Universidad Zaragoza
1542

30321 – Sistemas Distribuidos
Prácticas de Laboratorio

Grado en Ingeniería Informática

Curso 2019-2020

ESCUELA DE INGENIERÍA Y ARQUITECTURA
Departamento de Informática e Ingeniería de Sistemas

v 3.0
septiembre de 2019

Práctica 1. Introducción a las arquitecturas de Sistemas Distribuidos

1. Objetivos y Requisitos

Esta práctica tiene como objetivo analizar las arquitecturas *cliente-servidor* con sus variantes y así como la arquitectura *master-worker* y el uso que realizan de los recursos computacionales (la calidad del servicio que pueden ofrecer). Para ello, diseñaremos e implementaremos sistemas distribuidos muy sencillos en Elixir 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. **El número máximo de páginas permitido es de 4 en total**.

1.1 Objetivo

- Familiarización con las arquitecturas, cliente-servidor y máster-worker y el Quality of Service

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 Medición de tiempos de ejecución de una función

En Elixir el módulo Time contiene una serie de funciones relacionadas con el tiempo y que pueden utilizarse para medir tiempos de ejecución. La siguiente función

```
Iex(1)> Time.utc_now
```

devuelve el tiempo actual en UTC. De manera que, al invocarla sucesivas veces y restar los tiempos se puede saber el tiempo de ejecución de una secuencia de código. La función `Time.diff/3` puede usarse para restar fechas (en el tercer parámetro se puede especificar la unidad en que se expresa la diferencia, por ejemplo en milisegundos). Este aspecto se denomina habitualmente *instrumentación del código* y en este caso es intrusivo, en el sentido de que tiene que introducirse dentro del un código creado para otro fin.

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@127.0.0.1)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 virtual 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@127.0.0.1)1> Node.connect(:"node1@127.0.0.1")
true
```

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

```
iex(node1@127.0.0.1)2> Node.list
[:node2@127.0.0.1]

iex(node2@localhost)2> Node.list
[:node1@127.0.0.1]
```

Por supuesto, se pueden añadir más nodos.

```
$ iex --name node3@localhost
iex(node3@127.0.0.1)1> Node.connect(:"node2@127.0.0.1")
iex(node3@127.0.0.1)2> Node.list
[:node2@127.0.0.1, :node1@127.0.0.1]
```

Al hacerlo, automáticamente estarán todos conectados entre sí.

Una vez que se han creado y unido varios nodos se pueden usar las funciones `send` y `receive` para comunicar procesos en distintas máquinas virtuales. En caso de que los procesos no se conozcan a priori, 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.

2.2.1 Conexión de máquinas virtuales de Erland en distintas máquinas a través de la red

El mecanismo es similar al anterior, pero por cuestiones de seguridad, hay que realizar algunas cosas más. Para poder conectar dos nodos, tienen que acordar previamente una cookie – en este caso, una especie de frase de paso – que se verifica mutuamente durante el proceso de conexión. La primera vez que se arranca

una instancia de la máquina virtual de Erlang (BEAM), se genera una cookie aleatoria que se almacena en el directorio home en el fichero .erlang.cookie. Para ver una cookie, se puede usar la función `Node.get_cookie/0`:

```
iex(node1@localhost)1> Node.get_cookie
:JHSKSHDYEJHDKEDKDIEN
```

Se puede observar que la cookie se representa internamente como un átomo. Un nodo que se ejecute en otra máquina tendrá otra cookie diferente. Por tanto, por defecto no podrán comunicarse. Una forma de hacer que todas las máquinas compartan las cookies es a través de las opciones de arranque del intérprete iex, con la opción `--cookie nombre_cookie`. Mientras todas las máquinas virtuales compartan el mismo nombre de cookie, estas podrán comunicarse.

Por último, cuidado con los puertos en el laboratorio. Tenéis que lanzar la máquina virtual indicándole el rango de puertos habilitados. El comando que tiene en cuenta todo esto es el siguiente:

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

2.2.2 Deployment de un sistema distribuido

La puesta en ejecución de los distintos procesos distribuidos en las máquinas correspondientes puede ser una tarea muy compleja y tediosa. En la actualidad, existen distintas tecnologías para automatizar ese proceso. Elixir cuenta con una función que permite realizar el "deployment" de forma muy sencilla y conviene utilizarla. La función `Node.spawn` sirve para crear un *proceso* en una máquina remota y ejecutar allí la función anónima que se le pasa como argumento. Así:

```
iex(node1@127.0.0.1)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 2, en este caso, se escribe por pantalla el resultado de evaluar la cadena de caracteres en el nodo 1. Para el desarrollo de las prácticas es conveniente utilizar `Node.spawn/4` Finalmente, cabe destacar que esta función devuelve el PID del proceso creado remotamente.

3. Ejercicios

Junto con este enunciado, tenéis disponible el fichero `para_fibonacci.exs` que contiene algunas funciones para resolver esta práctica. En particular, contiene: (1) `fibonnacci?/1`, `Fibonacci_tr/1` y `of/1`. Funciones que calculan el Fibonacci del número pasado como argumento; no obstante, las implementaciones son muy distintas. Con esas funciones, tenéis que crear servidores y workers y realizar los ejercicios siguientes. (2) Un generador de carga de trabajo o programa cliente.

3.1 Diseño de una Arquitectura de SSDD

En la práctica, los sistemas distribuidos se construyen atendiendo a requisitos funcionales (qué debe de realizar el sistema) y requisitos no funcionales (cómo debe realizarlo y cuáles son sus prestaciones o coste). En los sistemas distribuidos actuales, es muy habitual que los requisitos no funcionales se materialicen en un acuerdo (o contrato formal entre el cliente y la empresa que proporciona el servicio) denominado acuerdo del nivel del servicio (*Service Level Agreement* SLA en inglés). Desde un punto de vista cuantitativo, el SLA se especifica en términos de unos indicadores, como, por ejemplo, tiempo de ejecución total, "*throughput*" (número de tareas ejecutadas por unidad de tiempo, consumo energético máximo o coste económico.

En tiempo de ejecución ("*runtime*"), los indicadores del SLA forman parte de un conjunto de variables denominadas "calidad del servicio" ("*Quality of Service"* QoS, en inglés). En esta práctica vamos a hacer uso de un único indicador de SLA, el tiempo de respuesta. En general el tiempo de respuesta de una tarea es el tiempo total de transmisión por la red de comunicación, más el tiempo de ejecución efectivo, más el tiempo de espera (o de interferencia), debido a un uso compartido de la infraestructura computacional:

$$T_{total} = T_{transmisión} + T_{ejecución} + T_{espera}$$

En esta práctica, el acuerdo va a consistir en que el tiempo total de respuesta de una tarea tiene que ser menor que 1,5 veces tiempo de ejecución de la tarea de forma aislada.

Para garantizar el QoS, deberéis:

1) realizar un estudio de la carga de trabajo en cada caso. ¿Es la carga de trabajo continua o variable en el tiempo?
2) Analizar las características de las máquinas del laboratorio L1.02, por ejemplo, determinado cuántos núcleos tienen, cuánta memoria y cuánto disco.
3) estudiar cuántas máquinas son necesarias para ejecutar la carga de trabajo. Para ello, es conveniente ejecutar primero las operaciones demandadas por la carga de trabajo de forma aislada y medir y analizar los tiempos. Nota1: Para garantizar la posible variabilidad en las mediciones, deberéis repetir cada medición 10 veces y observar el caso mejor y el caso peor para dimensionar el número de máquinas. Nota2: las máquinas están compartidas por todos vosotros, tendréis que coordinaros para no entorpecer vuestras mediciones.

En caso de que hagan falta varias máquinas para procesar la carga de trabajo, tendréis que utilizar la arquitectura máster-worker. En su versión más sofisticada, además del máster y de los workers existe un componente adicional que es el *pool de recursos*, que consiste en una estructura de datos que contiene todos los workers (procesos, almacenará el pid de cada proceso) disponibles y un proceso controlador que gestiona los recursos. El funcionamiento es el siguiente:

1) Las peticiones de trabajo le llegan al master
2) El máster le envía una petición (*request*) al controlador del pool para solicitarle un *worker* para una tarea.
3) El controlador se lo envía (*reply*) y de alguna manera marca a ese *worker* como ocupado. Notad que en caso de asignar 2 tareas al mismo *worker*, estaríamos secuencializando el procesamiento.
4) De alguna manera el controlador tiene que conocer cuándo un *worker* ha terminado, eso dependerá de cómo sea la interacción con el máster y con el cliente.
5) Además, el controlador podría tener la capacidad de gestionar los *workers* en tiempo de ejecución, esto es, de crear nuevos *workers* y de destruir *workers* existentes en función de la demanda de la carga de trabajo.

Para ello vamos a tener en cuenta los tres escenarios siguientes que realizan peticiones a un servidor.

Se pide

Diseñar la arquitectura software más adecuada para cada escenario y adaptar el código Elixir proporcionado para implementarla. Observar empíricamente que no se viola el *QoS* en cada escenario. En particular, se pide en la memoria:

1. Describir las características técnicas de cada máquina.
2. Analizar gráficamente la carga de trabajo de cada escenario.
3. Analizar el tiempo de ejecución, de forma aislada, de las operaciones que aparecen en la carga de trabajo.
4. Explicar razonadamente qué patrones arquitecturales (cliente-servidor, máster-worker) se podrían usar en cada escenario.

5. Explicar cómo sería una arquitectura software que tuviera que dar soporte a los tres escenarios simultáneamente.

Escenario 1.

Invocación> `cliente(pid, :uno), donde pid es el pid del servidor`

Escenario 2.

Invocación> `cliente(pid, :dos)`

Escenario 3.

Invocación> `cliente(pid, :tres)`

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. **Se considerará esta calificación en aquellos diseños**

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

arquitecturales de máster-worker que consideren el pool de recursos y la creación y destrucción de workers en función de la demanda de la carga de trabajo.

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 / 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	Arquitectura	Código	Memoria
10	A+	A+	A+
9	A+	A	A
8	A+	A	A
7	B	A	A
6	B	B	B
5	B-	B-	B-
suspenso	1C		

Tabla 1. Rúbrica

5. Entrega

Deberéis entregar un fichero zip que contenga: (i) los fuentes: escenario1.ex, escenario2.ex y escenario3.ex 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 día anterior al comienzo de la siguiente práctica, esto es, el 17 de octubre de 2019 para los grupos A y el 25 de octubre de 2019 para los grupos B (21 de octubre para el grupo del martes B).
Durante la segunda sesión de prácticas se realizará una defensa "in situ" de la práctica.

6. Bibliografía

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