Imagen que contiene Icono

El contenido generado por IA puede ser incorrecto.Dibujo en blanco y negro

El contenido generado por IA puede ser incorrecto.

Instituto Politécnico Nacional

Escuela Superior de Cómputo

Ingeniería en Sistemas Computacionales

Aplicaciones para comunicaciones en red

Academia “Sistemas Distribuidos”

Plan 2020

Práctica 19:

“**Protocolo SNMP**”

2015090269

González González Armando Omar

Profesor: Ojeda Santillan Rodrigo

Contenido

[Objetivo 3](#_Toc201738331)

[Introducción 3](#_Toc201738332)

[Desarrollo 4](#_Toc201738333)

[3.1 Implementación del Servidor RPC (server.py) 4](#_Toc201738334)

[3.2 Implementación del Cliente RPC (client.py) 5](#_Toc201738335)

[Conclusión 10](#_Toc201738336)

[Pregunta 10](#_Toc201738337)

[¿Cómo podrían generar un negocio a través de lo visto en la práctica? 10](#_Toc201738338)

[Bibliografía 10](#_Toc201738339)

## Objetivo

Implementar una aplicación cliente-servidor para la gestión remota de archivos utilizando el Protocolo de Llamada a Procedimiento Remoto (RPC). El objetivo es desarrollar un servidor en Python que exponga funciones para listar, leer y subir archivos, y un cliente con interfaz gráfica (GUI) que consuma estos servicios, demostrando cómo RPC abstrae la comunicación en red para simplificar el desarrollo de sistemas distribuidos.

## Introducción

El Protocolo de Llamada a Procedimiento Remoto, o RPC (Remote Procedure Call), es un modelo de comunicación que permite a un programa ejecutar código en otra máquina a través de una red, de la misma forma que si se tratara de una función local. La principal ventaja de RPC es que oculta al programador los detalles complejos de la comunicación en red, como la serialización de datos y la transmisión de mensajes, presentando la interacción como una simple llamada a una función.

El funcionamiento básico sigue un flujo definido:

1. El cliente llama a una función local conocida como "stub". Este stub se encarga de empaquetar los parámetros de la función (serialización) en un formato adecuado para la transmisión.
2. Los datos serializados se envían a través de la red hacia el servidor.
3. Un "stub" en el servidor recibe la petición, desempaqueta los datos (deserialización) y llama a la función real con los parámetros recibidos.
4. Una vez que la función se ejecuta en el servidor, el resultado se empaqueta y se envía de vuelta al cliente, donde es desempaquetado y retornado a la llamada original.

RPC es una tecnología fundamental en la construcción de sistemas distribuidos, microservicios (a través de implementaciones como gRPC) y en la administración remota de sistemas, como en el caso de NFS (Network File System).

## Desarrollo

Para la práctica se desarrolló un sistema de gestión de archivos compuesto por un servidor y un cliente implementados en Python. Se utilizó la biblioteca xmlrpc para la comunicación RPC y tkinter para la interfaz gráfica del cliente.

### **3.1 Implementación del Servidor RPC (**server.py**)**

El servidor utiliza la clase SimpleXMLRPCServer para crear un punto de acceso que escucha peticiones en el puerto 8000. Su función es exponer un conjunto de procedimientos para que puedan ser invocados por los clientes remotos. Las funciones expuestas (registradas con server.register\_function) son:

* **list\_files(directory):** Lista los archivos de un directorio en el servidor.
* **read\_file(file\_path):** Lee el contenido de un archivo específico y lo retorna como texto.
* **write\_file(file\_path, content):** Escribe contenido de texto en un archivo en el servidor.
* **upload\_file(file\_name, file\_data):** Recibe datos de un archivo codificados en base64, los decodifica y los guarda en el servidor con el nombre proporcionado.

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()

### **3.2 Implementación del Cliente RPC (**client.py**)**

El cliente proporciona una interfaz gráfica de usuario (GUI) creada con tkinter para interactuar con el servidor. Se conecta al servidor a través de un ServerProxy. Las funciones principales son:

* **listar\_archivos():** Invoca el método remoto list\_files del servidor y actualiza el listado de archivos en la GUI.
* **leer\_archivo():** Obtiene el archivo seleccionado de la lista, llama a la función remota read\_file y muestra el contenido devuelto en un área de texto.
* **enviar\_archivo():** Abre un diálogo para que el usuario seleccione un archivo local. Este archivo se lee en modo binario, se codifica en base64 y se envía al servidor llamando a la función remota upload\_file. Tras una subida exitosa, se actualiza la lista de archivos.

import tkinter as tk

from tkinter import messagebox, filedialog, scrolledtext

import xmlrpc.client

import base64

# Create XML-RPC proxy

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

def create\_gui():

root = tk.Tk()

root.title("Cliente de Gestión de Archivos")

root.geometry("800x600")

# Frame for file list and buttons

frame\_left = tk.Frame(root)

frame\_left.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, padx=10, pady=10)

# Frame for file content display

frame\_right = tk.Frame(root)

frame\_right.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True, padx=10, pady=10)

# File list components

tk.Label(frame\_left, text="Archivos en el servidor:").pack(anchor=tk.W)

global listbox

listbox = tk.Listbox(frame\_left, width=40, height=20)

listbox.pack(fill=tk.BOTH, expand=True, pady=5)

scrollbar = tk.Scrollbar(listbox)

scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

listbox.config(yscrollcommand=scrollbar.set)

scrollbar.config(command=listbox.yview)

# Buttons for file operations

button\_frame = tk.Frame(frame\_left)

button\_frame.pack(fill=tk.X, pady=5)

tk.Button(button\_frame, text="Actualizar lista", command=listar\_archivos).pack(side=tk.LEFT, padx=2)

tk.Button(button\_frame, text="Leer archivo", command=leer\_archivo).pack(side=tk.LEFT, padx=2)

tk.Button(button\_frame, text="Subir archivo", command=enviar\_archivo).pack(side=tk.LEFT, padx=2)

# tk.Button(button\_frame, text="Descargar archivo", command=descargar\_archivo).pack(side=tk.LEFT, padx=2)

# tk.Button(button\_frame, text="Eliminar archivo", command=eliminar\_archivo).pack(side=tk.LEFT, padx=2)

# File content display

tk.Label(frame\_right, text="Contenido del archivo:").pack(anchor=tk.W)

global text\_area

text\_area = scrolledtext.ScrolledText(frame\_right, width=50, height=25)

text\_area.pack(fill=tk.BOTH, expand=True)

# Initial file list load

listar\_archivos()

root.mainloop()

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}")

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.")

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}")

if \_\_name\_\_ == "\_\_main\_\_":

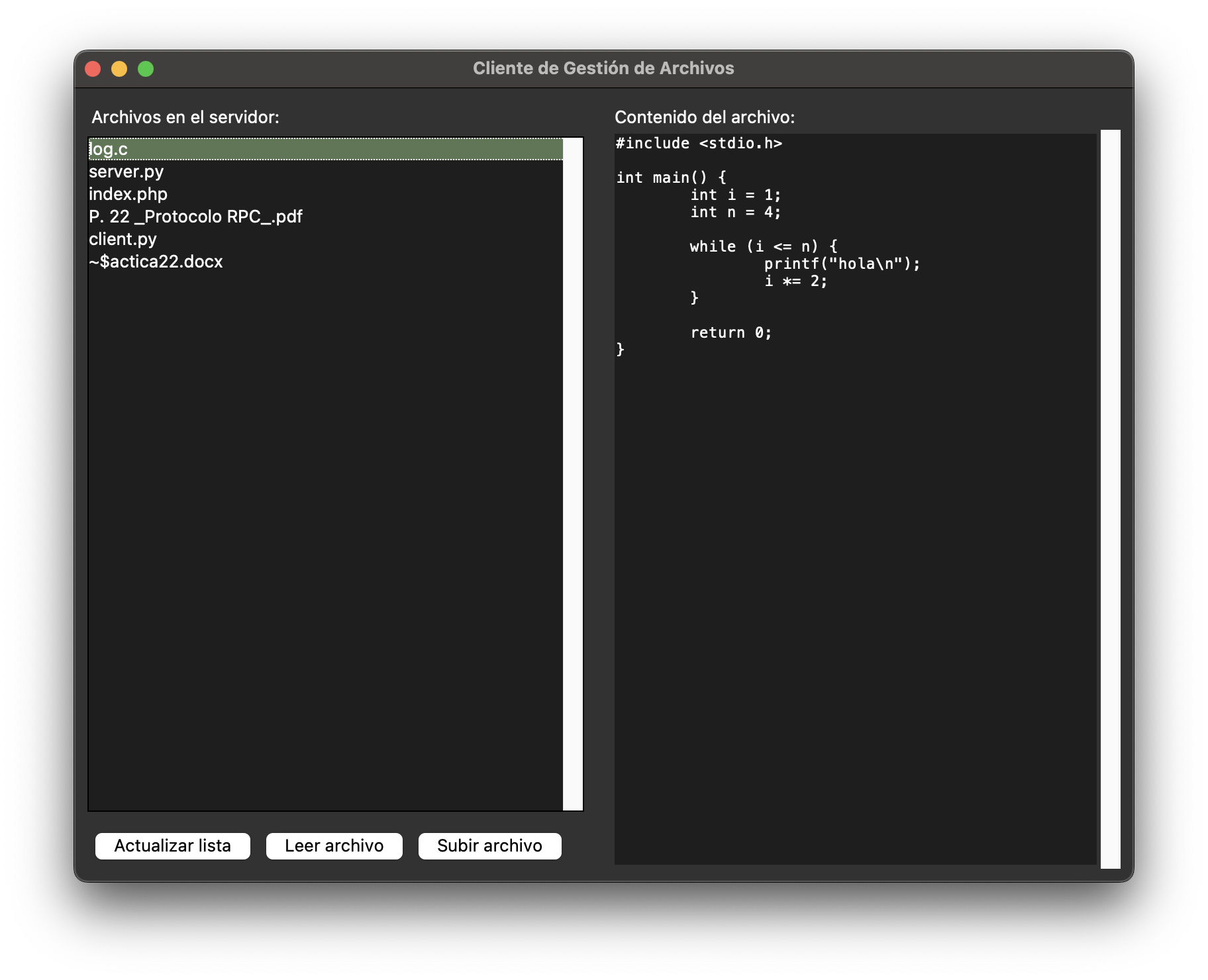
create\_gui()

A screenshot of a computer

AI-generated content may be incorrect.  
Figura 1. Lectura de archivos.

A screenshot of a computer

AI-generated content may be incorrect.  
Figura 2. Carga de archivos.

  
Figura 3. Lectura del archivo cargado.

A screenshot of a computer

AI-generated content may be incorrect.  
Figura 4. Servidor en ejecución.

## Conclusión

La práctica se completó exitosamente, logrando crear un sistema funcional de gestión de archivos remotos. Se demostró de manera práctica cómo el protocolo RPC, a través de la biblioteca xmlrpc.client y xmlrpc.server de Python, simplifica enormemente el desarrollo de aplicaciones distribuidas. La capacidad de llamar a funciones en un servidor remoto (proxy.list\_files(), proxy.read\_file(), etc.) con la misma sintaxis que una llamada a función local, es la principal ventaja de este paradigma.

Se concluye que RPC es una herramienta potente y eficaz que abstrae la complejidad de la comunicación en red. Permite a los desarrolladores centrarse en la lógica de la aplicación en lugar de en los detalles de bajo nivel de sockets y serialización de datos, facilitando la creación de sistemas robustos y modulares.

## Pregunta

### ¿Cómo podrían generar un negocio a través de lo visto en la práctica?

Este modelo puede ser la base para un servicio de "Backend como Servicio" (BaaS). En lugar de simples operaciones con archivos, el servidor RPC podría exponer funciones de cómputo complejas (ej. procesamiento de imágenes, cálculos científicos, análisis de datos). Se podría ofrecer este poder de cómputo a través de una API de pago, permitiendo que desarrolladores o empresas con hardware limitado accedan a servicios avanzados sin invertir en infraestructura propia, pagando por suscripción o por uso.

## Bibliografía

[1] A. S. Tanenbaum and D. J. Wetherall, Computer Networks, 5th ed. Boston: Pearson, 2011.

[2] The Python Software Foundation, "xmlrpc.client — XML-RPC client access," Python 3.12.4 documentation. [En línea]. Disponible: <https://docs.python.org/3/library/xmlrpc.client.html>. [Consultado: 25-jun-2025].

[3] J. F. Kurose and K. W. Ross, Computer Networking: A Top-Down Approach, 8th ed. Boston: Pearson, 2021.