UNIVERSITAT ROVIRA I VIRGILI

Tarea 1: Modelos de comunicación y *middleware*

Arey Ferrero Ramos

7 de noviembre del 2022

Índice

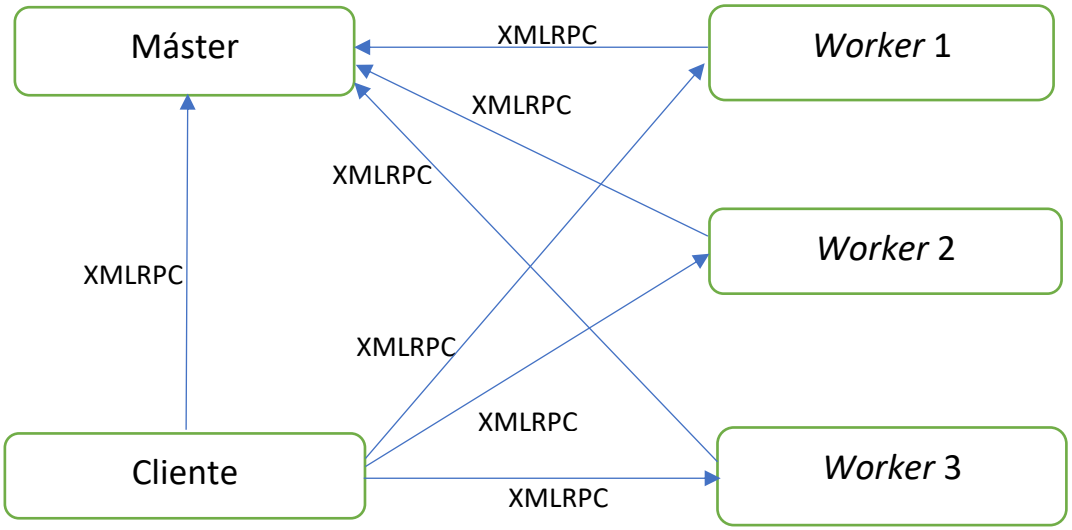
Especificaciones

Se pide la implementación de una arquitectura distribuida que paralelice una librería de Python llamada Pandas. Se recomienda tener como referencia la librería Dask, que ya se encarga de implementar dicha funcionalidad. La lista de funciones de la librería pandas disponibles para paralelizar son: read_csv(), apply(), columns(), groupby(), head(), isin(), items(), max() y min(). Se deben implementar dos tipos de arquitectura distribuida: la arquitectura distribuida con un *middleware* de comunicación directa y la arquitectura híbrida (usa un middleware de comunicación directa y uno de comunicación indirecta). Además, se puede implementar de forma optativa la arquitectura distribuida con un middleware de comunicación indirecta. Para llevar todo esto a cabo, se dispone de los *middlewares* de comunicación directa XMLRPC y gRPC, y de los *middlewares* de comunicación indirecta Redis y RabbitMQ.

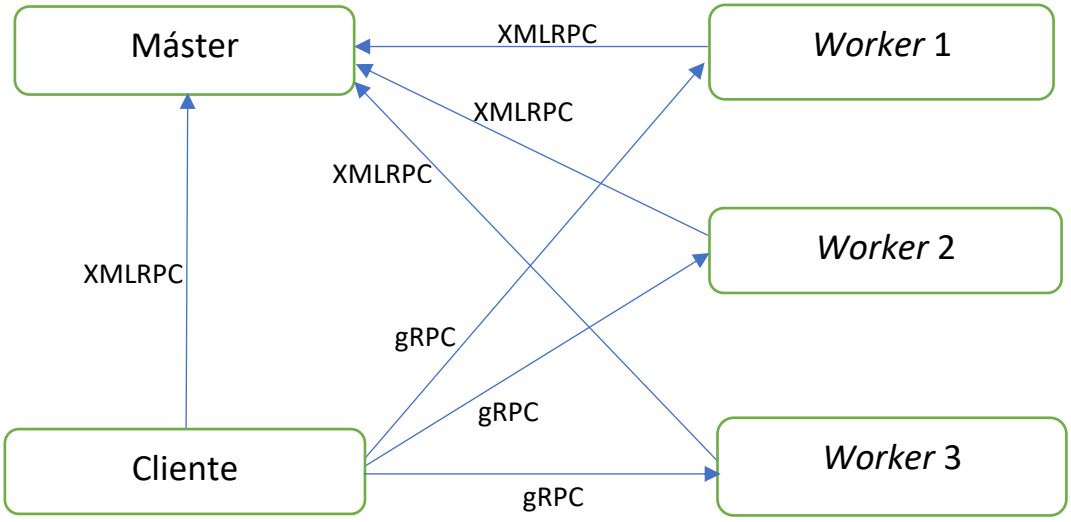
Diseño

Diagramas

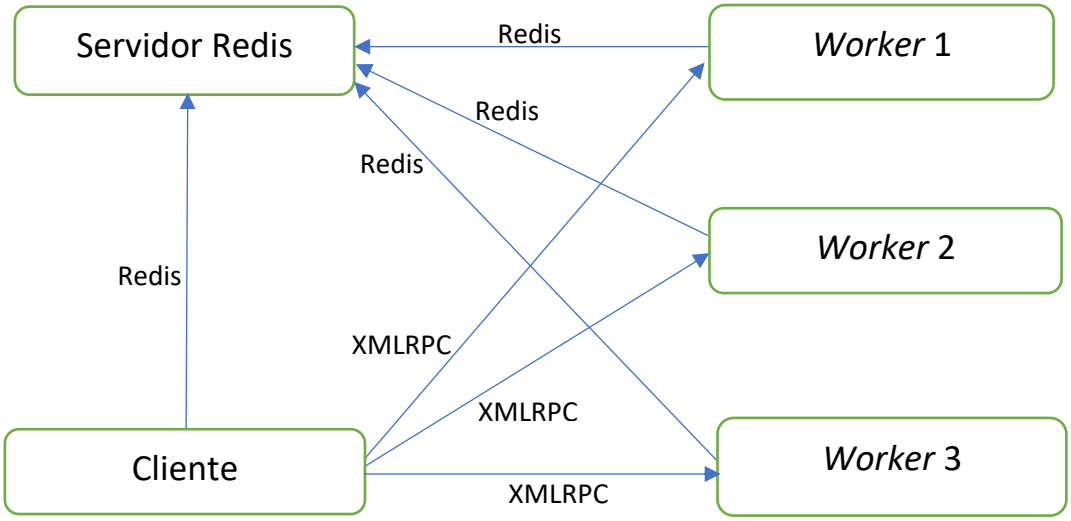
Arquitectura distribuida con el middleware de comunicación directa XMLRPC



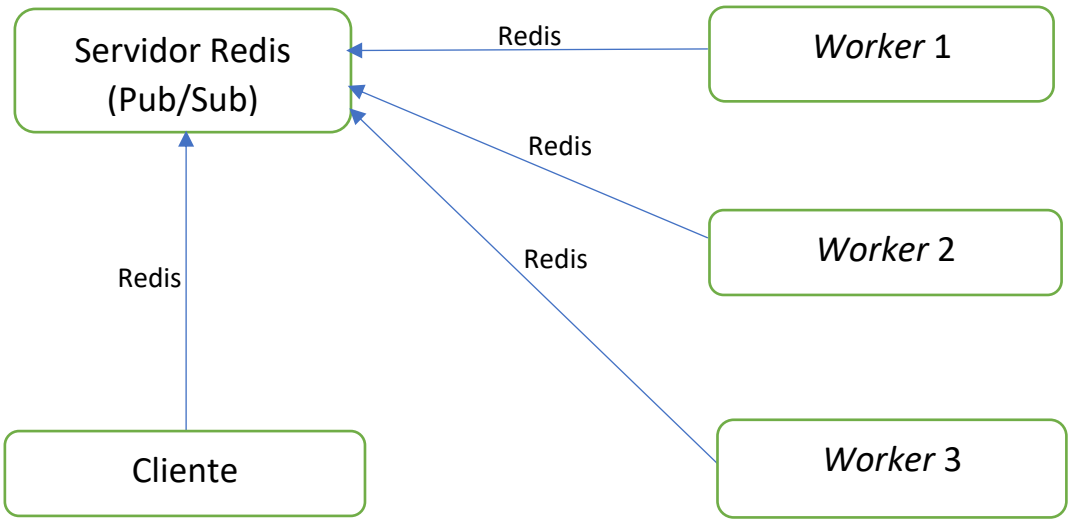
Arquitectura distribuida con los middlewares de comunicación directa XMLRPC y gRPC



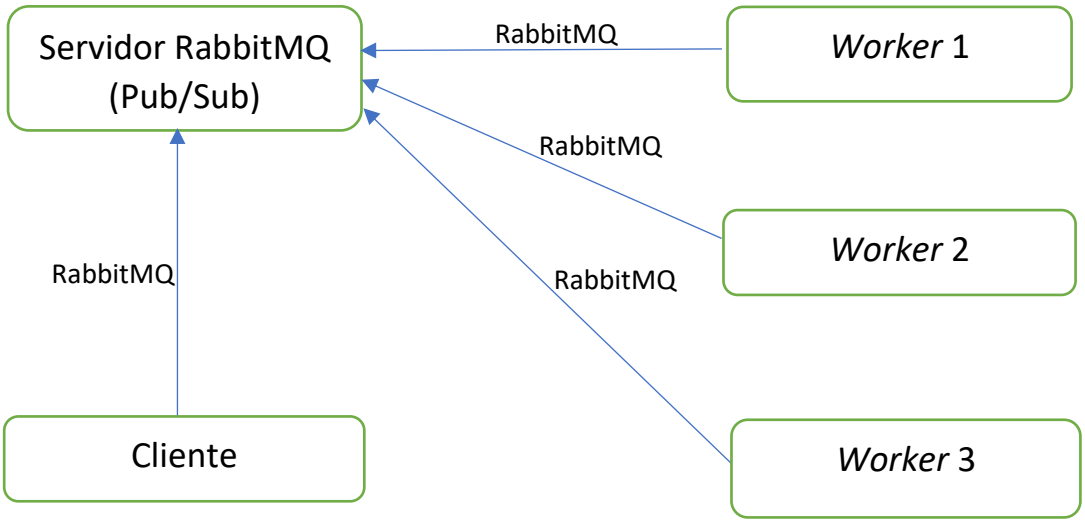
Arquitectura híbrida con el middleware de comunicación directa XMLRPC y el middleware de comunicación indirecta Redis



Arquitectura distribuida con el middleware de comunicación indirecta Redis



Arquitectura distribuida con el middleware de comunicación indirecta RabbitMQ



Comparativa de arquitecturas de comunicación directa e indirecta

Las arquitecturas que utilizan *middlewares* de comunicación directa se caracterizan porque cada nodo del clúster conoce la ubicación exacta de los demás nodos. Dicho de otro modo, cada nodo tiene acceso a la URL de los demás nodos. Esto permite que cada nodo pueda enviar los mensajes directamente a los demás nodos sin que se requiera un intermediario. Por otro lado, las arquitecturas que utilizan middlewares de comunicación indirecta se caracterizan porque cada nodo desconoce la ubicación de los demás nodos del clúster. Dicho de otro modo, no se almacena en ningún

sitio la URL de ningún nodo. Por lo tanto, para que los nodos puedan enviarse mensajes entre ellos se requiere un intermediario.

Se han implementado dos arquitecturas que utilizan exclusivamente middlewares de comunicación directa. La primera arquitectura utiliza únicamente el middleware XMLRPC. Aquí, el nodo máster define una clase con una estructura de datos tipo lista y un conjunto de métodos que realizan distintas operaciones sobre esta lista (añadir nodo, eliminar nodo, …). De este modo, cada vez que se crea un nodo *worker*, éste llamará al método que le permite registrarse en la lista del máster. Cuando el cliente quiera acceder a los nodos *worker* para distribuir su tarea, accederá primero al nodo máster para obtener la URL de cada worker y después distribuirá los ficheros entre los distintos *workers*.

La segunda arquitectura tiene exactamente el mismo diseño que la primera, pero utiliza el middleware XMLRPC sólo para la comunicación del cliente y los nodos *workers* con el nodo máster y el middleware gRPC para la comunicación del cliente con los nodos *workers*. Dado que el middleware gRPC es considerablemente más complejo que el middleware XMLRPC, se ha decidido mantener el uso del segundo para los casos ya mencionados. Además, se ha reducido la batería de funciones de pandas que se tiene que paralelizar a tres: read_csv(), max() y min(). He considerado que con estas funcionalidades ya queda demostrada la capacidad de manejar el middleware gRPC.

También se ha implementado una arquitectura híbrida, que utiliza el middleware de comunicación indirecta Redis para la comunicación del cliente y los nodos workers con el nodo máster y el middleware de comunicación directa XMLRPC para la comunicación del cliente con los nodos *workers*. Esta arquitectura es muy similar a las arquitecturas que utilizan middlewares de comunicación directa, aunque se distingue de éstas dado que no hay un nodo máster como tal. En su lugar se utiliza un servidor Redis que contiene una estructura de datos de tipo lista en la que los workers almacenan sus URLs al ser creados para que el cliente pueda acceder a ellas y comunicarse con estos. Dado que no se implementa ningún paradigma de comunicación indirecta, se podría decir que no se aprovechan las ventajas de este tipo de comunicación.

En todas las arquitecturas que utilizan middlewares de comunicación directa se han implementado dos versiones. En la primera, el envío de los ficheros csv para su procesamiento se lleva a cabo de forma iterativa. Esta versión, aunque permite familiarizarse con la API de los middlewares de comunicación directa, no permite aprovechar las ventajas de disponer de un sistema distribuido. Aunque el código de dicha versión no se ha presentado, si que se muestra en la documentación, así como su juego de pruebas y los gráficos con su tiempo de ejecución. En la segunda implementación se hace uso de hilos de ejecución (*threads*) para lograr un envío de los ficheros csv de forma paralela, de modo que sí que se logra aprovechar todas las ventajas de disponer de un sistema distribuido.

Además, se ha implementado una arquitectura que utiliza exclusivamente el middleware de comunicación indirecta Redis, que parte de un diseño completamente

diferente de las anteriores basado en el paradigma publish/subscribe. Igual que en la arquitectura híbrida, no hay un nodo máster como tal, sino que se utiliza un servidor Redis al que se conecta cada nodo worker y el cliente. Dado que ninguno de los nodos que participa en la comunicación conoce la ubicación de los demás, cada vez que un nodo deba comunicarse con los demás, deberá enviar un mensaje al servidor Redis. No obstante, Redis en realidad es un middleware de comunicación indirecta basado en un espacio de memoria compartida y, aunque soporta el paradigma publish/subscribe, no es un middleware pensado para ello. Esto plantea ciertas complicaciones que hacen que ésta haya sido la arquitectura más compleja de implementar de todas y, por lo tanto, en la que se han tenido que tomar las decisiones de diseño más distinguidas con respecto a lo explicado en los tutoriales sobre Redis.

La primera gran dificultad con la que me he encontrado es que cuando un nodo worker leía un mensaje del servidor Redis, no consumía dicho mensaje, de modo que los nodos workers en lugar de hacer cada uno una tarea, hacían todos todas las tareas. Para solucionar este problema he hecho que cada nodo cree un tópico diferente dentro del servidor Redis a través del cual se comunicará con el cliente. No obstante, para que el cliente pueda coordinarse correctamente con cada worker para establecer el nombre del tópico se ha tenido que crear toda una serie de instrucciones adicionales, así como una variable global (almacenada en el servidor Redis) que contiene el número de nodos worker que se han creado y un fichero adicional que sirve únicamente para inicializar esta variable a 0 (dado que los nodos workers no pueden llevar a cabo esta funcionalidad y el nodo máster es el servidor Redis). Además, ésta es la única arquitectura en la que al destruir un nodo se han tenido que ejecutar un conjunto de instrucciones. Estas instrucciones sirven precisamente para reescribir los nombres de los tópicos en cada nodo *worker* de manera que se mantenga la coherencia entre estos y el cliente. De lo contrario, la carga quedaría desbalanceada o incluso si el nodo destruido fuese el primero en ser creado, el clúster no funcionaría.

El segundo inconveniente con el que me he encontrado es coordinar la publicación de mensajes con el consumo de estos. Esto es problemático porque, en ocasiones, uno de los nodos puede consumir un mensaje antes de que otro lo haya publicado. Para solventar este problema, he creado bucles de espera que mantienen la ejecución del programa en un mismo punto hasta que se ha publicado el mensaje en el tópico correspondiente. Cabe destacar que las soluciones que se han propuesto para estos dos problemas están lejos de ser buenas prácticas de programación, sino que funcionan más bien como parches. Lo ideal hubiese sido que la función encargada de obtener los mensajes permitiese elegir entre únicamente leer el mensaje o consumirlo (borrarlo después de su lectura) y que el middleware de comunicación indirecta utilizado permitiese utilizar comunicación síncrona en lugar de asíncrona.

Por último, se ha implementado una arquitectura que utiliza el middleware de comunicación indirecta RabbitMQ partiendo de nuevo de un diseño basado en el paradigma publish/subscribe. Se utiliza un servidor RabbitMQ al que se conectan cada nodo worker y el cliente. Dado que los *workers* y el cliente van a actuar ambos como

Publisher y como *Consumer* en distintos puntos del programa, se ha decidido crear una clase Publisher y una clase Consumer. Así, los workers y el cliente crean objetos de dichas clases para llevar a cabo las funcionalidades requeridas. La clase Publisher no ha dado ningún problema, ya que es fácil de implementar. Por otro lado, el método callback(), perteneciente a la clase Consumer, ha de ser diferente en función de si el que actúa como consumidor es el cliente o uno de los workers. Por ello, se ha utilizado herencia, es decir, se ha convertido la clase Consumer en una clase abstracta, siendo el método callback() un método abstracto cuyo contenido es sobrescrito en el código del cliente y de los workers.

Uno de los problemas que he tenido que resolver es que, como ocurre con la arquitectura anterior, el middleware RabbitMQ soporta el paradigma publish/subscribe, pero no está pensado para ello, sino que está planteado en base al paradigma *message queue*. Dado que ambos paradigmas pertenecen al grupo de las arquitecturas basadas en eventos, debería haber menos problemas de compatibilidad. Aun así, además de que la API de RabbitMQ es más compleja que la de Redis, vuelve a existir el problema de que todos los nodos worker consumen todos los mensajes. Para solucionar esto, se debería etiquetar cada una de las colas de mensajes con un tópico, de manera que el cliente pudiese repartir los ficheros entre los distintos workers. Para hacer esto, el cliente debería conocer el número de workers, cosa que ya de por sí contradice la lógica de la comunicación indirecta dado que, si el cliente necesita conocer esta información, no estará completamente desacoplado de los workers. Además, el servidor RabbitMQ no permite crear ninguna variable global para almacenar el número de *workers*, como se ha hecho en Redis. Esto implica que no es posible, con los conocimientos de los que dispongo para esta práctica, implementar una arquitectura distribuida con el middleware de comunicación indirecta RabbitMQ que funcione para todos los casos de uso planteados en el resto de las arquitecturas. Así pues, se ha decidió programar una arquitectura que funcione para cualquier número de workers y cualquier número de ficheros csv, siempre y cuando ambos números coincidan o el número de ficheros csv sea inferior al número de workers. De este modo, será responsabilidad del usuario prever cuantos nodos worker va a necesitar en función del número de ficheros csv que quiera tratar.

La segunda problemática con la que me he encontrado es que la API de RabbitMQ está diseñada para que los nodos que adopten el rol de consumidor se mantengan bloqueados de forma síncrona a la espera de nuevos mensajes. Esto es conveniente para el funcionamiento de los workers, pero conlleva un gran problema en el diseño del cliente y es que este se queda bloqueado después de imprimir los resultados de la aplicación de las funciones sobre los ficheros csv. Tal y como se ha implementado, el usuario que adopte el rol de cliente deberá pulsar las teclas CTRL + C para que se muestren por pantalla las temperaturas máxima y mínima globales y el programa termine su ejecución. Existen implementaciones alternativas como hacer uso de un *timeout* para que la ejecución del código del cliente no sobrepase un tiempo específico. No obstante, debido a que la clase Consumer es una generalización de la que los workers y el cliente crean una instancia, el cambio de diseño de uno basado en la

utilización de las teclas CTRL + C para detener la ejecución del programa a uno que haga uso de un timeout también afectará a los workers, cosa que no nos interesa. Aunque esto también se podría solucionar convirtiendo el método consume() en otro método abstracto que sería implementado de forma diferente en el código de los workers y en el del cliente, esto no conlleva ninguna mejora real en el funcionamiento del programa, ya que al usuario que adopta el rol del cliente le puede interesar más detener la ejecución del programa a su conveniencia, en lugar de tener que adaptarse a un tiempo de espera preestablecido. Además, el uso de un *timeout* tampoco nos permitirá averiguar el tiempo de ejecución de real de RabbitMQ para hacernos una idea de su rendimiento, ya que el tiempo de ejecución será el del *timeout*.

Implementación

DaskFunctions.py

```
import pandas


class DaskFunctions:
    def readCSV(self, name_file):
        self.df = pandas.read_csv(name_file)
        return str(self.df)


    def apply(self, fields, code):
        return str(self.df[fields].apply(code))


    def columns(self):
        return str(self.df.columns)


    def groupby(self, field):
        return str(self.df.groupby(field))


    def head(self, num_rows):
        return str(self.df.head(num_rows))


    def isin(self, field, element):
        return str((element in self.df[field].values) == True)
```

```
    def item(self, row, column):
        return str(self.df.iloc[row, column])

    def items(self):
        return str(self.df.items())

    def max(self, field):
        return str(self.df.loc[:,field].max())

    def min(self, field):
        return str(self.df.loc[:,field].min())
```

Arquitectura distribuida con el middleware de comunicación directa XMLRPC

Master.py

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

master = SimpleXMLRPCServer(('localhost', 9000), logRequests=True)
logging.basicConfig(level=logging.INFO)

class WorkerFunctions:
    workers=[]

    def addWorker(self, worker):
        self.workers.append(worker)
        return ' '

    def listWorkers(self):
        return list(self.workers)

    def numWorkers(self):
```

```
        return len(self.workers)

    def removeWorker(self, port):
        for worker in self.workers:
            if port==worker.split(':')[2]:
                self.workers.remove(worker)
        return ' '

master.register_instance(WorkerFunctions())

try:
    print('Use control + c to exit the Master node')
    master.serve_forever()
except KeyboardInterrupt:
    print('Exiting master node')
```

Worker.py

```
import xmlrpc.client
import sys

from xmlrpc.server import SimpleXMLRPCServer
import logging
import daskFunctions

proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
proxy.addWorker('http://localhost:'+sys.argv[1])

worker = SimpleXMLRPCServer(('localhost', int(sys.argv[1])),
logRequests=True)
logging.basicConfig(level=logging.INFO)

worker.register_instance(daskFunctions.DaskFunctions())

try:
```

```
    print('Use control + c to exit the Worker node')
    worker.serve_forever()
except KeyboardInterrupt:
    print('Exiting Worker node')
    proxy.removeWorker(sys.argv[1])
```

Client.py (Iterativo)

```
import xmlrpc.client
import sys

client_master = xmlrpc.client.ServerProxy('http://localhost:9000')

maxs=[]
mins=[]
i=1

while i<len(sys.argv):
    client_worker =
    xmlrpc.client.ServerProxy(client_master.listWorkers()[(i-
    1)%client_master.numWorkers()])
    print(client_worker.readCSV(sys.argv[i])+"\n")
    print(client_worker.columns()+"\n")
    print(client_worker.head(5)+"\n")
    print(client_worker.isin('City', 'Tarragona')+"\n")
    print(client_worker.item(5, 3)+"\n")
    maxs.append(float(client_worker.max('Temp_max')))
    mins.append(float(client_worker.min('Temp_min')))
    i+=1

print("Temperatura maxima: "+str(max(maxs)))
print("Temperatura minima: "+str(min(mins)))
```

Client.py (Paralelizado)

```
import xmlrpc.client
import threading
import sys

def treat_file(client_worker, i, maxs, mins):
      print(client_worker.readCSV(sys.argv[i])+"\n")
      print(client_worker.columns()+"\n")
      print(client_worker.head(5)+"\n")
      print(client_worker.isin('City', 'Tarragona')+"\n")
      print(client_worker.item(5, 3)+"\n")
      maxs.append(float(client_worker.max('Temp_max')))
      mins.append(float(client_worker.min('Temp_min')))

client_master = xmlrpc.client.ServerProxy('http://localhost:9000')

threads=[]
maxs=[]
mins=[]

i=1
while i<len(sys.argv):
      num_worker=0
      while num_worker<client_master.numWorkers() and i<len(sys.argv):
            client_worker =
            xmlrpc.client.ServerProxy(client_master.listWorkers()[num_w
            orker])
            threads.append(threading.Thread(target=treat_file,
            name="thread%s" %i, args=(client_worker, i, maxs, mins)))
            threads[num_worker].start()
            num_worker+=1
            i+=1
      num_worker=0
      while num_worker<len(threads):
            threads[num_worker].join()
```

```
            num_worker+=1
        threads.clear()

print("Temperatura maxima: "+str(max(maxs)))
print("Temperatura minima: "+str(min(mins)))
```

Arquitectura distribuida con los middlewares de comunicación directa XMLRPC y gRPC

Master.py

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

master = SimpleXMLRPCServer(('localhost', 9000), logRequests=True)
logging.basicConfig(level=logging.INFO)

class WorkerFunctions:
    workers=[]

    def addWorker(self, worker):
        self.workers.append(worker)
        return ' '

    def listWorkers(self):
        return list(self.workers)

    def numWorkers(self):
        return len(self.workers)

    def removeWorker(self, port):
        for worker in self.workers:
            if port==worker.split(':')[2]:
                self.workers.remove(worker)
```

```
        return ' '

master.register_instance(WorkerFunctions())

try:
    print('Use control + c to exit the Master node')
    master.serve_forever()
except KeyboardInterrupt:
    print('Exiting master node')
```

DaskFunctions.py

```
import pandas

class DaskFunctions:
    def __init__(self):
        self.df = None

    def readCSV(self, name_file):
        self.df = pandas.read_csv(name_file)
        return str(self.df)

    def max(self, field):
        return self.df.loc[:,field].max()

    def min(self, field):
        return self.df.loc[:,field].min()

daskFunctions = DaskFunctions()
```

DaskFunctions.proto

```
syntax = "proto3";

message NameFile {
```

```
     string name_file = 1;
}

message Field {
     string field = 1;
}

message FileReturn {
     string value = 1;
}

message ValueReturn {
     float value = 1;
}

service DaskFunctions {
    rpc ReadCSV(NameFile) returns (FileReturn) {}
    rpc Max(Field) returns (ValueReturn) {}
    rpc Min(Field) returns (ValueReturn) {}
}
```

Worker.py

```
import xmlrpc.client
import sys
import os

import grpc
from concurrent import futures
import time

import daskFunctions_pb2
import daskFunctions_pb2_grpc

from DaskFunctions import daskFunctions
```

```
proxy = xmlrpc.client.ServerProxy('http://localhost:9000')
proxy.addWorker('http://localhost:'+sys.argv[1])


class
DaskFunctionsServicer(daskFunctions_pb2_grpc.DaskFunctionsServicer):
    def ReadCSV(self, request, context):
        response = daskFunctions_pb2.FileReturn()
        response.value = daskFunctions.readCSV(request.name_file)
        return response


    def Max(self, request, context):
        response = daskFunctions_pb2.ValueReturn()
        response.value = daskFunctions.max(request.field)
        return response


    def Min(self, request, context):
        response = daskFunctions_pb2.ValueReturn()
        response.value = daskFunctions.min(request.field)
        return response


server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))


daskFunctions_pb2_grpc.add_DaskFunctionsServicer_to_server(DaskFunctio
nsServicer(), server)


print('Starting Worker node server. Listening on port
'+sys.argv[1]+'.')
server.add_insecure_port('[::]:'+sys.argv[1])
server.start()


try:
    while True:
        time.sleep(86400)
except KeyboardInterrupt:
    print('Exiting worker node')
```

```
    server.stop(0)
    proxy.removeWorker(sys.argv[1])
```

Client.py (Iterativo)

```
import xmlrpc.client
import sys

import grpc
import daskFunctions_pb2
import daskFunctions_pb2_grpc

client_master = xmlrpc.client.ServerProxy('http://localhost:9000')

maxs=[]
mins=[]
i=1

while i<len(sys.argv):
    channel =
    grpc.insecure_channel('localhost:'+client_master.listWorkers()[(i-
    1)%client_master.numWorkers()].split(':')[2])
    client_worker = daskFunctions_pb2_grpc.DaskFunctionsStub(channel)
    name_file = daskFunctions_pb2.NameFile(name_file=sys.argv[i])
    print(client_worker.ReadCSV(name_file).value+"\n")
    field = daskFunctions_pb2.Field(field='Temp_max')
    maxs.append(float(client_worker.Max(field).value))
    field = daskFunctions_pb2.Field(field='Temp_min')
    mins.append(float(client_worker.Min(field).value))
    i+=1

print("Temperatura maxima: "+str(round(max(maxs), 2)))
print("Temperatura minima: "+str(round(min(mins), 2)))
```

Client.py (Paralelizado)

```
import xmlrpc.client
import threading
import sys

import grpc
import daskFunctions_pb2
import daskFunctions_pb2_grpc

def treat_file(client_worker, i, maxs, mins):
      name_file = daskFunctions_pb2.NameFile(name_file=sys.argv[i])
      print(client_worker.ReadCSV(name_file).value+"\n")
      field = daskFunctions_pb2.Field(field='Temp_max')
      maxs.append(float(client_worker.Max(field).value))
      field = daskFunctions_pb2.Field(field='Temp_min')
      mins.append(float(client_worker.Min(field).value))

client_master = xmlrpc.client.ServerProxy('http://localhost:9000')

threads=[]
maxs=[]
mins=[]

i=1
while i<len(sys.argv):
      num_worker=0
      while num_worker<client_master.numWorkers() and i<len(sys.argv):
            channel =
            grpc.insecure_channel('localhost:'+client_master.listWorker
            s()[num_worker].split(':')[2])
            client_worker =
            daskFunctions_pb2_grpc.DaskFunctionsStub(channel)
            threads.append(threading.Thread(target=treat_file,
            name="thread%s" %i, args=(client_worker, i, maxs, mins)))
            threads[num_worker].start()
```

```
            num_worker+=1
            i+=1
      num_worker=0
      while num_worker<len(threads):
            threads[num_worker].join()
            num_worker+=1
      threads.clear()

print("Temperatura maxima: "+str(round(max(maxs), 2)))
print("Temperatura minima: "+str(round(min(mins), 2)))
```

Arquitectura híbrida con el middleware de comunicación directa XMLRPC y el middleware de comunicación indirecta Redis

Servidor Redis (master.sh)

```
#!/bin/bash

redis-server --port 16379
```

Worker.py

```
import redis
from xmlrpc.server import SimpleXMLRPCServer
import logging
import sys
import daskFunctions

redis_cli = redis.Redis(host="localhost", port=16379)
redis_cli.rpush('workers', 'http://localhost:'+sys.argv[1])

worker = SimpleXMLRPCServer(('localhost', int(sys.argv[1])),
logRequests=True)
logging.basicConfig(level=logging.INFO)
```

```
worker.register_instance(daskFunctions.DaskFunctions())


try:
    print('Use control + c to exit the Worker node')
    worker.serve_forever()
except KeyboardInterrupt:
    print('Exiting Worker node')
    redis_cli.lrem('workers', 0, 'http://localhost:'+sys.argv[1])
```

Client.py (Iterativo)

```
import redis
import xmlrpc.client
import sys


redis_cli = redis.Redis(host="localhost", port=16379,
decode_responses=True, encoding="utf-8")


maxs=[]
mins=[]
i=1


while i<len(sys.argv):
    worker = redis_cli.lpop('workers')
    client_worker = xmlrpc.client.ServerProxy(worker)
    redis_cli.rpush('workers', worker)
    print(client_worker.readCSV(sys.argv[i])+"\n")
    print(client_worker.columns()+"\n")
    print(client_worker.head(5)+"\n")
    print(client_worker.isin('City', 'Tarragona')+"\n")
    print(client_worker.item(5, 3)+"\n")
    maxs.append(float(client_worker.max('Temp_max')))
    mins.append(float(client_worker.min('Temp_min')))
    i+=1
```

```
print("Temperatura maxima: "+str(max(maxs)))

print("Temperatura minima: "+str(min(mins)))
```

Client.py (Paralelizado)

```
import redis

import xmlrpc.client

import threading

import sys


redis_cli = redis.Redis(host="localhost", port=16379,
decode_responses=True, encoding="utf-8")


def treat_file(client_worker, i, max, mins):

      print(client_worker.readCSV(sys.argv[i])+"\n")

      print(client_worker.columns()+"\n")

      print(client_worker.head(5)+"\n")

      print(client_worker.isin('City', 'Tarragona')+"\n")

      print(client_worker.item(5, 3)+"\n")

      maxs.append(float(client_worker.max('Temp_max')))

      mins.append(float(client_worker.min('Temp_min')))


threads=[]

maxs=[]

mins=[]


i=1

while i<len(sys.argv):

      num_worker=0

      while num_worker<redis_cli.llen('workers') and i<len(sys.argv):

            worker = redis_cli.lpop('workers')

            client_worker = xmlrpc.client.ServerProxy(worker)

            threads.append(threading.Thread(target=treat_file,
            name="thread%s" %i, args=(client_worker, i, maxs, mins)))

            threads[num_worker].start()
```

```
            redis_cli.rpush('workers', worker)
            num_worker+=1
            i+=1
      num_worker=0
      while num_worker<len(threads):
            threads[num_worker].join()
            num_worker+=1
      threads.clear()

print("Temperatura maxima: "+str(max(maxs)))
print("Temperatura minima: "+str(min(mins)))
```

Arquitectura distribuida con el middleware de comunicación indirecta Redis

Servidor Redis (master.sh)

```
#!/bin/bash

redis-server --port 16379
```

DefineNumWorkers.py

```
import redis

redis_cli = redis.Redis(host="localhost", port=16379,
decode_responses=True, encoding="utf-8")
redis_cli.set('num_workers', str(0))
```

Worker.py

```
import redis
import daskFunctions
import time

redis_cli = redis.Redis(host="localhost", port=16379,
decode_responses=True, encoding="utf-8")
```

```
num_worker = redis_cli.get('num_workers')
redis_cli.set('num_workers', str(int(num_worker)+1))

pubsub_dask = redis_cli.pubsub()
pubsub_dask.subscribe('worker'+num_worker)

pubsub_restructure_nodes = redis_cli.pubsub()
pubsub_restructure_nodes.subscribe('restructure_nodes'+num_worker)

worker = daskFunctions.DaskFunctions()

try:
      print('Use control + c to exit the Worker node.')
      while True:



            name_file =
            pubsub_dask.get_message(ignore_subscribe_messages=True)
            if name_file and (name_file.get('type') == "message"):
                  redis_cli.publish(name_file.get('data'),
                  worker.readCSV(name_file.get('data')))
                  redis_cli.publish(name_file.get('data'),
                  worker.columns())
                  capturate = False
                  while not capturate:
                        num_rows =
                        pubsub_dask.get_message(ignore_subscribe_messag
                        es=True)
                        if num_rows and (num_rows.get('type') ==
                        "message"):
                              redis_cli.publish(name_file.get('data'),
                              worker.head(int(num_rows.get('data'))))
                              capturate = True
                        else:
                              time.sleep(0.1)
                  capturate = False
```

```
        while not capturate:
            field_element =
            pubsub_dask.get_message(ignore_subscribe_messag
            es=True)
            if field_element and (field_element.get('type')
            == "message"):
                redis_cli.publish(name_file.get('data'),
                worker.isin(field_element.get('data').spl
                it(':')[0],
                field_element.get('data').split(':')[1]))
                capturate = True
            else:
                time.sleep(0.1)
        capturate = False
        while not capturate:
            row_column =
            pubsub_dask.get_message(ignore_subscribe_messag
            es=True)
            if row_column and (row_column.get('type') ==
            "message"):
                redis_cli.publish(name_file.get('data'),
                worker.item(int(row_column.get('data').sp
                lit(':')[0]),
                int(row_column.get('data').split(':')[1])
                ))
                capturate = True
            else:
                time.sleep(0.1)
        redis_cli.publish(name_file.get('data'),
        worker.max('Temp_max'))
        redis_cli.publish(name_file.get('data'),
        worker.min('Temp_min'))
message_restructure =
pubsub_restructure_nodes.get_message(ignore_subscribe_messa
ges=True)
if message_restructure and (message_restructure.get('type')
== "message"):
        if int(message_restructure.get('data')) <
        int(num_worker):
            pubsub_dask.unsubscribe('worker'+num_worker)
            pubsub_restructure_nodes.unsubscribe('restructu
            re_nodes'+num_worker)
```

```
                        num_worker=str(int(num_worker)-1)
                        pubsub_dask.subscribe('worker'+num_worker)

                        pubsub_restructure_nodes.subscribe('restructure
                        _nodes'+num_worker)
            time.sleep(0.1)
except KeyboardInterrupt:
      print('Exiting worker node.')
      pubsub_dask.unsubscribe('worker'+num_worker)
      pubsub_restructure_nodes.unsubscribe('worker'+num_worker)
      i=0
      while i<int(redis_cli.get('num_workers')):
        if i != num_worker:
            redis_cli.publish('restructure_nodes'+str(i),
            str(num_worker))
        i+=1
      redis_cli.set('num_workers',
      str(int(redis_cli.get('num_workers'))-1))
```

Client.py

```
import redis
import sys
import time

redis_cli = redis.Redis(host="localhost", port=16379,
decode_responses=True, encoding="utf-8")
pubsub = redis_cli.pubsub()

maxs=[]
mins=[]
i=1

while i<len(sys.argv):

redis_cli.publish('worker'+str(i%int(redis_cli.get('num_workers'))),
sys.argv[i])
```

```
pubsub.subscribe(sys.argv[i])

capturate=False

while not capturate:

      message =
      pubsub.get_message(ignore_subscribe_messages=True)

      if message and (message.get('type') == 'message'):

            print(message.get('data')+"\n")

            while not capturate:

                  columns =
                  pubsub.get_message(ignore_subscribe_messages=Tr
                  ue)

                  if columns and (columns.get('type') ==
                  "message"):

                        print(columns.get('data')+"\n")

                        capturate = True

                  else:

                        time.sleep(0.1)


            redis_cli.publish('worker'+str(i%int(redis_cli.get('n
            um_workers'))), str(5))

            capturate = False

            while not capturate:

                  head =
                  pubsub.get_message(ignore_subscribe_messages=Tr
                  ue)

                  if head and (head.get('type') == "message"):

                        print(head.get('data')+"\n")

                        capturate = True

                  else:

                        time.sleep(0.1)


            redis_cli.publish('worker'+str(i%int(redis_cli.get('n
            um_workers'))), 'City'+':'+'Tarragona')

            capturate = False

            while not capturate:

                  isin =
                  pubsub.get_message(ignore_subscribe_messages=Tr
                  ue)

                  if isin and (isin.get('type') == "message"):
```

```
            print(isin.get('data')+"\n")
            capturate = True
      else:
            time.sleep(0.1)

redis_cli.publish('worker'+str(i%int(redis_cli.get('n
um_workers'))), str(5)+':'+str(3))
capturate = False
while not capturate:
      item =
      pubsub.get_message(ignore_subscribe_messages=Tr
      ue)
      if item and (item.get('type') == "message"):
            print(item.get('data')+"\n")
            capturate = True
      else:
            time.sleep(0.1)
capturate = False
while not capturate:
      maxm =
      pubsub.get_message(ignore_subscribe_messages=Tr
      ue)
      if maxm and (maxm.get('type') == "message"):
            maxs.append(float(maxm.get('data')))
            capturate = True
      else:
            time.sleep(0.1)
capturate = False
while not capturate:
      minm =
      pubsub.get_message(ignore_subscribe_messages=Tr
      ue)
      if minm and (minm.get('type') == "message"):
            mins.append(float(minm.get('data')))
            capturate = True
      else:
            time.sleep(0.1)
pubsub.unsubscribe(sys.argv[i])
```

```
            else:
                  time.sleep(0.1)
      i+=1

print("Temperatura maxima: "+str(max(maxs)))
print("Temperatura minima: "+str(min(mins)))
```

Arquitectura distribuida con el middleware de comunicación indirecta RabbitMQ

Servidor RabbitMQ (master.sh)

```
#!/bin/bash

docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672
rabbitmq:3.10-management
```

Publisher.py

```
import pika

class Publisher:
      def __init__(self, config):
            self.config = config
            self.connection = self.create_connection()

      def __del__(self):
            self.connection.close()

      def create_connection(self):
            return
            pika.BlockingConnection(pika.ConnectionParameters(host=self
            .config['host'], port=self.config['port']))

      def publish(self, exchange, routing_key, message):
            channel = self.connection.channel()
```

```
            channel.exchange_declare(exchange=exchange,
            exchange_type='direct')

            channel.basic_publish(exchange=exchange,
            routing_key=routing_key, body=message)
```

Consumer.py

```
import pika

import abc


class Consumer(abc.ABC):

      def __init__(self, config, exchange_name, binding_key):

            self.config = config

            self.exchange_name = exchange_name

            self.binding_key = binding_key

            self.connection = self.create_connection()


      def __del__(self):

            self.connection.close()


      def create_connection(self):

            return
            pika.BlockingConnection(pika.ConnectionParameters(host=self
            .config['host'], port=self.config['port']))


      @abc.abstractmethod

      def callback(self, channel, method, properties, body):

            pass


      def consume(self):

            channel = self.connection.channel()

            channel.exchange_declare(exchange=self.exchange_name,
            exchange_type='direct')

            result = channel.queue_declare(queue='', exclusive=True)

            queue_name =
            result.method.queuechannel.queue_bind(exchange=self.exchang
            e_name, queue=queue_name, routing_key=self.binding_key)
```

```
            channel.basic_consume(queue=queue_name,
            on_message_callback=self.callback, auto_ack=True)


            try:
                  print(' [*] Waiting for data. Use control + c to
                  exit.\n')
                  channel.start_consuming()
            except KeyboardInterrupt:
                  print(' [*] Exiting...\n')
                  channel.stop_consuming()
```

Worker.py

```
import daskFunctions
import publisher
import consumer
import sys


class ConsumerWorker(consumer.Consumer):
      def callback(self, channel, method, properties, body):
            print(" [x] Received new message %r" % (body))


            publisher_file = publisher.Publisher({'host': 'localhost',
            'port': 5672})
            worker = daskFunctions.DaskFunctions()


            publisher_file.publish('client', 'proves',
            worker.readCSV(body.decode().split(':')[0])+':'+worker.colu
            mns()+':'+worker.head(int(body.decode().split(':')[1]))+':'
            +worker.isin(body.decode().split(':')[2],
            body.decode().split(':')[3])+':'+worker.item(int(body.decod
            e().split(':')[4]),
            int(body.decode().split(':')[5]))+':'+worker.max('Temp_max'
            )+':'+worker.min('Temp_min'))


consumer_name = ConsumerWorker({'host':'localhost', 'port':5672},
'workers', 'worker'+sys.argv[1])
consumer_name.consume()
```

Client.py

```
import publisher

import consumer

import sys


class ConsumerClient(consumer.Consumer):

      def callback(self, channel, method, properties, body):

            print(" [x] Received new message.\n")

            print(body.decode().split(':')[0]+'\n')

            print(body.decode().split(':')[1]+'\n')

            print(body.decode().split(':')[2]+'\n')

            print(body.decode().split(':')[3]+'\n')

            print(body.decode().split(':')[4]+'\n')

            maxs.append(float(body.decode().split(':')[5]))

            mins.append(float(body.decode().split(':')[6]))


maxs=[]

mins=[]


publisher_name = publisher.Publisher({'host': 'localhost', 'port':
5672})


i=1

while i<len(sys.argv):

      publisher_name.publish('workers', 'worker'+str(i),
      sys.argv[i]+':'+str(5)+':'+'City'+':'+'Tarragona'+':'+str(5)+':'
      +str(3))

      i+=1


consumer_file = ConsumerClient({'host':'localhost', 'port':5672},
'client', 'proves')

consumer_file.consume()


print("Temperatura maxima: "+str(max(maxs)))

print("Temperatura minima: "+str(min(mins)))
```

Juegos de pruebas

Las pruebas son las mismas para todas las arquitecturas, aunque en la primera, la cual evalúa un único fichero csv como parámetro de entrada, se ha usado un fichero csv diferente para cada arquitectura. De este modo, se puede conocer el resultado que tendría que dar el programa para cada fichero csv. La salida incorporada en los juegos de pruebas es únicamente el resultado del cálculo del máximo y el mínimo de todos los ficheros csv pasados por parámetro. No obstante, en la ejecución real se muestran los resultados de otras funciones de la API de pandas, aunque no todas, como es el caso de la función apply(), cuyo correcto funcionamiento no se ha probado a pesar de que aparezca en la clase DaskFunctions de la mayor parte de las arquitecturas distribuidas.

Arquitectura distribuida con el middleware de comunicación directa XMLRPC (Iterativa)

Prueba	Descripción	Salida	¿Correcto?
1	Se crea un nodo *worker*. El cliente sólo recibe el fichero 'provas1.csv' como parámetro de entrada.	Temperatura máxima: 32.72. Temperatura mínima: 19.83.	Sí.
2	Se crea un nodo *worker*. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54 Temperatura mínima: 17.02.	Sí.
3	Se crean tres nodos *workers*. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
4	Se destruye el primer nodo *worker* creado de manera que quedan sólo dos nodos *workers* en ejecución. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
5	Se destruye el segundo nodo worker creado y se vuelve a arrancar el primero. En total hay dos nodos workers en ejecución. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
6	Pruebas adicionales.	Varias salidas diferentes.	Sí.

Arquitectura distribuida con el middleware de comunicación directa XMLRPC (Paralelizada)

Prueba	Descripción	Salida	¿Correcto?
1	Se crea un nodo *worker*. El cliente sólo recibe el fichero 'provas1.csv' como parámetro de entrada.	Temperatura máxima: 32.72. Temperatura mínima: 19.83.	Sí.
2	Se crea un nodo *worker*. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
3	Se crean tres nodos *workers*. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
4	Se destruye el primer nodo *worker* creado de manera que quedan sólo dos nodos *workers* en ejecución. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
5	Se destruye el segundo nodo worker creado y se vuelve a arrancar el primero. En total hay dos nodos workers en ejecución. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
6	Pruebas adicionales.	Varias salidas diferentes.	Sí.

Arquitectura distribuida con los middlewares de comunicación directa XMLRPC y gRPC (Iterativa)

Prueba	Descripción	Salida	¿Correcto?
1	Se crea un nodo *worker*. El cliente sólo recibe el fichero 'provas1.csv' como parámetro de entrada.	Temperatura máxima: 32.72. Temperatura mínima: 19.83.	Sí.
2	Se crea un nodo *worker*. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
3	Se crean tres nodos *workers*.	Temperatura máxima: 33.54.	Sí.

	El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura mínima: 17.02.	
4	Se destruye el primer nodo *worker* creado de manera que quedan sólo dos nodos *workers* en ejecución. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
5	Se destruye el segundo nodo worker creado y se vuelve a arrancar el primero. En total hay dos nodos workers en ejecución. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
6	Pruebas adicionales.	Varias salidas diferentes.	Sí.

Arquitectura distribuida con los middlewares de comunicación directa XMLRPC y gRPC (Paralelizada)

Prueba	Descripción	Salida	¿Correcto?
1	Se crea un nodo *worker*. El cliente sólo recibe el fichero 'provas1.csv' como parámetro de entrada.	Temperatura máxima: 32.72. Temperatura mínima: 19.83.	Sí.
2	Se crea un nodo *worker*. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
3	Se crean tres nodos *workers*. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
4	Se destruye el primer nodo *worker* creado de manera que quedan sólo dos nodos *workers* en ejecución. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
5	Se destruye el segundo nodo worker creado y se vuelve a arrancar el primero. En total hay dos nodos workers en ejecución. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
6	Pruebas adicionales.	Varias salidas diferentes.	Sí.

Arquitectura híbrida con el middleware de comunicación directa XMLRPC y el middleware de comunicación indirecta Redis (Iterativa)

Prueba	Descripción	Salida	¿Correcto?
1	Se crea un nodo *worker*. El cliente sólo recibe el fichero 'provas2.csv' como parámetro de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 18.7.	Sí.
2	Se crea un nodo *worker*. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
3	Se crean tres nodos *workers*. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
4	Se destruye el primer nodo *worker* creado de manera que quedan sólo dos nodos *workers* en ejecución. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
5	Se destruye el segundo nodo worker creado y se vuelve a arrancar el primero. En total hay dos nodos workers en ejecución. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
6	Pruebas adicionales.	Varias salidas diferentes.	Sí.

Arquitectura híbrida con el middleware de comunicación directa XMLRPC y el middleware de comunicación indirecta Redis (Paralelizada)

Prueba	Descripción	Salida	¿Correcto?
1	Se crea un nodo *worker*. El cliente sólo recibe el fichero 'provas2.csv' como parámetro de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 18.7.	Sí.
2	Se crea un nodo *worker*. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
3	Se crean tres nodos *workers*. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54.	Sí.

		Temperatura mínima: 17.02.	
4	Se destruye el primer nodo *worker* creado de manera que quedan sólo dos nodos *workers* en ejecución. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
5	Se destruye el segundo nodo worker creado y se vuelve a arrancar el primero. En total hay dos nodos workers en ejecución. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
6	Pruebas adicionales.	Varias salidas diferentes.	Sí.

Arquitectura distribuida con el middleware de comunicación indirecta Redis

Prueba	Descripción	Salida	¿Correcto?
1	Se crea un nodo *worker*. El cliente sólo recibe el fichero 'provas3.csv' como parámetro de entrada.	Temperatura máxima: 31.01. Temperatura mínima: 17.02.	Sí.
2	Se crea un nodo *worker*. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
3	Se crean tres nodos *workers*. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
4	Se destruye el primer nodo *worker* creado de manera que quedan sólo dos nodos *workers* en ejecución. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
5	Se destruye el segundo nodo worker creado y se vuelve a arrancar el primero. En total hay dos nodos workers en ejecución. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.
6	Pruebas adicionales.	Varias salidas diferentes.	Sí.

Arquitectura distribuida con el middleware de comunicación indirecta RabbitMQ

Prueba	Descripción	Salida	¿Correcto?
1	Se crea un nodo *worker*. El cliente sólo recibe el fichero 'provas3.csv' como parámetro de entrada.	Temperatura máxima: 31.01. Temperatura mínima: 17.02.	Sí.
2	Se crean dos nodos *worker*. El cliente recibe los ficheros 'provas1.csv' y 'provas2.csv' como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 18.7.	Sí.
3	Se crean tres nodos *worker*. El cliente recibe los ficheros 'provas1.csv' y 'provas3.csv' como parámetros de entrada.	Temperatura máxima: 32.72. Temperatura mínima: 17.02.	Sí.
4	Se crean tres nodos *workers*. El cliente recibe los tres ficheros csv como parámetros de entrada.	Temperatura máxima: 33.54. Temperatura mínima: 17.02.	Sí.

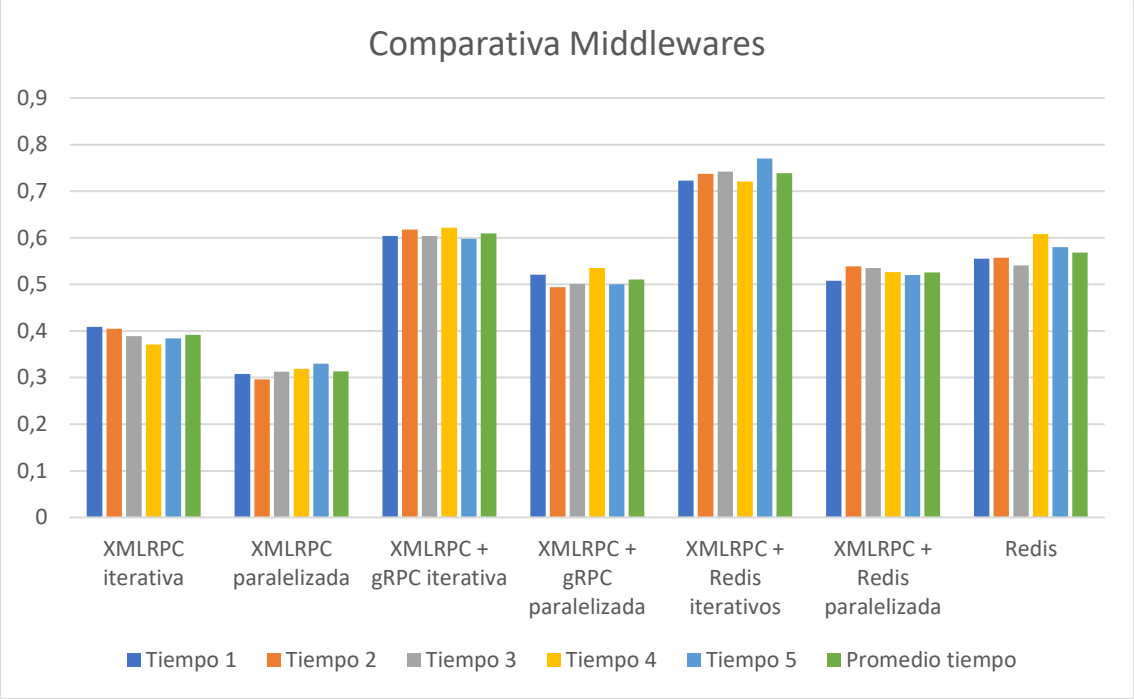
Gráfico comparativo con los tiempos de ejecución de cada middleware y discusión de los resultados

Las medidas del tiempo de ejecución se han obtenido para una versión del programa que utiliza tres nodos worker, a la que se le pasan tres ficheros csv y en la que el cliente únicamente aplica las funciones read_csv(), max() y min() sobre estos ficheros. Esto es debido a que estas son las únicas funciones que se han implementado en la arquitectura distribuida que incorpora el middleware de comunicación directa gRPC. A continuación, se muestra una tabla con las muestras tomadas para cada arquitectura distribuida y el promedio de éstas.

	Tiempo 1	Tiempo 2	Tiempo 3	Tiempo 4	Tiempo 5	Promedio tiempo
XMLRPC iterativa	0,409	0,405	0,389	0,371	0,384	**0,3916**
XMLRPC paralelizada	0,308	0,296	0,313	0,319	0,33	**0,3132**
XMLRPC + gRPC iterativa	0,604	0,618	0,604	0,622	0,598	**0,6092**
XMLRPC + gRPC paralelizada	0,521	0,494	0,501	0,535	0,5	**0,5102**

XMLRPC + Redis iterativos	0,723	0,737	0,742	0,721	0,77	**0,7386**
XMLRPC + Redis paralelizada	0,508	0,539	0,535	0,526	0,52	**0,5256**
Redis	0,555	0,557	0,541	0,608	0,58	**0,5682**

A partir de los datos de la tabla se obtiene el siguiente gráfico de columnas.



En el gráfico se puede apreciar que algunas de las previsiones se han cumplido. Para empezar, en las arquitecturas que utilizan *middlewares* de comunicación directa, vemos que, si el tratamiento de los ficheros se realiza de forma paralela se reduce significativamente el tiempo de ejecución con respecto al tratamiento de estos mismos ficheros de forma iterativa. Si se aumentase la cantidad de datos, por ejemplo, aumentando el número de ficheros csv o usando ficheros csv de tamaño gigantesco, se observaría una disminución considerable en el tiempo de ejecución de las arquitecturas paralelas con respecto a las arquitecturas iterativas. No se han hecho estas pruebas porque se considera que los resultados obtenidos ya prueban la mejora que supone el uso de paralelismo.

También se observa que Redis es el middleware de comunicación más lento, tal y como se ha explicado en la teoría.

Por otro lado, también se pueden observar algún resultado inesperado. En concreto, se observa que la arquitectura que combina los *middlewares* XMLRPC y gRPC es mucho más lenta que la que usa únicamente XMLRPC. Esto no debería ser así, ya que

gRPC es mucho más rápido que cualquiera de los otros *middlewares* utilizados. En un principio, pensé que estos resultados podían ser debidos a que la implementación de gRPC requiere de una mayor cantidad de código de programación y que, al ser la cantidad de datos tratada tan pequeña, la velocidad de ejecución de las funciones podía no lograr compensar el sobrecoste de todo este código adicional. No obstante, en las sesiones de teoría se ha explicado que el middleware que debería tener un sobrecoste en ese aspecto es XMLRPC ya que en gRPC el proceso de establecer la conexión únicamente consiste en abrir un socket. Atendiendo a esto, sólo puedo decir que no he conseguido hallar una explicación razonable para que el tiempo de ejecución de la arquitectura que utiliza el middleware gRPC sea superior a la que usa el middleware XMLRPC.

No se ha podido calcular el tiempo de ejecución de la arquitectura distribuida que usa el middleware de comunicación indirecta RabbitMQ ya que, como se ha mencionado en apartados anteriores, los nodos que adoptan el rol de consumidores se bloquean a la espera de un mensaje hasta que el usuario cliente pulse las teclas CTRL + C, por lo que el tiempo de finalización del programa no se correspondería con el tiempo de ejecución real de los métodos de la API de RabbitMQ. Según lo explicado en clase, este middleware debería tener un tiempo de ejecución inferior al de XMLRPC o Redis, pero superior al de gRPC. Se puede apreciar a ojo que la arquitectura que usa este middleware de comunicación indirecta es más rápida que la que usa Redis.

Lectura de un artículo y descripción de la solución desarrollada

El artículo empieza exponiendo las dificultades con las que tienen que lidiar algunos usuarios de los servicios del *cloud* debido al hecho de que los nodos que adoptan el rol de servidores generan un cuello de botella que evita que el clúster de dichos usuarios escale con facilidad.

Para lidiar con esta problemática, los autores del artículo presentan un prototipo de modelo de arquitectura distribuida llamado **PyWren**, cuyo código ha sido redactado en el lenguaje de programación Python y que se ejecuta sobre la plataforma AWS Lambda (proveedor de servicios del *cloud*). Este prototipo tiene un modelo de programación **Map-Reduce** y presenta dos características que permiten reducir los cuellos de botella de los servicios *cloud*.

La primera es el uso de funciones ***stateless***. Cuando este tipo de funciones terminan su ejecución, no almacenan ni los datos de entrada ni el resultado de la salida para que puedan ser usadas en llamadas posteriores a la función. No obstante, si las funciones no guardan el estado, estos datos se tendrán que almacenar de otra forma, para lo que se propone el uso de una **memoria de datos compartida distribuida**. Esta memoria de datos se implementa haciendo uso de un middleware de comunicación indirecta basado en un espacio de datos compartido, en concreto, **ObjectStorage** o **Redis**. Los datos se

almacenan de forma permanente en SSDs distribuidos porque, de este modo, su acceso será más rápido.

Por otro lado, la arquitectura del prototipo es ***serverless***. Esto no significa que el clúster carezca de servidores como me ha parecido entender inicialmente, sino que la arquitectura está conformada de tal forma que se libera al usuario que adopta el rol de cliente del manejo de los servidores. Aunque no se expone en el artículo, está implícito que, si la arquitectura sigue este paradigma, la comunicación se hará siguiendo el paradigma **pull**, en el que es el cliente el que pregunta periódicamente si la información está disponible.

Estas dos características permiten que el prototipo sea más rápido y que a la vez mantenga una alta **elasticidad**, que es un requisito indispensable ya que este recurso ha de ser accesible para un amplio rango de usuarios. La elasticidad es la capacidad de adaptarse dinámicamente a la demanda o, dicho de otro modo, la capacidad de escalar (aumentar el número de recursos cuando aumenta la demanda de estos) al alza o a la baja. Por esta misma razón es de suponer que la comunicación será **asíncrona** y ***transient***, aunque no se especifica en el artículo.

Además, éste prototipo también conserva la **simplicidad** y la **tolerancia a fallos** que son indispensables en los proveedores de servicio en el *cloud*.

Para terminar, debo decir que me ha resultado confuso tratar de comprender qué tipo de *middleware* de comunicación se utiliza. Esto es debido a que, en varios puntos del artículo, se habla de la ejecución de las funciones como si estas fuesen invocadas por el cliente y se ejecutasen en el servidor, algo que es propio de las RPCs (*Remote Producere Call*) que es un tipo de *middleware* de **comunicación directa**. Esto suena lógico atendiendo a que el *middleware* de comunicación usado por los proveedores de servicios *cloud* es **REST**, una simplificación de las RPCs que actúa a tan bajo nivel que apenas se considera un *middleware*. Por otro lado, no parece que los nodos conozcan la ubicación exacta de los demás nodos, cosa que sería característico de la **comunicación indirecta** (a pesar de que en otro momento se menciona que dichos nodos no están completamente desacoplados). Además, como se ha comentado en párrafos anteriores, se emplea **Redis**, probablemente el *middleware* de comunicación indirecta basado en espacio de datos compartido distribuido por excelencia. Se concluye, por lo tanto, que el *middleware* usado por el prototipo tiene características de ambos tipos de comunicación.

Fuentes documentales

- https://docs.python.org/es/3.9/library/xmlrpc.server.html
- https://medium.com/engineering-semantics3/a-simplified-guide-to-grpc-in-python-6c4e25f0c506
- https://lynn-kwong.medium.com/essentials-of-redis-all-you-need-to-know-as-a-developer-73da5d2fdc0b

- https://betterprogramming.pub/how-to-use-redis-for-caching-and-pub-sub-in-python-3851174f9fb0
- https://realpython.com/python-redis/#redis-as-a-python-dictionary
- https://www.rabbitmq.com/getstarted.html
- https://medium.com/@rahulsamant_2674/python-rabbitmq-8c1c3b79ab3d
- https://medium.com/analytics-vidhya/how-to-use-rabbitmq-with-python-e0ccfe7fa959
- https://www.geeksforgeeks.org/abstract-classes-in-python/
- https://www.tutorialspoint.com/abstract-base-classes-in-python-abc