Sistemas Distribuidos

Práctica 5, 2ª parte: Servicio de almacenamiento basado en primario/copia

Autor: Unai Arronategui y Víctor Medel

Resumen

Partiendo de lo realizado en la práctica anterior, en esta segunda parte se plantea desarrollar un servicio de almacenamiento distribuido clave/valor utilizando el servicio de vistas implementado.

Los protocolos a diseñar son los correspondientes al esquema de funcionamiento Primario/Copia planteado como ejemplo en la clase de teoría.

Estas prácticas incluyen redactar una memoria, escribir código fuente y elaborar un juego de pruebas. El texto de la memoria y el código deben ser originales. Copiar supone un cero en la nota de prácticas.

Notas sobre esta práctica

- Seguir, en lo posible, la guía de estilo de codificación de Elixir, en especial : fijar en el editor máxima longitud de línea de 80 columnas, como mucho 12 expresiones en una función (salvo situaciones especiales con bloques case y receive con multiples patrones, aunque 2 expresiones máximo por patrón), utilizar paréntesis, en lugar de espacios. Existen diferentes posibilidades de editores con coloración sintáctica: sublimetext, gvim, vim, ...
- La solución aportada deberá funcionar para los diferentes tests suministrados, y los que diseñeis e implementeis vosotros.

1. Objetivo de la práctica

El objetivo de esta práctica es diseñar e implementar un servicio de almacenamiento distribuido clave/valor que sea tolerante a fallos mediante el esquema Primario/Copia implementado en la primera parte de la práctica.

2. El servicio de almacenamiento distribuido clave/valor basado en aproximación primario/copia

La segunda parte de la práctica 5 consiste en utilizar las indicaciones definidas en clase para el ejemplo de servicio de replicación primario/copia. En términos generales, el objetivo es diseñar e implementar tanto el protocolo cliente del servicio clave/valor como el protocolo de replicación entre primario y copia.

Se definen 3 operaciones clave/valor que los clientes del servicio pueden utilizar :

- *lee(clave)* : devuelve el valor asociado a la clave, o cadena vacia ("") si no existe la clave.
- *escribe(clave, nuevo_valor)* : Actualizar valor asociado a *clave* con *nuevo_valor*. Si no existe clave, añade pareja (clave, nuevo_valor). Devuelve *nuevo_valor*.
- *escribe_hash(clave, nuevo_valor)* : *Para pruebas*. Actualiza, en la base de datos, el valor asociado a clave con el valor obtenido de ejecutar la función *hash(antiguo_valor <> nuevo_valor)* en base de datos y devuelve el antiguo_valor. *antiguo_valor* es el valor previo asociado a esa clave si existe, o la cadena vacía ("") si no hay valor previo.

Tanto las claves como los valores son de tipo *String.t*. La función *hash()* está disponible en el modulo *ServidorSA*.

Salvo en el tipo de dato que actualizan y en el valor devuelto, la operación *escribe()* y *escribe_hash()* deberían ser tratadas de forma idéntica (como se sugiere en el esqueleto de código).

Se aconseja la utilización de diagramas de interacción, diagramas de secuencia y diagramas de máquina de estados para modelar las interacciones entre clientes y servidores de almacenamiento, y para la interacción entre los diferentes servidores de almacenamiento con el protocolo de replicación.

Se debe gestionar la semántica "solo una vez" de la peticiones con reintentos y detección de duplicados (mediante secuenciación de peticiones). Los mensajes de propagación de peticiones de clientes entre primario y copia deben ser específicas para distinguirlas de las peticiones cliente al primario (por si se equivoca y lo envía al servidor copia).

La transferencia completa de la base de datos clave/valor a un nuevo nodo copia se puede realizar en un solo envío. El estado relacionado con detección de duplicaciones debe ser también replicado correctamente entre el primario y copia para tolerar correctamente los fallos.

Recomendaciones para los pasos en el desarrollo de la solución :

- En primer lugar, modificación del fichero *servidor_sa.exs* para que cada servidor de almacenamiento envíe latidos al servicio de vistas y pueda conocer la vista tentativa. Una vez que un servidor de almacenamiento conoce la vista tentativa, puede saber si es primario, copia o espera.

- Implementación del procesado ligado a la recepción de las operaciones *lee()* y *escribe()*. Para ello, se pueden almacenar las parejas clave/valor en el primario y la copia con estructura de datos *map* de Elixir. Suponemos que el tamaño del almacen es pequeño (pocos datos) y por lo tanto que suponen un retardo muy pequeño en las operaciones de copia de almacen entre nodos primario y copia.

- Modificación de la operación escribe para que el primario solicite confirmación de actualización al servidor copia antes de responder al cliente.

- Implementación de la copia de la base de datos clave/valor, por parte del primario, cuando un servidor se convierte en copia en una nueva vista.

Se debe tener en consideración que, aunque el servicio de vistas de la práctica 5 haya pasado las pruebas de la parte 1, pueden quedar errores en el programa que provoquen fallos en esta 2ª parte.

2.1. Notas sobre diseño e implementación en Elixir

Para la base de datos clave/valor en RAM se puede utilizar la estructura de datos *map* que teneis disponible en Elixir. Para poder introducir aleatoriedad en los fallos, se puede utilizar la función *uniform/1* del modulo Erlang "rand".

El fichero *cliente_gv.exs* sustituye al fichero *cliente_gv.exs* de la primera parte de la práctica, con algunas modificaciones. Implementa la misma API de interacción con el servidor de vistas, pero, a diferencia de la implementación cliente vistas anterior, no se crea un nodo Elixir nuevo. Esto es debido a que este código será utilizado directamente, por cada servidor de almacenamiento y cada cliente del servicio de almacenamiento, para interaccionar con el servidor de vistas.

Se debe substituir, *solamente*, el fichero *servidor_gv.exs* del directorio *ServidorVistas*, en el código que os ponemos disponible, por vuestro fichero del mismo nombre de la 1ª parte de la práctica 5.

Cualquier fichero adicional de código Elixir (**.exs*) que se hubiese creado para vuestra implementación del servicio de gestión de vistas de la 1ª parte, también debe incluirse en el directorio donde está ubicado todos los fichero de código del servicio de almacenamiento, incluidos los fcheros del servicio de gestión de vistas que le dan apoyo.

2.2. Validación

Junto al guión de la práctica, se provee el esqueleto básico del fichero de validación Elixir *servicio_almacenamiento_tests.exs*, basado en la infraestructura *ExUnit* de Elixir, los ficheros de apoyo *cliente_sa.exs*, *cliente_gv.exs*, esqueleto *servidor_sa.exs* y el fichero shell, *validar_servicio_almacenamiento.sh*, para lanzar ejecución de las pruebas. Los ficheros *nodo_remoto.exs* y *servidor_gv.exs* de la 1ª parte deben ser copiado integralmente en el directorio del código de la 2ª parte. Para este último fichero, unicamente deberiais necesitar adecuar el parámetro de las funciones *send* que enviaban mensajes de vuelta al

:cliente_gv. Ahora, este parámetro debería ser :servidor_sa, como proceso que toma la funcionalidad de cliente_gv.

Tener en cuenta que ejecuciones incabadas de validación pueden dejar VMs Erlang y demonio *epmd* sin eliminar (no completa el shell de jecución). Esto puede interferir en validaciones posteriores. Utilizar comandos "pkill erlƶ "pkill epmd"hasta eliminar todos estos procesos de sistema antes de volver a ejecutar una validación. También en las mauinas remotas en validaciones distribuidas.

En la fase final de desarrollo y validación de vuestra solución, para validar una ejecución realmente distribuida, modificar las direcciones IP de todos los hosts definidos en las constantes del modulo de validación *ServicioAlmacenamientoTest* para ubicar cada nodo de la aplicación en una máquina física diferente. Salvo el maestro, el servidor gestor de vistas y los clientes del servicio de almacenamiento, que pueden ser ubicados en la propia máquina física desde donde se inicia la aplicación.

Se deben tener en consideración la variaciones de latencia que pueden ocurrir en las diferentes configuraciones que se prueben. Adecuar los tiempos de expiración (timeouts) y retardos en la ejecución de algunas operaciones mediante *Process.sleep(milisegundos)* o en sección emphafter de bloques receive.

En el fichero *servicio_almacenamiento_tests.exs* contiene ya un pequeño juego de pruebas que debe ejecutarse correctamente.

Opcionalmente, se pueden implementar juegos de pruebas adicionales que contemplen los siguientes casos:

1. Parada de todos los servidores de almacenamiento y adición de uno nuevo que no debería estar activo...
2. Petición de escritura duplicada por perdida de respuesta (modificación realizada en BD), con primario y copia.
3. Petición de escritura inmediatamente después de la caída de nodo copia (con uno en espera que le reemplace).
4. Escrituras concurrentes de varios clientes sobre la misma clave, con comunicación con fallos (sobre todo pérdida de repuestas para comprobar gestión correcta de duplicados).
5. Comprobación de que un antiguo primario no debería servir operaciones de lectura.
6. Comprobación de que un antiguo primario que se encuentra en otra partición de red no debería completar operaciones de lectura.

Para llevar a cabo esta implementación, se recomienda basarse en el código disponible.

Recordad que, para ayudaros en la depuración, si ejecutais vuestras pruebas mediante el entorno ExUnit, podeis utilizar la función de Elixir *IO.inspect()* y/o la función de Erlang *:io.format()* para las trazas. En los tests se pueden utilizar, también, las macros tipo *assert* que vienen explicadas en la documentacion de ExUnit (ExUnit.Case). Adicionalmente, existe la posibilidad de utilizar la macro *Iex.pry* para depuración interactiva y *:observer.start* para depuración de nodos distribuidos.

3. Criterios de 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 / alguna funcionalidad.
- 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 problemaa través de la ejecución de la bateria de pruebas.
- Todos los programas tienen que seguir la guía de estilo de codificación Elixir.
- Se valorará negativamente una inadecuada estructuración de la memoria, así como la inclusión de errores gramáticales u ortográficos.

```
%% AUTOR: nombre y apellidos
%% NIA: n'umero de identificaci'on del alumno
%% FICHERO: nombre del fichero
%% TIEMPO: tiempo en horas de codificaci'on
%% DESCRIPCI'ON: breve descripci'on del contenido del fichero
```

3.1. 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 rúbrica en el Cuadro 1. Los valores de las celdas son los valores mínimos que hay que alcanzar para conseguir la calificación correspondiente y tienen el siguiente significado:

Calificación	Sistema	Tests	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			

Cuadro 1: Detalle de la rúbrica: los valores denotan valores mínimos que al menos se deben alcanzar para obtener la calificación correspondiente

- 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 funionan 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.

4. Entrega y Defensa

Se debe entregar, tanto la memoria como los ficheros de código fuente, en un solo fichero en formato tar.gz a través de moodle2 en la actividad habilitada a tal efecto. La fecha límite es el *11 de enero de 2016* para los grupos de semanas A y el *15 de enero de 2016*. La evaluación se efectuará en el laboratorio 1.02 el lunes 16 de 12 a 14h para los grupos de semanas A y martes 17 de enero de 15 a 19h para los grupos de semanas B.

El nombre del fichero tar.gz debe indicar apellidos del alumno y nº y parte de práctica.