UNIVERSITAT ROVIRA I VIRGILI

Tarea 1: Modelos de comunicación y *middleware*

Arey Ferrero Ramos

21 de julio del 2022

Índice

Especificaciones 4

Diseño 4

Diagramas 4

Arquitectura distribuida con el middleware de comunicación directa XMLRPC 4

.....

.....

.....

.....

.....

Arquitectura distribuida con los middlewares de comunicación directa XMLRPC y gRPC ... 5

.....

.....

.....

.....

.....

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 6

.....

.....

.....

.....

Comparativa de arquitecturas de comunicación directa e indirecta 6

Implementación 8

Arquitectura distribuida con el middleware de comunicación directa XMLRPC 8

Master.py 8

DaskFunctions.py 9

Worker.py 10

Client.py 11

Arquitectura distribuida con los middlewares de comunicación directa XMLRPC y gRPC 12
Master.py 12
DaskFunctions.py 13
DaskFunctions.proto 13
Worker.py 14
Client.py 16
Arquitectura híbrida con el middleware de comunicación directa XMLRPC y el middleware de comunicación indirecta Redis 17
Servidor Redis (master.sh) 17
DaskFunctions.py 17
Worker.py 18
Client.py 19
Arquitectura distribuida con el middleware de comunicación indirecta Redis 19
Servidor Redis (master.sh) 19
DefineNumWorkers.py 20
DaskFunctions.py 20
Worker.py 21
Client.py 24
Juego de pruebas 26
Arquitectura distribuida con el middleware de comunicación directa XMLRPC 27
Arquitectura distribuida con los middlewares de comunicación directa XMLRPC y gRPC 27
Arquitectura híbrida con el middleware de comunicación directa XMLRPC y el middleware de comunicación indirecta Redis 28
Arquitectura distribuida con el middleware de comunicación indirecta Redis 29
Lectura de un artículo y análisis de la solución desarrollada 29
Fuentes documentales 30

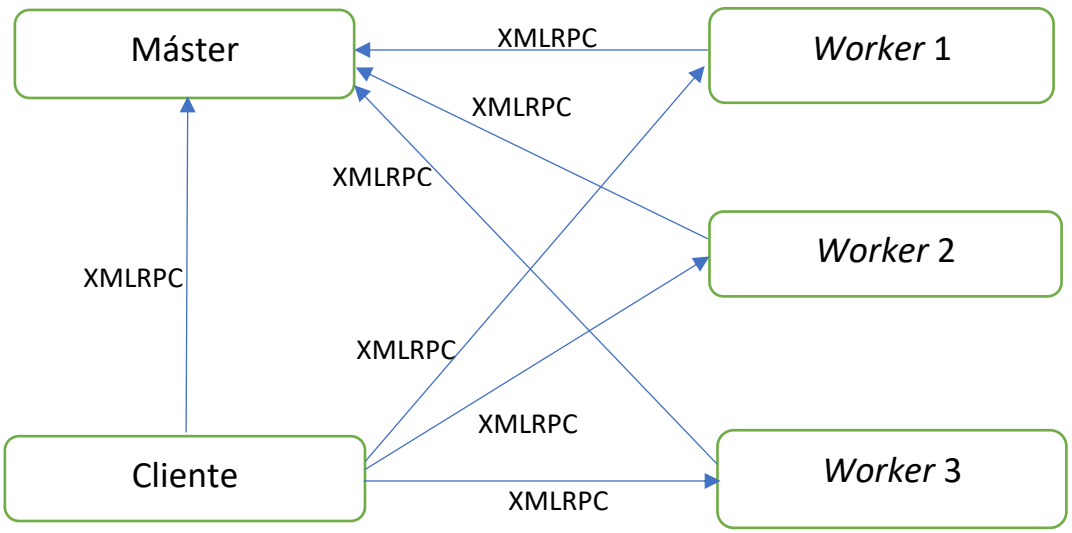
Especificaciones

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.

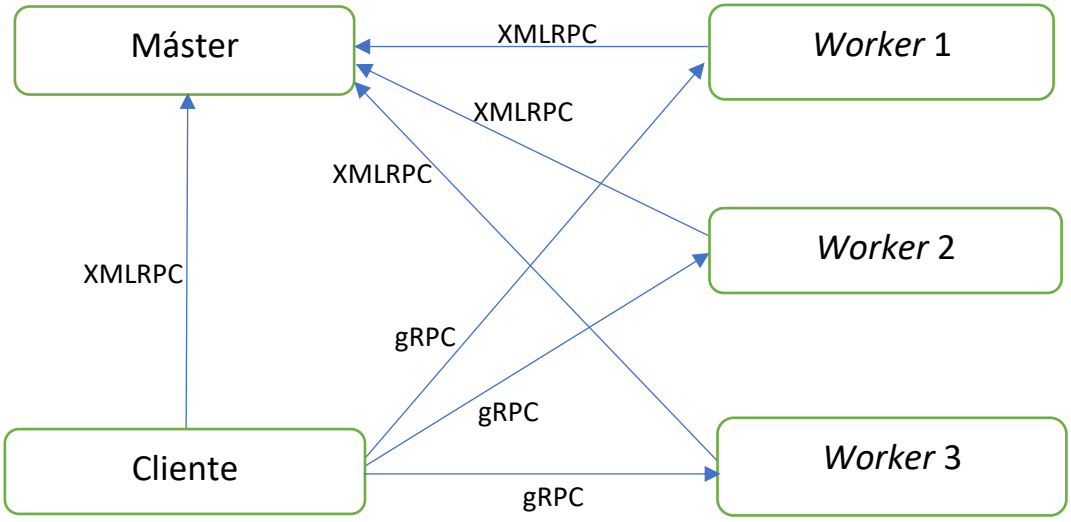
Diseño

Diagramas

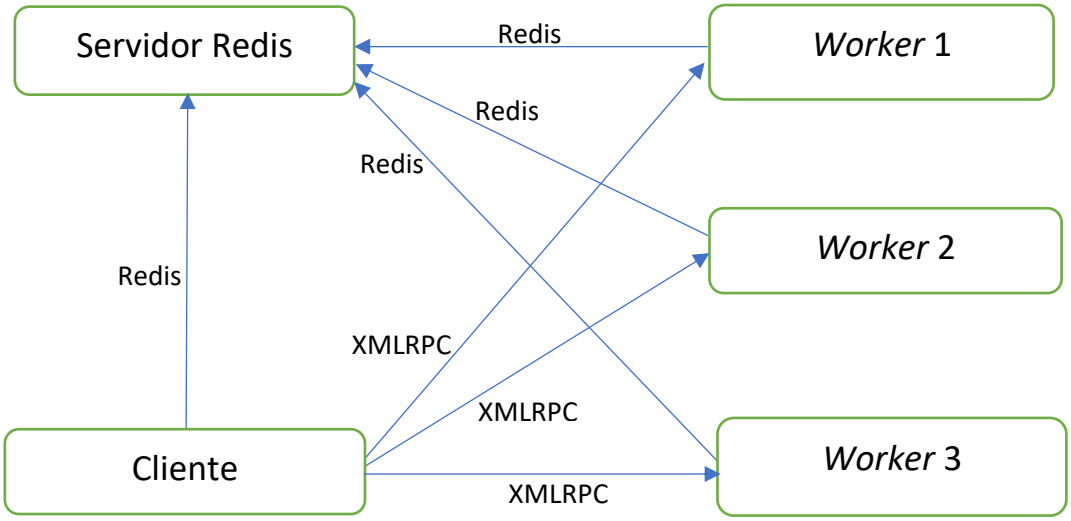
Arquitectura distribuida con el middleware de comunicación directa XMLRPC



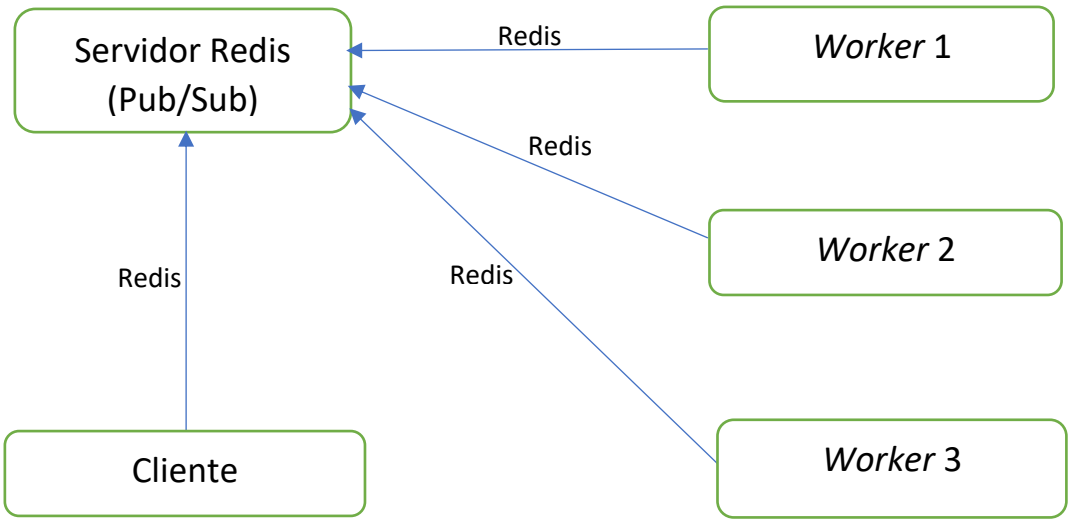
Arquitectura distribuida con los middlewares de comunicación directa XMLRPC y gRPC



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



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.

Por último, se ha implementado una arquitectura que utiliza exclusivamente el middleware de comunicación indirecta Redis. Esta arquitectura se ha desarrollado siguiendo el paradigma publish/subscribe por lo que es completamente diferente de las anteriores. Igual que en la arquitectura híbrida, no hay un nodo máster como tal, sino que se utiliza un servidor Redis al que se conectan 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. 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 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.

Implementación

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')
```

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())
```

Worker.py

```
import xmlrpc.client
import sys
import os

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

```
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)))
```

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

```
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)))
```

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
```

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())
```

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

```
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)))
```

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))
```

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())
```

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)))
```

Juego 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.

Arquitectura distribuida con el middleware de comunicación directa XMLRPC

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

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

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 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.

Lectura de un artículo y análisis de la solución desarrollada

Uno de los objetivos de los desarrolladores de la solución propuesta en el artículo era tener un modelo que tuviese una alta **elasticidad** y que fuese accesible para un rango de usuarios más amplio que el que normalmente tiene acceso a este recurso. 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 ello, tras la lectura del artículo, sé interpreta la siguiente información:

- La comunicación es **indirecta**.
- La arquitectura distribuida utilizada es una **arquitectura distribuida basada en un espacio de datos compartido**.

- El middleware de comunicación utilizado es ***stateless***. Además, en basa al objetivo de los desarrolladores, se puede suponer que es **asíncrono**, ***pull*** y ***transistent***.
- El modelo tiene **tolerancia a fallos**.
- El modelo es **sencillo**.

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