Universidad Zaragoza

1542

30321 – Sistemas Distribuidos
Prácticas de Laboratorio

Grado en Ingeniería Informática

Curso 2017-2018

ESCUELA DE INGENIERÍA Y ARQUITECTURA

Departamento de Informática e Ingeniería de Sistemas

v 1.0

septiembre de 2017

Práctica 1. Introducción a las arquitecturas de Sistemas Distribuidos

1. Objetivos y Requisitos

Esta práctica tiene como objetivo que analicéis la arquitectura cliente servidor con sus variantes y así como la arquitectura master-worker. Para ello, diseñaremos e implementaremos sistemas distribuidos muy sencillos y realizaremos distintos experimentos.

Deberéis entregar el código fuente y una memoria en la que analizaréis el comportamiento del cliente-servidor y resumiréis vuestro diseño del máster worker. No os extendáis más de *4* páginas en total.

1.1 Objetivo

- Familiarización con las arquitecturas, cliente-servidor y máster-worker

1.2 Requisitos

- Elixir y su entorno de desarrollo
- Seguir la guía de codificación de Elixir publicada aquí:

https://github.com/christopheradams/elixir_style_guide/blob/master/README.md

2. Aspectos de Elixir necesarios para esta práctica

2.1 Generación de timestamps

Se puede realizar una invocación a Erlang para que nos devuelva el tiempo actual del sistema

```
lex(1)> :os.system_time(:milliseconds)
```

El parámetro de la función system_time/1 puede ser también :`seconds` o :`micro_seconds`.

Puedes crear la función timestamp/1:

```
def timestamp do
  :os.system_time(:milliseconds)
end
```

De manera que, al invocarla sucesivas veces se puede saber el tiempo de ejecución de una secuencia de código. Este aspecto se denomina habitualmente *instrumentación del código* y en este caso es intrusivo.

2.2 Programación Distribuida en Elixir

Una máquina física puede ejecutar una o varias máquinas virtuales de Erlang. En cualquier caso, dichas máquinas virtuales pueden estar interconectadas y pueden permitir a sus procesos comunicarse mediante el paso de mensajes [1].

Para ello, hay que seguir los siguientes pasos, desde la línea de comandos de una máquina física ejecuta:

```
$ iex --name node1@127.0.0.1
```

El comando crea una máquina virtual de Erlang y nos permite acceder a ella a través de su intérprete de comandos. La opción `--name` convierte a la máquina virtual en un nodo de nombre `node1@127.0.0.1`. Esto va a hacer que otras máquinas virtuales creadas de la misma forma puedan comunicarse entre sí.

Una vez creado se puede obtener su nombre mediante la función `Kernel.node/0`.

```
iex(node1@localhost)1> node
:node1@127.0.0.1
```

Como puede observarse, se ha representado el nombre del nodo como un átomo.

En la misma máquina física o bien en otra, se puede crear otra máquina vritual de la misma forma e interconectarlas mediante la función Node.connect/1.

Si ejecutamos el siguiente comando, creará otra máquina virtual:

```
$ iex --name node2@127.0.0.1
```

Tras ejecutar esta función, interconectaremos el nodo 1 con el nodo 2 (las dos máquinas virtuales).

```
iex(node2@localhost)1> Node.connect(:"node1@localhost")
true
```

La función Node.list/0 nos dice cuántos nodos tenemos conectados a una máquina virtual:

```
iex(node1@localhost)2> Node.list
[:node2@localhost]

iex(node2@localhost)2> Node.list
[:node1@localhost]
```

Por supuesto, se pueden añadir más nodos.

```
$ iex --name node3@localhost
iex(node3@localhost)1> Node.connect(:"node2@localhost")
iex(node3@localhost)2> Node.list
[:node2@localhost, :node1@localhost]
```

Al hacerlo, automáticamente estarán todos conectados entre sí.

Una vez que se han creado varios nodos, existen varias formas de hacer que interactúen. Por ejemplo, mediante Node.spawn. Esta función sirve para crear un thread en una máquina remota y ejecutar allí una función. Así:

```
iex(node1@localhost)4> Node.spawn(
                         :"node2@127.0.0.1",
                         fn -> IO.puts "Hello from #{node}" end
                       )
Hello from node2@127.0.0.1
```

Ejecutará esa función lambda en el nodo 1, la prueba está en la salida que se obtiene.

Otra capacidad importante es la de enviar y recibir mensajes independientemente de la localización. Como a priori, los procesos no se conocen, existe un mecanismo, un registro (catálogo), que soluciona ese problema. Así un proceso se puede registrar en él con un nombre. La función `Process.register(self(), :server)` registrará al proceso que la invoque con el nombre de server.

```
iex(node1@localhost)3> Process.register(self(), :server)
true
```

De manera que, otro proceso puede enviarle mensajes de esta manera:

```
iex(node1@localhost)9> send(
                         {:server, :"node1@127.0.0.1"},
                         "Hola servidor, soy el cliente"
                       )
```

En Elixir se puede descubrir procesos (y sus pids, nombres, etc.) de forma automática, pero por el momento nosotros simplemente utilizaremos estos mecanismos y configuraremos el grupo de máquinas de forma manual.

Por último, cuidado con los puertos en el laboratorio. Tenéis que lanzar la máquina virtual indicándole el rango de puertos habilitados:

```
iex --name nodo1@155.210.154.200 \
      --erl  '-kernel inet_dist_listen_min 32000'
      --erl  '-kernel inet_dist_listen_max 32009'
```

3. Ejercicios

Junto con este enunciado, tenéis disponible el fichero para_primes.exs que contiene algunas funciones para resolver esta práctica. En particular: is_prime/1 y find_primes/1. La primera determina si un número entero dado es primo o no y devuelve true o false, respectivamente. La segunda toma como entrada un intervalo de números enteros y devuelve una lista con todos los números primos que contiene el intervalo. Con esas funciones, tenéis que realizar los 4 ejercicios siguientes.

3.1 Análisis de prestaciones de una arquitectura cliente-servidor

Caso 1. Cliente-Servidor, servidor secuencial

Programar un sistema distribuido en Elixir, siguiendo la arquitectura cliente-servidor, siendo el servidor secuencial. El cliente invocará al servidor, pasándole un intervalo de números enteros y esperará los resultados. El servidor atenderá la petición y le devolverá una lista con los números primos que contiene dicho intervalo.

Caso 2. Cliente-Servidor, servidor concurrente

Programar un sistema distribuido en Elixir siguiendo la arquitectura cliente-servidor, siendo el servidor concurrente. El cliente invocará al servidor, pasándole el intervalo [1, 100000] y esperará los resultados. El servidor atenderá la petición y le devolverá una lista con los números primos que contiene dicho intervalo.

Se pide:

1. Realizar 2 gráficas, una para cada caso, en las que tenéis que medir el tiempo de ejecución del cliente en 2 escenarios:
   - Escenario 1: solo están en ejecución el cliente y el servidor
   - Escenario 2: además del cliente y el servidor hay 2 clientes simultáneamente realizando peticiones al mismo servidor
   - Escenario 3: además del cliente y el servidor, hay 4 clientes simultáneamente realizando peticiones al mismo servidor

Nota: Para minimizar la variabilidad en las mediciones, deberéis repetir cada medición 10 veces y realizar la media aritmética de todas las mediciones obtenidas.

2. Analizar los resultados obtenidos y explicarlos razonadamente en función de las arquitecturas software y del hardware en el que se han ejecutado.

Ejercicio 2. Diseño de un master-worker

Caso 1 Master-Worker, con workers homogéneos

Programar un Sistema distribuido en Elixir siguiendo la arquitectura master-worker. El master deberá encontrar los números primos en el intervalo [1, 100000], para ello repartirá la carga entre 3 workers cuyo comportamiento es homogéneo (se espera que tarden más o menos lo mismo).

Caso 2 Máster-Worker, con workers heterogéneos

Programar un Sistema distribuido en Elixir siguiendo la arquitectura master-worker. El master deberá encontrar los números primos en el intervalo [1, 100000], para ello repartirá la carga entre 3 workers cuyo comportamiento es heterogéneo. Para similar la heterogeneidad, un worker, antes de enviar los resultados invocará a la función siguiente:

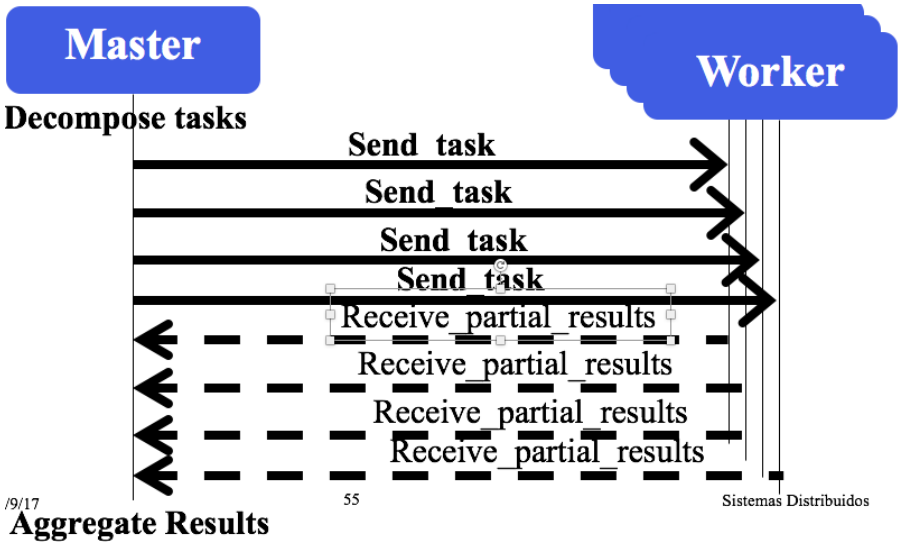
```
if :rand.uniform(100)>60, do: Process.sleep(round(:rand.uniform(100)/100 * 2000))
```

Se pide:

1. medir el tiempo de ejecución en ambos casos (repetir los experimentos al menos 10 veces)
2. explicar la estrategia de reparto de carga realizada en el caso 1 y en el caso 2.
3. comparar los resultados obtenidos en ambos casos con la ejecución secuencial del ejercicio 1.

En teoría, el máximo speedup sería texmaster-worker = texsecuencial / número_workers.

El tiempo total de un máster para realizar su tarea se puede explicar de la siguiente forma:



Tmaster = Tdecompose_tasks + max{T_sendtask_i} + max{T_worker_i} + max{T_receiveresults_i} + Taggregate

4. Evaluación

La realización de las prácticas es por parejas, pero los dos componentes de la pareja deberán entregarla de forma individual. En general estos son los criterios de evaluación:

- Deben entregarse *todos* los programas, se valorará de forma negativa que falte algún programa.
- Todos los programas deben compilar correctamente, se valorará de forma muy negativa que no compile algún programa.
- Todos los programas deben funcionar correctamente como se especifica en el problema.
- Todos los programas tienen que seguir el manual de estilo de Elixir, disponible en[1] (un 20% de la nota estará en función de este requisito). Además de lo especificado en el manual de estilo, cada fichero fuente deberá comenzar con la siguiente cabecera:

```
# AUTORES: nombres y apellidos
# NIAs: números de identificaci'on de los alumnos
# FICHERO: nombre del fichero
# FECHA: fecha de realizaci’on
# TIEMPO: tiempo en horas de codificación
# DESCRIPCI'ON: breve descripci'on del contenido del fichero
```

4.1. Rúbrica

Con el objetivo de que, tanto los profesores como los estudiantes de esta asignatura por igual, puedan tener unos criterios de evaluación objetivos y justos, se propone la siguiente rubrica en la Tabla 1. Los valores de las celdas son los valores mínimos que hay que alcanzar para conseguir la calificación correspondiente y tienen el siguiente significado:

A+ (excelente). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. Plantea correctamente el problema a partir del enunciado propuesto e identifica las opciones para su resolución. Aplica el método de resolución adecuado e identifica la corrección de la solución, sin errores. En el caso de la memoria, se valorará una estructura y una presentación adecuadas, la corrección del lenguaje, así como el contenido explica de forma precisa los conceptos involucrados en la práctica. En el caso del código, este se ajusta exactamente a las guías de estilo propuestas.

A (bueno). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. Plantea correctamente el problema a partir del enunciado propuesto e identifica las opciones para su resolución. Aplica el método de resolución adecuado e identifica la corrección de la solución, con ciertos errores no graves. Por ejemplo, algunos pequeños casos (marginales) no se contemplan o no funcionan correctamente. En el caso del código, este se ajusta casi exactamente a las guías de estilo propuestas.

B (suficiente). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. No plantea correctamente el problema a partir del enunciado propuesto y/o no identifica las opciones para su resolución. No aplica el método de resolución adecuado y / o identifica la corrección de la solución, pero con errores. En el caso de la memoria, bien la estructura y / o la presentación son mejorables, el lenguaje presenta deficiencias y

[1] https://github.com/christopheradams/elixir_style_guide/blob/master/README.md

/ o el contenido no explica de forma precisa los conceptos importantes involucrados en la práctica. En el caso del código, este se ajusta a las guías de estilo propuestas, pero es mejorable.

B- (suficiente, con deficiencias). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. No plantea correctamente el problema a partir del enunciado propuesto y/o no identifica las opciones para su resolución. No se aplica el método de resolución adecuado y/o se identifica la corrección de la solución, pero con errores de cierta gravedad y/o sin proporcionar una solución completa. En el caso de la memoria, bien la estructura y / o la presentación son manifiestamente mejorables, el lenguaje presenta serias deficiencias y / o el contenido no explica de forma precisa los conceptos importantes involucrados en la práctica. En el caso del código, hay que mejorarlo para que se ajuste a las guías de estilo propuestas.

C (deficiente). El software no compila o presenta errores graves. La memoria no presenta una estructura coherente y/o el lenguaje utilizado es pobre y/o contiene errores gramaticales y/o ortográficos. En el caso del código, este no se ajusta exactamente a las guías de estilo propuestas.

Calificación	S.Threads	S.Select	Código	Memoria
10	A+	A+	A+	A+
9	A+	A+	A	A
8	A	A	A	A
7	A	A	B	B
6	B	B	B	B
5	B-	B-	B-	B-
suspenso	1 C			

Tabla 1. Rúbrica

5. Entrega

Deberéis entregar un fichero zip que contenga: (i) los fuentes: ejercicio1.exs y ejercicio2.exs, y (ii) la memoria en pdf. La entrega se realizará a través de moodle2 en la actividad habilitada a tal efecto. La fecha de entrega será no más tarde del jueves anterior al comienzo de la siguiente práctica, esto es, el 19 de octubre de 2017 para los grupos A y el 26 de octubre de 2017 para los grupos B.

Los días 20 y 27 de octubre durante la sesión de prácticas se realizará una defensa “in situ” de la práctica.

6. Bibliografía

[1] “Elixir in action”, Sasa Juric, Manning, 2015.