Práctica 2. Fundación de los sistemas distribuidos

La realización de la práctica ha sido en todo momento conjunta por parte de los dos integrantes del grupo, Alberto Blanco Álvarez y Marcos Esteve Hernández, ya que hemos ido programando las funciones de manera conjunta, recabando información de páginas web y debatiendo las ideas sobre cómo realizar las distintas operaciones. En el video explicativo, hemos intentado dividir la explicación en partes iguales pero queríamos dejar constancia que las explicaciones que hacemos no corresponden con las partes individuales que ha hecho cada uno, sino que ambos hemos trabajado conjuntamente en todas las funciones de la práctica.

El objetivo de la práctica consiste en la elaboración de un sistema distribuido en el que se monitoriza la ejecución de las otras funciones, denominadas slaves, permitiendo obtener un sistema en el que se puede trabajar de manera conjunta sobre un fichero garantizando el sincronismo.

Funciones elaboradas

Función master(x, ibm_cos) retorna lista:

Esta función recibe los parámetros iniciales especificados en la función main() para monitorizar las funciones slave que se crean en el main mediante un map.

Cada x segundos, especificados en una llamada asíncrona en la función main(), listará todas los ficheros de petición que generen los slaves, y los ordenará por el orden de creación, mediante la creación de un diccionario que los ordenará por el parámetro 'LastModified' y la función update.
Una vez se tenga el diccionario, se obtendrá el primer fichero de petición (list.pop) y se subirá, mediante ibm_cos.put_object() el fichero de autorización de trabajo del slave, eliminando a su vez el fichero de petición correspondiente. También, se guardará el id del slave que se ha usado en la lista que se retornará.

Tras esto, se monitoriza el fichero result.json cada x segundos, descargándolo del bucket y mirando su parámetro 'LastModified', hasta que este parámetro se vea modificado respecto la última vez que se monitorizó. En ese momento, se volverá al inicio de la ejecución hasta que ya se hayan tratado todos los slaves.

Una vez se hayan tratado todos los slaves, se retornará la lista de los slaves que han ejecutado, con el orden de ejecución.

Función slave(id, x, ibm_cos) retorna null:

Esta función subirá un fichero "p_write_{id}", que indicará una petición de trabajo por parte de este slave al COS especificado.

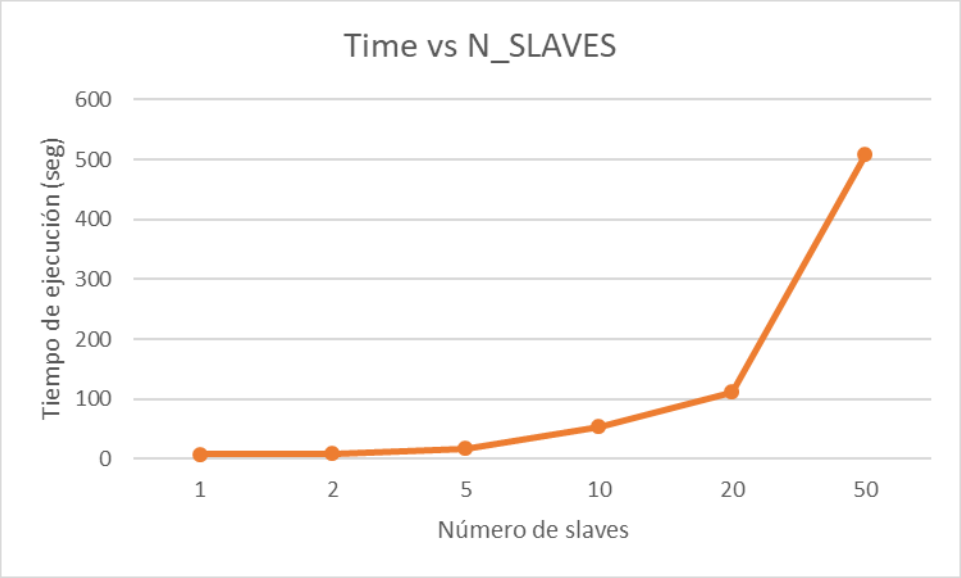
Estos ficheros de petición serán monitorizados por la función master(). Cuando sean tratados por el master, se subirán al COS unos ficheros que indicarán la autorización de trabajo para el slave. El slave mantendrá buscando este fichero en el COS hasta encontrarlo.

En cuanto detecte el fichero de autorización, el slave descargará el fichero results.json, y añadirá su id a la lista, subirá el fichero de nuevo al COS y finalizará su ejecución.
Como resultado, en el fichero results.json se tendrá una lista de los slaves que han ejecutado, en el orden de ejecución.

Función main():

En esta función se llamará a las funciones slave mediante un map, y a la función master mediante una llamada asíncrona por el módulo pywren. Se contabilizará el tiempo que tarda en la ejecución mediante el módulo time.
Al final, se descargará del COS el fichero results.json, se obtendrá una lista de su contenido y se comprarará con la lista obtenida de la función main para ver si la ejecución de los slaves coincide en ambas, y retornará un mensaje que indicará si coinciden o no.

Análisis de los resultados



En la gráfica podemos observar el aumento del tiempo de ejecución a medida que se incrementa el número de funciones slaves. Nótese como para un número de slaves bajo, la ejecución no tarda más que unos pocos segundos, pero al aumentar este número en mayor medida, el coste temporal aumenta exponencialmente. Esto se debe a que la escritura en el fichero se debe hacer de manera secuencial, permitiendo la ejecución de un solo slave mientras el resto han de esperar. Con esto, pese a encontrar un tiempo de espera óptimo para que se pueda ejecutar todo el programa sin fallos de sincronismos pero sin esperar demasiado, el tiempo total de ejecución seguirá condicionado al número de slaves que se tengan que ejecutar de manera secuencial.

Además se puede comprobar como la ejecución de los slaves no ha de coincidir con el orden alfanumérico de sus IDs:

```
[3, 1, 2, 4, 0]
[3, 1, 2, 4, 0]

Las listas son iguales
```

```
[0, 1, 10, 11, 12, 13, 14, 15, 16, 17, 19, 2, 4, 5, 6, 7, 8, 18, 3, 9]
[0, 1, 10, 11, 12, 13, 14, 15, 16, 17, 19, 2, 4, 5, 6, 7, 8, 18, 3, 9]

Las listas son iguales
Elapsed time: 111.51
```

Preguntas sobre la exclusión mútua

a) El modelo de master/slave es un modelo de comunicación asimétrica donde un proceso/dispositivo controla otros, sirviendo como un eje de comunicación. Es un modelo que se puede considerar distribuido ya que se puede ejecutar varios procesos de una manera prácticamente paralela y en varias máquinas. Sin embargo, no presenta un paralelismo completo, ya que siempre presenta un segmento de código crítico que se ha de ejecutar de forma secuencial para garantizar el sincronismo y el correcto funcionamiento. Por esto, si que se puede considerad distribuido.

b) Si el proceso master deja de funcionar, los procesos slaves que se hallaran dependiendo de este se quedarán en un deadlock, mientras que los que hayan recibido el aviso de ejecución finalizarían de manera correcta.
- Si el master crashea en un momento secuencial de la ejecución, en el que se fuera a dar permisos a un slave, este se quedaría esperando ejecución de forma permanente, afectando gravemente al funcionamiento del sistema y todos los que aún no hayan recibido permisos.
- Si el master crashea en un momento en el que ya se han cedido permisos a los slaves, entonces la ejecución no se verá afectada gravemente y se podrá acabar la ejecución del sistema.
Una forma de solucionar este problema sería mediante la introducción de unos timeouts a los slaves, de forma que si no reciben permiso en un determinado tiempo, puedan acabar su ejecución de forma prematura y evitar así el consumo de recursos y la inanición de estos.

c) Esto depende de la forma de ejecución. En el algoritmo propuesto, por ejemplo, no daría problema alguno, ya que al no introducir peticiones nuevas, se acabarán dando permisos a todos los slaves tarde o temprano, no habría inanición.
Por el contrario, si nos encontráramos en un sistema que permitiera añadir nuevas solicitudes continuamente, entonces se podría dar el caso de la inanición de un slave, sin recibir permisos de ejecución.

Conclusión

Mediante la monitorización por parte de una función master() de un sistema distribuido se puede trabajar de manera síncrona con un fichero sin que se produzcan problemas de accesibilidad.

Sin embargo, el empleo de este método puede afectar de manera destacable al tiempo de ejecución al tener que esperar a la autorización de un master para poder realizar la ejecución. Esto puede llegar a producir inconvenientes en sistemas de tiempo real, sobre todo a mayor tiempo que se establezca de espera.

Otro factor a tener en cuenta es que este tiempo de ejecución también dependerá de la conexión a los servicios de IBM cloud. Por lo tanto, tampoco se recomendaría la utilización del método MapReduce en el caso de que se carezca de una conexión estable.

Documentación

- GitHub - pywren/pywren-ibm-cloud: CloudButton Toolkit implementation for IBM Cloud Functions and IBM Cloud Object Storage: https://github.com/pywren/pywrenibm-cloud
- Funciones de IBM_COS: https://ibm.github.io/ibm-cos-sdk-python/reference/services/s3.html