Sistemes distribuïts
Departament d'Enginyeria Informàtica i Matemàtiques
Universitat Rovira i Virgili

Pràctica 1: High performance computing cluster

Josep Bello Curto
Oriol López Egea

23-04-2021

Índex

1. Decisions de disseny

Enllaç al repositori de GitHub: https://github.com/JosepBC/SDPR1
Com a protocols de comunicació directa hem decidit utilitzar XMLRPC i per comunicació indirecta Redis.

1.1 Client

Per la part de la CLI client hem decidit seguir el format que es va dir al Moodle, per exemple:

```
pyhton3 Client/client.py worker create
pyhton3 Client/client.py worker list
pyhton3 Client/client.py worker delete [id]
python3 Client/client.py job run-countwords [url1 url2 ...]
python3 Client/client.py job run-wordcount [url1 url2 ...]
```

Per fer aquesta CLI hem decidit utilitzar el mòdul argparse de Python que ens permet fer de forma simple una CLI bàsica.

1.2 Clúster

El codi del clúster es divideix en dues classes

- Master: Té tota la lògica del node master referent a la gestió de workers i tasques.
- Worker: Esta escoltant la cua de redis on es fiquen les tasques per executar-les.

1.2.1 Master

El master és qui exposa l'API per tal que l'usuari pugui interactuar amb el clúster.

1.2.1.1 Gestió de workers

Hem decidit que els workers s'executin cadascun a un procés diferent d'aquesta forma poden treballar de forma paral·lela diversos workers. Per controlar quins workers tenim al clúster hem decidit utilitzar un diccionari, on la clau és l'id del worker i el valor el procés del worker.

Per crear un worker simplement afegim un nou element a aquest diccionari i iniciem el procés. Per eliminar-lo l'eliminem del diccionari i parem el procés i finalment per llistar els workers retornem la llista de claus del diccionari, és a dir, les ID dels workers.

1.2.1.2 Gestió de tasques

Hem decidit utilitzar cues de redis per guardar les tasques i que els workers les agafin d'allí.

Per tal de fer el clúster el màxim genèric hem decidit que es passin les funcions que es vol que el clúster executi per paràmetre utilitzant el mòdul dill de la següent forma

```
import dill
def foo():
    print("Hello")

if __name__ == '__main__':
    serFoo = dill.dumps(foo)
```

La funció dill.dumps() serialitza la funció per tal que es pugui executar des del clúster.

Per demanar que el clúster executi una tasca, amb algun dels workers disponibles, tenim dues opcions, enviar una sola tasca o enviar un conjunt de tasques les quals després hem d'acumular.

Per tal que les funcions d'enviar una tasca, o un grup de tasques, siguin asíncrones hem decidit que aquestes no retornin el resultat, sinó que sigui una altra funció, getTaskResult, la que ens retorni el resultat un cop generat.

El procediment per enviar una sola tasca seria per exemple:

```
import dill
import xmlrpc.client
def foo(urlContent):
    return urlContent

if __name__ == '__main__':
    proxy = xmlrpc.client.ServerProxy("http://localhost:9000")
    serFoo = dill.dumps(foo)
    proxy.submitTask(1, serFoo, http://x.com)
    jobId, res = proxy.getTaskResult(1)
    print("Task:", jobId, "=", res)
```

I per enviar dues o més tasques

```
import dill
import xmlrpc.client
def foo(urlContent):
    return urlContent

def reduceFoo(urlContentList):
    reduced = ""
    for elem in urlContentList:
        reduced += elem
    return reduced

if __name__ == '__main__':
    proxy = xmlrpc.client.ServerProxy("http://localhost:9000")
    serFoo = dill.dumps(foo)
    serReduceFoo = dill.dumps(reduceFoo)
    urls = ["http://x.com/", "http://x.com/"]
    taskID = proxy.submitTask(1, serFoo, urls, serReduceFoo)
    jobId, res = proxy.getTaskResult(taskID)
    print("Task:", jobId, "=", res)
```

Internament la funció submitTask construeix un diccionari amb totes les dades de la tasca que li han passat per paràmetre i fa una tupla amb aquest diccionari i una funció interna anomenada mapTask. Aquesta funció mapTask és la que executarà realment el worker, per tasques individuals o subtasques, i és l'encarregada d'executar, dintre del worker, la funció que l'usuari ens ha demanat i retornar, per redis, el resultat.

Per un altre costat quan enviem més d'una tasca al clúster, submitTaskGrup, es crida per cada URL a la funció submitTaskGroup, d'aquesta forma els workers aniran executant aquestes funcions i guardant el seu resultat a redis. A continuació submitTaskGroup construeix un diccionari amb tota l'informació necessària per executar el reduce dels resultats parcials i llavors crea una tupla amb aquestes dades i una funció reduceTask. Aquesta funció reduceTask, que és la que executa el worker, s'espera que tots els altres workers acabin la seva feina, trau el resultat de la cua de redis on s'han guardat, explicat mes endavant, i finalment executa el reduce pròpiament i el guarda a redis.

1.2.2 Worker

Els workers estan esperant que arribin tasques a fer per una cua de redis, quan trau alguna cosa de la cua es desbloqueja, carrega la tupla que ha enviat o bé submitTask o submitTaskGroup, que es tracta d'una funció a executar que serà mapTask o reduceTask respectivament, i els arguments d'aquesta funció. I executa aquesta funció mapTask o reduceTask.

1.2.3 Cues de redis

Hem decidit utilitzar les cues de redis per fer la comunicació entre el master i els workers. Hem decidit tenir tres tipus de cues, una cua on es ficaran totes les tasques a executar pels workers i dos tipus més on es guardaran els resultats.

- jobs: Cua on es fiquen totes les tasques per part del master, on els workers estan bloquejats.
- jobid: Cua amb el nom del jobid on es guarden els resultats, parcials o d'una tasca de nomes un element.
- recueJobId: Cua amb un identificador generat de forma pseudoaleatoria on es guardara el resultat final d'una tasca amb múltiples subtasques.

2. Joc de proves

Per tal de provar aquesta pràctica hem fet un fitxer per anar fent proves sense utilitzar la CLI i la pròpia CLI. A continuació hi ha algunes captures de l'execució de la CLI i els seus resultats.

2.1 Gestió de workers

```
bello@bellopc:UNI/SDPR1 ‹master*›$ python3 Client/client.py worker create
bello@bellopc:UNI/SDPR1 ‹master*›$ python3 Client/client.py worker create
bello@bellopc:UNI/SDPR1 ‹master*›$
```

```
bello@bellopc:UNI/SDPR1 ‹master*›$ python3 Cluster/runCluster.py
Control-C to exit
127.0.0.1 - - [23/Apr/2021 17:22:29] "POST /RPC2 HTTP/1.1" 200 -
Locking worker  0
127.0.0.1 - - [23/Apr/2021 17:22:34] "POST /RPC2 HTTP/1.1" 200 -
Locking worker  1
```

Figura 2.1: Crear un worker

```
bello@bellopc:UNI/SDPR1 ‹master*›$ python3 Client/client.py worker list
[0, 1]
```

Figura 2.2: Llistar els workers

```
bello@bellopc:UNI/SDPR1 ‹master*›$ python3 Client/client.py worker list
[0, 1]
bello@bellopc:UNI/SDPR1 ‹master*›$ python3 Client/client.py worker delete 1
bello@bellopc:UNI/SDPR1 ‹master*›$ python3 Client/client.py worker list
[0]
```

Figura 2.3: Eliminar un worker

2.2 Tasques

```
bello@bellopc:UNI/SDPR1 ‹master*›$ curl http://localhost:8000/file1.txt
Hello there%
bello@bellopc:UNI/SDPR1 ‹master*›$ python3 Client/client.py job run-countwords http://localhost:8000/file
1.txt
Task: 0.9860658591636012 = 2
```

```
bello@bellopc:UNI/SDPR1 ‹master*›$ curl http://localhost:8000/file1.txt
Hello there%
bello@bellopc:UNI/SDPR1 ‹master*›$ python3 Client/client.py job run-countwords http://localhost:8000/file
1.txt http://localhost:8000/file1.txt
Task: 0.06875608924768062 = 4
```

Figura 2.4: Enviar job count words

```
bello@bellopc:UNI/SDPR1 ‹master*›$ curl http://localhost:8000/file1.txt
Hello there%
bello@bellopc:UNI/SDPR1 ‹master*›$ python3 Client/client.py job run-wordcount http://localhost:8000/file1
.txt
Task: 0.46747350824304723
Hello = 1
there = 1
```

```
bello@bellopc:UNI/SDPR1 ‹master*›$ curl http://localhost:8000/file1.txt
Hello there%
bello@bellopc:UNI/SDPR1 ‹master*›$ python3 Client/client.py job run-wordcount http://localhost:8000/file1
.txt http://localhost:8000/file1.txt
Task: 0.4252639305458322
Hello = 2
there = 2
```

Figura 2.5: Enviar job count words

Índex de figures