LABORATORIO 1

SISTEMAS DISTRIBUIDOS 2020

Barja, María Marta

1.

Para este punto se crea un bucle en el cliente dependiendo de una variable lógica ClienteActivo inicializada en true. Dentro de este bucle se solicita que el usuario ingrese un mensaje por consola, y luego este mensaje es transmitido al servidor. Al recibir su respuesta chequea que el mensaje recibido no le indique que debe finalizar, y lo muestra por pantalla.

En el servidor también se crea un bucle dependiente de una variable lógica ServerActivo inicializada en true. Dentro de ese bucle recibe los mensajes del cliente, chequea que estos mensajes no indiquen que debe finalizar su ejecución, y envía una respuesta. Si el servidor recibe un mensaje equivalente a Exit, invierte el valor de serverActivo, envía como respuesta el comando ExitCliente que indicará al cliente que debe finalizar, y finaliza su ejecución. Es decir que cuando el cliente envíe el comando Exit, el servidor finalizará su ejecución y le indicará al cliente a través del mensaje ExitCliente que también finalice la suya.

2.

a. Servidor con estados: son aquellos servidores que guardan información de las sesiones de los clientes de manera que la información que debe transmitir el cliente es mucho menor.

Servidor sin estados: son los servidores que no retienen información de la sesión. El cliente envía los datos de sesión relevantes al receptor de tal manera que cada paquete de información transferido puede entenderse de forma aislada, sin información de contexto de paquetes anteriores en la sesión.

b. Es un servidor con estados. Acumula valores del cliente en la variable acumulador presente en el servidor.

c. En este punto, se creó una variable acumuladora en el cliente, se descarta la misma del lado del servidor y utilizando la misma estructura ya definida, en vez de enviar el id del cliente en el primer campo, se envía el número acumulado guardado en el cliente. De esta manera el servidor solo realiza la suma del acumulado y el valor ingresado y devuelve el nuevo total. El método obtener que indica el valor total acumulado es un método local del cliente.

3.

a. Primero se definió el protocolo del servicio en el archivo sistem_fyle.proto. Allí se detallan los métodos y los mensajes que envían y reciben cliente y servidor.

```
syntax = "proto3";

message Path {
   string value = 1;
}
message PathFiles {
    repeated string values = 2;
}
message ReadFileParameters{
    string nombre_archivo = 1;
    int32 offset = 2;
    int32 cant_bytes = 3;
}
message ResultadoOpen {
   string respuesta = 1;
}
message ResultadoRF {
   bytes contenido = 1;
}
message ResultadoClose {
   string respuesta = 1;
}

service FS {
   rpc ListFiles(Path) returns (PathFiles){};

   rpc OpenFile(Path) returns (ResultadoOpen){};

   rpc ReadFile(ReadFileParameters) returns (ResultadoRF){};

   rpc CloseFile(Path) returns (ResultadoClose){};
}
```

A partir de este archivo gRPC genera las clases correspondientes. Luego se implementaron el cliente y el servidor.

Inicialmente realice una clase servidor y una clase cliente. La clase cliente instanciaba un FSStub, generado por gRPC y se llamaba directamente a los métodos. En la clase Servidor implementé los métodos definidos en el stub generado por gRPC FSServicer. Esta primera implementación funcionaba correctamente. Luego de implementar la solución con sockets utilizando la estructura de base brindada por Pedro, reestructure las clases del punto a, de manera que para ejecutar el punto a y el punto b solo basta con importar los stubs cliente y servidor de los distintos paquetes (p3a y p3b)

El cliente:

Para el cliente implemente la clase Stub (dentro del archivo ClientStub.py) que se encarga de hacer la conexión con el servidor recibiendo el numero de puerto, y define una variable interna Stub que instancia un objeto de la clase FSStub que genera RPC. Dentro de la clase Stub se implementan todos los métodos requeridos, dentro de esos métodos se hace una llamada a los métodos del stub generado por RPC. Es decir, que la clase Stub que generé envuelve el Stub que genera RPC.

```
class Stub:
    def __init__(self, host_port):
        self._appliance = host_port
        self._channel = None
        self._stub = None

    def connect(self):
        try:
            self._channel = grpc.insecure_channel(self._appliance)
            self._stub = file_system_pb2_grpc.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, directorio):
        if self.is_connected():
            path = file_system_pb2.Path(value=directorio)
            response = self._stub.ListFiles(path)
            return response
        return None

    def open_file(self,archivo):
        if self.is_connected():
            path = file_system_pb2.Path(value=archivo)
            response = self._stub.OpenFile(path)
            return response
        return None

    def read_file(self,archivo):
        if self.is_connected():
            archivoLocal = open('.' + '//' + archivo, 'wb')
            offset=0
            cant_bytes=4000
            ts_inicio = int(round(time.time() * 1000))
            while True:

                RFParametros =
file_system_pb2.ReadFileParameters(nombre_archivo=archivo, offset=offset,
cant_bytes=cant_bytes)
                resp = self._stub.ReadFile(RFParametros)
                bytes_recibidos=sys.getsizeof(resp.contenido)
                archivoLocal.write(resp.contenido)
                if bytes_recibidos>(cant_bytes):
                    offset+=cant_bytes
                else:
                    print('transferencia finalizada')
                    archivoLocal.close()
                    break
            ts_final = int(round(time.time() * 1000))
            print('tiempo de transferencia: '+str(ts_final-ts_inicio))
            return True
        return None

    def close_file(self,archivo):
        if self.is_connected():
            path = file_system_pb2.Path(value=archivo)
            response = self._stub.CloseFile(path)
            return response
        return None
```

A su vez se crea una clase Client (dentro del archivo Client.py), que contiene un adaptador (será una instancia de la clase Stub mencionada anteriormente), y define los métodos. Se verá que para el punto 3, se utiliza esta misma clase, con la diferencia que el Stub que recibe para llamar a los métodos del mismo, esta implementado con Sockets.

```
class Client:
    def __init__(self, adapter):
        self.adapter = adapter
    def conectar(self):
        try:
            self.adapter.connect()
        except Exception as e:
            print('Connection error {e}')
    def desconectar(self):
        self.adapter.disconnect()
    def esta_conectado(self):
        return self.adapter.is_connected()
    def listar_archivos(self, path):
        return self.adapter.list_files(path)
    def abrir_archivo(self,path):
        return self.adapter.open_file(path)
    def leer_archivo(self,path):
        return self.adapter.read_file(path)
    def cerrar_archivo(self,path):
        return self.adapter.close_file(path)
```

Finalmente la clase cliente.py, solo instancia un objeto de la clase Stub (indicando IP y puerto) y luego instancia un objeto cliente de la clase Client y le pasa el Stub creado por parámetros. Luego en un while se informa al usuario los comandos disponibles.

El servidor.

Se crea una clase Stub (dentro del archivo ServerStub.py) que crea e inicializa un servidor con métodos de gRPC y a ese servidor le define un adaptador de la clase StubFSServicer (dentro del archivo ServerStub.py) que extiende de la clase FSServicer que genera RPC e implementa los métodos definidos en dicha clase. Para todos los servidores se creo una clase file_system (dentro del archivo file_system.py) que implementa la funcionalidad de todos los métodos. Es decir que todos los servidores utilizan un objeto de esta clase, llamado adaptador, para realizar la funcionalidad necesaria.

Luego se crea la clase server.py que recibe un adaptador (será un objeto de la clase Stub) y ejecuta los métodos inicializar y run del stub. Esta clase también se utiliza en todos los servidores.

```
class Server:
    def __init__(self, adapter):
        self.adapter = adapter
    def inicializar(self):
        print('Inicializando el servidor')
        self.adapter.run()
```

Por último, se crea el servidor.py que instancia un objeto de file_system.py e instancia un Stub pasándole el adaptador mencionado por parámetro. Luego instanciando la clase Server mencionada arriba a la cual le pasa el Stub creado.

```
def main():
    adaptador=FS()
    stub = Stub(adaptador, 50051)
    servidor = Server(stub)
    servidor.inicializar()
```

b. Para este punto, teniendo definida toda la estructura mencionada en el punto A, se definen nuevamente las clases Stub del servidor y del cliente de la siguiente manera:

El Stub del cliente tiene definidos todos los métodos incluidos el conectar y desconectar implementados con sockets, luego instancia un objeto de la clase FSSTub enviándole el canal de la conexión ya establecida con el servidor. Esta clase tiene definidos los métodos necesarios (Listar archivos, abrir y cerrar archivo y leer archivo), el envío y recepción de todos los mensajes al servidor se realizan en esta clase, se utilizó la librería pickles para serializar las estructuras que se deben enviar al servidor, y para des serializar los mensajes recibidos por el mismo. La clase Stub, dentro de sus métodos solo realiza llamadas a los métodos de esta clase.

```
class Stub:
    def __init__(self, host, port):
        self._appliance = (host, port)
        self._channel = None
        self._stub = None

    def connect(self):
        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):
        if self.is_connected():
            self._stub.Desconectar()
        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.strip())
        return 'El servidor esta desconectado'

    def open_file(self,path):
        if self.is_connected():
            return self._stub.OpenFile(path.strip())
        return 'El servidor esta desconectado'

    def read_file(self,path):
        if self.is_connected():
            return self._stub.ReadFile(path.strip())
        return 'El servidor esta desconectado'

    def close_file(self,path):
```

```
        if self.is_connected():
            return self._stub.CloseFile(path.strip())
        return 'El servidor esta desconectado'
```

En el Stub del servidor se crea un socket y se lo pone a escuchar dentro de un while. Luego se implementó una clase FSStub, que recibe el adaptador (un objeto de la clase file_system.py) el canal con la conexión ya establecida con el cliente: atiende las solicitudes con la librería pickes des serializa el mensaje recibido, y según el valor del parámetro OP, ejecuta los métodos necesarios utilizando el adaptador. Luego serializa la respuesta y la envía a través del canal. Es decir que las clases que se intercambian mensajes a través del socket son FSStub del lado del cliente y FSStub del lado del servidor.

```
class Stub:
    def __init__(self, adapter, port):
        self._port = port
        self._adapter = adapter
        self.server = None
        self._stub = None

    def _setup(self):
        self.server = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        self.server.bind(('192.168.0.106', self._port))

    def run(self):
        self._setup()
        self.server.listen()
        try:
            while True:
                print('server activo')
                connection, client_address = self.server.accept()
                self._stub = FSStub(connection, self._adapter)

        except KeyboardInterrupt:
            connection.close()
            self.server.stop(0)
```

c. La implementación con gRPC es mucho más sencilla ya que no debemos ocuparnos de manejar conexiones, ni de serializar/des serializar las estructuras de los mensajes entre cliente y servidor. En la implementación con sockets se debe especificar dónde se aceptan las conexiones a los clientes, y utilizar el canal para enviar los mensajes que además se deben serializar si se trata de una estructura que contiene parámetros. En gRPC las estructuras de los mensajes; y los parámetros y resultados de los métodos que se ejecutan remotamente se definen en el archivo de protocolo, y luego a partir del protocolo definido gRPC genera los stubs del cliente y del servidor, solo hay que implementar los métodos especificados del lado del servidor. En la implementación con sockets todo esto debe realizarse en forma manual. En conclusión, es más sencillo (o menos laborioso) implementar servicios remotos utilizando gRPC.

4.

a. Se observa que el servidor atrapa una excepción al detectar que la conexión fue terminada por el host remoto, pero al lanzar el cliente acepta la conexión y funciona nuevamente.

Consola del servidor:

```
Inicializando el servidor
server activo
ERROR!!!  [WinError 10054] Se ha forzado la interrupción de una conexión existente por el host remoto
server activo
```

b. Al detener el servidor se observa el mismo error mencionado anteriormente, pero del lado del cliente, detecta que la conexión al servidor ha sido terminada por el host remoto.

Consola del cliente:

```
Ingrese un directorio
.
['file_system.py', 'ServerStub.py', 'Servidor.py', '__pycache__']
************************************************************
Ingrese un comando ([L]ist, [O]pen, [R]ead, [C]lose, [S]alir
P
************************************************************
Ingrese un comando ([L]ist, [O]pen, [R]ead, [C]lose, [S]alir
R
Ingrese un archivo del directorio
file_system.py
ERROR en REadfile  [WinError 10054] Se ha forzado la interrupción de una conexión existente por el host remoto
Error al abrir archivo
```

c. Es un servidor sin estados. Si bien el servidor almacena en un objeto file_manager, todos los descriptores de los archivos abiertos y el ultimo path del cual listó archivos, el cliente para realizar el read file, debe enviar el nombre del archivo, el offset y la cantidad de bytes, actualizando los últimos 2 valores por cada solicitud que haga al servidor, hasta terminar de transferir el archivo.

5. Para implementar un servidor concurrente, se modificaron las clases Stub y FSStub del servidor definidas en el punto 3 b. En primer lugar la clase FSStub extiende de la clase threading.Thread, y el método process_request, se redefine como run(). Luego en el Stub, cuando se establece una conexión con un cliente, se instancia un hilo de esta clase y se ejecuta el método Run() del mismo.

```
class Stub:
    def __init__(self, adapter, port):
        self._port = port
        self._adapter = adapter
        self.server = None
        self._stub = None

    def _setup(self):
        self.server = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        self.server.bind(('192.168.0.106', self._port))

    def run(self):
        self._setup()
        self.server.listen()
        try:
            while True:
                connection, client_address = self.server.accept()
                newthread = FSStub(connection,self._adapter)
                newthread.start()
        except KeyboardInterrupt:
            connection.close()
            self.server.stop(0)


class FSStub(threading.Thread):
    def __init__(self, canal, file_system_adapter):
        self._channel = canal
        self._adapter = file_system_adapter
        threading.Thread.__init__(self)

    def run(self):
        try:
            while True:
                print(threading.get_ident())
                print(self._adapter._file_manager)
                data = self._channel.recv(1000000)
                data_loaded = pickle.loads(data)
                comando = data_loaded["op"]
                path = data_loaded["path"]
                if comando == '1':
                    self.list_file(path)
                if comando == '2':
                    self.open_file(path)
                if comando == '3':
                    self.read_file(path, data_loaded["offset"],
data_loaded["cant_bytes"])
                if comando == '4':
                    self.close_file(path)
                if comando == '5':
                    print("cliente desconectado")
                    self._channel.close()
                    break
        except Exception as e:
```

```
            print('ERROR!!! ', e)
            return
```

6.

a. Se agregó un print(threading.get_ident()) dentro del método run del hilo del servidor y se observó que las solicitudes de distintos clientes son atendidas por distintos hilos. Es decir, al aceptar la conexión del cliente en el servidor, se crea un hilo para atender las solicitudes que llegarán a través de esa conexión, de esta manera cada cliente es atendido por un hilo distinto.

b. A partir de los descripto anteriormente, esta implementación tiene un thread por conexión. Se crea un hilo por cada cliente, el mismo hilo atiende todas las solicitudes y muere al desconectarse el cliente.

c. Se transfirió un archivo de 878347 KB – en ambos servidores con un buffer de 4000 bytes

- Con sockets

  tiempo de transferencia: 57368 ms

- Con gRPC

  tiempo de transferencia: 156755 ms

7.

Para este punto implemente un ThreadPoolExecutor en el Stub del servidor, con un límite de 2 workers, para ello solo fue necesario modificar la clase Stub del servidor del punto 3.b. Cuando se acepta una conexión de un cliente, se ejecuta el método submit del executor (que fue inicializado como un pool de hilos), y allí se llama al método process_request que se encarga de crear un objeto FSStub

```
class Stub:
    def __init__(self, adapter, port):
        self._port = port
        self._adapter = adapter
        self.server = None
        self._executor = None

    def _setup(self):
        self.server = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        self.server.bind(('192.168.0.106', self._port))
        self._executor = ThreadPoolExecutor(max_workers=2)

    def run(self):
        self._setup()
        self.server.listen()
        try:
            while True:
                connection, client_address = self.server.accept()
                self._executor.submit(self.process_request,
connection)
        except KeyboardInterrupt:
            connection.close()
            self.server.stop(0)

    def process_request(self, connection):
        try:
            newthread = FSStub(connection, self._adapter)
        except Exception as e:
            print('ERROR!!! ', e)
            return
```

A mi parecer, es útil esta implementación si se desea limitar el número de clientes conectados en simultáneo.