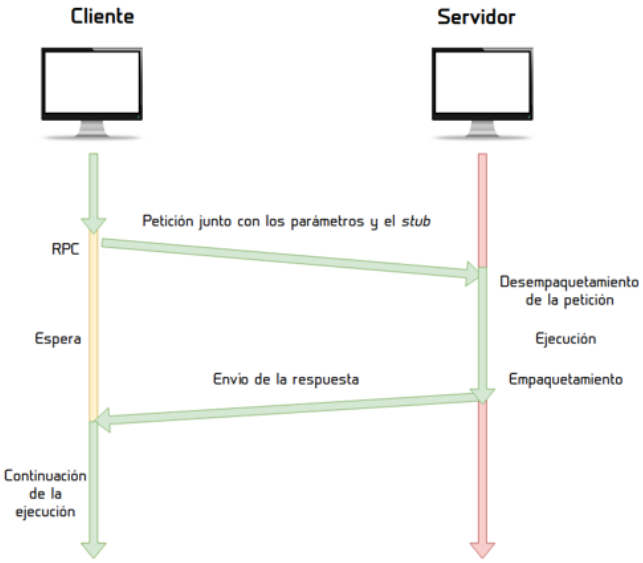
Protocolo RPC

El Protocolo de Llamada a Procedimiento Remoto o RPC (Remote Procedure Call) es un protocolo que permite a un programa ejecutar funciones o procedimientos en otro equipo (servidor) dentro de una red como si fuera una función local. El RPC es una forma de comunicación entre procesos que abstrae la complejidad de la red y oculta los detalles de la transmisión de mensajes entre cliente y servidor.

El funcionamiento se puede visualizar en la siguiente imagen:



Cliente: El programa que solicita la ejecución de una función remota llama a un procedimiento.

Stub Cliente: El cliente llama a una función "stub", que convierte los parámetros de la función en un formato adecuado para enviarlos a través de la red (serialización).

Transporte: Los parámetros serializados se envían a través de la red hacia el servidor.

Stub Servidor: El servidor recibe los datos, los deserializa y ejecuta la función solicitada.

Resultado: La respuesta es enviada de vuelta al cliente mediante el mismo proceso de transporte.

Aplicaciones de RPC

Sistemas Distribuidos: Facilita la creación de sistemas en los que los componentes están distribuidos en varios servidores.

Servicios Web: RPC es un precursor de protocolos más modernos como SOAP o REST, y sigue siendo utilizado en algunas arquitecturas para invocar servicios distribuidos.

Administración de Sistemas: Se utiliza en sistemas operativos para gestionar recursos remotos, como en NFS (Network File System) y NIS (Network Information Service).

Microservicios: Aunque muchos microservicios utilizan HTTP/REST, algunos utilizan gRPC (una implementación moderna de RPC de Google).

Desarrollo:

Para la presente práctica se va a realizar un programa que permita consultar, leer y enviar archivos a un servidor utilizando el protocolo RPC.

Para ello el lenguaje a utilizar será Python aunque se puede realizar en cualquier lenguaje de programación utilizando las bibliotecas adecuadas.

Se va a realizar del lado del cliente una interfaz gráfica con apoyo de la biblioteca de tkinter, para ello también se utiliza la biblioteca de xmlrpc.

Comenzaremos estableciendo la lógica para el cliente para ello crearemos un archivo con el nombre de cliente-rpc.py:

```
import tkinter as tk
from tkinter import messagebox, filedialog
import xmlrpc.client
import base64
```

Esta función se encarga de conectarse al servidor RPC y obtener una lista de archivos almacenados en el servidor. Utiliza el método remoto list_files() para recibir esta lista, limpia cualquier contenido previo en la lista visual (Listbox) de la GUI, y luego inserta los nombres de los archivos que ha recibido para mostrarlos al usuario. Si algo falla durante la solicitud al servidor, muestra un mensaje de error.

```
def listar_archivos():
    try:
        archivos = proxy.list_files(".")
        listbox.delete(0, tk.END)  # Limpiar la lista
        for archivo in archivos:
            listbox.insert(tk.END, archivo)  # Añadir archivo a la lista
    except Exception as e:
        messagebox.showerror("Error", f"No se pudieron listar los archivos: {e}")
```

Esta función permite al usuario seleccionar un archivo de la lista de archivos disponibles en el servidor, y luego recupera su contenido utilizando la función remota read_file() del servidor. Una vez que obtiene el contenido del archivo, lo despliega en un área de texto en la interfaz gráfica. También incluye manejo de errores en caso de que el archivo no pueda ser leído correctamente o no se seleccione un archivo.

```
def leer_archivo():
    archivo_seleccionado = listbox.get(tk.ACTIVE)
    if archivo_seleccionado:
        try:
            contenido = proxy.read_file(archivo_seleccionado)
            text_area.delete(1.0, tk.END)
            text_area.insert(tk.END, contenido)  # Mostrar el contenido del
archivo en la GUI
        except Exception as e:
            messagebox.showerror("Error", f"No se pudo leer el archivo: {e}")
    else:
        messagebox.showwarning("Advertencia", "Por favor, selecciona un archivo
para leer.")
```

Esta función permite al usuario seleccionar un archivo local desde su sistema, lo lee en formato binario y lo codifica en base64 para poder enviarlo a través de XML-RPC al servidor. Utiliza el método remoto upload_file() del servidor para transferir el archivo. Una vez que el archivo se ha subido correctamente, la función actualiza la lista de archivos del servidor para reflejar el nuevo archivo disponible y muestra un mensaje de éxito. Si ocurre algún error, se notifica al usuario mediante un mensaje de error.

```
def enviar_archivo():
    # Seleccionar archivo local
    archivo_seleccionado = filedialog.askopenfilename()
    if archivo_seleccionado:
        try:
            # Leer el archivo y codificarlo en base64
            with open(archivo_seleccionado, "rb") as file:
                archivo_codificado = base64.b64encode(file.read()).decode('utf-
8')
            # Obtener solo el nombre del archivo
            nombre_archivo = archivo_seleccionado.split("/")[-1]
            # Enviar el archivo al servidor
            resultado = proxy.upload_file(nombre_archivo, archivo_codificado)
            messagebox.showinfo("Éxito", resultado)
            listar_archivos()  # Actualizar lista de archivos en el servidor
        except Exception as e:
            messagebox.showerror("Error", f"No se pudo enviar el archivo: {e}")
```

Del lado del servidor se va a establecer la lógica para que se pueda hacer la recepción de los archivos y se pueda establecer conexión con el cliente:

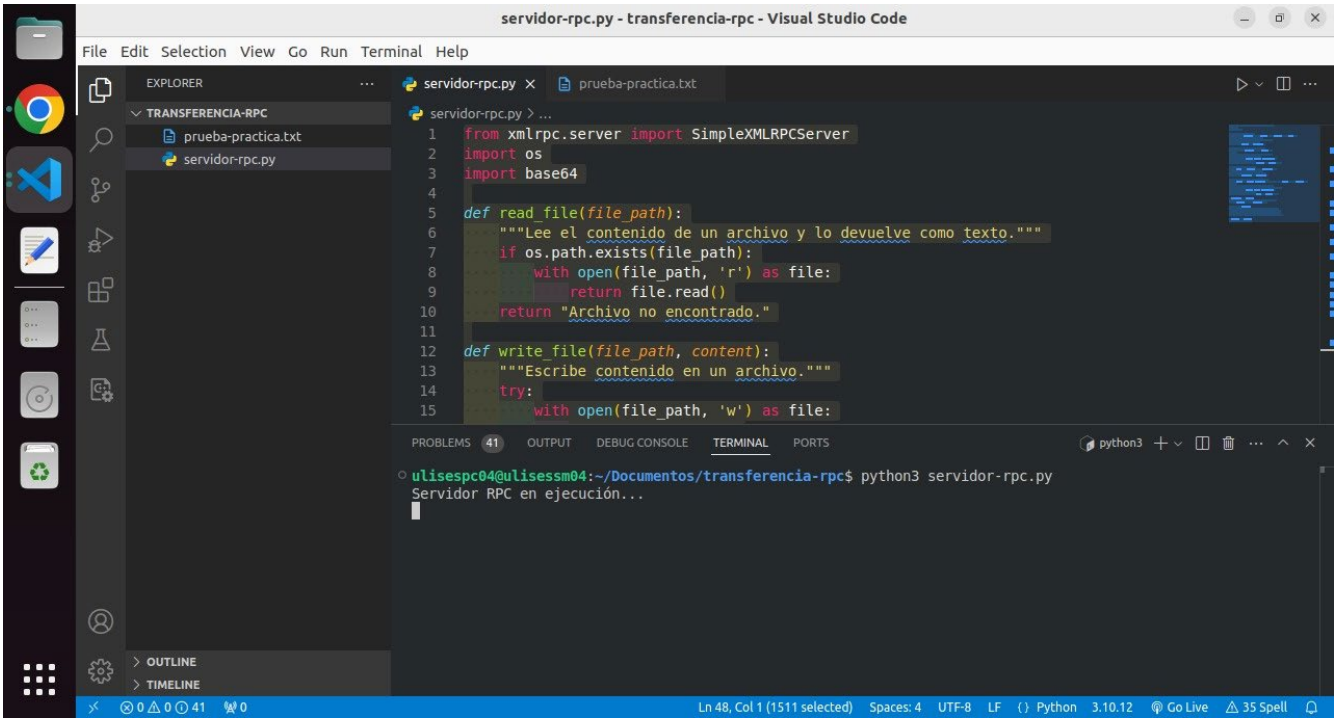
```
from xmlrpc.server import SimpleXMLRPCServer
import os
import base64

def read_file(file_path):
    """Lee el contenido de un archivo y lo devuelve como texto."""
    if os.path.exists(file_path):
        with open(file_path, 'r') as file:
            return file.read()
    return "Archivo no encontrado."

def write_file(file_path, content):
    """Escribe contenido en un archivo."""
    try:
        with open(file_path, 'w') as file:
            file.write(content)
        return "Archivo escrito exitosamente."
    except Exception as e:
        return f"Error al escribir el archivo: {e}"

def list_files(directory):
    """Lista los archivos de un directorio."""
    if os.path.exists(directory):
        return os.listdir(directory)
    return "Directorio no encontrado."

def upload_file(file_name, file_data):
```

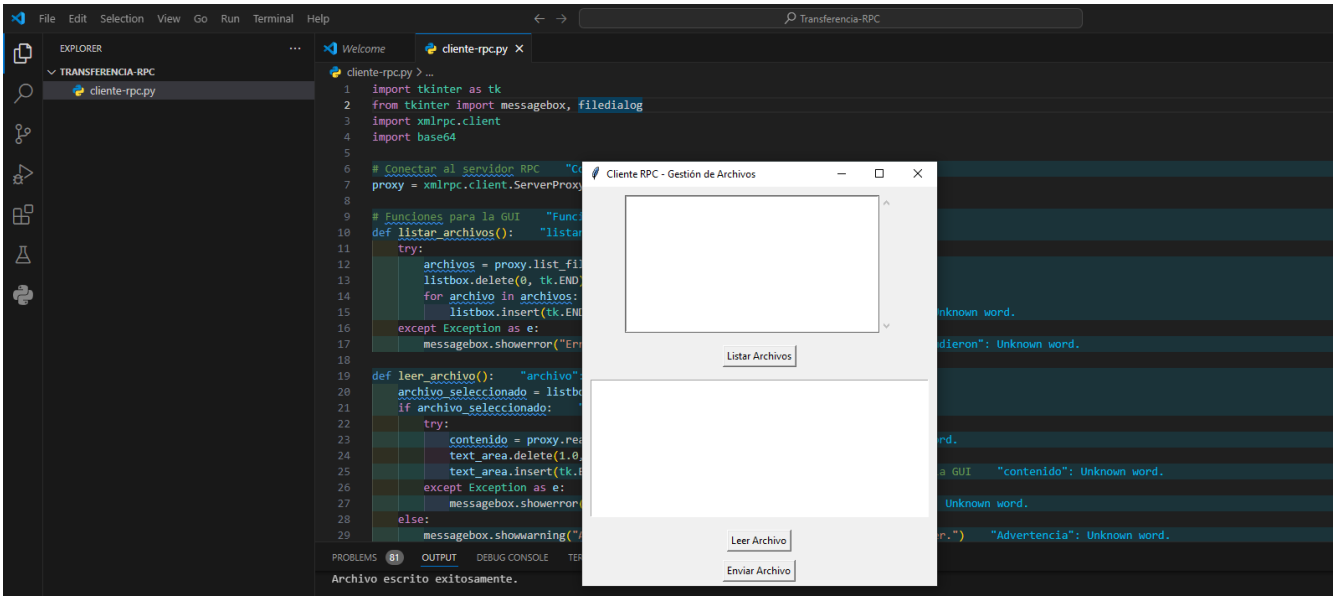
```
    """Recibe un archivo codificado en base64 y lo guarda en el servidor."""
    try:
        with open(file_name, 'wb') as file:
            file.write(base64.b64decode(file_data))
        return "Archivo subido exitosamente."
    except Exception as e:
        return f"Error al subir el archivo: {e}"

# Configurar el servidor
server = SimpleXMLRPCServer(("0.0.0.0", 8000))
print("Servidor RPC en ejecución...")

# Registrar funciones
server.register_function(read_file, "read_file")
server.register_function(write_file, "write_file")
server.register_function(list_files, "list_files")
server.register_function(upload_file, "upload_file")

# Ejecutar el servidor
server.serve_forever()
```

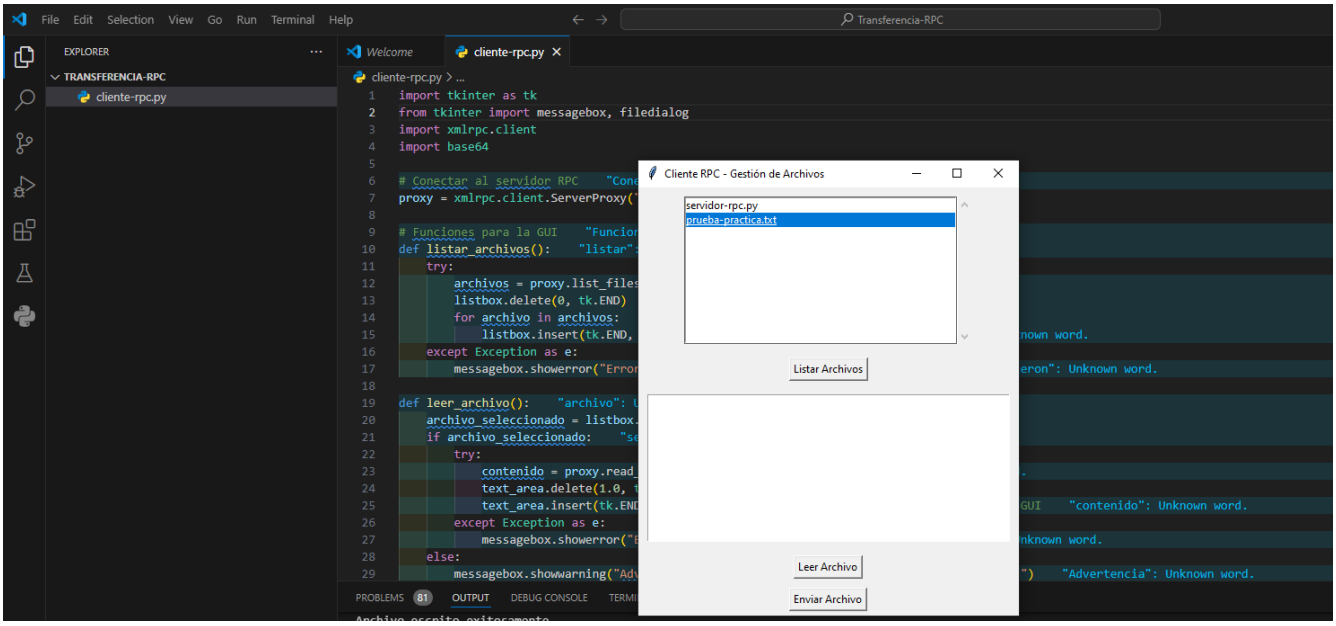
El servidor se encarga de ofrecer servicios relacionados con la gestión de archivos, permitiendo al cliente remoto leer, escribir, listar, y subir archivos utilizando llamadas RPC. El servidor se inicia utilizando SimpleXMLRPCServer, configurado para escuchar en todas las interfaces (0.0.0.0) y en el puerto 8000. Una vez iniciado, este servidor espera solicitudes de los clientes y ejecuta las funciones que se registran para su uso remoto.

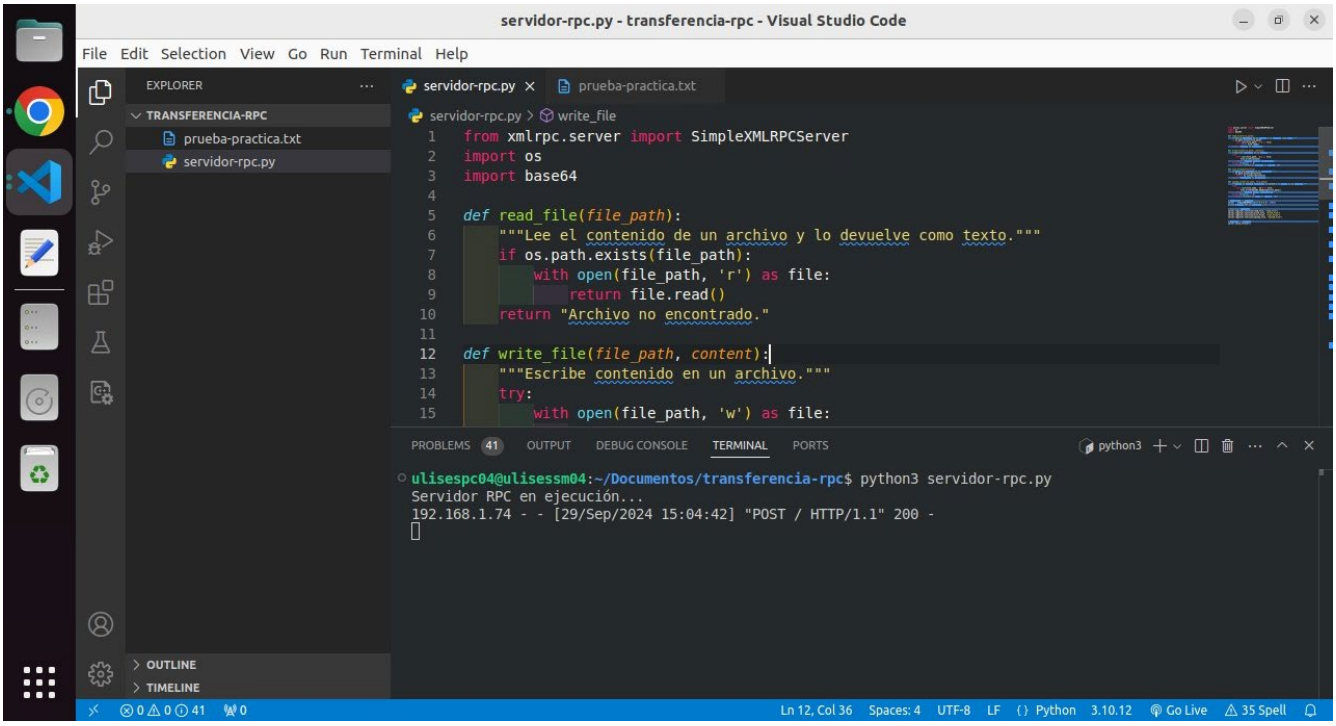
- **read_file(file_path)**: Esta función recibe la ruta de un archivo como argumento y se encarga de leer el contenido del archivo si existe. Abre el archivo en modo de lectura y devuelve su contenido en forma de texto. Si el archivo no existe, devuelve un mensaje que indica "Archivo no encontrado".
- **write_file(file_path, content):** Esta función permite escribir contenido en un archivo especificado. Recibe la ruta del archivo y el contenido que se va a escribir. Si el proceso de escritura se realiza con éxito, devuelve un mensaje de confirmación. Si ocurre algún error durante la operación, como un problema de permisos, devuelve un mensaje de error con la descripción del problema.
- **list_files(directory):** Esta función lista los archivos en un directorio dado. Verifica si el directorio especificado existe, y si es así, utiliza os.listdir() para devolver una lista de los nombres de los archivos en ese directorio. Si el directorio no se encuentra, devuelve un mensaje de error indicando que el directorio no existe.
- **upload_file(file_name, file_data):** Esta función se encarga de recibir un archivo codificado en base64 (en formato binario seguro para la transmisión) y guardarlo en el servidor. El cliente envía el archivo codificado como una cadena de texto, y la función lo decodifica usando base64.b64decode() y lo guarda con el nombre proporcionado. Si la operación es exitosa, devuelve un mensaje de confirmación; si ocurre un error (por ejemplo, durante la escritura del archivo), se captura la excepción y se devuelve un mensaje de error con la descripción.

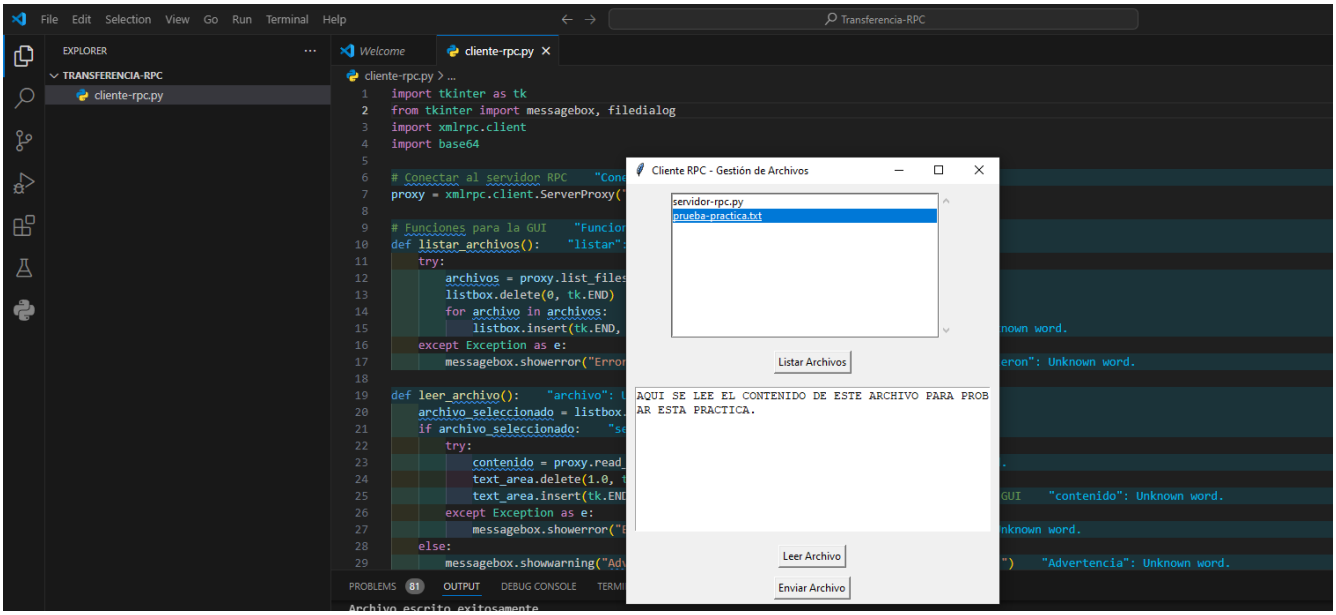
Ahora se va a probar el código donde el servidor se va a ejecutar en un sistema operativo Linux y el cliente en un sistema operativo de Windows:

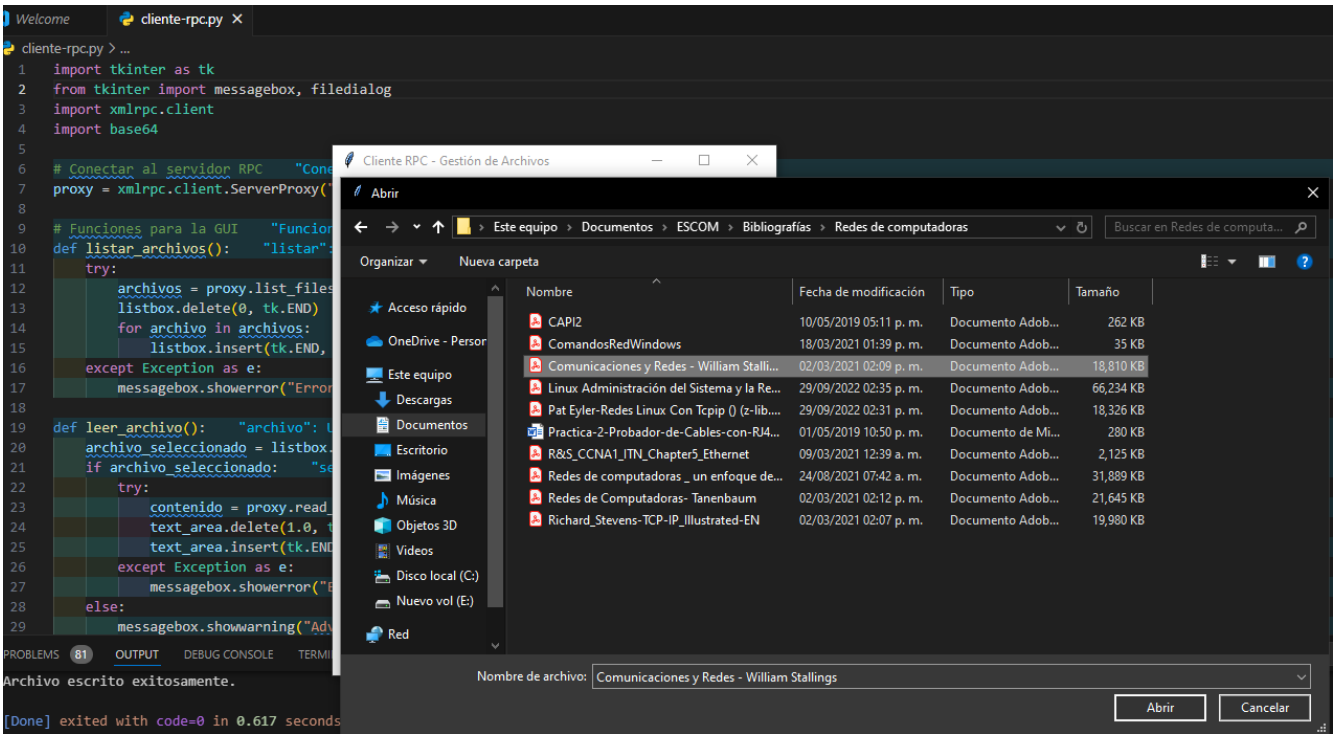


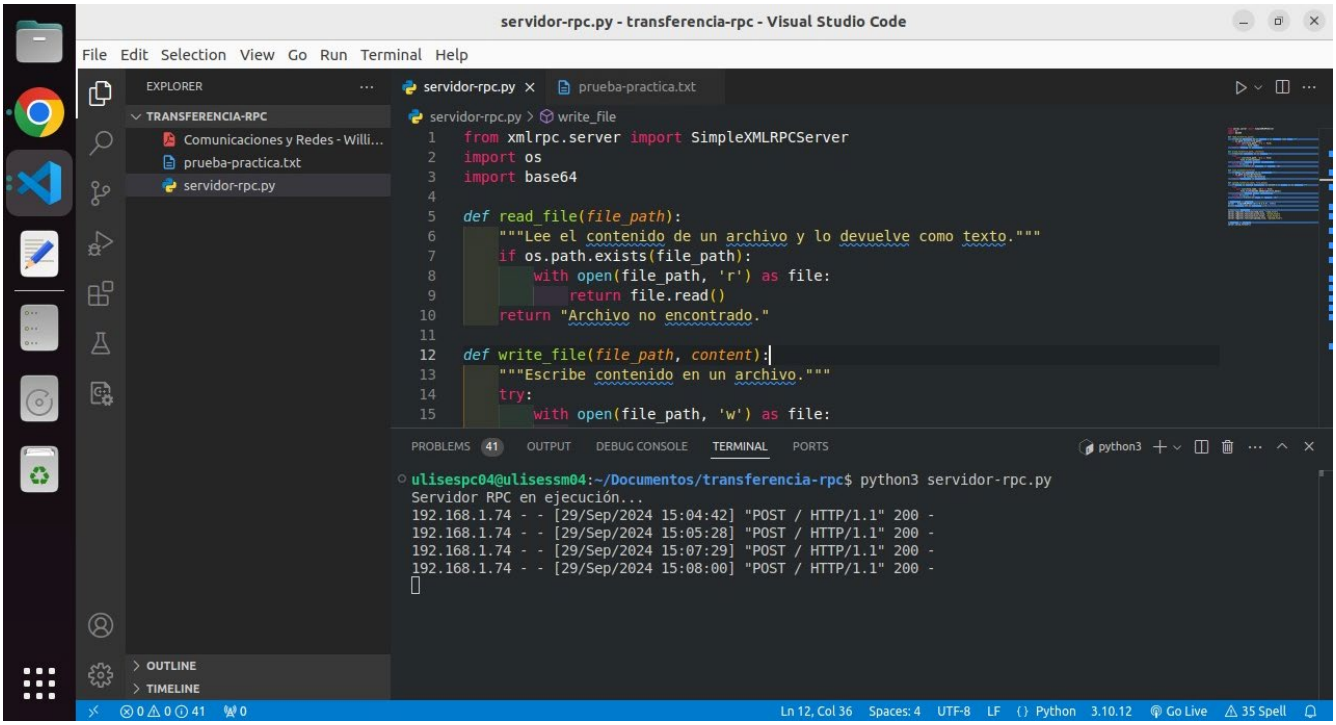


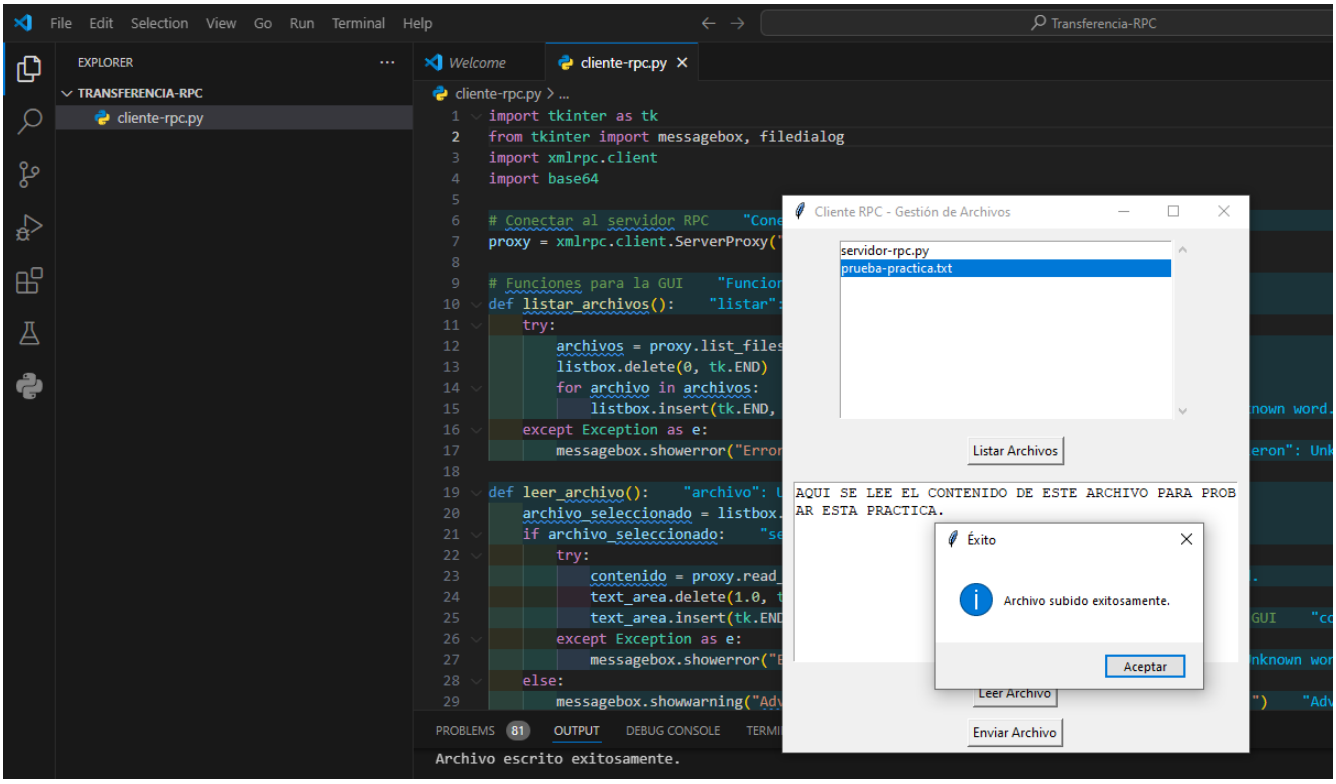

File Edit Selection View Go Run Terminal Help
Transferencia-RPC
EXPLORER
TRANSFERENCIA-RPC
cliente-rpc.py
Welcome
cliente-rpc.py
import tkinter as tk
from tkinter import messagebox, filedialog
import xmlrpc.client
import base64
Cliente RPC - Gestión de Archivos
servidor-rpc.py
prueba-practica.txt
Listar Archivos
Leer Archivo
Enviar Archivo
PROBLEMS 81
OUTPUT
DEBUG CONSOLE

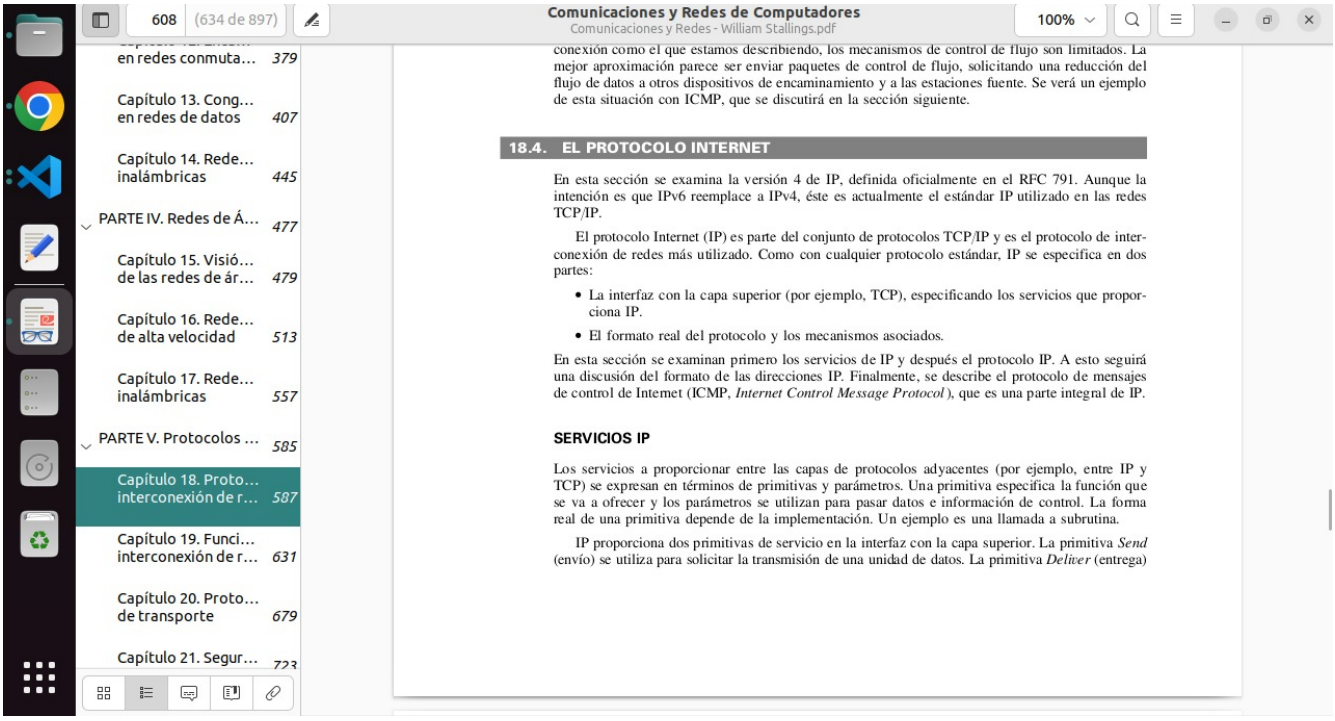

servidor-rpc.py - transferencia-rpc - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
TRANSFERENCIA-RPC
prueba-practica.txt
servidor-rpc.py
from xmlrpc.server import SimpleXMLRPCServer
import os
import base64
def read_file(file_path):
"""Lee el contenido de un archivo y lo devuelve como texto."""
if os.path.exists(file_path):
with open(file_path, 'r') as file:
return file.read()
return "Archivo no encontrado."
def write_file(file_path, content):
"""Escribe contenido en un archivo."""
try:
with open(file_path, 'w') as file:
PROBLEMS 41
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
ulisespc04@ulisessm04:~/Documentos/transferencia-rpc$ python3 servidor-rpc.py
Servidor RPC en ejecución...
192.168.1.74 - - [29/Sep/2024 15:04:42] "POST / HTTP/1.1" 200 -
OUTLINE
TIMELINE
Ln 12, Col 36
Spaces: 4
UTF-8
LF
Python
3.10.12
Go Live
35 Spell


Cliente RPC - Gestión de Archivos
servidor-rpc.py
prueba-practica.txt
Listar Archivos
AQUI SE LEE EL CONTENIDO DE ESTE ARCHIVO PARA PROBAR ESTA PRACTICA.
Leer Archivo
Enviar Archivo
Archivo escrito exitosamente.


Cliente RPC - Gestión de Archivos
Abrir
Este equipo › Documentos › ESCOM › Bibliografías › Redes de computadoras
Nombre
Fecha de modificación
Tipo
Tamaño
Comunicaciones y Redes - William Stalli... 02/03/2021 02:09 p. m. Documento Adob... 18,810 KB
Nombre de archivo: Comunicaciones y Redes - William Stallings
Abrir
Cancelar
Archivo escrito exitosamente.
[Done] exited with code=0 in 0.617 seconds


servidor-rpc.py - transferencia-rpc - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
TRANSFERENCIA-RPC
Comunicaciones y Redes - Willi...
prueba-practica.txt
servidor-rpc.py
servidor-rpc.py
prueba-practica.txt
servidor-rpc.py > write_file
from xmlrpc.server import SimpleXMLRPCServer
import os
import base64
def read_file(file_path):
"""Lee el contenido de un archivo y lo devuelve como texto."""
if os.path.exists(file_path):
with open(file_path, 'r') as file:
return file.read()
return "Archivo no encontrado."
def write_file(file_path, content):
"""Escribe contenido en un archivo."""
try:
with open(file_path, 'w') as file:
PROBLEMS 41
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
python3
ulisespc04@ulisessm04:~/Documentos/transferencia-rpc$ python3 servidor-rpc.py
Servidor RPC en ejecución...
192.168.1.74 - - [29/Sep/2024 15:04:42] "POST / HTTP/1.1" 200 -
192.168.1.74 - - [29/Sep/2024 15:05:28] "POST / HTTP/1.1" 200 -
192.168.1.74 - - [29/Sep/2024 15:07:29] "POST / HTTP/1.1" 200 -
192.168.1.74 - - [29/Sep/2024 15:08:00] "POST / HTTP/1.1" 200 -
OUTLINE
TIMELINE
Ln 12, Col 36
Spaces: 4
UTF-8
LF
Python
3.10.12
Go Live
35 Spell


File Edit Selection View Go Run Terminal Help
Transferencia-RPC
EXPLORER
TRANSFERENCIA-RPC
cliente-rpc.py
Welcome
cliente-rpc.py
cliente-rpc.py > ...
import tkinter as tk
from tkinter import messagebox, filedialog
import xmlrpc.client
import base64
# Conectar al servidor RPC
proxy = xmlrpc.client.ServerProxy("
# Funciones para la GUI
def listar_archivos():
try:
archivos = proxy.list_files
listbox.delete(0, tk.END)
for archivo in archivos:
listbox.insert(tk.END,
except Exception as e:
messagebox.showerror("Error
def leer_archivo():
archivo_seleccionado = listbox.
if archivo_seleccionado:
try:
contenido = proxy.read_
text_area.delete(1.0,
text_area.insert(tk.END
except Exception as e:
messagebox.showerror("E
else:
messagebox.showwarning("Adv
Cliente RPC - Gestión de Archivos
servidor-rpc.py
prueba-practica.txt
Listar Archivos
AQUI SE LEE EL CONTENIDO DE ESTE ARCHIVO PARA PROB
AR ESTA PRACTICA.
Éxito
Archivo subido exitosamente.
Aceptar
Leer Archivo
Enviar Archivo
PROBLEMS 81
OUTPUT
DEBUG CONSOLE
Archivo escrito exitosamente.


608 (634 de 897)
Comunicaciones y Redes de Computadores
Comunicaciones y Redes - William Stallings.pdf
100%
en redes conmuta... 379
Capítulo 13. Cong... en redes de datos 407
Capítulo 14. Rede... inalámbricas 445
PARTE IV. Redes de Á... 477
Capítulo 15. Visió... de las redes de ár... 479
Capítulo 16. Rede... de alta velocidad 513
Capítulo 17. Rede... inalámbricas 557
PARTE V. Protocolos ... 585
Capítulo 18. Proto... interconexión de r... 587
Capítulo 19. Funci... interconexión de r... 631
Capítulo 20. Proto... de transporte 679
Capítulo 21. Segur... 723
conexión como el que estamos describiendo, los mecanismos de control de flujo son limitados. La mejor aproximación parece ser enviar paquetes de control de flujo, solicitando una reducción del flujo de datos a otros dispositivos de encaminamiento y a las estaciones fuente. Se verá un ejemplo de esta situación con ICMP, que se discutirá en la sección siguiente.
18.4. EL PROTOCOLO INTERNET
En esta sección se examina la versión 4 de IP, definida oficialmente en el RFC 791. Aunque la intención es que IPv6 reemplace a IPv4, éste es actualmente el estándar IP utilizado en las redes TCP/IP.
El protocolo Internet (IP) es parte del conjunto de protocolos TCP/IP y es el protocolo de interconexión de redes más utilizado. Como con cualquier protocolo estándar, IP se especifica en dos partes:
• La interfaz con la capa superior (por ejemplo, TCP), especificando los servicios que proporciona IP.
• El formato real del protocolo y los mecanismos asociados.
En esta sección se examinan primero los servicios de IP y después el protocolo IP. A esto seguirá una discusión del formato de las direcciones IP. Finalmente, se describe el protocolo de mensajes de control de Internet (ICMP, Internet Control Message Protocol), que es una parte integral de IP.
SERVICIOS IP
Los servicios a proporcionar entre las capas de protocolos adyacentes (por ejemplo, entre IP y TCP) se expresan en términos de primitivas y parámetros. Una primitiva especifica la función que se va a ofrecer y los parámetros se utilizan para pasar datos e información de control. La forma real de una primitiva depende de la implementación. Un ejemplo es una llamada a subrutina.
IP proporciona dos primitivas de servicio en la interfaz con la capa superior. La primitiva Send (envío) se utiliza para solicitar la transmisión de una unidad de datos. La primitiva Deliver (entrega)