

Práctica 5: Servicio de almacenamiento basado en primario/copia

Autor: Unai Arronategui y Víctor Medel

Resumen

En esta práctica se plantea desarrollar un servicio de almacenamiento distribuido clave/valor en memoria RAM, tolerante a fallos, utilizando el servicio de vistas implementado en la Práctica 4. Las operaciones que se pondrán a disposición de los clientes serán: lectura de valor y escritura de valor. Los protocolos a diseñar son los correspondientes al esquema de funcionamiento Primario/Copia planteado como ejemplo en la clase de teoría. Se plantearan situaciones de fallo adicionales, como por ejemplo, el envío de escrituras duplicadas o la pérdida de mensajes.

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

- Se debe 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: gedit, sublimetext, gvim, vim, ...
- La solución aportada deberá funcionar para los diferentes tests suministrados, y los que diseñéis e implementéis vosotros. Tener cuidado con los errores ligados a las temporizaciones (retardos ejecución y comunicación, y tiempos de expiración), son complicados de resolver.

1. Objetivo de la práctica

El objetivo de esta práctica es diseñar e implementar un servicio de almacenamiento distribuido clave/valor, en memoria RAM, que sea tolerante a fallos mediante el esquema Primario/Copia planteado en clases de teoría, y apoyandose en la implementación del servicio de vistas de la práctica 4.

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

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 tres operaciones clave/valor que los clientes del servicio pueden utilizar :

- *lee(clave)* : devuelve el valor asociado a la clave, o cadena vacía (“”) 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 máquina de estados, diagramas de interacción y diagramas de secuencia 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.

Para el desarrollo de la solución, se indican las siguientes recomendaciones :

- 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 supone 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 4 haya pasado las pruebas, pueden quedar errores, en dicho programa, que provoquen fallos en esta práctica.

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.

Solamente se debe sustituir el fichero *servidor_gv.exs* del directorio *ServidorVistas*

del código suministrado, por el fichero del mismo nombre desarrollado en la Práctica 4. Cualquier fichero adicional de código Elixir (**.exs*), que se hubiese creado para la implementación del servicio de gestión de vistas de la práctica, debe incluirse en el directorio donde está ubicado todos los fichero de código del servicio de almacenamiento, incluidos los ficheros 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 copiados 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`.

Se debe tener en cuenta que ejecuciones inacabadas (por errores) de validación pueden dejar VMs Erlang y demonio *epmd* sin eliminar (no completa el shell de ejecución). Esto puede interferir en validaciones posteriores. Utilizar comandos "pkill beam; pkill epmd" hasta eliminar todos estos procesos de sistema antes de volver a ejecutar una validación. Aplicarlo, también, en las máquinas remotas para validaciones distribuidas.

En la fase final de desarrollo y validación de vuestra solución, se debe contemplar una ejecución realmente distribuida. Para ello, se tiene que modificar las direcciones IP de todos los hosts definidos en las constantes del módulo 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.

Tener cuidado con la variaciones de latencia que pueden ocurrir en las diferentes configuraciones que se prueben. Pueden dar errores complicados a detectar y resolver. Los **tiempos de expiración** (timeouts) y **retardos** en la ejecución de algunas operaciones deben adecuarse correctamente mediante *Process.sleep(milisegundos)* o en sección *after* de bloques *receive*.

El fichero *servicio_almacenamiento_tests.exs* contiene ya el siguiente juego de pruebas que debe ejecutarse correctamente.

1. Arranque correcto de máquinas.

2. La operación de escritura se realiza correctamente
3. Escrituras concurrentes y comprobación de consistencia tras caída de primario.

Se pide implementar los siguientes test adicionales:

4. Escrituras concurrentes y comprobación de consistencia tras caída de primario y copia. Se puede gestionar con cuatro nodos o con el primario rearrancado.
5. Petición de escritura inmediatamente después de la caída de nodo copia (con uno en espera que le reemplace).
6. Petición de escritura duplicada por pérdida de respuesta (modificación realizada en BD), con primario y copia.
7. Comprobación de que un antiguo primario no debería servir operaciones de lectura.

La superación de las pruebas 1 a 3 supone la obtención de una nota B en la parte correspondiente a test. Para obtener una calificación de A, se debiera superar las pruebas 4 y 5. La superación de los test 6 y 7 supone tener una calificación de A+. Para llevar a cabo esta implementación, se recomienda basarse en el código disponible.

Como ayuda a la depuración, si las pruebas se ejecutan mediante el entorno ExUnit, se puede utilizar la función de Elixir *IO.inspect()* y/o la función de Erlang *:io.format()* para obtener trazas. En los tests se pueden utilizar las macros tipo *assert* que vienen explicadas en la documentación 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 funcionar correctamente como se especifica en el problema a través de la ejecución de la batería de pruebas.
- Todos los programas tienen que seguir la guía de estilo de codificación Elixir. Debe utilizarse el formateo estándar mediante "**mix format** <ficheros>"
- Se valorará negativamente una inadecuada estructuración de la memoria, así como la inclusión de errores gramaticales u ortográficos.

Calificación	Sistema	Tests	Código	Memoria
10	A+	A+ (1-7)	A+	A+
9	A+	A+ (1-7)	A	A
8	A	A (1-5)	A	A
7	A	A (1-5)	B	B
6	B	B (1-3)	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

- La memoria debería incluir diagramas de máquinas de estado y/o diagramas de secuencia para explicar los protocolos.
- Cada uno de los servidores de almacenamiento debe ejecutarse en una máquina física diferente.

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, en el Cuadro 1 se propone una rúbrica. Los valores de las celdas indican el mínimo que hay que alcanzar para conseguir la calificación correspondiente. Su significado es el siguiente:

- 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 / 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 *manifestamente* 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 (longitud máxima de 8 páginas) 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 *13 de diciembre de 2019*, a las 8h de la mañana, para los grupos de semanas A y el *10 y 14 de enero de 2020*, a las 8 de la mañana, para los grupos de semanas B. La evaluación se efectuará en el laboratorio 1.02 el viernes 14 de diciembre para los grupos de semanas A y viernes 11 de enero para grupos de semanas B.

La entrega debe contener los diferentes ficheros de código Elixir y la memoria, en formato pdf. El nombre del fichero tar.gz debe indicar apellidos del alumno y nº y parte de práctica. Aquellos alumnos que no entreguen la práctica a través de moodle2 no serán calificados.