UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS D'ALCOI

DSIC

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Sistemas de almacenamiento y procesado distribuido

Laboratorio 3

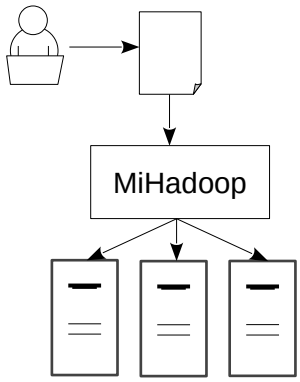
Javier Esparza Peidro – jesparza@dsic.upv.es

Contenido

1. Introducción

Con este laboratorio comenzaremos el proyecto 2 de la asigatura, cuyo objetivo consiste en diseñar una plataforma de procesamiento distribuido, que nada tendrá que envidiar a otras como Hadoop o Spark ;-).

En este laboratorio en particular se pondrán en práctica los conocimientos adquiridos sobre MapReduce en las clases de teoría. Para ello, se prentende implementar una versión limitada de Hadoop, capaz de ejecutar tareas MapReduce sobre un clúster de computadoras.

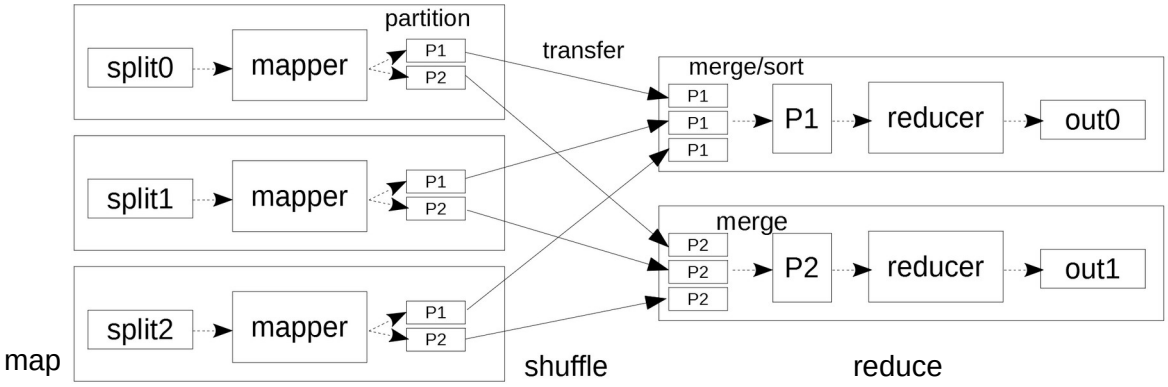


2. El proceso MapReduce

Tal y como se ha visto en clase, el proceso consta de las siguientes fases globales:

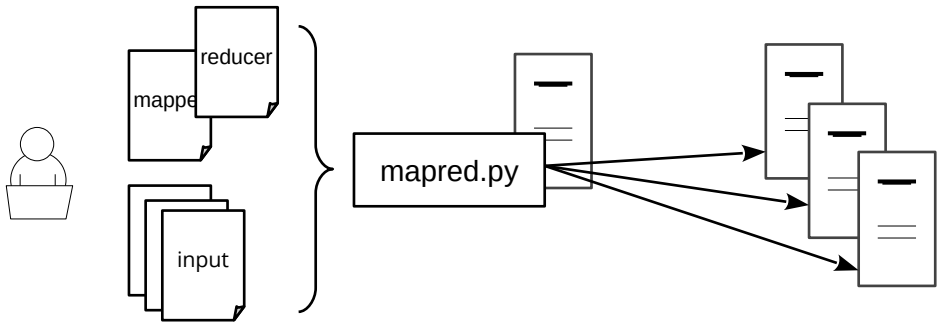
- Map: cada línea de un fragmento (split) es procesada por el *mapper*, que emite cero o más pares (k_1,v_1).
- Shuffle: los pares (k_1,v_1) se particionan en *r* particiones, donde *r* representa el número de *reducers*. El particionado se produce en el mismo nodo donde se ejecutó el *mapper*. Después cada partición se transfiere a su correspondiente *reducer*, que fusiona todas las particiones procedentes de todos los *mappers* y las ordena por clave.
- Reduce: el *reducer* recibe los pares (k_1, list(v_1)) y genera nuevos pares (k_2,v_2) que se escriben en un fichero.

A continuación se muestra una figura que ilustra todo el proceso.



3. Arquitectura

Todo el proceso será dirigido por un único script *mapred.py*, que se ejecutará en un nodo origen. Este script recibirá los datos iniciales y los scripts *mapper*/*reducer* y se encargará de ejecutar todas las fases del algoritmo MapReduce, transfiriendo los datos y ejecutando las tareas *mapper*/*reducer* en los distintos nodos. Finalmente obtendrá los resultados del proceso. Para ello, siempre que sea posible, hará uso de distintas utilidades Unix y de los comandos *ssh* y *scp*. La siguiente figura ilustra el procedimiento.



Para empezar, crearemos el script *mapred.py* y definiremos en su interior la función *run()*.

```
def run(input, output, mapper, reducer, config={})
```

Esta función ejecutará el algoritmo MapReduce. A continuación se describen los parámetros de la función:

- *input*: es el directorio donde se encuentran los ficheros a procesar.
- *output*: es el directorio donde se almacenarán los ficheros con los resultados de la operación.
- *mapper*: path donde se encuentra el script con el *mapper*.
- *reducer*: path donde se encuentra el script con el *reducer*.
- *config*: diccionario con distintos parámetros de configuración. Algunos de estos parámetros serán:
  - *numOfMappers*: número de tareas *mapper* que procesarán los datos de entrada.
  - *numOfReducers*: número de tareas *reducer* que generarán los resultados.
  - *nodes*: contiene información sobre los nodos que procesarán las tareas MapReduce.

Para facilitar la ejecución del proceso, sería conveniente disponer de un fichero *mapred.json* que contenga los parámetros de configuración que se cargan por defecto. Por ejemplo:

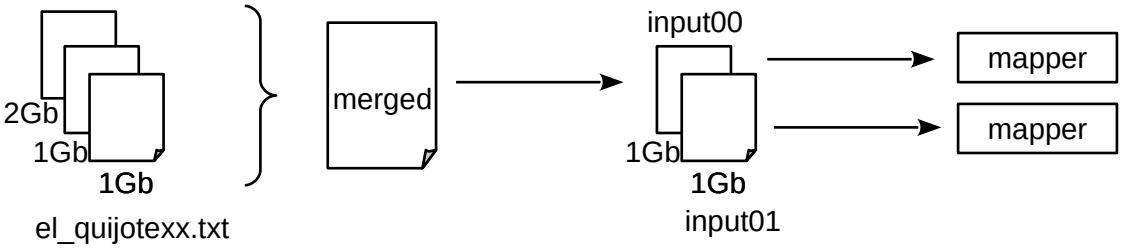
```
{
  "numOfMappers": 2,
  "numOfReducers": 2,
  "nodes": [
    {"host": "worker1", "user": "alumno", "password": "alumno"},
    {"host": "worker2", "user": "alumno", "password": "alumno"}
  ]
}
```

Estos parámetros se cargarán y aplicarán, si no son sobrescritos por nuevos parámetros en *config*.

En los siguientes apartados se discutirá acerca de cómo implementar cada etapa del algoritmo MapReduce. Para ello, se pondrán ejemplos de una tarea MapReduce simple, que cuente el número de veces que aparece cada palabra en un conjunto de ficheros de texto. Es posible obtener los ficheros de ejemplo el_quijote.txt, mapper.py y reducer.py en PoliformaT.

4. Entrada

La entrada del proceso es una colección de ficheros ubicados en un mismo directorio. Estos ficheros deberán ser troceados en fragmentos de dimensiones similares, para ser transferidos a los distintos *mappers*. Por tanto, deberán efectuarse *numOfMappers* fragmentos. Hay que tener en cuenta que cada fichero puede tener una dimensión diferente, así que no sería buena opción enviar un fichero diferente a cada *mapper*. Otra estrategia sería fusionar todos los ficheros en uno único, y fragmentar éste último en *numOfMappers* trozos. Esta aproximación se puede conseguir fácilmente utilizando las herramientas Unix *cat* y *split*, tal y como se presenta a continuación.



Los comandos necesarios para ello serían:

```
> cat el_quijote*.txt > merged  # fusiona los ficheros el_quijote*.txt en un fichero merged
> split -n l/2 -d merged input   # fragmenta merged en dos trozos de igual tamaño con prefijo input
```

Para ejecutar estos comandos en Python:

```
import os
files = " ".join([f"{input}/{file}" for file in os.listdir(input)]) # ficheros dentro del directorio input
os.system(f"cat {files} > merged")
os.system("split -n l/2 -d merged input")  # genera dos ficheros input00, input01
```

5. Map

Una vez se dispone de los fragmentos, es necesario asignar cada fragmento a un nodo diferente, donde se ejecutará el *mapper* para procesar dicho fragmento. Para planificar esta asignación se pueden utilizar distintos algoritmos, que tengan en cuenta las capacidades de cada nodo, o su carga. Para simplificar, efectuaremos una asignación *round-robin*. A partir de la lista *nodes*, se irán asignando los fragmentos por orden. Por ejemplo:

Fragmento	**Nodo**
input00	worker1
input01	worker2

Una vez efectuada la asignación de fragmentos, es necesario transferirlos a los distintos nodos. También será necesario transferir el script *mapper* que procesará el fragmento. Para ello utilizaremos la librería Python *Fabric*, que permite ejecutar comandos *ssh* y *scp* con código nativo Python:

https://www.fabfile.org/

Para instalar esta librería en un entorno virtual Python:

```
> pip install fabric
```

O de manera global:

```
> sudo apt install fabric
```

Una vez instalada, el primer paso consiste en abrir una conexión contra las máquinas remotas:

```
from fabric import Connection
con1 = Connection(host="worker1", user="alumno", connect_kwargs={"password": "alumno"})
con2 = Connection(host="worker2", user="alumno", connect_kwargs={"password": "alumno"})
```

Para transferir ficheros a la máquina remota, indicamos el fichero en la máquina origen, y el fichero en la máquina destino:

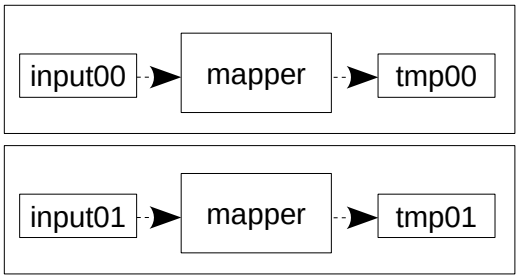
```
con1.put("input00", "input00")    # se trasfiere el fragmento XX a la máquina
con1.put(mapper, "mapper.py")     # se trasfiere el mapper a la máquina
con2.put("input01", "input01")    # se trasfiere el fragmento XX a la máquina
con2.put(mapper, "mapper.py")     # se trasfiere el mapper a la máquina
```

Una vez transferidos los fragmentos, será necesario ejecutar el *mapper*. El *mapper* es un script Python que recibe los datos de la entrada estándar y genera pares (k_1,v_1) por la salida estándar. Estos pares poseen el formato k_1\t$v_1$, es decir, clave y valor están separados por un tabulador '\t'.

Para ejecutar el *mapper* en el nodo remoto utilizaremos de nuevo la librería *Fabric.* Por ejemplo:

```
con1.run("cat input00 | python3 mapper.py > tmp00")
con2.run("cat input01 | python3 mapper.py > tmp01")
```

La salida obtenida es un resultado temporal que deberá pasar la fase de shuffle.

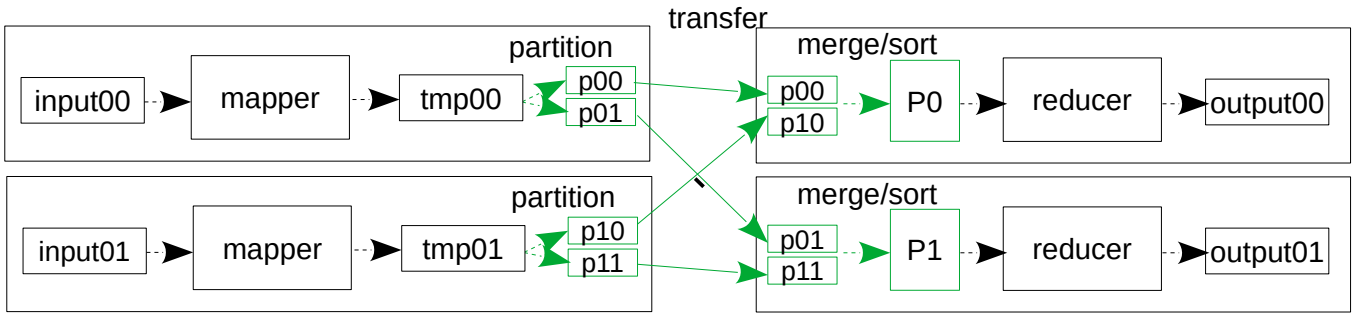


6. Shuffle

La fase de shuffle consta de tres etapas:

- Partition: los pares (k_1,v_1) son particionados en *r* particiones.
- Transfer: las particiones son transferidas a su correspondiente *reducer*.
- Merge/Sort: en cada *reducer*, las particiones son fusionadas y el resultado se ordena.

La siguiente figura ilustra el proceso global, resaltando las etapas de shuffle:



Discutimos cada subfase en los siguientes apartados.

6.1 Partition

El resultado temporal del *mapper* debe ser particionado en *r* subconjuntos. Cada subconjunto será transferido a un *reducer* diferente. Los subconjuntos se configuran en base a la clave k_1 de cada par (k_1,v_1). Para ello, se suelen utilizar técnicas de hashing.

Crearemos un script *partition.py* que se encargará de efectuar el particionado. Este script recibirá los pares (k_1,v_1) por la entrada estándar y creará *r* ficheros. Cada fichero contendrá una partición diferente. Por ejemplo:

```
#!/bin/python3
import sys
import hashlib

output = sys.argv[1]               # recibimos el prefijo de los ficheros output
numOfReducers = int(sys.argv[2])   # recibimos el número de reducers r

splits = [open(f"{output}_{i}", "wt") for i in range(numOfReducers)]
for line in sys.stdin:
    key, value = line.split("\t")
    splitIndex = int(hashlib.sha1(key.encode()).hexdigest(),16)%len(splits)
    splits[splitIndex].write(f"{key}\t{value}")

for split in splits: split.close()
```

Este fichero debe ser transferido al nodo remoto.

```
con1.put("partition.py", "partition.py")    # se trasfiere partition.py a la máquina
con2.put("partition.py", "partition.py")    # se trasfiere partition.py a la máquina
```

Finalmente, efectuamos el particionado.

```
con1.run("cat tmp00 | python3 partition.py p_00 2")    # particionar entrada 00 en 2 particiones:
                                                       # p_00_0 y p_00_1
con2.run("cat tmp01 | python3 partition.py p_01 2")    # particionar entrada 01 en 2 particiones
                                                       # p_01_0 y p_01_1
```

6.2 Transfer

El siguiente paso consiste en transferir las particiones parciales a los nodos donde se ejecutarán los *reducer*. Para ello, primero debe existir una asignación de tareas *reducer* a los distintos nodos. Por ejemplo:

Partición	**Nodo (reducer)**	**Ficheros a transferir**
Partición 0	worker1	p_00_0 (en worker1), p_01_0 (en worker2)
Partición 1	worker2	p_00_1 (en worker1), p_01_1 (en worker2)

Una vez efectuada la asignación, es necesario transferir las particiones al nodo correspondiente. La transferencia de ficheros se efectúa desde el nodo que posee las particiones parciales al nodo en que se ejecutará el *reducer*. La transferencia utilizará el comando *scp* en el nodo remoto. Para poder automatizar la autenticación es necesario instalar en el nodo remoto la utilidad *sshpass*.

```
> sudo apt install sshpass
```

Después, la transferencia de ficheros podría efectuarse del siguiente modo.

```
con1.run(f"sshpass -p 'alumno' scp -o 'StrictHostKeyChecking no' p_00_0 alumno@worker1:")
con1.run(f"sshpass -p 'alumno' scp -o 'StrictHostKeyChecking no' p_00_1 alumno@worker2:")
```

```
con2.run(f"sshpass -p 'alumno' scp -o 'StrictHostKeyChecking no' p_01_0 alumno@worker1:")
con2.run(f"sshpass -p 'alumno' scp -o 'StrictHostKeyChecking no' p_01_1 alumno@worker2:")
```

Es evidente que no son necesarias todas las transferencias, ya que algunas particiones parciales ya están en el nodo local.

6.3 Merge/Sort

Una vez las particiones parciales han sido transferidas a cada nodo, es necesario fusionarlas y ordenar todos los pares (k_1,v_1) por clave k_1. Para ello, es posible ejecutar los siguientes comandos:

```
con1.run(f"cat p_00_0 p_01_0 | sort  -t$'\t' > p0")
con2.run(f"cat p_00_1 p_01_1 | sort  -t$'\t' > p1")
```

7. Reduce

Una vez se dispone de las particiones fusionadas y ordenadas, es el momento de procesarlas utilizando el *reducer*. El *reducer* es un script Python que recibe los pares (k_1,v_1) ordenados por clave por la entrada estándar, y generará pares (k_2,v_2) por la salida estándar. Se utiliza por tanto una estrategia similar a la de Hadoop Streaming, en la que el *reducer* deberá detectar cuándo acaba un conjunto de pares (k_1,v_1) con la misma clave.

Antes de ejecutar el *reducer s*erá necesario transferir el script a los distintos nodos:

```
con1.put(reducer, "reducer.py")    # se trasfiere reducer.py al nodo
con2.put(reducer, "reducer.py")
```

A continuación se ejecutarán los *reducer*:

```
con1.run(f"cat p0 | python3 reducer.py > output00")
con2.run(f"cat p1 | python3 reducer.py > output01")
```

8. Obtención de resultados

El último paso consiste en recopilar los resultados obtenidos en los distintos nodos que ejecutaron el script *reducer*. Dichos resultados deberán estar disponibles en el nodo director, donde se ejecuta el script *mapred.py*.

```
con1.get("output00","output00")
con2.get("output01","output01")
con1.close()
con2.close()
```

9. Automatización, optimización y limpieza

Todos los pasos vistos en las secciones anteriores deberán ser automatizados en el script *mapred.py*, que debe ser capaz de ejecutar con cualquier script *mapper/reducer*, y sobre cualquier configuración de *numOfMappers/numOfReducers* y de nodos.

Además, existen múltiples puntos donde el proceso puede ser optimizado. Por ejemplo:

- Si los nodos que ejecutan los *mapper* y los nodos que ejecutan los *reducer* coinciden, hay transferencias de ficheros innecesarias, pues ya se dispone de los ficheros en los nodos locales.
- Muchos resultados parciales pueden ser evitados. Por ejemplo, las fases Map/Partition pueden ser conectadas con tuberías, sin necesidad de generar los ficheros temporales

tmpXX. Otro ejemplo serían las fases Merte/Sort/Reduce, que también puedes ser conectadas, sin necesidad de generar los resultados intermedios pXX.

- Todas las operaciones que se efectúan en nodos diferentes pueden ejecutarse en paralelo, utilizando para ello threads.

Por último, el proceso genera muchos ficheros intermedios que se quedan en los nodos. Es necesario eliminarlos una vez dejan de ser necesarios.