Práctica 2: Implementación de calculadora usando Thrift

Introducción

-El objetivo de esta práctica consiste en aprender a utilizar la funcionalidad que nos ofrece Thrift para poder hacer comunicaciones entre cliente y servidor, y con la posibilidad de programar el cliente y el servidor en una variedad de idiomas.

-En este caso, tendremos que construir un fichero .thrift que modificaremos usando el propio lenguaje que tiene y podremos definir las funciones que queramos, para luego usar: *"thrift -gen py calculator.thrift"* que generará todos los archivos necesarios (para el lenguaje Python solo, porque hemos usado -py-), exceptuando el cliente y el servidor. En RPC se generaba una plantilla base para que pudiésemos programar el cliente y servidor, pero para esta práctica he utilizado los ficheros de ejemplo mostrados por el profesor.

Primera parte

-Como ya he dicho esta primera parte era definir el .thrift que empieza de la siguiente manera:

```
typedef i32 int
service CalculatorService
{
        // Acepta dos doubles para sumarlos
        double mul(1:double n1, 2:double n2),

        // Acepta dos double para multiplicarlos
        double sum(1:double n1, 2:double n2),

        // Acepta dos double para restarlos
        double sub(1:double n1, 2:double n2),

        // Acepta dos double para dividirlos
        double div(1:double n1, 2:double n2),

        // Acepta dos dobles para hacer la potencia
        double pow(1:double n1, 2:double n2),
```

-La primera línea es para hacer más vistoso el int ya que en este lenguaje se definen como i32. El lenguaje nos permite definir bastantes variables y estructuras. Las operaciones además reciben argumentos ordenados por un número. Estas primeras operaciones reciben valores double, pero si se le pasan enteros se hace casting (desde el cliente) para dar más flexibilidad.

```
// Modulo de un número
int mod(1:int n1, 2:int n2)

// Máximo común divisor con un mínimo de 4 números, si se pasan menos de 4 se controla
// desde la definición de la función
int mcd(1:list<int> nums),

// Mínimo común divisor con un mínimo de 4 números, si se pasan menos de 4 se controla
// desde la definición de la función
int mcm(1:list<int> nums),

// Factorial de un número
int factorial(1:int num),
```

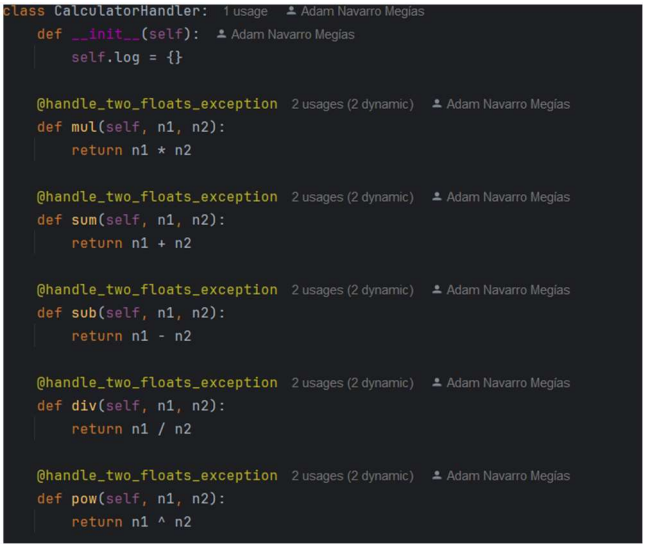
-Como se puede ver solo he definido las funciones más elementales y faltaría implementar algo más elaborado como vectores o matrices. Todas reciben 2 números menos estas tres últimas que reciben una lista o un único número, aunque luego se verá que desde el cliente solo se pueden pasar 4 números para mcd y mcm.

Programa servidor

-Como ya he dicho, se ha utilizado como plantilla para los dos programas los ejercicios de ejemplo dados por el profesor, para el servidor es la siguiente parte:

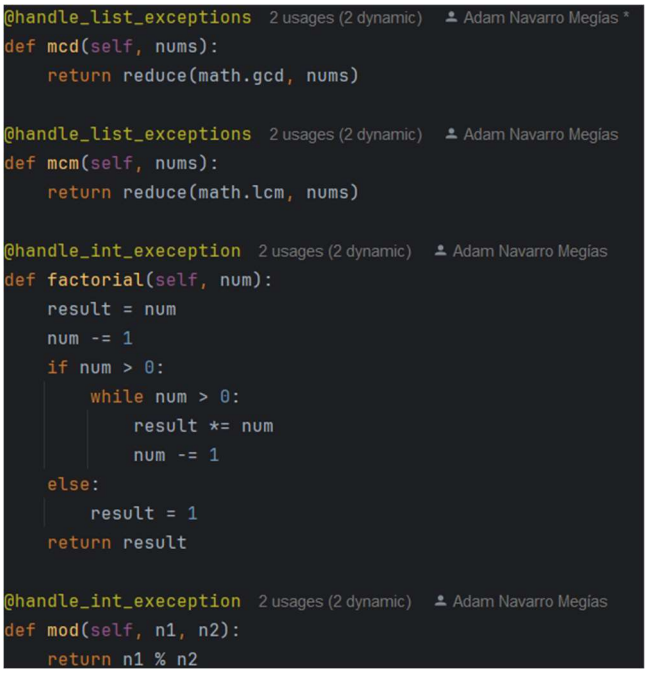
```
##################################################################
if __name__ == "__main__":
    handler = CalculatorHandler()
    processor = CalculatorService.Processor(handler)
    transport = TSocket.TServerSocket(host="127.0.0.1", port=9090)
    tfactory = TTransport.TBufferedTransportFactory()
    pfactory = TBinaryProtocol.TBinaryProtocolFactory()

    server = TServer.TSimpleServer( *args: processor, transport, tfactory, pfactory)

    print("iniciando servidor...")
    server.serve()
    print("fin")
```

-Cuando se ejecute el programa servidor se alojará en localhost. También se hacen los imports pertinentes al principio.



```
class CalculatorHandler:  1 usage  Adam Navarro Megías
    def __init__(self):  Adam Navarro Megías
        self.log = {}

    @handle_two_floats_exception  2 usages (2 dynamic)  Adam Navarro Megías
    def mul(self, n1, n2):
        return n1 * n2

    @handle_two_floats_exception  2 usages (2 dynamic)  Adam Navarro Megías
    def sum(self, n1, n2):
        return n1 + n2

    @handle_two_floats_exception  2 usages (2 dynamic)  Adam Navarro Megías
    def sub(self, n1, n2):
        return n1 - n2

    @handle_two_floats_exception  2 usages (2 dynamic)  Adam Navarro Megías
    def div(self, n1, n2):
        return n1 / n2

    @handle_two_floats_exception  2 usages (2 dynamic)  Adam Navarro Megías
    def pow(self, n1, n2):
        return n1 ^ n2
```

-Como podemos ver la implementación de las operaciones es bastante sencilla ya que son muy elementales, se ha añadido decoradores de Python (más abajo se muestran), pero se ha podido comprobar que no son del todo efectivos, ya que la mayor parte de los errores se captan antes si quiera de que lleguen los datos al servidor.



```
@handle_list_exceptions  2 usages (2 dynamic)  Adam Navarro Megías *
def mcd(self, nums):
    return reduce(math.gcd, nums)

@handle_list_exceptions  2 usages (2 dynamic)  Adam Navarro Megías
def mcm(self, nums):
    return reduce(math.lcm, nums)

@handle_int_exception  2 usages (2 dynamic)  Adam Navarro Megías
def factorial(self, num):
    result = num
    num -= 1
    if num > 0:
        while num > 0:
            result *= num
            num -= 1
    else:
        result = 1
    return result

@handle_int_exception  2 usages (2 dynamic)  Adam Navarro Megías
def mod(self, n1, n2):
    return n1 % n2
```

-Este es el resto de las operaciones, para mcm y mcd se ha usado la biblioteca math, pero se podrían haber hecho en la misma función también

```python
# Decorador de python para manejar excepciones
def handle_list_exceptions(operation):  2 usages  Adam Navarro Megías
    def wrapper(self, *args): # Encapsula las operaciones para añadirle la funcionalidad  Adam Navarro Megías
        try:
            # Si tiene una lista como primer argumento
            if args and isinstance(args[0], list) and len(args[0]) == 1:
                nums = args[0]
                if nums is None or len(nums) == 0: # Hace esta verificación
                    raise EmptyListError("The list passed by argument is empty")
                if all(isinstance(num, int) for num in nums):
                    raise ValueError("The passed arguments are incorrect")

            if (operation.__name__ == "mcd" or operation.__name__ == "mcm") and len(args[0]) < 4:
                raise EmptyListError("The list passed by argument doesnt have enough elements")


            return operation(self, *args) # Si es correcto ejecuta la función normal

        except EmptyListError as e:
            print(e) # En caso de error se muestra
            return e
        except ValueError as e:
            print(e) # En caso de error se muestra
            return e
    return wrapper
```

-Este es el primero de los decoradores (hay otros dos en el código) que funciona muy simple: durante la ejecución se ejecutará este código antes de la propia función como si fuera parte de la misma. En la parte de wrapper(self, *args) se está refiriendo a la función que está encapsulando, es decir, que self es el que se le pasa a la función original y que *args son todos los argumentos tal cual se pasan a la función (se puede usar en funciones de distinto número de argumentos). El bloque de código que son 2 if, hacen unas comprobaciones, en este caso sobre listas, y luego si no se ha lanzado ninguna excepción, se devuelve “operation(self,*args)” siendo operation la operación en concreto y se devuelve para que se ejecute el código ahora si de la operación.

Programa cliente

-En el programa cliente, igual que en el servidor, se ha usado de plantilla el código del profesor, que lo que hace es definir un objeto “client” que será a través del cual se podrán lanzar las operaciones definidas en el servidor, y se hará entre las líneas de “transport.open()” y “transport.close()”.

-Para el cliente no voy a poner capturas de todo el código porque es más largo, pero si voy a explicar el proceso de hacerlo. Mi idea original era hacer algo parecido a la primera parte de la práctica 2 pero más sencillo, pero luego apareció la idea de hacer una interfaz gráfica (no se pedía en la práctica) para así practicar con alguna biblioteca de Python, y es lo que hice. Para el cliente entonces he

usado tkinter de Python, que permite hacer interfaces gráficas, aunque he hecho una bastante simple porque es la primera vez que uso la biblioteca.

```
def __init__(self):  Adam Navarro Megías
    # Creamos las caracteristicas iniciales de nuestra ventana
    self.root = tk.Tk()
    self.root.geometry('800x800')
    self.max_text_length = 50

    # Creamos dos pestañas, una tendrá una calculadora más básica y otra tendrá
    # Operaciones como vectores o mcm y mcd
    self.notebook = ttk.Notebook(self.root)
    self.tab1 = ttk.Frame(self.notebook)
    self.tab2 = ttk.Frame(self.notebook)

    self.notebook.add(self.tab1, text='Basic')
    self.notebook.add(self.tab2, text='Advanced')
    self.notebook.pack(expand=1, fill='both')

    self.notebook.select(self.tab1)
```

-El constructor de la clase “Calculator” es donde está el grueso del programa, ya que se hace toda la construcción de la interfaz y define los botones que harán llamadas a las funciones pertinentes para hacer los cálculos.

-Se define un root que será toda la ventana y un tamaño, en este caso he elegido 800x800 porque me parecía suficiente. Además he definido 2 pestañas usando Notebook para así separar una calculadora normal de las operaciones de mcm y mcd y es un diseño más modular que permite añadir más pestañas para otras operaciones. Luego se definen los botones y labels para ambas pestañas.

```
self.result = tk.StringVar()
self.result.trace_add( mode: "write", self.limit_tam)
self.entry = tk.Entry(self.tab1, state="readonly" , textvariable=self.result, font=("Arial", 12), b
self.entry.pack(padx=10, pady=10)
```

-Este fragmento es importante porque se define “self.result” que es una variable que se va a modificar durante ejecución para poner los operandos y donde se va a poder ver el resultado, además está ligada a un “Entry” para poder visualizarlo.

```
row = 0
column = 0
for button in self.buttons:
    if button == "V" or button == "mcm" or button == "mcd":
        tk.Button(self.buttonframe, text=button, command=self.changeFrame).grid(row=row, column=column,
                                                                                 sticky=tk.W+tk.E+tk.S+tk.N)
    else:
        (tk.Button(self.buttonframe, text=button, command=lambda b=button: self.parse_button(b))
            .grid(row=row, column=column, sticky=tk.W+tk.E+tk.S+tk.N))

    if column < 3:
        column+=1
    else:
        row+=1
        column = 0
```

-Este bloque es importante también porque lo que hago aquí es recorrer un listado de strings (definido al principio del class) y para cada uno de los símbolos se crea un botón que se va a colocar en el grid y tendrá una llamada de función que será la que modifique los resultados. La mayor parte de los botones llaman la función "parse_button(b)" que hace lo siguiente:

```
def parse_button(self, button):  1 usage   Adam Navarro Megías
    if button == '=': self.calculate()
    elif button == 'C': self.clear()
    elif button == 'back': self.go_back()
    elif button in '^!+-*/%':
        self.result.set(self.result.get() + f' {button} ')
    else:
        self.result.set(self.result.get() + button)
```

-Es una función que dependiendo del botón pues calculará el resultado, borrará el resultado o simplemente escribirá el contenido del botón en la calculadora.

-Cabe remarcar que la calculadora puede hacer un conjunto de operaciones pero **no sigue ningún orden de operaciones, las realiza de manera secuencial según aparecen.**

```
def calculate(self):  1 usage   Adam Navarro Megías
    text = self.result.get()
    text.strip()
    tokens = text.split() # Tokenizamos la entrada y la recorremos
    if len(tokens) < 2: return

    res = 0



    if tokens[1] == "!":
        res = client.factorial(int(float(tokens[0])))
        tokens.remove(tokens[0])
        tokens.remove(tokens[0])
    elif tokens[1] in '^+-*/%' and len(tokens) >= 3:
        if tokens[1] == '%':
            res = self.operations[tokens[1]](int(tokens[0]), int(tokens[2]))
        else:
            res = self.operations[tokens[1]](float(tokens[0]), float(tokens[2]))
        tokens.remove(tokens[0])
        tokens.remove(tokens[0])
        tokens.remove(tokens[0])
    else:
        res = "Operacion mal construida"
```

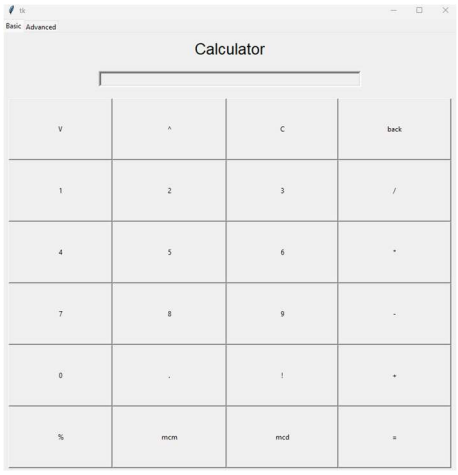
-Esta es la primera parte del cálculo. Básicamente se tokeniza la entrada (se divide todo el string en substrings separados por espacios) y se hace una primera operación. Se comprueba si es factorial porque es la operación que recibe un solo número y se comporta distinto.

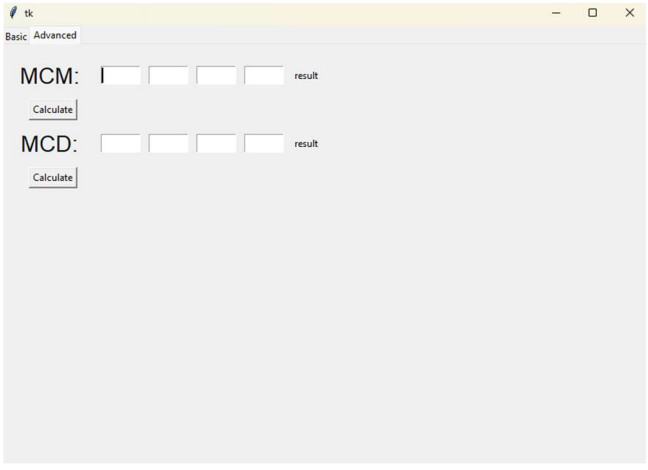
```
for indice, token in enumerate(tokens):  # Parseamos y vamos llamando a las funciones pertinentes
    try:
        if token in '^!+-*/%':
            if token == "!":
                res = client.factorial(int(float(res)))
                tokens.remove(token)
            else:
                if token == '%':
                    res = self.operations[token](int(res), int(tokens[indice + 1]))
                else:
                    res = self.operations[token](float(res), float(tokens[indice + 1]))

                tokens.remove(tokens[indice + 1])
                tokens.remove(token)
        else:
            raise ValueError

    except ValueError:
        print("Valor incorrecto introducido")


self.result.set(str(res))
```

-Luego si hay más contenido y operaciones se realizan en un for siguiendo la misma estructura y usando el valor ya calculado.

-Mcm y mcd se hacen en la otra pestaña y tienen su propia función de calcular.

-Para utilizar el programa es bastante simple, se ejecuta el servidor y luego se ejecuta el cliente que se verá intuitivo de usar ya que es como una calculadora normal:





Conclusión

-He realizado una calculadora simple con una interfaz gráfica, se podrían implementar más operaciones modificando el .thrift, añadiéndola al servidor y luego modificando la interfaz y el cliente con más pestañas o en la misma pestaña añadiendo la funcionalidad. En la memoria no se ha mostrado todo el código, se pueden ver en más detalle todas las funciones del cliente en el código (se han mostrado las partes más relevantes).