Sistemas Distribuidos

TRABAJO DE LABORATORIO - 1

2021

Docentes

- Profesor Adjunto: **Lic. Cristian Javier Parise**
- Auxiliar: **Lic. Pedro Konstantinoff**

Alumno

- Conti Martin

Trabajo de Laboratorio N° 1

Cliente / Servidor. Sockets. gRPC. Threads. Concurrencia. RFS

1. Dado el código provisto en el repositorio, analice los fuentes client11.py y server11.py y modifíquelos para que la consulta del cliente y la respuesta del servidor sea más interactiva, ya sea mediante interfaz gráfica o línea de comandos. Además agregue un comando mediante el cual el cliente indica al servidor la finalización de su ejecución.

https://github.com/pkonstantinoff/unpsjb_distribuidos/tree/master/tl1/p1

2. Servidores con y sin estado.

a) Definir brevemente qué es un servidor con estados y qué es un servidor sin estados.

b) Analice los fuentes client2.py y server2.py del repositorio, e indique si se trata de un servidor con o sin estados.

https://github.com/pkonstantinoff/unpsjb_distribuidos/tree/master/tl1/p2

c) Dado el código analizado, documente los pasos, e implemente un servidor opuesto al ya implementado (Si es sin estado realice uno con estado o viceversa) con la misma funcionalidad general.

3. Dada la siguiente especificación Protocol Buffer (proto3):

Generar el archivo file_system.proto

```
syntax = "proto3";

message Path {
        string value = 1;
}
message PathFiles {
   repeated string values = 2;
}
service FS {
        rpc ListFiles(Path) returns (PathFiles){};
}
```

a) Agregue las operaciones OpenFile, ReadFile, CloseFile al .proto de la especificación e implemente la solución completa. Emplear como ayuda las filminas correspondientes a gRPC.

b) Implemente el equivalente al ejercicio anterior, generando los stubs en forma manual, sin gRPC en lenguaje Python. Para ello se deberán crear también las estructuras a enviar en cada tipo de mensaje además de las clases stub del cliente y del servidor.

c) Comparar y comentar la complejidad de la implementación con gRPC y la realizada con los stubs a mano.

Sistemas Distribuidos

Emplee las fuentes del repositorio para guiarse en el desarrollo de este punto.

https://github.com/pkonstantinoff/unpsjb_distribuidos/tree/master/tl1/p3

4. Teniendo en cuenta el servidor del ejercicio anterior, 3.b:
   a) Pruebe cancelar un cliente cuando se está en el medio del funcionamiento y vuelva a arrancar el cliente. Observe y documente qué ocurre.
   b) Pruebe cancelar el servidor cuando se está en el medio del funcionamiento y vuelva a arrancar el cliente. Observe y documente qué ocurre.
   c) Determine si es un servidor con estados o sin estados.

5. Modifique la solución del punto 3.b empleando hilos para que los requerimientos de los clientes puedan responderse de manera concurrente.

6. Concurrencia.
   a) Disparar dos clientes a la vez y verificar (documentar) si las solicitudes de ambos clientes son atendidas por el mismo o por distintos hilos en el servidor
   b) Determine si la implementación del ejercicio 3.- tiene un thread por requerimiento, por conexión o por recurso (en términos del cap. 6 de Coulouris).
   c) Transferir un archivo de gran tamaño (GBs) completo en la versión con RPC y en la versión con Sockets (utilizando las primitivas de lectura o escritura con un tamaño de buffer igual en ambos casos) y cronometrar (que el mismo cliente muestre cuánto demoró) en ambos casos, sacando las conclusiones del caso respecto a la performance de cada solución.

7. Proponga una modificación del ejercicio 3.b de manera tal que se tenga un conjunto (pool) de threads creados con anterioridad a la llegada de los requerimientos y donde se administren los threads de acuerdo a las llegadas de requerimientos ¿Sería útil cambiar la cantidad de threads administrados de esta manera? Justifique.

1.

README.md

Para la ejecución del cliente y servidor es necesario python versión 3 o superior.

Ejecutando en la línea de comandos: python server11.py se inicializa el servidor, quedando a la espera de clientes.

Por otro lado, mediante: python client11.py se crea un cliente que se conecta al servidor y envía un saludo.

Para ejecutar el servidor en un contenedor docker, mediante el comando: docker image build . --tag <<nombre_de_la_imagen>> se genera una imagen con el fuente del servidor11.py.

Seguidamente, para corroborar la existencia de la imagen: docker image ls mostrará en pantalla las imagenes disponibles.

Para ejecutar el servidor: docker run -p 8090:8080 <<nombre_de_la_imagen>>

- en donde 8090 es el puerto del host,
- 8080 es el puerto definido para el servidor,
- y finalmente el nombre de la imagen creada previamente.

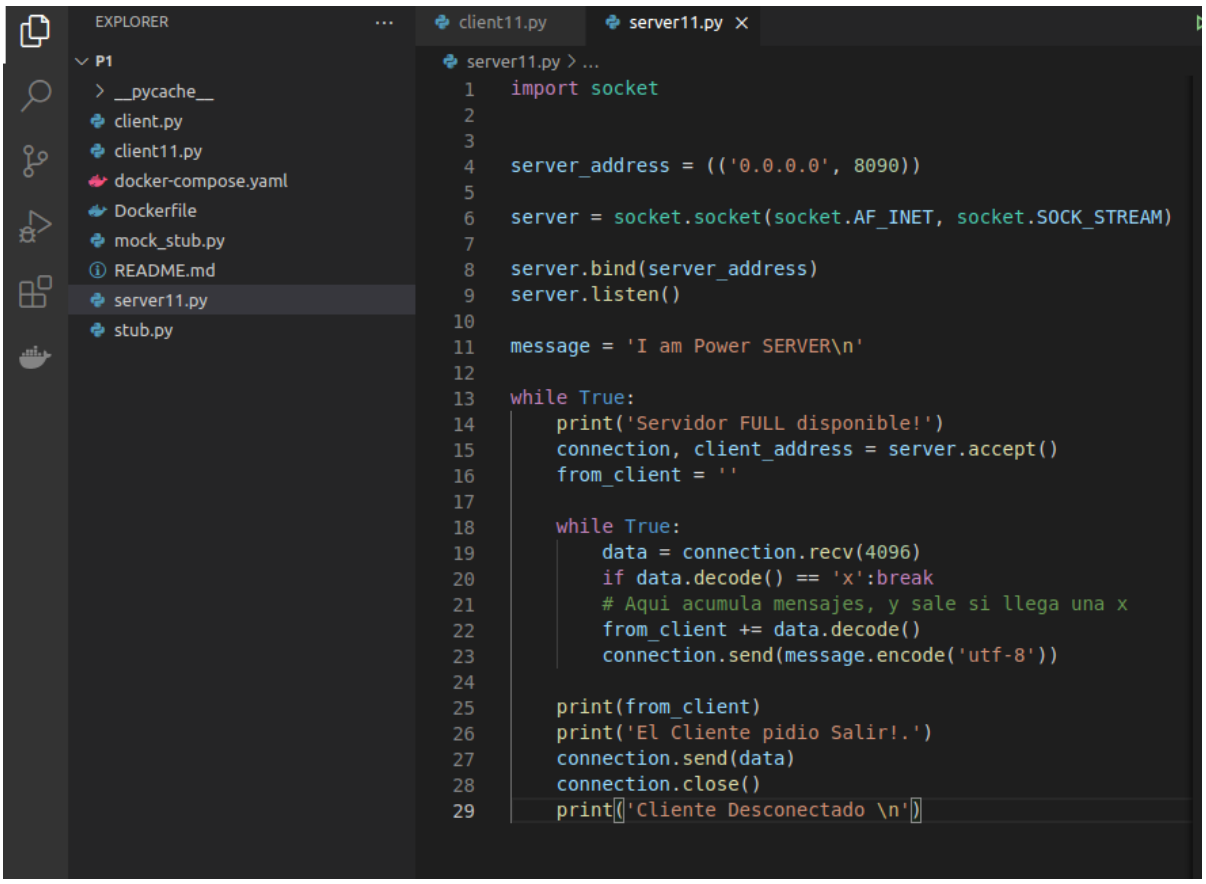
Konstantinoff Pedro. 18/08/2020.

RTA:

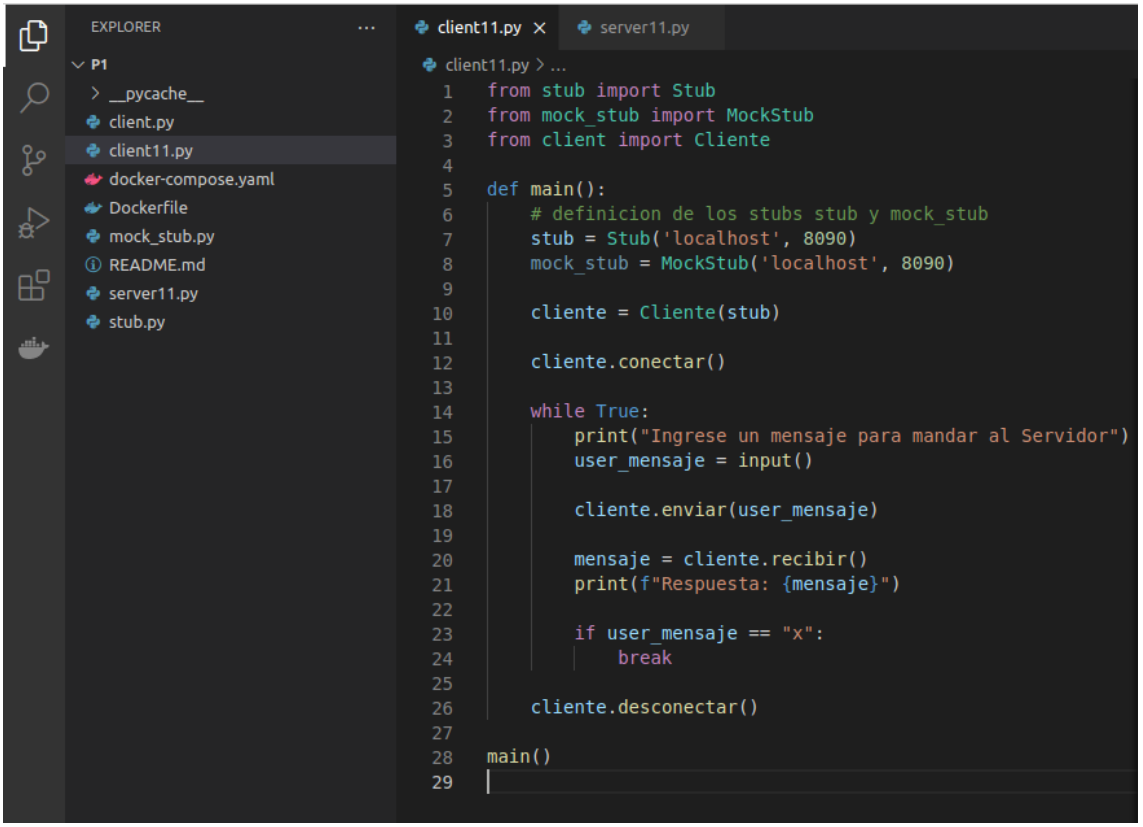
En este caso lo que se optó es por hacer un bucle de repetición “while” tanto en el server11.py como en el client11. A continuación detallo cada uno.

Primero vemos el Servidor, quien es el que se debe ejecutar primero, y quedarse a la espera de conexión de un cliente.

server11.py


EXPLORER
P1
__pycache__
client.py
client11.py
docker-compose.yaml
Dockerfile
mock_stub.py
README.md
server11.py
stub.py
client11.py
server11.py
server11.py > ...
import socket
server_address = (('0.0.0.0', 8090))
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(server_address)
server.listen()
message = 'I am Power SERVER\n'
while True:
print('Servidor FULL disponible!')
connection, client_address = server.accept()
from_client = ''
while True:
data = connection.recv(4096)
if data.decode() == 'x':break
Aqui acumula mensajes, y sale si llega una x
from_client += data.decode()
connection.send(message.encode('utf-8'))
print(from_client)
print('El Cliente pidio Salir!.')
connection.send(data)
connection.close()
print('Cliente Desconectado \n')

client11.py


EXPLORER
P1
__pycache__
client.py
client11.py
docker-compose.yaml
Dockerfile
mock_stub.py
README.md
server11.py
stub.py
client11.py
server11.py
client11.py > ...
from stub import Stub
from mock_stub import MockStub
from client import Cliente
def main():
# definicion de los stubs stub y mock_stub
stub = Stub('localhost', 8090)
mock_stub = MockStub('localhost', 8090)
cliente = Cliente(stub)
cliente.conectar()
while True:
print("Ingrese un mensaje para mandar al Servidor")
user_mensaje = input()
cliente.enviar(user_mensaje)
mensaje = cliente.recibir()
print(f"Respuesta: {mensaje}")
if user_mensaje == "x":
break
cliente.desconectar()
main()

Corremos el servidor con python3 server11.py

El mismo se queda esperando a que se conecte un cliente y envíe un mensaje.

```
(distribuidos) pepito@pepito-UX:~/unpsjb_distribuidos-PUNTO 1/tl1/p1$ python3 server11.py
Servidor FULL disponible!
```

Corremos el cliente con python3 client11.py

Y debemos enviar algo como texto, el servidor va a juntar luego todos los mensajes y replicarlo. En el caso de que se envíe una x, se entiende como un mensaje para salir, y la conexión se cierra.

```
(distribuidos) pepito@pepito-UX:~/unpsjb_distribuidos-PUNTO 1/tl1/p1$ python3 client11.py
Ingrese un mensaje para mandar al Servidor
```

Mandamos un mensaje de Hola Servidor!

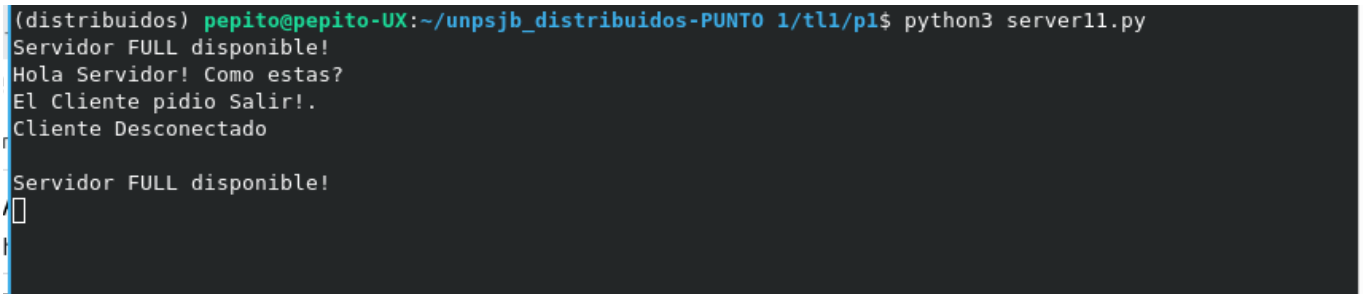
A lo que el servidor responde con un I am Power SERVER

Espera por un nuevo mensaje porque no fue el mensaje de “x” que indicaba salir.

Lo que si se hace en la última línea, indicando la salida.

```
(distribuidos) pepito@pepito-UX:~/unpsjb_distribuidos-PUNTO 1/tl1/p1$ python3 client11.py
Ingrese un mensaje para mandar al Servidor
Hola Servidor!
Respuesta: I am Power SERVER

Ingrese un mensaje para mandar al Servidor
Como estas?
Respuesta: I am Power SERVER

Ingrese un mensaje para mandar al Servidor
x
Respuesta: x
(distribuidos) pepito@pepito-UX:~/unpsjb_distribuidos-PUNTO 1/tl1/p1$
```

El servidor réplica los mensajes acumulados. Y cuando recibe una “x” finaliza la sesión con el cliente.



Hay que aclarar aquí que el servidor va a seguir siempre conectado esperando una nueva conexión.

2. Servidores con y sin estado.

a) Definir brevemente qué es un servidor con estados y qué es un servidor sin estados.

RTA:

Sistemas sin estado

- Un proceso o una aplicación sin estado se refiere a los casos en que estos están aislados.
- No se almacena información sobre las operaciones anteriores ni se hace referencia a ellas.
- Cada operación se lleva a cabo desde cero, como si fuera la primera vez.
- Las aplicaciones sin estado proporcionan un servicio o una función y usan servidores de impresión, de red de distribución de contenido (CDN) o web para procesar estas solicitudes a corto plazo.
- Ejemplo: Búsqueda en línea de la respuesta a una pregunta que se le haya ocurrido. Escribimos la pregunta en un motor de búsqueda y presiona la tecla Entrar. Si la operación se interrumpe o se cierra por accidente, simplemente inicia una nueva.
- Hay que pensar en las operaciones sin estado como en máquinas expendedoras: a una solicitud, una respuesta.

Sistemas con estado

- Las aplicaciones y los procesos con estado son aquellos a los que se puede volver una y otra vez, como la banca en línea o el correo electrónico.

- Se realizan con el contexto de las operaciones anteriores, y la operación actual puede verse afectada por lo que ocurrió previamente.
- Por estos motivos, las aplicaciones con estado utilizan los mismos servidores cada vez que procesan la solicitud de un usuario.
- En caso de interrumpirse una operación de este tipo, se puede retomar más o menos desde donde la dejó gracias al almacenamiento del contexto y el historial. Las aplicaciones con estado rastrean, por ejemplo, la ubicación de las ventanas, las preferencias de configuración y la actividad reciente.
- Hay que pensar en ellas como en una conversación periódica y constante con la misma persona.

La mayoría de las aplicaciones que usamos diariamente son de este tipo; pero conforme la tecnología avanza, los microservicios y los contenedores facilitan el diseño y la implementación de aplicaciones en la nube.

b) Analice los fuentes client2.py y server2.py del repositorio, e indique si se trata de un servidor con o sin estados.

RTA:

Se trata de un servidor CON estados.

c) Dado el código analizado, documente los pasos, e implemente un servidor opuesto al ya implementado (Si es sin estado realice uno con estado o viceversa) con la misma funcionalidad general.

RTA:

La idea aquí es eliminar todo lo que sea que guarde un estado, digamos que debería borrar todo lo relacionado al Acumulador.

Muestro primero el ServidorSinEstados

```
import socket



# Utilizaremos el módulo Pickle.

# Este nos permite almacenar casi cualquier objeto Python directamente en un archivo o cadena

# sin necesidad de realizar ninguna conversión.

# Lo que el módulo pickle realiza en realidad es lo que se llama serialización de objetos, es decir,

# convertir objetos a y de cadenas de bytes. El objeto que va a ser pickled se serializará en un flujo de
bytes,

# los cuales se pueden escribir en un archivo, por ejemplo, y restaurar en un punto posterior
```

```
import pickle


# De esta inicializacion no cambia nada
server_address = ("0.0.0.0", 8081)
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(server_address)
server.listen()


while True:
   print("Server disponible!")
   connection, client_address = server.accept()


   # Esto es nuevo, aca
   client = ":".join(list(map(str, client_address)))
   print(f"Cliente {client}")


   while True:


       # A data le pongo lo que viene de la conexion
       data = connection.recv(4096)


       # Aca se produce la magia de Pickle para que interprete que se quiere hacer
       payload = pickle.loads(data)


       comando = payload.get("command")




       # Si data no es nada, se va y sale.
```

```
        if not data: break



        if comando == 1:

            valor = payload.get("valor")

            response = {"valor": valor}

            response_serialized = pickle.dumps(response)

            connection.sendall(response_serialized)



        else:

            print(f"El cliente solicito el comando {comando} con el valor {payload.get('valor')}")

            break



    connection.close()

    print("cliente desconectado \n")
```

Luego muestro elClienteSinEstados

```
# Este cliente es Igual practicamente al cliente 2 pero no guarda los estados.



import socket

# Utilizaremos el módulo Pickle.

# Este nos permite almacenar casi cualquier objeto Python directamente en un archivo o cadena

# sin necesidad de realizar ninguna conversión.

# Lo que el módulo pickle realiza en realidad es lo que se llama serialización de objetos, es decir,

# convertir objetos a y de cadenas de bytes. El objeto que va a ser pickled se serializará en un flujo de
bytes,

# los cuales se pueden escribir en un archivo, por ejemplo, y restaurar en un punto posterior

import pickle
```

```
server_address = ("0.0.0.0", 8081)

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

client.connect(server_address)



keep_working = True



# Basicamente se elimino la opcion de Obtener

COMMAND = {"A": 1}



while keep_working:

   print("Ingrese un comando ([A]cumular, [S]alir)")

   comando = input()



   if comando == "A":

       print("Ingrese un valor a acumular")



       #  ----------------------------------



       try:

           valor = int(input())

           payload = {"command": COMMAND[comando], "valor": valor}



           # Pickle lo que hace es serializar el "valor"

           payload_serialized = pickle.dumps(payload)

           client.sendall(payload_serialized)

           data = client.recv(4096)



           # Aca si ya con pickle para que serializa el valor

           result = pickle.loads(data)
```

```
            print(f"El valor acumulado es: {result.get('valor')}")

        except ValueError:

            print("Debe ingresar un numero")



    # El salir queda igual

    elif comando == "S":

        keep_working = False

        client.close()

        print("Sesion finalizada")



    # Este queda igual por si ingresa algo no valido..

    else:

        print("El comando ingresado no es valido")
```

```
(distribuidos) pepito@pepito-UX:~/unpsjb_distribuidos_PUNTO1/tl1/p2$ python3 serverSinEstado.py
Server disponible!
Cliente 127.0.0.1:36652
cliente desconectado

Server disponible!
Cliente 127.0.0.1:36654
```

Como se ve, el servidor no guarda nada del estado del cliente, para él es como si se tratara de otro cliente. Nos podemos dar cuenta porque el número de procesos cambia.

```
(distribuidos) pepito@pepito-UX:~/unpsjb_distribuidos_PUNTO1/tl1/p2$ ^C
(distribuidos) pepito@pepito-UX:~/unpsjb_distribuidos_PUNTO1/tl1/p2$ python3 clientSinEstado.py
Ingrese un comando ([A]cumular, [S]alir)
A
Ingrese un valor a acumular
23
El valor acumulado es: 23
Ingrese un comando ([A]cumular, [S]alir)
A
Ingrese un valor a acumular
34
El valor acumulado es: 34
Ingrese un comando ([A]cumular, [S]alir)
S
Sesion finalizada
(distribuidos) pepito@pepito-UX:~/unpsjb_distribuidos_PUNTO1/tl1/p2$ python3 clientSinEstado.py
Ingrese un comando ([A]cumular, [S]alir)
```

3. Dada la siguiente especificación Protocol Buffer (proto3):

Generar el archivo file_system.proto

```
syntax = "proto3";

message Path {
        string value = 1;
}
message PathFiles {
   repeated string values = 2;
}
service FS {
        rpc ListFiles(Path) returns (PathFiles){};
}
```

a) Agregue las operaciones OpenFile, ReadFile, CloseFile al .proto de la especificación e implemente la solución completa. Emplear como ayuda las filminas correspondientes a gRPC.

RTA:

Simplemente para que me quede en el práctico, pongo algo de teoría que encontre en internet sobre gRPD

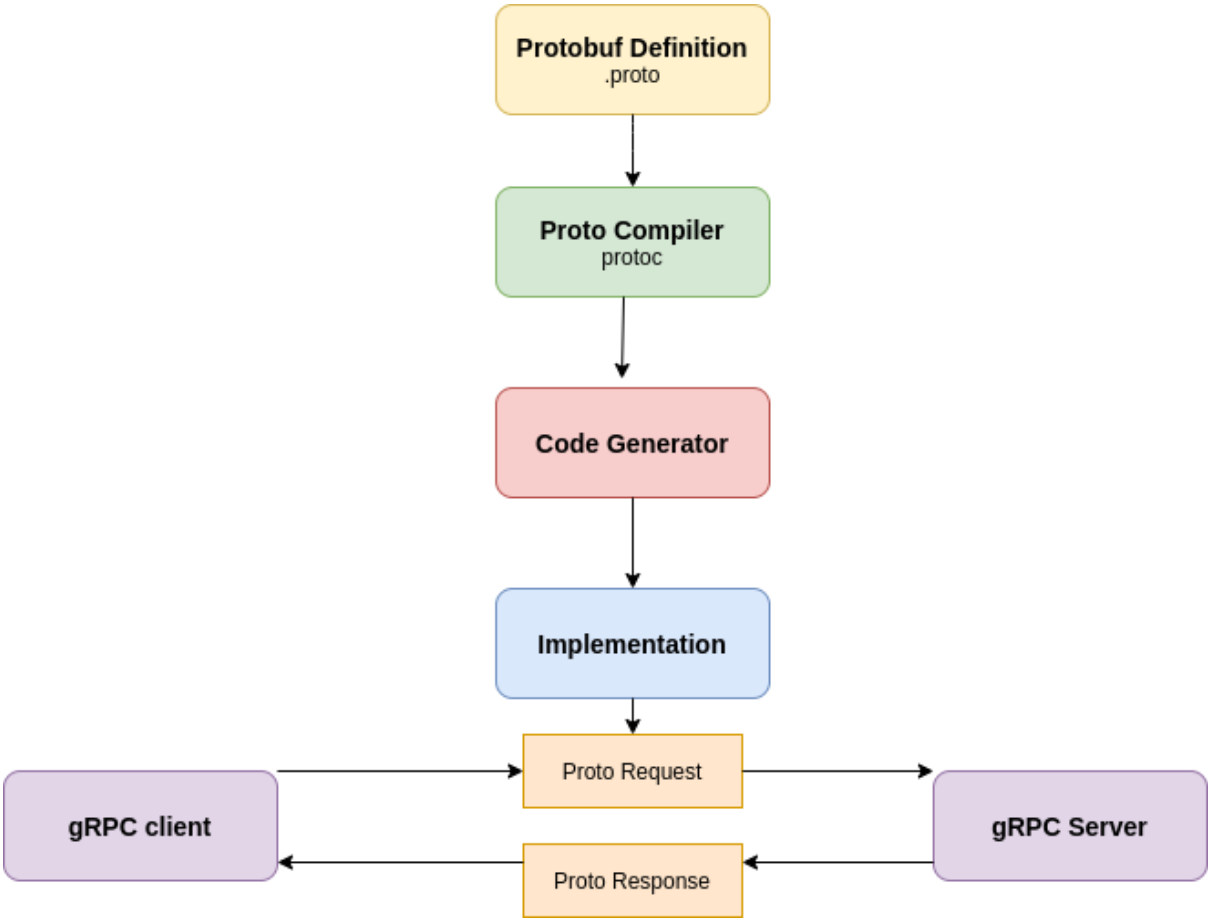
¿**Qué es gRPC**?, gRpc es un protocolo RPC (Remote Procedure Call), el cual puede ser ejecutado sobre cualquier entorno para realizar conexiones o llamadas entre microservicios. Este protocolo fue creado por Google y puede ser utilizado como una alternativa a REST, WebHook o GraphQL.

Entre sus principales características podemos encontrar:

- Transmisión bidireccional y autenticación conectable totalmente integrada con transporte basado en HTTP / 2.
- Destaca el rendimiento, debido a su bajo consumo de CPU, ancho de banda,etc.
- Ofrece JSON encondings y serialización con PROTO 3.
- Genera automáticamente stubs idiomáticos de cliente y servidor para su servicio en una variedad de idiomas y plataformas
- Autenticación incorporada soportando SSL.

Componentes principales

En el siguiente diagrama se puede ver el flujo a realizar para poder hacer uso de gRPC, que será detallado a continuación.



Protocol Buffers o protobuf

Protocol Buffer o Protobuf, ya que pueden ser llamados de la misma manera, es un lenguaje implementado, independiente de Google.

El objetivo que hay que intentar conseguir al crear un Protobuf es la de que sea Contract

First, es decir, hagamos un esfuerzo para definir primero el contrato.

Para poder realizar a continuación su desarrollo nos vamos a basar en su IDL (Interface Definition Languaje), que gracias a RPC, podremos configurar nuestras comunicaciones entre el cliente y el microservicio. Con protobuf crearemos tanto la interfaz del servicio, como sus métodos y sus mensajes. Para poder llevar a cabo su implementación, se realizará en un tipo de fichero con extensión .proto, de manera que pueda ser legible por un humano.

El hecho de que se piense en API First, es básicamente, porque si después de su implementación, tenemos que hacer cambios en nuestro fichero .proto, podemos tener consecuencias, tanto en el cliente, como en el servicio que es consumido, y los cambios nos podrían llevar demasiado esfuerzo.

Proto Compiler

El objetivo de gRPC es que fuera multilenguaje y multiplataforma, por lo que teniendo en cuenta estas premisas, tan solo nos tendremos que preocupar de generar nuestro Protobuf y compilarlo en el lenguaje que necesitemos.

Por lo que para poder generar nuestro fichero Protobuf en el lenguaje que necesitamos, se hace uso de Proto Compiler (protoc).

Implementación

Teniendo claro, o al menos entendiendo, los puntos anteriores sobre protobuf y proto compiler, podemos dar paso a la implementación y desarrollo del código.

Para poder realizar el desarrollo y que tanto la parte cliente como servidora, se puedan comunicar será necesario hacerlo a través de Proto Request y Proto Response. Es decir, haremos uso de unos tipos concreto tanto para realizar las peticiones como para las respuestas que devolverá el servicio.

Se uso la libreria time de Python, para la toma de los tiempos

- https://python-para-impacientes.blogspot.com/2017/03/el-modulo-time.html

Ejemplo de Menú en Python, sacado de aca

- https://www.discoduroderoer.es/crear-un-menu-de-opciones-en-consola-en-python/

Ejemplo de manipulación de archivos en Python

- https://www.youtube.com/watch?v=5svp6GIdzzA

Bueno, lo primero que pide el ejercicio es tomar el proto3 e ingresar los servicios que vamos a ofrecer. El servicio que ofrece es:

ListFiles que pide como parámetro un objeto “Path” y retorna un “PathFiles” que es una variable string.

Debemos generé las clases gRPC para python con los comandos

$ pip install grpcio

$ pip install grpcio-tools

y genere las clases

$ python -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. file_system.proto

---

Con el cliente, desde la consola recibe el un nombre de archivo y un nombre para crear el archivo de salida y va a listar.

El Stub se instancia con:

- Ip: localhost
- Puerto: 50051

CAMINO

Justo en ese Stub se inyecta a una instanciación de un cliente, para decir que el cliente va a utilizar ese stub específicamente.

Luego se conecta el cliente. Y es aquí donde se define una capa de abstracción donde el cliente llama a su stub. (aqui separamos por ejemplo el punto 3a y 3b, ya que cada uno llama a su Stub.)

En el Stub del cliente se definen las conexiones a utilizar, en este caso va a ser gRPC, pero en los otros puntos cambia..

Por otro lado, desde la mirada del servidor se instala la clase Stub donde se instancia una clase llamada FS y un puerto.

El puerto es el 50051. Luego se instancia un Servidor con el stub antes creado (lo mismo que nombre antes para el Cliente.).

Se inicializa el servidor.

La clase Stub, dentro de la carpeta servidor define las conexiones de grpc y las operaciones para conectar con el cliente.

También hay que destacar que gRPC genera clases automáticamente. Las cuales vemos en los archivos file_system_pb2.

Caminito desde el lado del servidor!

```
from file_system import FS

from server import Server

from p3a import ServerStub
```

- Arrancamos por el Servidor.py
- El Servidor.py llama o importa del:
    - file_system el FS
    - server el Server
    - p3a el ServerStub

El Stub del servidor, que está en la carpeta de 3a/server es quien tiene la lógica.

Caminito desde el lado del cliente!

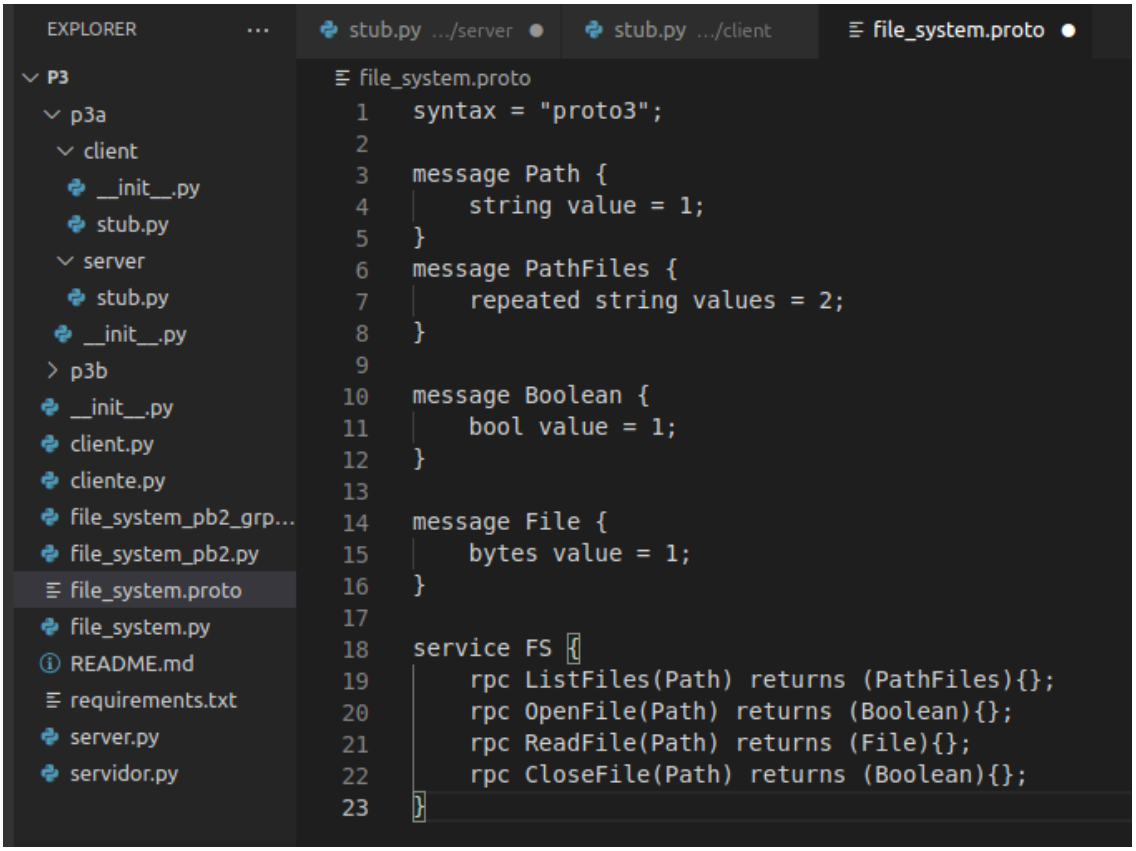
```
from client import Client

from p3a import ClientStub
```

Luego lanzamos el cliente, y el caminito del cliente es:

- Arrancar el cliente con Cliente.py
- El Cliente llama o importa a:
    - client el Client
    - p3a el ClientStub

file_system.proto

Aquí si agregue los **servicios y tipos de datos** que vamos a implementar



Esos números que se ven ahí son los números que usa gRPC para serializar y deserializar y saber donde se encuentra cada dato.

Definiciones del Client.py (Nuevas)

```
def abrir_archivo(self, path):
    return self.adapter.open_file(path)

def leer_archivo(self, path, offset, cant_bytes):
    return self.adapter.read_file(path, offset, cant_bytes)

def cerrar_archivo(self, path):
    return self.adapter.close_file(path)
```

Definiciones del Cliente.py

En este cliente, como no tenía nada más que un listado de archivos en ruta actual, se optó por hacer un Menu para que sea más interactivo y ordenado; no solo para este punto sino para los siguientes.

```python
from client import Client
from p3a import ClientStub



# Para el p 3a
from p3a import ClientStub

# Para el p 3b
#from p3b import ClientStub

from datetime import datetime

# Obtiene la extencion del archivo, lo que este despues de un punto.
def Obtener_extension_de_archivo(path):
   extension = path.split(".").pop()
   return extension



def leer_archivo(cliente, path):

   can_open_file = cliente.abrir_archivo(path)

   if can_open_file:

       extension = Obtener_extension_de_archivo(path)

       today = datetime.now()

       # https://rico-schmidt.name/pymotw-3/datetime/index.html
       file_name = f"{today.strftime('%d-%m-%Y_%H:%M:%S')}.{extension}"

       # Open es una funcion de python para abrir archivos
       # necesita la ruta y los permisos
       # y lo que hacemos es que lo almacenamos en file
       file = open(file_name, "wb")

       print("Copiando el archivo...")
       file.close()
       cliente.cerrar_archivo(path)
```

```
        return True
    else:
        return False



def listar_archivos(path, cliente):
    archivos = cliente.listar_archivos(path)

    # Iterar por toda la lista y muestra archivo por archivo.
    for archivo in archivos:
        print(f"{archivo}")
    return True



def menu():
    print(". Donde buscar el archivo?:..")
    path = input()
    return path



def main():
    # Este Main no cambia por ahora
    stub = ClientStub("localhost", "50051")
    cliente = Client(stub)
    cliente.conectar()

    # Variable para poder salir del while
    salir = False

    while not salir:
        print(" - - - - - - - -0- - - - - - - -")
        print(" Menu - Sistemas Distribuidos 2021")
        print(" L - Leer y copiar Archivo")
        print(" V - Ver los archivos de un directorio")
        print(" S - Salir")
        print(" - - - - - - - -0- - - - - - - -")

        # Opcion ingresada por consola
        camino = input()
```

```
try:

    # Segun la opcion entra en un camino o el otro
    camino = str(camino)
    if camino == "L":
        path = menu()
        print(f"Ruta ingresada: {path}")

        operation_result = leer_archivo(cliente, path)

        if operation_result:
            print("Copia exitosa!")
        else:
            print("Archivo no existe!")
    if camino == "V":
        path = menu()
        print(f"Ruta ingresada: {path}")
        operation_result = listar_archivos(path, cliente)

        if not operation_result:
            print(f"Directorio vacio. {path}")
    elif camino == "S":

        # Desconecta y saluda el cliente.
        print("Saliendo")
        cliente.desconectar()
        print("Cliente Desconectado.")

        # Talvez esta de mas
        salir = True
        break
    else:
        print("Reimprimir MENU!")
except ValueError:
    print("Opción Incorrectas")
except KeyboardInterrupt:
    cliente.desconectar()
    salir = True
```

```
            break


if __name__ == '__main__':
    main()
```

Del lado del Server.py

```
class Server:


    def __init__(self, adapter):
        self.adapter = adapter


    def inicializar(self):
        print("  ")
        print(" - - - - - - - -0- - - - - - - -")
        print('Inicializando el servidor')
        print("  ")
        print(" - - - - - - - -0- - - - - - - -")
        self.adapter.run()
```

Servidor.py

```
from file_system import FS
from server import Server
from p3a import ServerStub


def main():
    stub = ServerStub(FS(), '50051')
    servidor = Server(stub)
    servidor.inicializar()


if __name__ == '__main__':
    main()
```

Ahora sí los correspondientes al punto 3a y algunas capturas de pantalla del uso.

Dentro de la carpeta 3a, Cliente, está el Stub del cliente, el cual tiene agregadas estas 3 funciones (las que pedía el ejercicio)

```
    # Agregados
    def open_file(self, path):
        path = Path(value=path)
        response = self._stub.OpenFile(path)
        return response.value


    def read_file(self, path):


        path = Path(value=path)
        response = self._stub.ReadFile(path)
        return response.value


    def close_file(self, path):
        path = Path(value=path)


        response = self._stub.CloseFile(path)


        return response.value
```

Agregados del servidor, en la carpeta 3a.

```
    def OpenFile(self, request, context):
        response = file_system_pb2.Boolean()
        can_open = self._adapter.open_file(request.value)
        response.value = can_open
        return response


    def ReadFile(self, request, context):
```

```
        response = file_system_pb2.File()

        return response


    def CloseFile(self, request, context):

        response = file_system_pb2.Boolean()

        can_open = self._adapter.close_file(request.value)

        response.value = can_open

        return response
```

Inicializamos el servidor

```
(distribuidos) pepito@pepito-UX:~/Distribuidos-TP1/tl1/p3$ python3 servidor.py

 - - - - - - - -0- - - - - - - -
Inicializando el servidor

 - - - - - - - -0- - - - - - - -
```

Inicializamos el cliente, y tenemos 3 caminos.

```
(distribuidos) pepito@pepito-UX:~/Distribuidos-TP1/tl1/p3$ python3 cliente.py
 - - - - - - - -0- - - - - - - -
 Menu - Sistemas Distribuidos 2021
 L - Leer y copiar Archivo
 V - Ver los archivos de un directorio
 S - Salir
 - - - - - - - -0- - - - - - - -
L
. Donde buscar el archivo?:..
/home/pepito/requirements.txt
Ruta ingresada: /home/pepito/requirements.txt
Copiando el archivo...
Copia exitosa!
Reimprimir MENU!
 - - - - - - - -0- - - - - - - -
 Menu - Sistemas Distribuidos 2021
 L - Leer y copiar Archivo
 V - Ver los archivos de un directorio
 S - Salir
 - - - - - - - -0- - - - - - - -
V
. Donde buscar el archivo?:..
/home
Ruta ingresada: /home
pepito
 - - - - - - - -0- - - - - - - -
 Menu - Sistemas Distribuidos 2021
```

Vemos que la primer opción “L” Lee y crea una copia local del archivo, con los datos de la fecha y hora.

La segunda opción es “V” que es para ver el directorio, y listar los archivos del mismo.

En el ejemplo solo había una sola carpeta.

Última opción “S” salir.

b) Implemente el equivalente al ejercicio anterior, generando los stubs en forma manual, sin gRPC en lenguaje Python. Para ello se deberán crear también las estructuras a enviar en cada tipo de mensaje además de las clases stub del cliente y del servidor.

RTA:

Aquí para la implementación del ejercicio manual, digamos sin gRPC, se mantuvo el cliente y servidor inicial.

Deje lo indispensable para realizar la lógica del cliente, separado de la parte de conexiones.

También utilice la librería pickle de python, muy útil porque de esta forma se emula las estructuras que creaban con gRPC.

Cliente sin gRPC:

```
import sockets


from p3b.structures import (
   Path,
   PathFiles,
)


# Constante para el tamaño del buffer
cant_buff = 1024


class FSStub:


   def __init__(self, canal):
       self._channel = canal
```

```
    def ListFiles(self, path):
        path = Path(path=path, operacion=1)
        self._channel.sendall(path)

        path_files = PathFiles()
        list_files = []
        while self._channel.recv_into(path_files):
            list_files.append(path_files.values)

        return list_files

    def Read(self,path):
        self._channel.sendall(path)
        request = self._channel.recv(cant_buff)
        return request

class Stub:

    def __init__(self, host='0.0.0.0', port='8090'):
        self._appliance = (host, port)
        self._channel = None
        self._stup = None

    def connect(self):
```

```
        """ Returns a gRPC open channel """

        try:

            self._channel = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)

            self._channel.connect(self._appliance)

            self._stub = FSStub(self._channel)

            return True if self._channel else False

        except Exception as e:

            print('Error when openning channel {}'.format(e))

            return False


    def disconnect(self):

        self._channel.close()

        self._channel = None


    def is_connected(self):

        return self._channel


    def list_files(self, path):

        if self.is_connected():

            return self._stub.ListFiles(path)

        return None




    def read_file(self, path):

        if self.is_connected():
```

```
            return self._stub.Read(path)

        return None
```

Servidor sin gRPC

```
    def _process_request(self):

        data = self._channel.recv(cant_buff)

        if data:

            dataPickle = pickle.loads(data)

            if dataPickle is not None:

                if dataPickle['operacion'] == 1:

                    path_files =
self._adapter.list_files(dataPickle['value'])

                    request = { 'value': path_files}

                    requestPickle= pickle.dumps(request)

                    self._channel.send(requestPickle)



                elif dataPickle['operacion'] == 2:

                    read_file = self._adapter.read_file(dataPickle)

                    request = { 'value': read_file}

                    requestPickle= pickle.dumps(request)

                    self._channel.sendall(requestPickle)

                return 0

        else:

            return 1
```

c) Comparar y comentar la complejidad de la implementación con gRPC y la realizada con los stubs a mano.

RTA:

Tiene mucho sentido comentar aquí que si bien ambas implementaciones tienen mucha complejidad, como así mucho nivel de abstracción, gRPC como que tiene más puntos a favor ya que crea muchas cosas de forma automática.

Sistemas Distribuidos

Emplee las fuentes del repositorio para guiarse en el desarrollo de este punto.

https://github.com/pkonstantinoff/unpsjb_distribuidos/tree/master/tl1/p3

4. Teniendo en cuenta el servidor del ejercicio anterior, 3.b:

a) Pruebe cancelar un cliente cuando se está en el medio del funcionamiento y vuelva a arrancar el cliente. Observe y documente qué ocurre.

RTA:

b) Pruebe cancelar el servidor cuando se está en el medio del funcionamiento y vuelva a arrancar el cliente. Observe y documente qué ocurre.

RTA:

c) Determine si es un servidor con estados o sin estados.

RTA:

Desde ya que el servidor es sin estado, porque no guarda información del cliente entre peticiones en ningún momento.

5. Modifique la solución del punto 3.b empleando hilos para que los requerimientos de los clientes puedan responderse de manera concurrente.

RTA:

6. Concurrencia.

a) Disparar dos clientes a la vez y verificar (documentar) si las solicitudes de ambos clientes son atendidas por el mismo o por distintos hilos en el servidor

RTA:

b) Determine si la implementación del ejercicio 3.- tiene un thread por requerimiento, por conexión o por recurso (en términos del cap. 6 de Coulouris).

RTA:

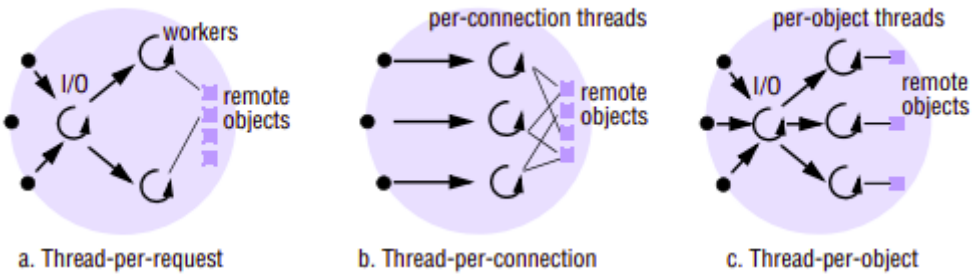
Depende de cómo esté implementado. Primero detallamos cada uno.

Hilo por requerimiento: el subproceso de E / S genera un nuevo subproceso de trabajo para cada solicitud, y ese trabajador se destruye a sí mismo cuando ha procesado la solicitud en su objeto remoto designado. Esta arquitectura tiene la ventaja de que los subprocesos no compiten por una cola compartida y el rendimiento se maximiza potencialmente porque el subproceso de E / S puede crear tantos trabajadores como solicitudes pendientes hay. Su desventaja es la sobrecarga de las operaciones de creación y destrucción de subprocesos.

Hilos por conexión: La arquitectura hilo por conexión asocia un hilo con cada conexión. El servidor crea un nuevo hilo de trabajo cuando un cliente establece una conexión y lo destruye cuando el cliente cierra la conexión. En el medio, el cliente puede realizar muchas solicitudes a través de la conexión, dirigidas a uno o más.

Hilos por objetos: La arquitectura hilo por objeto asocia un hilo con cada objeto remoto. Un subproceso de E / S recibe solicitudes y las pone en cola para los trabajadores, pero esta vez hay una cola por objeto.

Por lo comentado anteriormente, este caso es por conexión, atiende al cliente hasta que muere.



a. Thread-per-request b. Thread-per-connection c. Thread-per-object

c) Transferir un archivo de gran tamaño (GBs) completo en la versión con RPC y en la versión con Sockets (utilizando las primitivas de lectura o escritura con un tamaño de buffer igual en ambos casos) y cronometrar (que el mismo cliente muestre cuánto demoró) en ambos casos, sacando las conclusiones del caso respecto a la performance de cada solución.

RTA:

7. Proponga una modificación del ejercicio 3.b de manera tal que se tenga un conjunto (pool) de threads creados con anterioridad a la llegada de los requerimientos y donde se administren los threads de acuerdo a las llegadas de requerimientos ¿Sería útil cambiar la cantidad de threads administrados de esta manera? Justifique.

RTA: