UNIVERSITAT ROVIRA I VIRGILI

Tarea 2: Arquitectura distribuida

Arey Ferrero Ramos

2 de enero del 2023

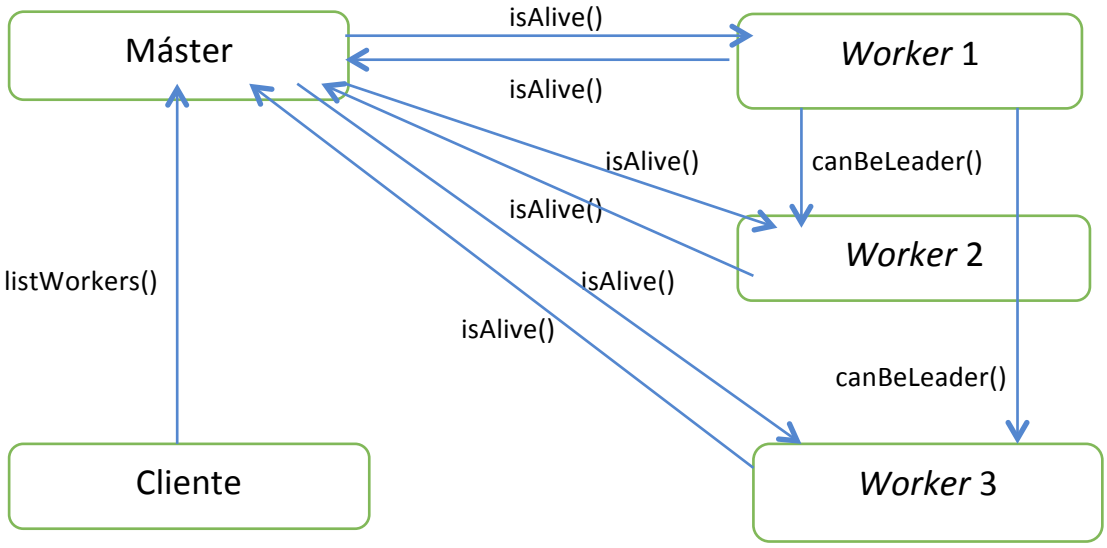
Índice

Especificaciones

Se pide complementar la arquitectura distribuida de la tarea 1 que usa el middleware de comunicación directa XMLRPC con dos características adicionales: Consistencia y tolerancia a fallos. Para tener consistencia se debe implementar un sistema dinámico que permita al cliente ser consciente, en tiempo real, de todos los cambios que se efectúen sobre la lista de workers (añadir worker, eliminar worker, …). Para tener tolerancia a fallos se deben implementar detectores de fallos. Uno de estos, que estará localizado en el master, deberá detectar si alguno de los nodos worker ha caído y, en caso afirmativo, el nodo máster lo eliminará de la lista de workers. El otro, que estará localizado en los nodos worker, deberá detectar si el nodo máster ha caído y, en caso afirmativo, mediante un proceso de elección de líder (que requerirá comunicación entre los nodos worker), uno de los nodos worker se convertirá en el nuevo nodo máster.

Diseño

Diagrama



Decisiones de diseño

La solución para conseguir consistencia ha sido trivial y ha consistido en que el proxy (cliente del servidor) del nodo máster que es creado por el cliente de la arquitectura invoca la lista de workers cada vez que quiera obtener un nuevo nodo worker. Si un nodo worker es eliminado de la lista, toda la lista de workers se actualiza automáticamente de manera que cuando el cliente acceda a la siguiente posición, accederá a la posición correcta. Cabe destacar que dicha solución no se ha tenido que implementar en el desarrollo de esta fase de la práctica, sino que ya se había implementado para la tarea 1.

En el caso de la tolerancia a fallos, las soluciones han requerido de más complejidad. El primer problema con el que he tenido que lidiar ha sido que, a diferencia de en la primera tarea, los roles que adoptan los nodos en la arquitectura no son fijos. En la primera tarea, el nodo lanzado como máster siempre sería el máster y un nodo lanzado como worker siempre sería un worker. En esta segunda tarea, todos los nodos worker pueden convertirse en un master, por lo que todos los nodos han de tener almacenado un objeto que permite a los clientes que se conecten a los servidores de dichos nodos acceder tanto a las funciones de administración de la lista de workers (almacenadas en el máster en la tarea 1) como a las funciones de tratamiento de los ficheros csv (almacenadas en los workers en la tarea 1). Dado que no se pueden almacenar objetos en un servidor una vez éste ya ha sido lanzado, se han agrupado las dos clases que contenían cada grupo de funciones en una única clase denominada 'ServerFunctions', de la cual se crea un objeto que será enviado al servidor antes de que sea lanzado. Aunque probablemente no sea una solución elegante, es efectiva en la medida en la que permite a los clientes del servidor de un nodo worker cambiar dinámicamente el tipo de funciones a las que accede en caso de que dicho nodo se convierta en un máster.

El segundo problema con el que he tenido que lidiar es que hay elementos que forman parte de la arquitectura, como la URL del máster o la lista de workers, cuyo contenido cambiará durante la ejecución del programa y, a su vez, han de ser accesibles dinámicamente por todos los nodos y por el cliente. En condiciones normales, este problema se solucionaría con variables globales o, si el diseño tuviese que ser más robusto, usando un patrón de diseño Singleton. No obstante, en este caso se utiliza una arquitectura distribuida, por lo que cada nodo es lanzado desde un terminal diferente. Esto implica que cada nodo tendrá su propio intérprete de Python con su propia sección de memoria privada, cosa que elimina la posibilidad de tener una variable global o un objeto Singleton compartidos entre los distintos nodos. La primera solución que se me ha ocurrido al problema inicial es almacenar la URL del máster como una variable de Redis y la lista de workers como una lista de Redis. Tratándose Redis de un motor de almacenamiento externo que permite compartir la información entre todos los nodos que han establecido la misma conexión, se soluciona cualquiera de los problemas antes mencionados. No obstante, aunque esta

implementación se llevó a cabo de forma exitosa, tiene el inconveniente de que, si cayese el servidor Redis, caería toda la arquitectura. Por lo tanto, la arquitectura desarrollada sería centralizada y, aunque se podría argumentar que la dependencia se establece con respecto a un servidor externo que proporciona ciertas garantías, es más interesante desarrollar una arquitectura completamente distribuida.

Por ello, la solución final que se ha propuesto para la lista de nodos workers es que sea una lista de Python estándar y que el detector de fallos de cada nodo worker se descargue dicha lista cada vez que hace la comprobación de que el servidor master siga funcionando correctamente. De este modo, si el nodo máster cae, se tiene una lista de workers actualizada y descargada localmente cuyo contenido se copiará en la lista de workers del nodo worker que se convierta en máster la cual, hasta ese momento, estaba vacía. Por otro lado, cada nodo worker tendrá una copia de la URL del máster y, cada vez que un nodo worker se convierta en máster, éste deberá actualizar dicha variable en todos los workers. Esto último tiene el inconveniente de que, cada vez que se lance un nodo o se ejecute el cliente, uno de los parámetros que se tendrá que pasar será el puerto del nodo máster, cosa que obliga al usuario de esta arquitectura a conocer cómo se produce el proceso de elección de líder para así conocer cuál es la URL del máster en todo momento.

Conceptualmente, los detectores de fallos se basan en hacer *pings* o *keep alives* al nodo cuyos fallos se quieren detectar. Una de las dificultades que me he encontrado en el desarrollo de la práctica ha sido suponer que dichos *pings* o *keep alives* debían ser funciones de Python que hacían literalmente *ping* o *keep alive* a un nodo de la arquitectura. No obstante, todos los nodos de la arquitectura comparten la misma dirección IP (“localhost”) y, por lo tanto, se tendría que especificar el puerto para distinguir dichos nodos, cosa que es imposible en las funciones de Python para hacer ping. Para poder diferenciar nodos mediante el puerto se tendrían que utilizar *raw sockets*, algo que no es necesario ni recomendable para esta práctica. Por ello, se han diseñado un mecanismo que emula el funcionamiento de un *ping* o de un *keep alive*. Este mecanismo parte de una estructura ‘try: except:’. En la sección ‘try:’ se creará un cliente del nodo cuyos fallos quieren ser detectados y este cliente invocará a una función llamada is_alive() (la cual no hace nada aparte de devolver True), de modo que, si el nodo ha caído, no se podrá invocar esta función y saltará la excepción. En la sección ‘except:’ se llevará a cabo todo el proceso para retirar el nodo caído del sistema. Esto tiene mucho sentido si se tiene en cuenta que internamente los *pings* funcionan así.

Por último, considero conveniente explicar cómo funciona el proceso de elección de líder. El primer diseño que implementé tenía una estructura de tipo *First Come First Served* (FIFO). Por lo tanto, se basaba en que el worker que se convertía en máster fuese el primero en detectar que el master había caído y, para evitar que hubiese conflictos con los demás nodos, empleé exclusión mutua. Es decir, convertí todas las instrucciones que formaban parte del proceso de elección de líder en una sección crítica delimitada por *mutexs* de Python (semáforos), con los que ya había

trabajado en C. En el segundo diseño que implementé, el líder ya era elegido en base a un criterio, por lo que pude eliminar los semáforos, aunque el diseño seguía siendo muy rudimentario e inaceptable dado que no había comunicación entre los nodos worker.

Esta comunicación la logré en mi tercer diseño con lo que se podría entender como un sistema de paso de mensajes XMLRPC. El proceso consiste en lo siguiente: Cuando un nodo worker detecta que el nodo máster ha caído, recorrerá todos los workers de la lista de workers y, para cada uno, creará un proxy (cliente de su servidor) que le permitirá invocar a la función canBeLeader() y pasarle su puerto y el puerto de este nodo, que se está evaluando. Esta función comparará ambos puertos y devolverá True en caso de que el nodo que ha creado el proxy para invocar la función tenga prioridad sobre el nodo evaluado. En caso contrario, la función devolverá False y el nodo worker terminará el recorrido dado que ya no podrá ser un candidato a máster en esta elección. El nodo worker que se convertirá en máster será aquel que termine de recorrer toda la lista de workers porque siempre habrá recibido el valor True como parámetro de retorno para todas las invocaciones de la función mencionada, es decir, el worker que tenga el puerto con el valor más elevado. Una vez haya terminado el proceso de elección, el nuevo nodo máster informará a los nodos worker de quien es el nuevo máster a través de la función setMaster().

Implementación

NodeType.py

```
class NodeType:
     def __init__(self, node_type):
          self.node_type = node_type

     def getNodeType(self):
          return self.node_type

     def setNodeType(self, node_type):
          self.node_type = node_type
```

ManageNodes.py

```
import xmlrpc.client
import time


def pingNodes(node, port, event):
     proxy =
     xmlrpc.client.ServerProxy('http://localhost:'+port,
     allow_none=True)
     if (node.getNodeType() == "worker"):
          workers=[]


     while True:
          if event.is_set():
               break
          else:
               if (node.getNodeType() == "master"):
                    for worker in proxy.listWorkers():
                         try:
                              xmlrpc.client.ServerProxy(
                              worker,
                              allow_none=True).isAlive()
                         except:
                              proxy.removeWorker(
                              worker.split(':')[2])
               else:
                    try:
                         proxy_master =
                         xmlrpc.client.ServerProxy(
                         proxy.getMaster(),
                         allow_none=True)
                         proxy_master.isAlive()
                         workers =
                         proxy_master.listWorkers()
                    except:
```

```
                    for worker in workers:
                        proxy_worker =
                        xmlrpc.client.ServerProxy(
                        worker, allow_none=True)
                        isLeader =
                        proxy_worker.canBeLeader(
                        worker.split(':')[2], port)
                        if (isLeader == False):
                            break
                    if (isLeader == True):
                        node.setNodeType("master")
                        workers.remove(
                        'http://localhost:'+port)
                        proxy =
                        xmlrpc.client.ServerProxy(
                        'http://localhost:'+port,
                        allow_none=True)
                        for worker in workers:
                            proxy.addWorker(worker)
                            proxy_worker =
                            xmlrpc.client.
                            ServerProxy( worker,
                            allow_none=False)
                            proxy_worker.setMaster(
                            'http://localhost:'+port
                            )
        time.sleep(1)
```

ServerFunctions.py

```
import pandas

class ServerFunctions:
    def __init__(self, master):
        self.master = master
        self.workers = []
```

```
def getMaster(self):
     return self.master

def setMaster(self, master):
     self.master = master

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 ' '

def isAlive(self):
     return True

def canBeLeader(self, self_port, candidate_port):
     if int(self_port) <= int(candidate_port):
          return True
     else:
          return False
```

```
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())
```

Master.py

```
from xmlrpc.server import SimpleXMLRPCServer
import threading
import logging
import sys
import nodeType
import manageNodes
import serverFunctions


node = nodeType.NodeType("master")


master = SimpleXMLRPCServer(('localhost',
int(sys.argv[1])), logRequests=True, allow_none=True)
logging.basicConfig(level=logging.INFO)
master.register_instance(serverFunctions.ServerFunctions(
'http://localhost:'+sys.argv[1]))


event = threading.Event()
name_thread =
threading.Thread(target=manageNodes.pingNodes, args=(node,
sys.argv[1], event,))


try:
     print('Use control + c to exit the
     '+node.getNodeType()+' node.')
     name_thread.start()
     master.serve_forever()
except KeyboardInterrupt:
     print('Exiting '+node.getNodeType()+' node...')
     event.set()
     name_thread.join()
```

Worker.py

```
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client
import threading
import logging
import sys
import nodeType
import manageNodes
import serverFunctions


node = nodeType.NodeType("worker")


proxy =
xmlrpc.client.ServerProxy('http://localhost:'+sys.argv[1],
allow_none=True)
proxy.addWorker('http://localhost:'+sys.argv[2])


worker = SimpleXMLRPCServer(('localhost',
int(sys.argv[2])), logRequests=True, allow_none=True)
logging.basicConfig(level=logging.INFO)
worker.register_instance(serverFunctions.ServerFunctions(
'http://localhost:'+sys.argv[1]))


event = threading.Event()
name_thread =
threading.Thread(target=manageNodes.pingNodes, args=(node,
sys.argv[2], event,))


try:
     print('Use control + c to exit the
     '+node.getNodeType()+' node.')
     name_thread.start()
     worker.serve_forever()
```

```
except KeyboardInterrupt:
     print('Exiting '+node.getNodeType()+' node...')
     event.set()
     name_thread.join()
```

Client.py

```
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:'+sys.argv[1])

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

i=2
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_worker])
        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)))
```

Juegos de pruebas

La arquitectura distribuida de la que se parte para hacer las pruebas de ejecución es la que se ha mostrado en el diagrama, es decir, un nodo máster (puerto 9000) y tres nodos worker (puertos 8999, 8998 y 8997). Para lanzar el nodo máster se deberá utilizar la comanda 'python3 master.py 9000' y para lanzar cada uno de los nodos worker se deberán ejecutar las comandas 'python3 worker.py 9000 8999', 'python3 worker.py 9000 8998' y 'python3 worker.py 9000 8997'. También será necesario especificar siempre el puerto del máster al ejecutar el cliente. Esta arquitectura distribuida variará a medida que se vayan llevando a cabo las diferentes pruebas de ejecución.

Consistencia

Dado que los ficheros csv que utiliza mi programa son muy pequeños, el ciclo completo del cliente tiene un tiempo de ejecución tan breve que es imposible eliminar un nodo antes de que finalice todas sus tareas. Por ello, para hacer las pruebas necesarias para evaluar la consistencia, se añadió una función time.sleep() (que ya se había utilizado en sección de detección de fallos para aliviar la carga de la CPU) en el bucle principal del cliente y se le pasó el parámetro '5' (que indica que la CPU entrará en reposo durante cinco segundos en cada iteración del bucle), de modo que se pudieron hacer las pruebas enunciadas a continuación.

Prueba	Descripción	¿Correcto?
1	Se envían a ejecutar únicamente dos nodos worker. El cliente recibe el puerto del máster y los tres ficheros csv como parámetros de entrada. Cuando ha terminado el tratamiento del primer fichero csv, se añade el primer nodo worker. El cliente se adapta dinámicamente y envía el tercer fichero csv al tercer nodo para ser tratado.	Sí.
2	Se envían a ejecutar los tres nodos worker como se ha especificado al inicio de este apartado. El cliente recibe el puerto del máster y los tres ficheros csv como parámetros de entrada. Cuando ha terminado el tratamiento del primer fichero csv, se tumba el tercer nodo worker. El cliente se adapta dinámicamente y envía el tercer fichero csv al primer nodo para ser tratado.	Sí.
3	Se envían a ejecutar los tres nodos worker como se ha especificado al inicio de este apartado. El cliente recibe el puerto del máster y cuatro ficheros csv (uno de los tres ficheros de que se dispone está repetido) como parámetros de entrada. Cuando ha terminado el tratamiento del primer fichero csv, se tumba el primer nodo worker. El cliente se adapta dinámicamente y, después de tratar el segundo y tercer ficheros, envía el cuarto fichero csv al segundo nodo de la lista (ahora el primero) para ser tratado.	Sí.

Tolerancia a fallos

Prueba	Descripción	¿Correcto?
1	El cliente recibe el puerto del máster y el fichero 'provas1.csv' como parámetros de entrada.	Sí.

2	El cliente recibe el puerto del máster y el fichero 'provas2.csv' como parámetros de entrada.	Sí.
3	El cliente recibe el puerto del máster y el fichero 'provas3.csv' como parámetros de entrada.	Sí.
4	El cliente recibe el puerto del máster y los tres ficheros csv como parámetros de entrada.	Sí.
5	Se tumba el nodo worker con puerto 8997 (CTRL + C). El cliente recibe el puerto del máster y los tres ficheros csv como parámetros de entrada.	Sí.
6	Se relanza el nodo worker con puerto 8997. El cliente recibe el puerto del máster y los tres ficheros csv como parámetros de entrada.	Sí.
7	Se tumba el nodo worker con puerto 8998 (CTRL + C). El cliente recibe el puerto del máster y los tres ficheros csv como parámetros de entrada.	Sí.
8	Se relanza el nodo worker con puerto 8998. El cliente recibe el puerto del máster y los tres ficheros csv como parámetros de entrada.	Sí.
9	Se tumba el nodo worker con puerto 8999 (CTRL + C). El cliente recibe el puerto del máster y los tres ficheros csv como parámetros de entrada.	Sí.
10	Se relanza el nodo worker con puerto 8999. El cliente recibe el puerto del máster y los tres ficheros csv como parámetros de entrada.	Sí.
11	Se tumban los nodos worker con puertos 8999 y 8998 (CTRL + C). El cliente recibe el puerto del máster y los tres ficheros csv como parámetros de entrada.	Sí.
12	Se relanzan los nodos worker con puertos 8999 y 8998. El cliente recibe el puerto del máster y los tres ficheros csv como parámetros de entrada.	Sí.
13	Se tumba el máster (CTRL + C). A través del proceso de elección de líder, el nodo worker con puerto 8999 se convierte en el nuevo máster. El cliente recibe el puerto de este nuevo máster y los tres ficheros csv como parámetros de entrada.	Sí.
14	Se tumba el nodo worker con puerto 8998 (CTRL + C). El cliente recibe el puerto del segundo máster y los tres ficheros csv como parámetros de entrada.	Sí.
15	Se relanza el nodo worker con puerto 8998. El cliente recibe el puerto del segundo máster y los tres ficheros csv como parámetros de entrada.	Sí.
16	Se tumba el segundo máster (CTRL + C). A través del proceso de elección de líder, el nodo worker con puerto 8998 se convierte en el nuevo máster. El cliente	Sí.

	recibe el puerto del tercer máster y los tres ficheros csv como parámetros de entrada.	
17	Se relanza el nodo worker con puerto 8999. El cliente recibe el puerto del tercer máster y los tres ficheros csv como parámetros de entrada.	Sí.
18	Se lanza un nodo worker con puerto 8996. El cliente recibe el puerto del tercer máster y los tres ficheros csv como parámetros de entrada.	Sí.
19	Se tumba el nodo worker con puerto 8996 (CTRL + C). El cliente recibe el puerto del tercer máster y los tres ficheros csv como parámetros de entrada.	Sí.
20	Se relanza el nodo worker con puerto 8996. El cliente recibe el puerto del tercer máster y los tres ficheros csv como parámetros de entrada.	Sí.
21	Se tumba el tercer máster (CTRL + C). A través del proceso de elección de líder, el nodo worker se con puerto 8999 se convierte en el nuevo máster. El cliente recibe el puerto del cuarto máster y los tres ficheros csv como parámetros de entrada.	Sí.
22	Se tumba el nodo worker con puerto 8997 (CTRL + C). El cliente recibe el puerto del cuarto máster y los tres ficheros csv como parámetros de entrada.	Sí.
23	Se tumba el cuarto máster (CTRL + C). A través del proceso de elección de líder, el nodo worker se con puerto 8996 se convierte en el nuevo máster. Dado que no quedan nodos worker disponibles, la única forma de comprobar que dicho nodo se ha convertido en el quinto máster es tumbar dicho nodo.	Sí.

Fuentes documentales

A lo largo del desarrollo de la práctica se han hecho varios experimentos para tratar de lograr un código más elaborado y robusto. Muchos de estos experimentos no han llevado a nada en el contexto de la práctica, a pesar de que han servido para ampliar mis conocimientos del lenguaje de programación Python. Por esta razón, quería incorporar algunas de las páginas web consultadas a pesar de que los conceptos explicados en la mayor parte de éstas no aparecen en la entrega que he proporcionado (Algunas son mencionados en el apartado 'decisiones de diseño'). La mayor parte de elementos que incorpora la versión final de la práctica provienen de mi propio razonamiento y de información proporcionada en las sesiones de teoría y en consultas al profesor de la asignatura por lo que la webgrafía que podría citar es muy limitada.

Webgrafía

- https://es.wikipedia.org/wiki/Singleton
- https://stackoverflow.com/questions/6760685/creating-a-singleton-in-python
- https://dev.to/pila/constructors-in-python-init-vs-new-2f9j
- https://santoshk.dev/posts/2022/__init__-vs-__new__-and-when-to-use-them/
- https://stackoverflow.com/questions/2953462/pinging-servers-in-python
- https://stackoverflow.com/questions/70764097/how-can-i-ping-a-specific-port-and-report-the-results
- https://www.reddit.com/r/learnpython/comments/6hnn30/ping_a_server_with_port_using_python/
- https://superfastpython.com/thread-event-object-in-python/
- https://superfastpython.com/thread-mutex-lock/
- https://stackoverflow.com/questions/3310049/proper-use-of-mutexes-in-python