Universidad de Zaragoza
Grado en Ingeniería Informática

Trabajo Fin de Grado (TFG)

Validación de la resiliencia de sistemas distribuidos mediante Chaos Engineering

Distributed systems resiliency validation using Chaos Engineering

Autor
Sergio Torres Castillo

Coordinador
Miguel Julián Ramos

Ponente
Unai Arronategui Arribalzaga

Año Académico 2021-2022

Resumen

Este proyecto tiene como fin **validar la resiliencia de los sistemas distribuidos mediante** una nueva disciplina de pruebas llamada **Chaos Engineering**. Los sistemas distribuidos son sistemas complejos que necesitan ser sometidos a pruebas de resiliencia para asegurar que están en buen estado o poder encontrar los problemas fácilmente para corregirlos más adelante. En esto ayuda Chaos Engineering, **simulando condiciones turbulentas en sistemas distribuidos** para generar caos de igual forma que podría ocurrir ante acontecimientos imprevistos en nuestros sistemas.

Sysdig es una empresa dedicada a la monitorización y seguridad en la nube y será el entorno de trabajo de este proyecto, dentro del departamento de calidad del software. Su servicio está construido sobre un sistema distribuido, por lo que se usará este escenario para simular ahí las condiciones turbulentas y **comprobar su resiliencia** ante una serie de **inyección de fallos** en algunos componentes que forman parte del sistema.

Se definen varios casos de uso que describen varias hipótesis sobre cómo debería de comportarse nuestro sistema ante **caídas de servicio** o **actualizaciones de componentes**. Cada uno de estos casos de uso prueban distintos componentes de nuestro sistema, concretamente los ***datastores*** Redis, NATS Streaming y Kafka, en ese orden y de forma incremental de dificultad. Estos componentes forman parte importante del servicio de Sysdig y por eso validar que son resilientes para este tipo de condiciones resulta de importancia para asegurar una buena calidad de servicio.

Para la ejecución de las pruebas de NATS Streaming y Kafka se utiliza el **framework de Chaos Engineering *Chaos Toolkit***, que abstrae el diseño, implementación y ejecución de las pruebas de forma sencilla y extensible para los distintos casos de uso que se añaden.

Finalmente, se **validan los resultados** para cada uno de los casos de uso, especificando los problemas que se han encontrado y si se ha encontrado alguna solución. Además, después de haber realizado las pruebas y tras comprobar los resultados obtenidos, se concluye si Chaos Engineering es una buena opción a largo plazo para seguir trabajando en la empresa.

Índice

1. Introducción

1.1. Contexto

Este documento hace referencia a la realización de un trabajo de fin de grado (TFG) de la carrera de Ingeniería Informática en la **Universidad de Zaragoza**, dentro de la rama de *Tecnologías de la Información*. La labor principal de este trabajo trata sobre mejorar la resiliencia de los sistemas distribuidos mediante una técnica de pruebas innovadora, llamada ***Chaos Engineering***, pilar fundamental del proyecto.

En este punto entra **Sysdig**[1], una empresa orientada a la monitorización y seguridad en la nube. Dentro de su departamento de calidad, entorno de trabajo de este TFG, se busca que el producto o servicio que se ofrece (Sysdig como producto, anexo 8.1) tenga unos estándares de calidad mínimos para el cliente final.

Este TFG estudia Chaos Engineering de forma teórica-práctica usando un entorno real y orientado al cliente final, por lo que es el escenario perfecto para comprobar si esta tecnología es útil, no solo de forma general, sino también para estudiar si merece la pena seguir trabajando en pruebas de Chaos como una solución de *Quality Assurance* (a partir de ahora, QA) a largo plazo dentro de la empresa.

1.2. Motivación

En una época donde los servicios en la nube son más relevantes que nunca, utilizados en gran medida por la sociedad en Internet, surge la necesidad de monitorizar y asegurar el buen funcionamiento de nuestros sistemas de forma continua. Las empresas quieren estar mas preparadas ante condiciones turbulentas en producción y para ello acuden a experimentar con nuevas tecnologías de pruebas, con el objetivo de encontrar debilidades en sus sistemas de forma más rápida en entornos tan complejos como lo son los sistemas distribuidos.

Investigar sobre esto resultará de gran ayuda para entender cómo Chaos Engineering puede ayudar en la mejora de la resiliencia de los sistemas distribuidos de cualquier servicio en la nube en un contexto en el que cada vez es más difícil encontrar alternativas eficientes para realizar este tipo de pruebas.

1.3. Objetivos

El objetivo principal de este proyecto es mejorar la resiliencia de los sistemas distribuidos. La resiliencia de éstos es un factor muy importante a tener en cuenta y depende de muchos factores, ya que existen varios puntos de fallo.

Para ayudar en esta tarea, se estudia *Chaos Engineering*. Esta filosofía consiste en probar la resiliencia de nuestros sistemas a través de la inyección de fallos de forma premeditada

[1] https://sysdig.es/

en un entorno de producción, o como dice la definición: *La ingeniería del caos es la disciplina de experimentar en un sistema distribuido para generar confianza en la capacidad del sistema para soportar condiciones turbulentas e inesperadas en la producción* de tal forma que puede ocurrir algo que no esperábamos, siendo esto el objetivo. Esto ayuda a saber de forma más plausible las debilidades de nuestros sistemas, que de otra forma no podíamos haber conocido, siendo capaces entonces de tomar acciones ante ello.

El resultado de esta investigación servirá también para decidir si esta disciplina merece ser implementada a largo plazo en Sysdig, como uno de los procesos de QA en la empresa.

1.4. Metodología

Para la realización de este trabajo se ha empezado con la tarea de investigación. Todo lo relacionado con la disciplina de Chaos Engineering se ha estudiado a fondo como primer paso para entender qué es y cómo se utiliza, así como de qué manera lo utilizan las empresas para mejorar la resiliencia de sus sistemas y detectar debilidades. Después de esto se ha pasado a implementar de forma práctica la disciplina en Sysdig con su producto, realizando las pruebas en un entorno real y recogiendo los resultados. Tras estas pruebas se toman las conclusiones pertinentes sobre si se han cumplido los objetivos y si ha merecido la pena el esfuerzo en dicho trabajo.

Esta memoria refleja todo este proceso cronológicamente, en una rampa ascendente de dificultad, pasando por definir unos requisitos sobre qué es lo que se quiere probar, seguidamente con el diseño e implementación de las distintos casos de uso que se quieren llevar a cabo para comprobar la resiliencia del sistema con varias tecnologías de bases de datos (a partir de ahora, *datastores*), y finalmente realizando la validación de la resiliencia del sistema estudiando si se han cumplido los objetivos con la ejecución de las pruebas de Chaos.

Por último, se han seguido las pautas e instrucciones que recomienda la Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza (más conocida como *EINA*) para la elaboración de la memoria del TFG [1].

2. Estado del arte

Los **sistemas distribuidos** [3] se han convertido en la arquitectura más atractiva a la hora de construir infraestructuras basadas en la nube. Estos sistemas tienen diversas ventajas respecto a los sistemas centralizados, los cuales se basan en que todo el trabajo se realiza en una sola máquina en una única ubicación. En un sistema centralizado, una misma máquina maneja todas las peticiones de los clientes, lo que puede llevar a sobrecarga y consecuentemente al fallo del servidor.

En un sistema distribuido existen varias máquinas que pueden estar repartidas en ubicaciones distintas con el objetivo de repartir la carga de trabajo entre los clientes y en el caso de fallo en una de las máquinas o nodos del sistema, siempre podremos manejar esas peticiones para que se redirijan a otra máquina funcionando al mismo tiempo, eliminando el punto único de fallo de los sistemas centralizados. Estos sistemas también pueden utilizarse como una forma de repartir las conexiones de los clientes para leer de una base de datos, por ejemplo, aportando concurrencia de lectura o incluso escritura de datos (concurrencia de lectura/escritura). Además, si un sistema distribuido se satura, siempre se puede escalar tanto vertical como horizontalmente, añadiendo más máquinas para aumentar la capacidad del sistema para aceptar nuevas peticiones.

Aun así, los sistemas distribuidos al ser más complejos son más difíciles de mantener. Esto es así porque existen más interacciones entre los distintos componentes, los datos no están en un mismo sitio y hay que tener en cuenta su sincronización para replicación de datos, o la fragmentación de datos para una escritura concurrente en el que una BBDD se encuentra fragmentada en distintos nodos distribuidos, etc.

Por otro lado tenemos los sistemas basados en **microservicios**. Éstos también son sistemas distribuidos, ya que descompone una aplicación en distintas partes desacopladas aunque dependientes entre sí hasta cierto punto. Esto hace que cada componente tenga más independencia a la hora de realizar cambios en su software (actualizaciones) y que en el caso de fallo en uno de los componentes de nuestra aplicación repercuta lo menos posible en el estado global del sistema.

De hecho, ningún sistema distribuido está a salvo de las fallas de red. Un punto muy importante en este sentido es el **Teorema CAP** [4]. Este teorema expone que es imposible que se garanticen más de 2 de las 3 siguientes características en un sistema distribuido de forma simultánea: Consistencia, Disponibilidad y la Tolerancia al particionado. La consistencia hace referencia a que un sistema sea capaz de leer siempre el último dato escrito o por el contrario devuelva un error. La disponibilidad hace referencia a que la respuesta a una petición siempre se efectúa (un sistema siempre devuelve un dato no erróneo). La tolerancia al particionado consiste en la capacidad del sistema para seguir funcionando si varios mensajes se pierden o retrasan en la red por cualquier motivo.

Por otro lado tenemos los llamados orquestadores de sistemas distribuidos, uno de ellos

es **Kubernetes** (más conocido como K8s). K8s es un sistema de código abierto para automatizar el despliegue, el escalado y la gestión de aplicaciones en contenedores de forma distribuida. Los contenedores [5] son una forma de virtualización del sistema operativo (*SO*), con la diferencia de que utilizan la misma arquitectura del equipo anfitrión para su ejecución. Un solo contenedor se puede usar para ejecutar desde un microservicio o proceso de software a una aplicación de mayor tamaño. Dentro de un contenedor se encuentran todos los ejecutables, el código binario, las bibliotecas y los archivos de configuración necesarios, pero sin los binarios y archivos del sistema operativo, que son innecesarios ya que se utilizan los del propio sistema anfitrión o host, consiguiendo así un mayor rendimiento.

K8s puede albergar una infraestructura entera con sus distintos microservicios que trabajan conjuntamente para una aplicación global (como por ejemplo una página web, con sus componentes de BBDD, backend, frontend, etc.), todo ello de forma semi-autogestionada, ya que K8s guarda el estado tanto del sistema en su conjunto así como el de sus componentes o microservicios. Esto ayuda a la hora de automatizar los reinicios de un componente cuando este falla, el autoescalado cuando existe saturación, autobalanceo de datos, etc.

También es necesario conocer la diferencia entre los sistemas **SaaS** (*Software as a Service*) y **on-premise** (traducido como *en las instalaciones propias*). Un sistema SaaS es aquel que se pone a disposición de los clientes como un servicio en la nube, en el que el usuario común solo tiene que acceder sin realizar ningún tipo de instalación, simplemente con una conexión a Internet. En un sistema SaaS, la infraestructura ya se encuentra disponible y es desplegada por parte del dueño del servicio, dejando todas las tareas de mantenimiento y control a dicho dueño. Sin embargo, un sistema on-premise consiste en que son los clientes los que instalan dicha aplicación en su propia infraestructura, lo que le da al usuario más control sobre la aplicación pero él es el que se hace cargo del mantenimiento de la infraestructura.

Normalmente un sistema on-premise es elegido a favor del sistema SaaS cuando el cliente quiere tener más control sobre los datos y quiere mas seguridad al respecto, ya que en un sistema SaaS los datos se encuentran en el dominio del dueño de la infraestructura y los datos de los distintos clientes se guardan en el mismo sitio.

En el ciclo de vida de un producto o servicio, **CI/CD** [6] se encarga de automatizar las etapas de desarrollo, éstas son la integración y distribución continuas (aunque también puede referirse a la implementación continua). Dejando a un lado la semántica, la integración continua hace referencia a la automatización de la primera etapa en el desarrollo de la aplicación, diseñando, probando y combinando los cambios nuevos en el código de la aplicación con regularidad en un repositorio compartido.

La siguiente etapa, la distribución continua, hace referencia a los cambios que implementa un desarrollador en una aplicación para que se carguen en un repositorio para luego pasar a producción, además de volver a lanzar pruebas que validen los cambios. A veces se habla también de la implementación continua conjuntamente con la distribución continua, ya que hace referencia al lanzamiento automático de los cambios que implementa el desarrollador

desde el repositorio hasta la producción, para ponerlos a disposición de los clientes.

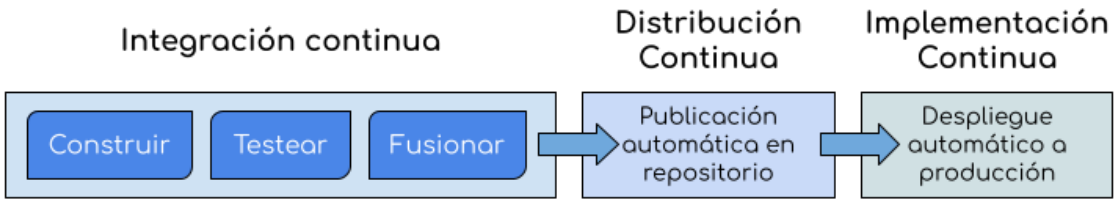


Figura 1: Etapas del ciclo de desarrollo de un producto con CI/CD

En la práctica hay empresas que implementan un canal CI/CD en el ciclo de desarrollo de sus productos, empezando con la implementación de CI y seguidamente acaban automatizando sus últimas etapas con la distribución e implementación continuas.

Para comprobar la resiliencia de los sistemas distribuidos es necesario realizar un buen conjunto de pruebas, no solo del mismo tipo, también tener en cuenta que existen distintas capas en el dominio de prueba, y se deben de probar cada una de ellas. Aquí entra la **pirámide de los tests**, en ella se aprecia (ver Figura 2) los distintos tipos de pruebas que se deben de hacer en el desarrollo de una aplicación. El grueso son los test unitarios, que son los de más bajo nivel y se encargan de comprobar que nuestro código (aparte de estar auto documentado con la elaboración de estos test) hace lo que se espera que haga y falla cuando debería. Estos test deberían de ser los más abundantes ya que de esta forma se evita que los fallos puedan ir hacia niveles de abstracción más altos y por tanto sean más difíciles de tratar.

Luego tenemos los test de integración, que testean las funcionalidades y servicios/componentes de la aplicación (siempre desde un punto de vista aislado del resto de la aplicación global). Y finalmente los test end-to-end (conocidos como E2E), que testean la aplicación punto a punto a través de la interfaz de usuario, ya sea con el testeo de las funcionalidades y servicios de la aplicación pero de forma que se compruebe su impacto en la aplicación de forma global.

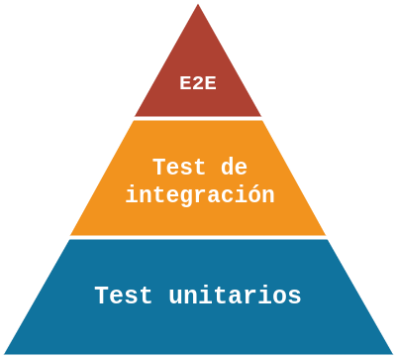


Figura 2: Pirámide de los tests

Cuando un sistema decimos que debe ser resiliente, también nos podemos referir a que tenga un buen rendimiento en temas de calidad del sistema como las condiciones de escalado, fiabilidad y uso de los recursos. Para testear estos requisitos se utilizan las pruebas de rendimiento (***Performance testing***) [7], con el objetivo de mejorar el rendimiento global de nuestra aplicación en materia de diseño, antes incluso de la implementación del código.

Por otro lado, las empresas también buscan **regresiones**, esto es asegurar que los casos de prueba que ya habían sido probados y fueron exitosos permanezcan así ante evoluciones y cambios funcionales en el código. Para cumplir con este objetivo se hacen las llamadas *Pruebas de regresión* [8]. Estas pruebas se suelen automatizar ya que se ejecutan por cada nueva liberación del software, lo que implicaría mucho esfuerzo lanzarlas de forma manual.

Sin embargo, existen nuevas técnicas que han surgido recientemente con el objetivo de probar la resiliencia de nuestros sistemas de forma más efectiva. Una de estas técnicas es la filosofía de ***Chaos Engineering***, objeto principal de estudio de este proyecto. Podemos definir Chaos Engineering como *la disciplina de experimentar en un sistema distribuido para generar confianza en la capacidad del sistema para soportar condiciones turbulentas en la producción*. Además [9], hoy en día, gran parte del software que comprende la infraestructura es de código abierto, y que a su vez se compone de muchas partes diferentes (componentes), lo que hace que el comportamiento sea a veces impredecible. *Chaos Engineering* es un intento de reconocer este hecho y desarrollar software en consecuencia.

El concepto de Chaos Engineering es relativamente nuevo, es por ello que no hay muchas herramientas dedicadas a esto. Aún así existen algunas que son dignas de estudio. La principal es Chaos Monkey[2], que se puede considerar la precursora del Chaos Testing (rama principal de Chaos Engineering). Fue creada por Netflix con el objetivo de buscar una nueva forma de detectar errores que no se conocían en su infraestructura y poder mejorar la calidad de servicio de sus clientes. Su principal característica es poder realizar caídas de componentes de Netflix (failovers) de forma automatizada y de esta forma comprobar si otros componentes críticos para el usuario final, así como las funcionalidades principales de la plataforma quedan inalteradas o su impacto es mínimo en la experiencia del usuario final.

Existen muchos **frameworks** [10] para realizar Chaos Engineering, algunos de los más conocidos son Chaos Toolkit[3] y Chaos Mesh[4]. De forma breve, Chaos Toolkit es uno de los frameworks de referencia con una gran cantidad de casos de uso. Escrito en Python y ejecutado de forma externa a la plataforma que se prueba, está basado en plugins extensibles para integrarse con las pruebas, como por ejemplo el plugin de K8s para realizar acciones como ejecuciones de comandos en contenedores de *pods* (anexo 8.2). Por otro lado, Chaos Mesh es un framework centrado en Kubernetes. Su funcionamiento se basa en un conjunto de operadores desplegados sobre el propio clúster donde se ejecutan los experimentos. A

[2]https://netflix.github.io/chaosmonkey/
[3]https://chaostoolkit.org/
[4]https://chaos-mesh.org/

diferencia de Chaos Toolkit, Chaos Mesh está diseñado para ejecutar pruebas de más larga duración, donde el caos se genera de forma continuada durante un tiempo para su evaluación.

El hecho de que se realicen estas **pruebas en producción** asegura que el sistema sea resiliente ante fallos durante el uso de la aplicación, de cara al cliente, ya que la simulación de los fallos pueden impactar directamente a la experiencia del usuario final. Esto es importante ya que de esta forma se consigue tener una concepción lo más cercana posible o lo que ocurriría si no se provocaran los fallos intencionadamente. En ocasiones se hacen en alguna etapa previa del ciclo de desarrollo como podría ser en **pre-producción**, aunque no aprovecharíamos el potencial de Chaos Engineering en su totalidad. A pesar de esto, es una alternativa viable ante la imposibilidad, difícil implementación o simplemente no se quiere correr el riesgo de impactar en la experiencia del usuario en la rama de producción final.

3. Análisis de requisitos

Como se ha explicado anteriormente, nuestro objetivo es hacer que los sistemas de Sysdig sean más resilientes probando en sí su resiliencia para encontrar errores. Y no solo poniendo el foco en unos componentes en concreto, que serán objeto de pruebas más adelante, sino que se estudiará el impacto que tienen estas condiciones de fallo de estos componentes en todo nuestro sistema, ya que cada uno de los servicios en un entorno de desarrollo como Sysdig requieren de muchos microservicios conectados entre si, compartiendo flujos de datos y que en muchas ocasiones son dependientes entre sí.

Existen muchos motivos por los que se quiere comprobar la resiliencia de nuestros sistemas, de forma más específica se tiene que tener en cuenta que se está realizando la transición hacia sistemas CD (Continuous Delivery) partiendo iniciamente desde sistemas CI (Continuous Integration). Esto quiere decir que después de realizar CI y cada vez que hay nuevas versiones del software, éstas se fusionan a producción de forma automatizada. Esto causa que los microservicios que se ven modificados por los desarroladores para realizar actualizaciones, por ejemplo, se deban de reiniciar. Esto último es importante, porque cada vez que se debe de actualizar un micorservicio ésto se traduce en failovers que deben de ser controlados recuperados sin errores.

Nuestro sistema ha pasado de ser monolítico a ser un sistema basado en microservicios, esto quiere decir que la complejidad de todo el sistema aumenta, porque hay más interacciones entre componentes y cada uno de los microservicios, aunque estén desacoplados los unos de los otros, tengan cierta relación con los demás cuando se habla de comunicación ante ciertas operaciones cómo funcionamiento normal del servicio. Esto es relativamente nuevo en Sysdig, y se desconoce que ocurre ante ciertas situaciones como las caídas de algunos microservicios y cómo podrían afectar al resto de ellos o incluso al estado global del servicio de Sysdig.

Además, históricamente se han detectado fallos relacionados con actualizaciónes de componentes (microservicios), caídas de servicios, recuperación de componentes a un estado distinto con el que dejaron de estar disponibles, problemas con la tolerancia a fallos, etc. Es por ello que se desea seguir investigando en estos temas.

Por otro lado, la comunicación entre los desarrolladores de los distintos microservcios de Sysdig es un requisito fundamental para elaborar sistemas que sean resilientes. El trabajo de cada uno de los grupos encargados del desarrollo de cada componente es relativamente aislado, por lo que resulta en una falta de información del comportamiento de componentes circundantes y que tienen relación de uso con el que se está trabajando en ese momento. Es por esto que a veces no se tienen en cuenta ciertas interacciones entre los distintos componentes del sistema y esto puede dar lugar a errores desconocidos hasta ahora.

Dentro del departamento de QA se elaboran distintas pruebas para asegurar la calidad del servicio de Sysdig, pero no son suficientes y continuamente se necesitan no solo más

pruebas sobre una misma metodología, como son las pruebas de rendimiento o las pruebas de regresión (que buscan la causa de errores ya conocidos, carencias de funcionalidades o divergencias funcionales), sino también probar nuevas metodologías de pruebas que cubran nuevos puntos de fallo.

Dicho esto, se va a comprobar la resiliencia del sistema a través de varios casos de uso, cada uno de ellos definido para comprobar que una parte del sistema es resiliente ante fallos, y que está relacionado con un microservicio en concreto (en esencia se experimenta con varios *datastores*), ver si el impacto de algunas condiciones turbulentas en estos componentes afectan al estado global del sistema o simplemente si la degradación y recuperación de ciertos componentes es como se esperaba en un principio.

En primer lugar se va a comprobar la **resiliencia con Redis**, un gestor de BBDD no relacional usado para almacenar ciertos datos en Sysdig y que forman parte del flujo de datos de funcionamiento del servicio principal. Como pequeño contexto, Sysdig dispone de un clúster de Redis de *alta disponibilidad*, a partir de ahora *HA*, además de aportar tolerancia a fallos. Por tanto, el objetivo es comprobar si dicha tolerancia a fallos es efectiva y de si su comportamiento es el esperado, además de averiguar si existen nuevos puntos de fallo que no se conocían previamente. Para ello se comprobará que cuando alguna réplica de Redis deja de estar disponible, ya sea por una caída de servicio (failover de uno o varios nodos del clúster) o por una actualización en el microservicio (actualización de uno o varios nodos del clúster) no hay inconsistencias en el clúster una vez que se recuperan los nodos.

En segundo lugar se va a comprobar la **resiliencia con NATS Streaming**, una tecnología de cola de mensajes que hace de intermediario para algunos flujos de datos dentro de Sysdig. Cuando Sysdig se instala en modo HA, consecuentemente se instala un clúster de 3 nodos de NATS Streaming, por lo que se desea comprobar también que pasa cuando algún nodo deja de estar disponible como en el caso anterior. El objetivo es comprobar que el estado del clúster sigue estando saludable y no existen inconsistencias antes y después del suceso. Por otro lado, se quiere comprobar que la lectura y escritura de datos es posible antes, durante y después de la caída/actualización. NATS Streaming debería de permitir leer y escribir con una mayoría de nodos (2/3) activos, por lo que se tendrá en cuenta a la hora de hacer las pruebas.

En tercer y último lugar se va a comprobar la **resiliencia con Kafka**, otra tecnología de cola de mensajes similar al caso anterior, aunque su funcionamiento es más complejo. Este microservicio se instala también en modo de HA, formado por un clúster de 3 nodos Kafka, también conocidos como *brókers*, junto con Zookeeper (ver anexo 8.6.1) en modo HA, que recoge información sobre el clúster de Kafka y su estado, formado de igual forma por 3 nodos Zookeeper para aportar tolerancia a fallos y HA. El objetivo es comprobar que ocurre cuando algunos brókers de Kafka dejan de estar disponibles simulando una caída de servicio o una actualización. De igual forma que en casos anteriores, se comprobará que el estado del clúster sigue siendo saludable antes y después, y que Sysdig puede seguir leyendo datos de la cola de Kafka antes, durante y después de esta condición, siempre que sea posible.

Se comprobará que cada componente llega a un estado saludable en un período de tiempo razonable. Dicho período de tiempo será analizado de forma individual para cada uno de los componentes y casos de fallo que se encuentren.

Como punto final, tras haber comprobado la resiliencia de estas tres tecnologías, se analizará la forma en que se ha hecho y su eficiencia a la hora de encontrar errores. En esencia, si puede servir de utilidad para la empresa y si puede aportar más que otras alternativas que puedan haber en el mercado. Desde QA, estos tipo de estudios son regulares ante la necesidad de búsqueda de nuevas técnicas para aumentar la eficiencia de nuestras pruebas de calidad y de encontrar nuevos fallos ante un sistema tan cambiante.

4. Diseño e implementación

Para satisfacer los requisitos mencionados anteriormente, y como ya se sabe, se ha decidido utilizar la disciplina de *Chaos Engineering*, con el objetivo de asegurar la resiliencia del sistema. Para ello se diseñarán e implementarán pruebas de *Chaos Testing* para construir y ejecutar las pruebas que cubren los requisitos mencionados previamente en los casos de uso.

Antes de pasar al diseño e implementación de las pruebas, se va a explicar cómo se diseña una buena prueba de Chaos, en qué consisten y qué etapas tienen de forma genérica para entender los próximos pasos.

4.1. Diseño de una prueba de Chaos

Para definir una prueba de Chaos tenemos que tener en cuenta qué es el estado **estable del sistema** (o *steady-state*). Consiste en el estado que debería de tener un sistema saludable en todo momento. Si el estado estable cambia, puede significar que hay una debilidad en el sistema, porque no es el esperado.

Por otro lado tenemos la **hipótesis** de una prueba. Una hipótesis de Chaos consiste en realizar una suposición sobre lo que se cree que ocurrirá durante el trascurso de una condición turbulenta en nuestro sistema. A efectos prácticos, la hipótesis es la base sobre la que se construye la prueba de Chaos.

También es necesario conocer el concepto de ***hipótesis continua***, que no es más que comprobar que el estado estable no cambia de forma continua durante todo el experimento. Este concepto será clave en algunas próximas pruebas.

Dicho esto, los experimentos de Chaos constan de 4 pasos principales [2]:

1. Definir el comportamiento normal del sistema (su estado estable o *steady-state*) en función de resultados que se puedan medir como el rendimiento y estado general, las tasas de error, la latencia, lectura y escritura, etc.
2. Formular la hipótesis sobre el comportamiento del estado estable de un grupo experimental, en comparación con un grupo de control estable. A efectos prácticos, un grupo de control estable puede ser el estado del sistema previo al experimento, en el que el sistema funciona con normalidad, y el grupo experimental puede referirse al estado del sistema durante o después del experimento.
3. Exponer al grupo experimental a eventos simulados del mundo real, como fallos en el servidor (failovers, actualizaciones, fallos de rendimiento, particionados de red...), respuestas mal formadas, picos de tráfico, etc.
4. Probar la hipótesis comparando el estado estable del grupo de control y el grupo experimental. Cuanto más pequeñas sean las diferencias, más confianza tendremos en que el sistema es fiable y, por tanto, resiliente ante dichas condiciones turbulentas.

4.2. Caso de uso 1: Redis

Antes de pasar al diseño y la implementación de las pruebas de resiliencia, se debe de conocer el funcionamiento básico de Redis y qué versión se utiliza en Sysdig para HA y tolerancia a fallos para entender los próximos pasos. En nuestro contexto se utiliza la versión de *Redis Sentinel* (ver anexo sección 8.4 para más información sobre este componente) para cumplir con estas necesidades.

En esencia se tienen 3 nodos de Redis formando un único clúster, cada uno de estos nodos están formados por 2 componentes: Redis y Sentinel (por tanto tenemos 3 Redis y 3 Sentinel). Redis es la máquina que guarda los datos (dispone de las BBDD) y éstos se replican continuamente a las otras máquinas de Redis (réplicas), por lo que tenemos un Redis maestro y 2 Redis que hacen la labor de replicar la información del maestro.

Por otro lado, Sentinel no es más que el componente que materializa la toleracia a fallos y la alta disponibilidad, ya que éste se encarga de iniciar la votación con los otros Sentinel (a través del algoritmo RAFT, ver anexo sección 8.3 para más información sobre cómo funciona el algoritmo de elección de nodo maestro) para elegir al Redis maestro tras la inicialización del clúster o tras la ocurrencia de fallos en el servidor (como un failover).

Además, el nodo de Redis maestro será el que acepte todas las peticiones de los clientes para leer y escribir, sin embargo, para que un cliente intercambie información con Redis, antes deberá de contactar con un Sentinel para que éste le diga cuál de los 3 nodos de Redis es el nodo maestro y poder iniciar la comunicación con él.

4.2.1. Diseño

Para comprobar la resiliencia de nuestro sistema se diseñará una prueba de Chaos que termine un nodo de Redis con el rol de maestro y otro nodo de Redis con el rol de réplica y comprobar que se cumplen los requisitos. Esta prueba se trata de una prueba de integración, porque estamos comprobando la resiliencia de Redis en un contexto que se encuentra fuera del clúster de Sysdig. Más adelante la complejidad de las pruebas irá aumentando, pero por ahora este será nuestro punto de partida.

Para ello, y tal y como se ha explicado en el diseño de una prueba de Chaos, primero se debe de definir el **estado estable del sistema**. Éste consistirá en que el clúster de Redis tiene sus 3 nodos activos y funcionando correctamente y que todos los Sentinel están de acuerdo en qué nodo de Redis es el maestro.

La **hipótesis** es la siguiente: si matamos el nodo maestro y un nodo réplica, al cabo de cierto tiempo y tras la recuperación de los nodos caídos, los 3 deberán de volver a funcionar correctamente y ponerse de acuerdo en la elección de un nodo maestro.

El diseño de la prueba de Redis se puede ver en la Figura 3.

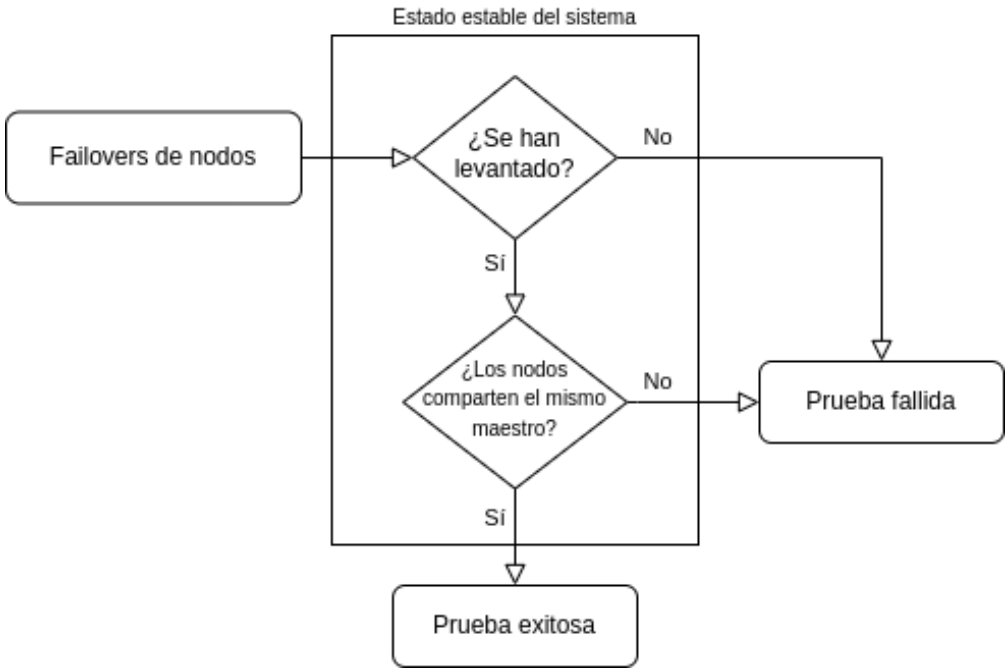


Figura 3: Diseño de la prueba para el caso de uso 1

4.2.2. Implementación

Para implementar la prueba del diseño anterior, primero se tiene que tener en cuenta que ya existían pruebas para comprobar la resiliencia de Redis en Sysdig, aunque no como la que se quiere llevar a cabo, por lo que se reutilizará esta infraestructura para construir y ejecutar las nuevas pruebas.

Estas pruebas están construidas con Bats[5], un framework que utiliza bash como lenguaje de programación para la construcción y ejecución de pruebas. Éstas se ejecutan en un clúster local de K8s desplegado por kind[6] (ver anexo 8.2.2), donde se instala el clúster de Redis como tal (empaquetado por Bitnami[7]) con el gestor de paquetes de K8s Helm[8]. En la Figura 4 se puede ver la implementación del proceso.

[5]https://github.com/bats-core/bats-core
[6]https://kind.sigs.k8s.io/
[7]https://bitnami.com/
[8]https://helm.sh/

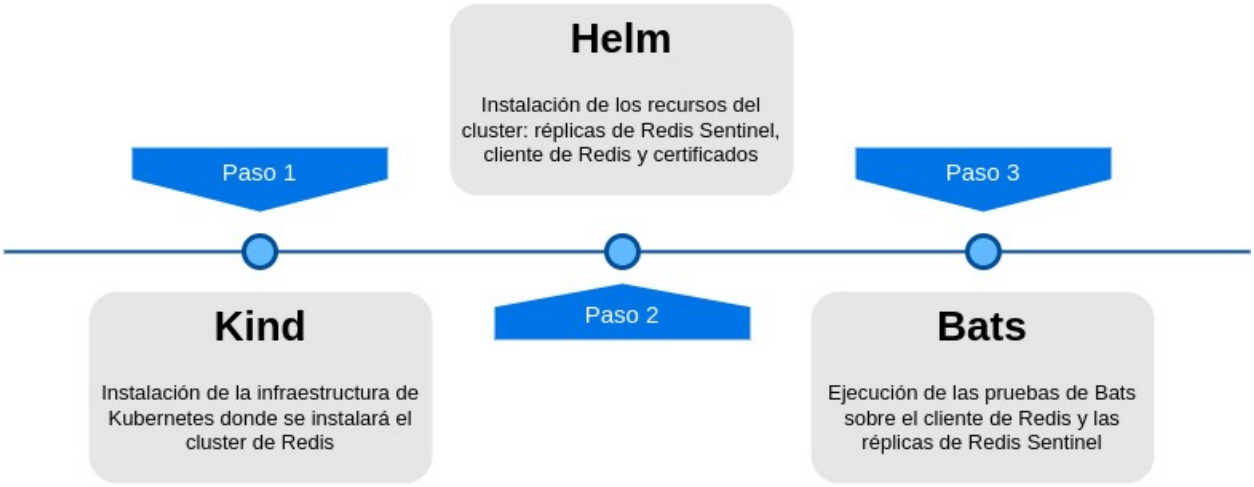


Figura 4: Proceso de ejecución de las pruebas de Redis

Otro punto a tener en cuenta es que Redis permite su operación tanto sin TLS como con TLS activado, para proveer mayor seguridad en las comunicaciones entre los nodos de Redis y entre éstos y los clientes. Sysdig utiliza ambas configuraciones de Redis para diferentes versiones de Sysdig, por lo que se realizará la misma prueba para ambos casos.

Para la realización de las pruebas se utilizan varios módulos de Bats, éstos aportan más funcionalidades para tener más opciones a la hora de construirlas. Estos módulos son *bats-support*, *bats-assert* y *bats-detik*. Algunas de las funcionalidades que aportan, utilizadas en las pruebas son: comprobar que un comando se ha ejecutado correctamente (a través de *assert_success*), comprobar que dos resultados coinciden (a través de *assert_equal*), ejecutar comandos de K8s predefinidos a través de un lenguaje natural (*bats-detik*) como comprobar que un contenedor está *Ready* con un número de reintentos cada cierto tiempo, etc.

Por un lado, para implementar el estado estable del sistema en Bats y comprobar que sus 3 nodos están de acuerdo en quién es el maestro, se ejecuta el comando *redis-cli sentinel get-master-addr-by-name primary* en cada uno de los Sentinel a través del *CLI* de K8s, que devuelve la dirección IP del nodo maestro. *redis-cli* no es más que el *CLI* de Redis para configuración y monitorización. Después se comparan los resultados de los 3 nodos y si todos están de acuerdo en quién es el maestro, el estado estable se considera validado.

La estructura de la prueba será la siguiente: en primer lugar, para eliminar el nodo primario y el nodo secundario, se obtienen los nombres de los pods a través del *CLI* de K8s. Es decir, se obtiene la nombre del nodo con el rol de primario y uno de los nodos con el rol de secundario. Una vez obtenidos, se utiliza el comando *kubectl delete pod* para eliminarlos. Seguidamente se espera a que los dos nodos se vuelvan a levantar con un comando de *bats-detik* y finalmente se verifica que se cumple el estado estable del sistema.

Ver anexo sección 8.4.1 para ver el código de las pruebas.

4.3. Framework de Chaos: Chaos Toolkit

Para el desarrollo de las pruebas de los siguientes casos de uso se va a utilizar el framework Chaos Toolkit. Este framework se ha elegido porque es el más extensible de los que se han analizado y puede cubrir distintos casos de uso diferentes con la implementación de nuevos módulos de código. Además, al ser compatible con diversas infraestructuras cloud como AWS, GCP, Azure y otras a través de los llamados *drivers* es posible extender las pruebas a otras infraestructuras sin mayor esfuerzo.

Por otro lado, para nuestros casos de uso, que el framework se ejecute externamente a la infraestructura de prueba es una ventaja ya que de esta forma nos aseguramos que el comportamiento del framework es independiente y aislado de la infraestructura con la que se quiere probar su resiliencia.

En este contexto, al ser un framework de código abierto, Sysdig ha adaptado el framework para que sea compatible con su producto al 100 %, añadiendo módulos y funcionalidades diseñadas específicamente para este contexto. A partir de ahora diferenciaremos Chaos Toolkit (framework original) de *Chaos Toolkit Sysdig*, como la versión adaptada para ser compatible con su infraestructura y utilizada para la ejecución de las nuevas pruebas de Chaos Engineering en Sysdig y que se recogen en los siguientes casos de uso.

En cuanto a la semántica de las pruebas de Chaos, ver anexo sección 8.7 para más información sobre la estructura de la pruebas en Chaos Toolkit (y *Chaos Toolkit Sysdig*).

Después de conocer la semántica, el diseño de las pruebas con el framework se puede entender con la Figura 5. De forma breve, primero cada experimento verifica que el estado estable del sistema se cumple, tras ello se ejecutan las acciones programadas junto con la hipótesis continua, si hubiera. Si todo ello resulta sin errores, se termina verificando de nuevo el estado estable del sistema para concluir si el experimento de Chaos ha tenido éxito o no, en cuyo caso se habrá encontrado una debilidad en el sistema.

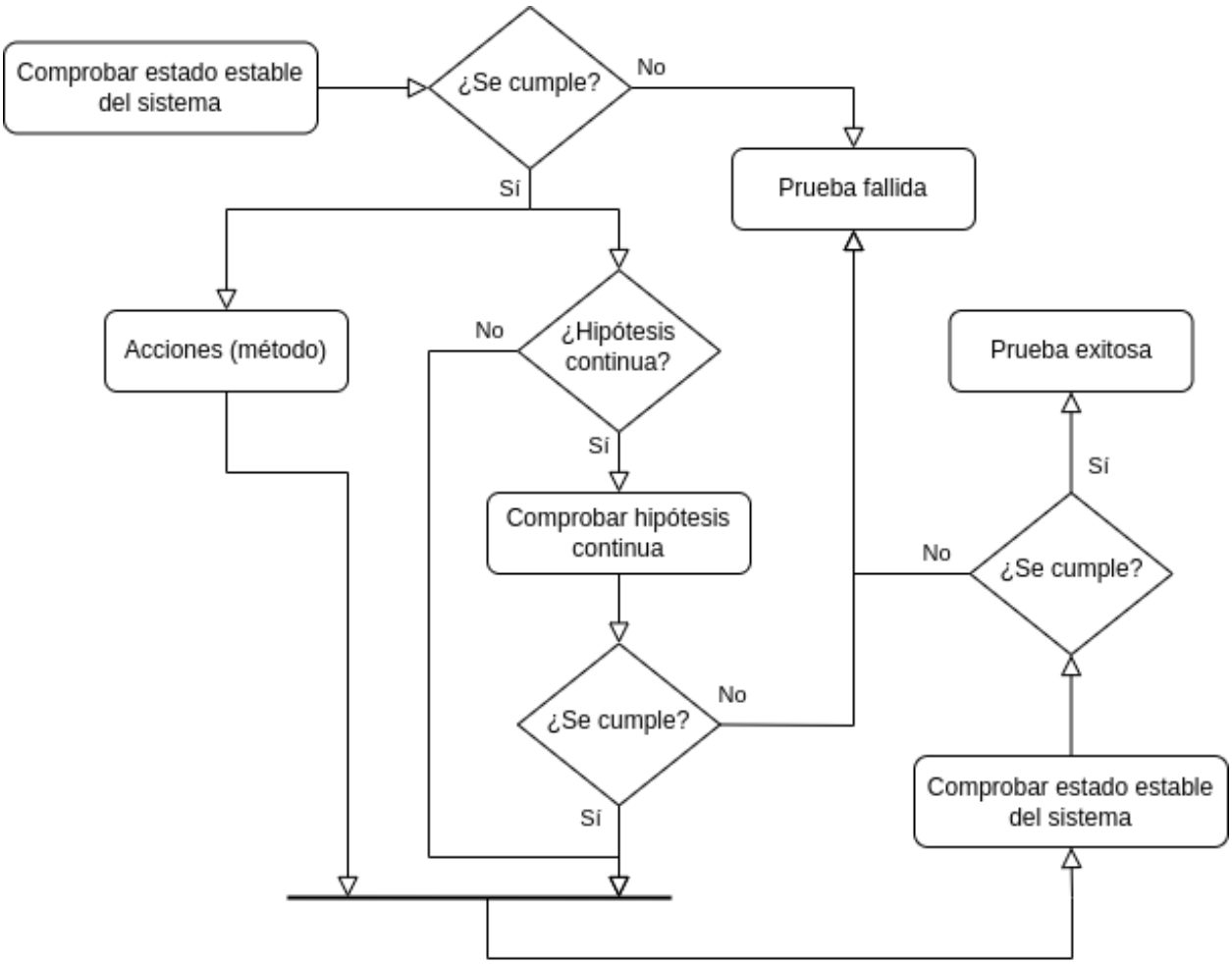


Figura 5: Diseño de las pruebas con Chaos Toolkit y *Chaos Toolkit Sysdig*

4.4. Caso de uso 2: NATS Streaming

De igual forma que en el caso de uso anterior, se va a explicar brevemente el funcionamiento de NATS Streaming en su versión de tolerancia a fallos y HA con un clúster formado por 3 nodos para poder entender los próximos pasos de diseño e implementación. NATS Streaming usa NATS (su versión base) para su funcionamiento (ver anexo sección 8.5 para más información sobre este componente), pero se puede obviar esto por el momento para centrarnos en el funcionamiento de NATS Streaming.

Este componente no es más que un sistema de transmisión de datos (cola de mensajes) a través de una cola cíclica, lo que quiere decir que va guardando los mensajes de entrada y si no hay salida de mensajes, la cola se llenará y se sobre escribirán los mensajes más antiguos.

Por otro lado, se debe de entender el patrón de Productor-Consumidor, ya que NATS

Streaming lo utiliza para la transmisión de mensajes. De forma simple, el productor es aquel proceso que *produce* mensajes, y el consumidor es aquel proceso que *consume* los mensajes que el productor ha enviado. NATS Streaming actúa como un intermediario entre los productores de mensajes y los consumidores, guardando de forma temporal o permanentemente la información. Estos procesos van a velocidades independientes, por lo que NATS Streaming se encargará de desacoplar las velocidades con su cola, guardando los mensajes que un productor envía para que un consumidor pueda leer de la cola la información más adelante.

Cada nodo del clúster puede obtener la exclusividad de poder leer y escribir a través del inicio de una votación para elegir al líder del clúster (a través del algoritmo RAFT). Esta votación se iniciará durante la inicialización del clúster o cuando ocurra un fallo en el servidor como un failover. Los demás nodos del clúster se limitan a replicar la información del nodo líder. Los clientes, por tanto, se comunicarán con el nodo líder para leer y escribir datos.

4.4.1. Diseño

Para comprobar la resiliencia de nuestro sistema se diseñarán varias pruebas de Chaos, en todas ellas se realizarán failovers en el clúster: se eliminará un nodo secundario (o réplica), un nodo primario (o líder), tanto un primario como un secundario, dos secundarios y los 3 nodos del clúster. Todas estas pruebas se tratan de pruebas end-to-end, porque se probará con NATS Streaming desplegado dentro del clúster de Sysdig. El objetivo es comprobar que se cumplen los requisitos mencionados en el *Análisis de requisitos*.

Para ello, se define la prueba de Chaos con el siguiente **estado estable del sistema**: el clúster de NATS Streaming deberá de tener los 3 nodos activos y funcionando correctamente (todos ellos tienen que ser conscientes de que están 3 nodos en el clúster) y se podrá leer y escribir siempre que haya suficientes nodos disponibles en el clúster (2/3) para realizar la elección del nodo líder y que pueda obtener la exclusividad para leer y escribir.

Las **hipótesis** de nuestras pruebas dependerán del tipo de nodo con el que se quiera simular una caída. En el caso de que se quiera terminar un nodo primario o un nodo secundario, seguirá existiendo mayoría en el clúster para que pueda haber líder, por lo que la hipótesis será la siguiente: si matamos al nodo primario, o por el contrario matamos al nodo secundario, NATS Streaming deberá de poder seguir leyendo y escribiendo datos antes, durante y después del experimento (con una tolerancia de tiempo para la ventana temporal entre la caída del nodo y la elección de un nodo líder por parte de los otros dos nodos, en el caso de la caída del nodo primario).

En el caso de que se quieran terminar dos o más nodos, la **hipótesis** será la siguiente: si se matan dos o más nodos del clúster de NATS Streaming, existirá una imposibilidad durante el tiempo en el que se recupera al menos un nodo más y se pueda elegir al maestro que acepte las peticiones de los clientes para leer y escribir datos. Al final los 3 nodos deberán de recuperarse correctamente y cumplir el estado estable del sistema.

El diseño de las pruebas de NATS Streaming se puede ver en la Figura 6.

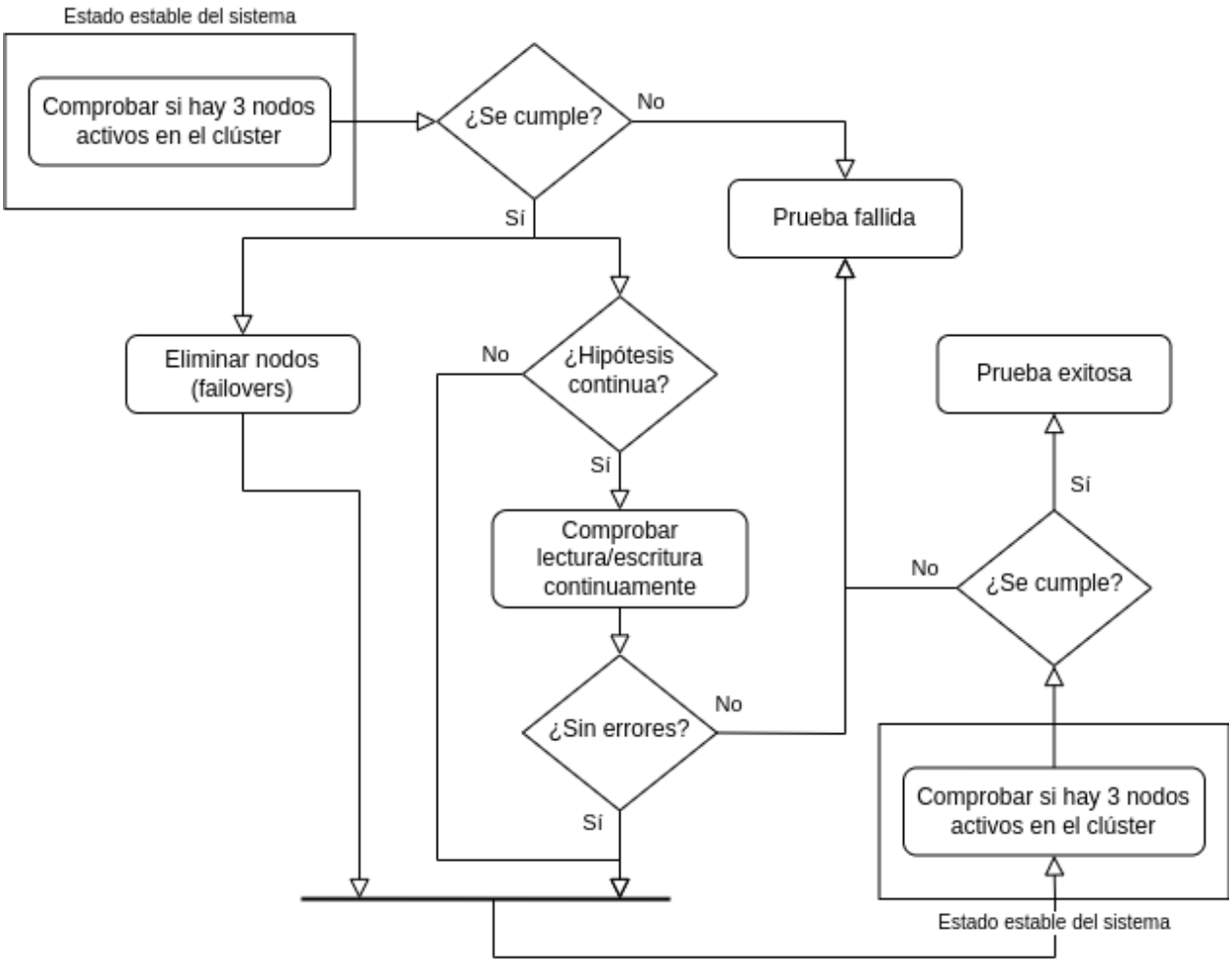


Figura 6: Diseño de las pruebas para el caso de uso 2

4.4.2. Implementación

Para la implementación de las pruebas mencionadas en el apartado de diseño, a diferencia de lo que se ha hecho con Redis, se utilizará *Chaos Toolkit Sysdig* para facilitar la construcción y ejecución de las pruebas de Chaos.

En Sysdig ya existía con anterioridad una base del framework con módulos de código y pruebas para otras tecnologías de bases de datos como Cassandra o Postgres (otros *datastores)*, por lo que se reutilirarán muchos de los módulos y pruebas existentes para implementar el diseño de las nuevas pruebas de NATS Streaming. *Chaos Toolkit Sysdig* también implementa una API de K8s para la ejecución de comandos tipo *CLI de K8s* y poder, por ejemplo,

terminar un pod desde el framework, que nos servirá de utilidad para provocar los failovers de los nodos, así como también ejecutar comandos en contenedores de pods.

Para simular un cliente de NATS se utilizará *NATS Box*, un contenedor con varias utilidades como el *CLI* de NATS para monitorización y mantenimiento del clúster, para ejecutar comandos de lectura y escritura a través del patrón Productor/Consumidor, así como una herramienta para realizar una prueba de rendimiento en el clúster y realizar de forma automática la ejecución de uno o varios Productores y Consumidores de forma concurrente y ver las estadísticas.

Algunos de los nuevos módulos que se han implementado en *Chaos Toolkit Sysdig* se explican a continuación. Por un lado tenemos el módulo que despliega *NATS Box* (como un *deployment*, ver anexo 8.2) y lo instala en el clúster de Sysdig, lo configura con las credenciales de acceso a NATS Streaming de Sysdig e instala los certificados necesarios para su uso. Esto se hace a través de un archivo *YAML* que incorpora la configuración para el *deployment*. El objetivo es usar esta herramienta para ejecutar comandos de lectura/escritura como si se tratara de un cliente end-to-end en Sysdig (simulando otro microservicio, por ejemplo). En las pruebas, se instalará *NATS Box* en el clúster de Sysdig al principio del experimento y se eliminará la instancia una vez que el experimento haya terminado (el módulo también implementa la eliminación de *NATS Box*) a través de los *Controls* del framework.

Se ha dicho que parte del estado estable del sistema de nuestras pruebas es comprobar que el clúster esta saludable y que cada uno de los nodos es consciente de que hay 3 nodos activos. Para ello se implementa un módulo que obtiene el número de nodos activos en el clúster a través de un endpoint de monitorización. El módulo ejecuta un comando para recuperar la información de este endpoint en cada uno de los 3 nodos del clúster a través de la API de K8s, procesa la información y devuelve finalmente el número de nodos conocidos en él por cada uno de ellos, si coincide.

En cuanto a la funcionalidad de lectura/escritura, se implementa un módulo que utiliza *NATS Box* para ejecutar una prueba de rendimiento con el comando *stan-bench*, activando un Productor que enviará un mensaje y un Consumidor que lo recibirá de forma concurrente. En el caso de que haya un error en este proceso, se implementa con *Tenacity* un mecanismo de reintentos con *Backoff* exponencial, es decir, existen 6 intentos y cada vez que uno falla, el tiempo entre reintentos aumenta. De esta forma se simula la política de reintentos que un componente de Sysdig utiliza para enviar y recibir información de NATS Streaming.

Finalmente tenemos el módulo que termina uno o varios nodos del clúster. Está implementado de tal forma que puedes definir el número de nodos que quieres terminar y de qué tipo (nodo primario, nodos secundarios, o ambos). De forma breve, su funcionamiento se basa en obtener la información de cuál es el nodo primario y cuáles son los nodos secundarios a través de un endpoint de monitorización ejecutado en cada uno de los 3 nodos, para luego eliminar el o los nodos que hayamos requerido por parámetros. Si en algún momento alguno de los nodos no tiene definido un estado de *Leader* (primario) o *Follower* (secundario), sal-

tará una excepción de código y la ejecución fallará (puede ocurrir que el estado de un nodo sea *Candidate*, lo que quiere decir que está en proceso de votación para elección de nodo líder).

Ver anexo sección 8.5.1 para ver la implementación de los módulos y su explicación de forma más detallada.

Por otro lado, *Chaos Toolkit Sysdig* implementa la ejecución de pruebas unitarias con el objetivo de hacer más sencillo la documentación del código y de chequeo del nuevo código que se implementa en los módulos del framework (para comprobar que el funcionamiento del código es el esperado). Es por ello que se crean nuevas pruebas unitarias para los nuevos módulos. Ver anexo 8.9.1 para ver la implementación de dichas pruebas.

Tras definir los módulos necesarios para las pruebas, se va a explicar cómo se han implementado las pruebas en sí. Cada una de las pruebas son archivos *YAML* que ejecuta Chaos Toolkit en su proceso de ejecución. Ya existían pruebas para Cassandra y Postgres, por ejemplo, así que se han tomado como base y referencia para la construcción de las nuevas.

Básicamente todas ellas tienen la misma estructura (ver anexo sección 8.7 para más información sobre la construcción de las pruebas de Chaos Toolkit). La definición de la hipótesis del estado estable del sistema sigue el mismo proceso para todas las pruebas: primero se comprueba que el estado de los pod de NATS están *Ready* (comprobación del *StatefulSet*, anexo 8.2), acto seguido se comprueba que el clúster esté saludable (con la comprobación del número de nodos en el clúster) y finalmente se verifica la lectura/escritura a través del patrón Productor/Consumidor. También se realiza la *hipótesis continua* para comprobar la lectura/escritura durante el experimento (en el caso de eliminar un nodo primario o un nodo secundario).

En cuanto al *Método* del experimento, la única diferencia entre las distintas pruebas es el número y tipo de nodos que se quieren terminar (parámetros del módulo para terminar nodos de NATS Streaming). Tras ejecutar dicha acción, se realiza una espera de X segundos para dar tiempo a que se reinicien y recuperen los nodos caídos, antes de pasar a la comprobación final del estado estable del sistema.

Ver anexo sección 8.5.2 para ver el código de las pruebas.

4.5. Caso de uso 3: Kafka

Apache Kafka es otra tecnología de transmisión de datos a través de colas de mensajes, que utiliza el patrón productor-consumidor para desacoplar velocidades. Su funcionamiento básico se explica a continuación para entender los próximos pasos.

De forma introductoria, en Sysdig se utiliza un clúster de Apache Kafka con 3 nodos (o brókers) para tolerancia a fallos y HA. Al contrario que NATS Streaming, Kafka no tiene una cola cíclica, su limitación se encuentra en la cantidad de datos que pueda albergar el

disco duro.

La forma en que se guardan los mensajes es muy distinta, Kafka utiliza *Topics* y *Particiones* para guardar los datos, replicarlos y particionarlos. Para que se entienda, de forma breve, un *topic* es un conjunto de datos sobre un contexto determinado, y las *particiones* de un *topic* son los subconjuntos de datos de un *topic*, por lo que su función principal es la de particionar los datos para distribuirlos en distintos brókers de Kafka y poder tener concurrencia en la lectura y escritura de los datos.

Estas particiones también pueden estar replicadas en los distintos brókers, por lo que de esta forma se consigue la tolerancia a fallos. Si un bróker se cae, habrá más particiones con la misma información en otros brókers. Por otro lado, por cada partición replicada N veces, habrá una partición líder y el resto serán réplicas. Si un cliente quiere leer o escribir datos en un *topic*, éste se comunicará con el bróker que tenga la partición líder correspondiente.

Cuando ocurre un failover o cualquier otro fallo en uno de los brókers, las particiones líderes de ese nodo dejan de estar disponibles, por lo que la lectura y escritura se hace imposible. En ese caso, otro bróker que tenga la partición replicada, puede hacerse con el estado de líder de la partición, aceptando las lecturas y escrituras de los clientes a partir de ese momento de forma automática y sin comprometer la disponibilidad.

Otro punto importante a tener en cuenta es el bróker con el rol de *ActiveController*, responsable de controlar los estados de las particiones y las réplicas y de llevar a cabo tareas administrativas como reasignación de particiones. Éste se ayuda de Apache Zookeeper (anexo 8.6.1) para obtener cierta información acerca del estado del clúster de Kafka, como los brókers activos en el clúster o información sobre *topics* y *particiones*, entre otros.

4.5.1. Diseño

Como en el caso de uso anterior, se comprobará la resiliencia de nuestro sistema con varias pruebas de Chaos, simulando caídas de servicio (failovers): se eliminará el bróker con el rol de *ActiveController* y cualquier otro que no tenga ese rol y se comprobarán que se cumplen los requisitos impuestos en el *Análisis de requisitos*. Estas pruebas también son end-to-end, ya que el clúster de Apache Kafka se despliega dentro del clúster de Sysdig y, además, se comprobará que Sysdig puede seguir leyendo datos de Kafka.

Siguiendo los pasos habituales, se define por tanto un **estado estable del sistema** con el que se construirán las futuras pruebas: el clúster de Kafka debe de estar saludable, sin inconsistencias (se debe de comprobar que solo hay un controlador activo, que los 3 brókers pueden comunicarse entre sí, y que no existen particiones *UnderReplicated* en ningún bróker) y Sysdig debe de poder leer datos de la cola de Kafka sin errores.

Para ambos casos de prueba tendremos la misma **hipótesis**: si matamos un nodo de Kafka (ya sea *ActiveController* o no), Sysdig debe de poder seguir leyendo datos de la cola

(y no solo después del failover, también durante y antes del experimento), ya que los roles de las particiones líderes del bróker caído deberán de pasar a otro bróker que disponga de las mismas particiones replicadas, tal y como se ha explicado en el funcionamiento de Kafka, permitiendo a los clientes seguir leyendo y escribiendo datos sin comprometer la disponibilidad del sistema.

El diseño de las pruebas de Kafka se puede ver en la Figura 7.

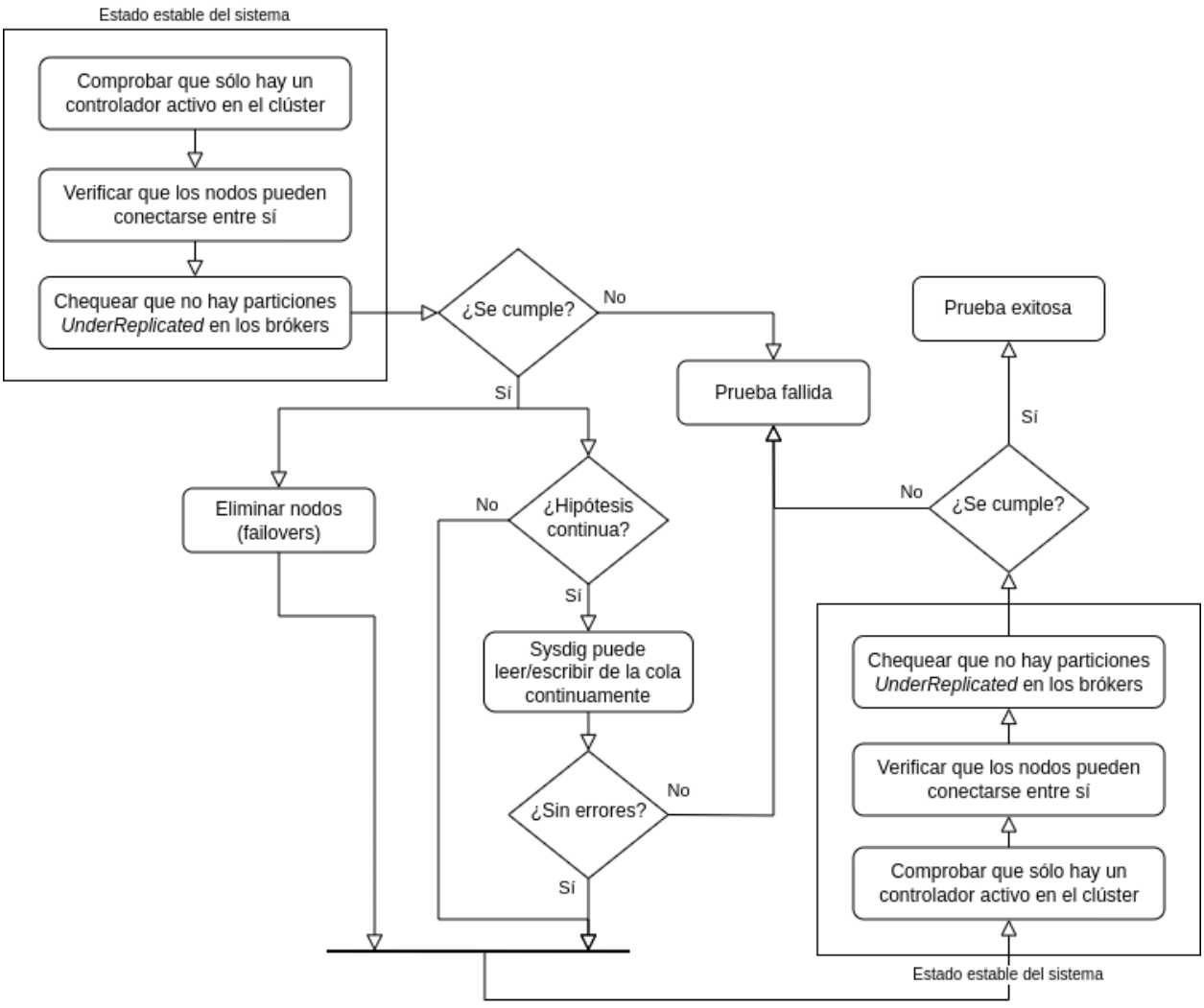


Figura 7: Diseño de las pruebas para el caso de uso 3

4.5.2. Implementación

Las pruebas de Chaos se van a implementar con el framework de Chaos de Sysdig, como se ha hecho en el caso anterior. Los nuevos módulos que se crean para poder ejecutar las

pruebas, según se ha descrito en el apartado de diseño, se explican a continuación.

Por un lado, para comprobar que el clúster de Kafka está saludable y poder implementar la comprobación del estado estable del sistema, se define un módulo con las 3 comprobaciones que se han descrito en el apartado de diseño: en primer lugar se define una función que comprueba la conexión de cada uno de los brókers de Kafka con el resto del clúster a través de la ejecución en cada nodo del comando *nc -vz*, conocido como *Netcat* (para realizar tareas de red en Linux), y comprobar la conexión punto a punto entre los 3 nodos. Tanto este comando como los 2 siguientes se ejecutan con la API de K8s de Chaos Toolkit, al igual que en el caso de uso anterior.

En segundo lugar, se define otra función para comprobar que solo hay un nodo con el rol de *ActiveController* a través de la aplicación JMX, que recoge las métricas del bróker de Kafka. Para ejecutar la aplicación de JMX se utiliza el comando *kafka-run-class* con una serie de opciones para especificar que se quiere obtener la métrica de *ActiveController*, devolviendo si el bróker tiene asignado ese rol (devuelve 1) o no (devuelve 0). Tras esto se hace un conteo de los nodos que se consideran *ActiveController*, y si son más de 1 saltará una excepción de código y fallará la ejecución.

En tercer lugar, se implementa la función para comprobar que no existen particiones *UnderReplicated* en cada uno de los nodos, a través de la ejecución de un script remoto implementado previamente, utilizado en la *Readiness Probe* (anexo 8.2) de K8s en cada pod de Kafka.

Otro módulo que se implementa, y al igual que en el caso de uso anterior, es la terminación de un pod. En este caso se quiere simular una caída de un bróker de Kafka, con una función para terminar el nodo de Kafka con el rol de *ActiveController* y otra para terminar uno o dos nodos (definido por parámetro) sin dicho rol. Para ello se utiliza otro módulo externo que se ha implementado con una función para obtener el nombre de red (DNS) del nodo de Kafka con el rol de *ActiveController* a través de la ejecución de un comando en uno de los nodos de Zookeeper (a través *zookeeper-shell*) con una serie de opciones.

Ver anexo sección 8.6.2 para ver la implementación de los módulos y su explicación de forma más detallada.

Al igual que en el caso anterior se implementa la ejecución de pruebas unitarias con el objetivo de hacer más sencillo la documentación del código y comprobar que los módulos creados tienen un comportamiento esperado. Ver anexo 8.9.2 para ver la implementación de dichas pruebas.

En cuanto a la definición de las dos pruebas de Chaos Toolkit, se sigue el mismo procedimiento con una estructura similar a las pruebas con NATS Streaming (definición de los *YAML*). En ambas pruebas se define la misma hipótesis del estado estable del sistema, con la comprobación del estado de los pod de Kafka (*StatefulSet* debe estar *Ready*), la verificación

de que Sysdig sigue activo (a través de un ping a su API) y la verificación de que se puede obtener datos de la API de Sysdig. Todos estos módulos ya se encontraban definidos con anterioridad en el framework, por lo que han sido reutilizados. Además, se define la hipótesis continua de las pruebas (verificación continua durante le ejecución de la pruebas) con la comprobación de la conexión con Sysdig (ping a API) y la lectura de datos de Sysdig cada 0.2 segundos (a través de la API).

Finalmente se define el *Método* del experimento: eliminar el pod de Kafka (eliminar nodo con *ActiveController* o terminar nodo sin ese rol, según el tipo de prueba). Tras ejecutar dicha acción, se realiza una espera de X segundos para dar tiempo a que se reinicien y recuperen los nodos caídos, antes de pasar a la comprobación final del estado estable del sistema.

Ver anexo sección 8.6.3 para ver el código de las pruebas.

5. Validación

5.1. Automatización de las pruebas

En el caso de Redis, se utiliza *Github Actions*[9] para realizar la automatización con CI en el repositorio de Sysdig para la infraestructura de Redis. De forma breve, cada vez que realizamos una *Pull Request* en dicho repositorio, se ejecuta el conjunto de pruebas de Redis con el objetivo de validar que la infraestructura sigue estando en buen estado. Nuestras pruebas no se incluyen aún en este proceso debido a que por el momento no se ha conseguido encontrar una solución al problema de resiliencia que se describirá más adelante.

Para los casos de uso 2 (NATS Streaming) y 3 (Kafka) se automatizan a través de la plataforma de automatización Jenkins. El objetivo de esta herramienta en Sysdig es cubrir la parte de CI. En la Figura 8 se puede ver el proceso que realiza el *Job* de Chaos para la ejecución de las pruebas. Este *Job* se ejecuta de forma diaria lanzando con el *OSC* un clúster con kops[10] (ver anexo 8.2.1) en AWS, instalando Sysdig y después ejecutando todo el conjunto de pruebas de Chaos con *Chaos Toolkit Sysdig*.

[9] https://docs.github.com/es/actions
[10] https://kops.sigs.k8s.io/

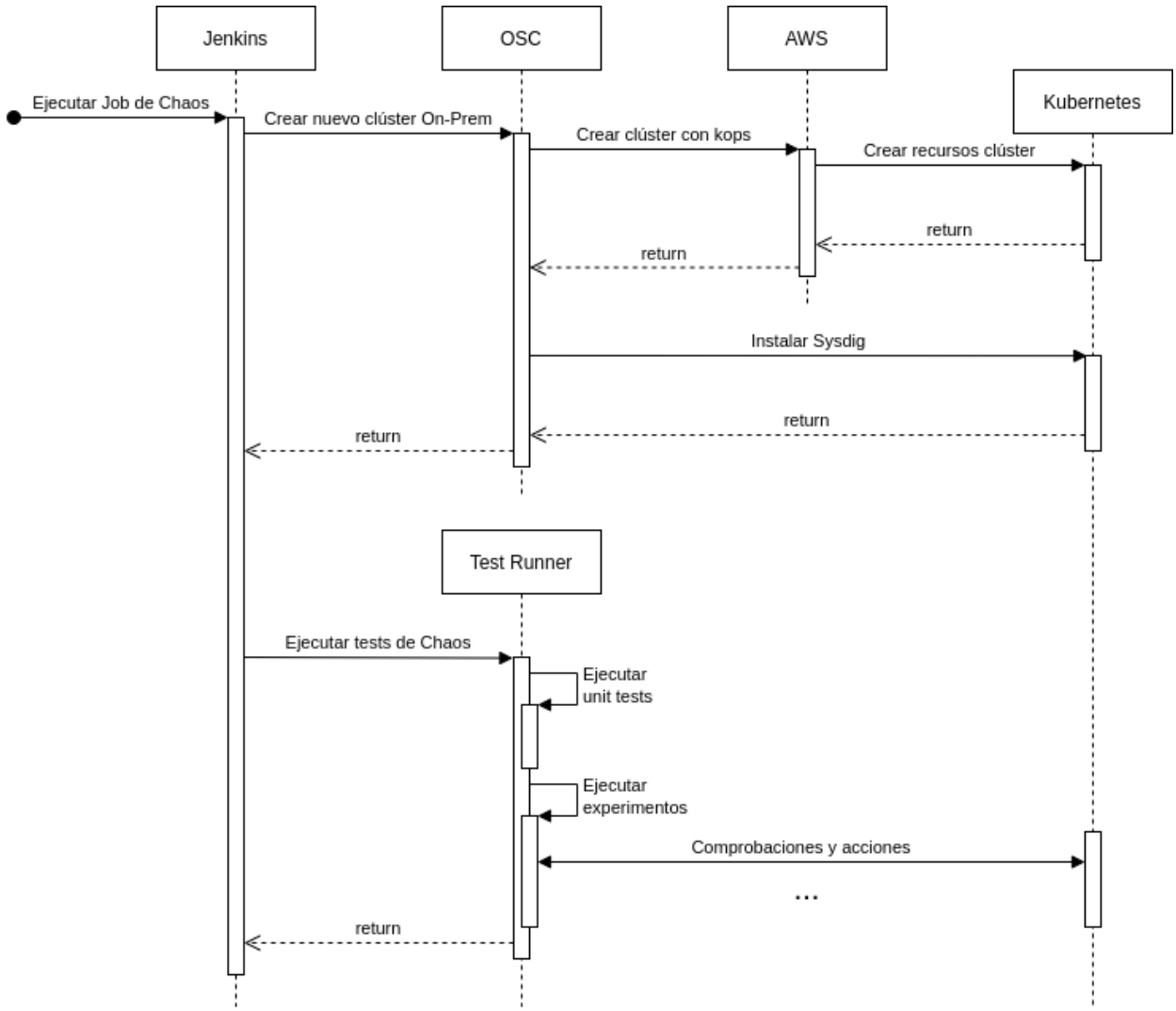


Figura 8: Automatización de las pruebas con Jenkins

Los resultados de estas ejecuciones también se han tenido en cuenta a la hora de valorar la resiliencia del sistema, y a partir de su implementación y gracias a CI, se seguirán ejecutando continuamente con el objetivo de encontrar errores futuros ante cambios en los componentes afectados como actualizaciones u otro tipo de errores.

Según se ha visto en las ejecuciones de las pruebas, se han encontrado varias desviaciones en las hipótesis de las pruebas de Chaos. Esto quiere decir que, o la hipótesis se ha formulado mal, o se ha implementado mal, o realmente el sistema no es resiliente antes estos casos de uso. Hay que ser conscientes del error humano y por eso es necesario poner sobre la mesa todas esas opciones. Por tanto, si se quisiera pasar estas pruebas a la parte de CD de Sysdig, para que cada vez que se crea una nueva versión del producto de Sysdig se ejecuten las pruebas, primero se debe de estar seguro de que las hipótesis son las correctas y de que si las pruebas fallan es porque realmente tenemos un problema de resiliencia en el sistema. Esto

se tomará como posible trabajo a futuro en el apartado de *Conclusiones*.

Por otro lado la depuración de errores en Jenkins es muy importante para saber qué ha ocurrido cuando una prueba de Chaos falla. Actualmente esto se hace con la generación de *logs* que crea automáticamente *Chaos Toolkit Sysdig* cuando se ejecutan las pruebas. Estos *logs* quedan disponibles tanto si falla el sistema como si no, siendo accesibles para cada Job de Jenkins que ejecuta las pruebas de Chaos. Ver anexo sección 8.10 para más información acerca de la depuración de errores en Jenkins.

5.2. Análisis de resultados

Una vez diseñadas e implementadas las pruebas, se ejecutan para comprobar si las hipótesis que se han definido para cada una de ellas se cumplen. Si esto es así, el sistema es resiliente para ese caso de uso, si no lo es, se habrá encontrado una debilidad en el sistema. Estas conclusiones se toman después de varias ejecuciones manuales (anexo 8.4.2 para Redis y anexo 8.8 para NATS y Kafka) y mediante las ejecuciones automatizadas.

5.2.1. Caso de uso 1: Redis

En el caso de la prueba de Redis sin TLS, Redis nunca puede recuperarse ya que los nodos que se levantan son incapaces de conectarse con el otro único nodo del clúster, por lo que no puede haber una elección para elegir un nodo maestro que acepte las peticiones de los clientes. En el caso de la prueba con TLS, ocurre lo mismo en un 50 % de los casos.

Se sabe que se trata de un problema de configuración de los archivos que se usan para desplegar los contenedores de los pods que contienen a *Redis* y *Sentinel*. Se ha dicho que la infraestructura de Redis está empaquetada por *Bitnami*, al igual que estos archivos, que han sido modificados para funcionar con Sysdig. Es por ello que o se seguirá buscando el problema para corregirlo manualmente, o se probará a realizar directamente una actualización completa de las nuevas versiones de los archivos originales de *Bitnami*.

5.2.2. Caso de uso 2: NATS Streaming

En la gran mayoría de los casos las hipótesis de las pruebas se cumplen, terminando en pruebas exitosas, pero en aproximadamente un 10-20 % de los casos, tras un failover de un nodo primario, la comprobación del estado estable del sistema nos devuelve que existen 4 nodos en el clúster, en vez de 3 que sería lo normal. Este comportamiento se está estudiando ya que es un problema que ha ocurrido durante la ejecuciones automatizadas y no se ha conseguido replicar manualmente, además de que no se sabe cuál es el impacto real de este problema y si afecta al usuario final de Sysdig.

5.2.3. Caso de uso 3: Kafka

En un 50 % de los casos para cualquiera de las dos pruebas que se ejecutan, durante un período de tiempo después del failover, la lectura/escritura de Sysdig a través de su API

falla, resultando imposible leer los últimos 10s de datos. A pesar de ello, tras unos 10-20 segundos estos datos pueden ser leídos de nuevo, por lo que se considera que el impacto que puede tener es mínimo y se tolerarán de igual forma que hace Sysdig.

5.3. Validación de la resiliencia en pre-producción

Habiendo validado la resiliencia de nuestro sistema para los 3 casos de uso, surge la siguiente pregunta: ¿es suficiente con ejecutar las pruebas en pre-producción? Para responder a esta pregunta se tiene que tener en cuenta que tipo de pruebas se están realizando. Por poner un ejemplo, si eliminamos 2 nodos de NATS Streaming, sabemos que durante el tiempo en el que al menos uno de ellos se levanta, el/los clientes (en este caso el propio Sysdig, con otros microservicios que leen y escriben en él) es incapaz de recibir y enviar datos nuevos porque el servicio no esta disponible, y eso se sabe con total certeza, por lo que afectará a la experiencia del usuario. Es por eso que se considera que determinadas pruebas como ésta no son eficientes a la hora de implementarlas en producción.

Sin embargo, si se extiende la prueba para que se ejecute en una porción pequeña de usuarios de producción y se añade un factor de prueba tal como el testeo de si los usuarios siguen sin abandonar la interfaz de usuario ante posibles inconsistencias de datos, o si realizan otras acciones tolerables en la UI a criterio de la empresa, entonces podríamos hacer la implementación en producción y aportaría valor.

En caso contrario, si que tendría sentido comprobar la resiliencia del sistema en un ambiente de producción con el failover de un solo nodo de NATS Streaming (por poner otro ejemplo), ya que la hipótesis es que no debería de fallar y la lectura/escritura se debe de poder hacer sin problemas. Sin embargo, en este último caso tendría más sentido extender la prueba y comprobar que el rendimiento de lectura/escritura sigue estando dentro de umbrales tolerables cuando solo hay 2 nodos activos en el clúster, por ejemplo.

Como se ve, si se quieren hacer pruebas de Chaos en producción, se deben de plantear de esta forma y con el objetivo prefijado para realizar las pruebas allí, pero tal y como se han diseñado e implementado las pruebas, ejecutarlas en el escenario de pre-producción actual es, posiblemente, la mejor decisión.

6. Conclusiones

6.1. Resolución del trabajo

Después de las horas de trabajo dedicado a este proyecto se considera que se han cumplido los objetivos impuestos inicialmente. Se han implementado diversas pruebas de Chaos Engineering para comprobar la resiliencia del sistema, y se ha visto que es efectivo a la hora de encontrar nuevos errores que antes no se conocían en el sistema. No solo se ha probado con una sola tecnología, sino que se ha probado con 3 distintas (Redis, NATS Streaming y Kafka) con las que simular las condiciones turbulentas en un sistema distribuido real que implica la filosofía de Chaos, lo que le da bastante valor a la empresa.

Al principio se hizo complicado la elaboración de pruebas con el framework, especialmente con la primera tecnología con la que se realizaron las pruebas (NATS Streaming) porque era el primer contacto que se tenía con *Chaos Toolkit Sysdig*. Con Kafka se hizo más simple ya que se adaptaron los módulos y pruebas ya creadas con NATS Streaming, además de que ya se tenía experiencia con el funcionamiento del framwwork. Por otro lado con la automatización de las pruebas se cree que aporta mucho valor a largo plazo, ya que como se ha dicho anteriormente esto supone poco esfuerzo y formaría parte del ciclo CI/CD. En general, sí ha merecido la pena la implementación de Chaos Engineering en Sysdig.

Por otro lado, los casos de uso elegidos se han formulado de esta forma para ir entendiendo la complejidad del sistema progresivamente. Además, estos *datastores* se han elegido porque forman parte de la infraestructura del producto de Sysdig y son pilares fundamentales en su funcionamiento, por lo que se hace necesario asegurar una buena resiliencia.

6.2. Trabajo a futuro

En cuanto al trabajo que se podría hacer en un futuro, las opciones pasan por probar otros frameworks y comparar de forma práctica que opción se ajusta mejor a las necesidades de la empresa. En su momento ya se tomó la decisión de utilizar Chaos Toolkit, pero la valoración podría cambiar si se encuentran más ventajas a la hora de usar otro framework como *Chaos Mesh*, por poner un ejemplo.

Este proyecto se ha basado en una serie de casos de uso para comprobar la resiliencia del sistema con una serie de experimentos de chaos mediante la inyección de failovers, pero esto puede ir más allá con la inyección de otros tipos de fallos como una mayor latencia de red o aislar ciertos componentes del resto. Además se pueden probar nuevas tecnologías de bases de datos u otros componentes utilizados en la infraestructura de Sysdig, aumentar el número de pruebas y extenderlas para un mismo caso de uso con el objetivo de hacerlas más robustas (con más comprobaciones) o incluso con la ejecución de ciertas pruebas de forma automática con CI/CD cada vez que se produce un cambio en un componente específico del sistema. Se tiene un framework de Chaos que es fácilmente extensible con módulos y *drivers*, por lo que sería fácil de llevar a cabo.

Otro de los temas que se podría considerar es pasar a realizar los experimentos en producción para una pequeña porción de usuarios. Como ya se ha comentado en la sección de *Validación*, esto tiene una serie de ventajas y desventajas, por lo que habría que estudiar bien que riesgos conlleva y si finalmente las pruebas en producción merecen la pena. Lo mismo ocurre con pasar los experimentos a la línea de CD, para que cada vez que se lance una nueva versión de Sysdig, se lancen también las pruebas. Para ello, primero será necesario asegurar una estabilidad en la ejecución de las pruebas con Jenkins CI.

Además, las pruebas ejecutadas con Jenkins de forma automatizada solo se lanzan en un clúster de K8s con kops. Sysdig utiliza muchos más *sabores* donde se lanzan las versiones de Sysdig como una forma de diversidicar las opciones disponibles para el cliente. Sería una buena opción incluir las ejecuciones en los disntintos *sabores* que acepta Sysdig para adaptarse a todas las posibilidades existentes y así comprobar la resiliencia del sistema en su totalidad.

6.3. Valoración personal

Personalmente, este proyecto me ha servido para ganar mucha experiencia no solo en temas técnicos como los que se han ido mencionando durante el trabajo: Chaos Engineering, tecnologías de bases de datos, pruebas de QA, sistemas distribuidos, la importancia de la resiliencia en nuestros sistemas, conocimientos de Python, etc. sino también para ganar experiencia laboralmente, ya que es mi primer contacto laboral, lo que me aporta mucho valor.

En general el trabajo me ha gustado, a pesar de que en ocasiones y al tratarse de una tecnología de pruebas nueva y no muy conocida, surgían bloqueos. A pesar de esto el esfuerzo realizado ha merecido la pena y se han logrado los objetivos iniciales que se presentaron a la hora de diseñar el objetivo del proyecto. Se ha probado con éxito parte de la resiliencia del sistema de Sysdig con Chaos Engineering y el hecho de haber encontrado inconsistencias hace que haya merecido aún más la pena, ya que esto se tiene en cuenta para mejorar el sistema.

Como punto final, agradecer a todo el personal de Sysdig y al equipo con el que he estado trabajando en el departamento de QA por toda la ayuda prestada, en especial a David Pemán Ruiz, que ha sido un verdadero mentor para mí y me ha ayudado con cualquier duda que me surgía con el trabajo, y también a Miguel Julián Ramos por su dedicación, motivación, apoyo y guía en el proyecto. A los profesores que me animaron a comenzar con las prácticas de empresa en Sysdig y apoyarme con el proyecto, en especial a Unai Arronategui Arribalzaga y Francisco J. Lopez-Pellicer, y a toda mi familia y amigos que siempre han sido una inspiración y motivación para conseguir todos mis objetivos.

7. Bibliografía

[1] EINA. Recomendaciones e instrucciones para cumplimentar la memoria del TFG https://eina.unizar.es/sites/eina.unizar.es/files/archivos/secretaria/20210706_instrucciones_tfg_tfm.pdf

[2] Mathias Lafeldt, Gremlin. The Discipline of Chaos Engineering https://www.gremlin.com/blog/the-discipline-of-chaos-engineering/

[3] Andrew S. Tanenbaum, Maarten van Steen, CreateSpace Independent Publishing Platform. *Distributed Systems: Principles and Paradigms 2nd Edition*

[4] IBM. Teorema CAP https://www.ibm.com/es-es/cloud/learn/cap-theorem

[5] Scott Surovich, Marc Boorshtein, Packt Publishing. *Kubernetes and Docker - An Enterprise Guide: Effectively containerize applications, integrate enterprise systems, and scale applications in your enterprise*. Edición 2020

[6] RedHat. ¿Qué son la integración y la distribución continuas (CI/CD)? https://www.redhat.com/es/topics/devops/what-is-ci-cd

[7] Stackify. The Ultimate Guide to Performance Testing and Software Testing https://stackify.com/ultimate-guide-performance-testing-and-software-testing/

[8] Gerardus Blokdyk, CreateSpace Independent Publishing Platform. *Regression testing: Beginner's Guide, Second Edition*

[9] Casey Rosenthal, Nora Jones, O'Reilly Media Inc. *Chaos Engineering: System Resiliency in Practice*. Edición 2020

[10] Román Martín y Alberto Grande, Paradigma Digital. Chaos engineering: herramientas y frameworks de chaos https://www.paradigmadigital.com/techbiz/chaos-engineering-herramientas-y-frameworks/

[11] Kubernetes.io. Documentación de conceptos de Kubernetes https://kubernetes.io/es/docs/concepts/_print/

8. Anexos

8.1. Sysdig (como producto/servicio)

Sysdig es un producto o servicio orientado a la monitorización y seguridad de contenedores, Kubernetes y la nube. Existen dos principales productos dentro de Sysdig, *Monitor* y *Secure*. El primero está más orientado a la monitorización de Kubernetes y Prometheus en la nube, mientras que el segundo está más orientado a la seguridad para ejecutar aplicaciones de Kubernetes en la nube. Por otro lado el servicio que se ofrece puede incluir ambos productos conjuntamente o de forma separada.

Sysdig está formado por distintos componentes o microservicios y todos ellos están desplegados sobre un clúster de Kubernetes. Además, Sysdig puede ponerse a disposición de los clientes de dos formas distintas, a través de SaaS o a través de un clúster on-premise, es decir, a través de una plataforma online gestionada por la propia empresa de Sysdig sin necesidad de que el cliente gestione la infraestructura, o a través de la instalación de Sysdig en la propia infraestructura del cliente, respectivamente.

En este proyecto se hablará continuamente del clúster de Sysdig como al clúster de Kubernetes on-premise que contiene a Sysdig, con todos sus microservicios para su funcionamiento, desplegado a través de la plataforma de computación en la nube AWS para realizar allí las pruebas y experimentos de Chaos Engineering.

8.2. Conceptos de Kubernetes

Hay varios conceptos que se utilizan en este proyecto, de forma breve [11]:

- **Nodos**: son parte de la arquitectura de K8s y son las máquinas virtuales o físicas que forman parte del clúster de K8s.
- **Clúster**: es un conjunto de nodos en K8s.
- **Pods**: son unidades de computación más pequeñas que los nodos. Un nodo puede estar formado por varios Pods y un Pod puede estar formado por uno o más contenedores que comparten los recursos de almacenamiento y de red.
- **Deployment**: Es un controlador que es capaz de manejar el estado de una aplicación en el clúster y que puede estar formado por una o varias réplicas de Pods de forma auto gestionada. De tal forma que cuando hay una diferencia con la especificación del *Deployment*, K8s gestiona de forma automática la solución, como por ejemplo la instanciación de un nuevo Pod cuando éste falla.
- **StatefulSet**: Es lo mismo que un *Deployment* con la diferencia de que los Pods nunca pierden su identidad, garantizando su orden y unicidad. Es decir, si un Pod falla, la instanciación de un nuevo Pod seguirá teniendo la misma identidad de red y almacenamiento persistente previo a su fallo.
- **Readiness**: Una *Readiness Probe* es una comprobación regular y automática que hace K8s para un contenedor concreto con el objetivo de determinar si éste es capaz de aceptar tráfico. Si lo es, entonces el contenedor se marca como *Ready*.

8.2.1. Kops

Kops es una herramienta de código abierto que permite gestionar la infraestructura de K8s sobre servicios de nubes públicas. En este proyecto se utiliza para desplegar un clúster on-premise de K8s con Sysdig en la nube pública de Amazon *AWS*. Kops se encarga de crear y configurar todos los componentes de K8s para el aprovisionamiento de Sysdig.

8.2.2. Kind

Kind es otra herramienta que permite gestionar la infraestructura de K8s al igual que Kops, con la diferencia de que Kind despliega la infraestructura de K8s de forma local usando contenedores de *Docker* como nodos del clúster. En este proyecto se usa para realizar las pruebas de integración de Redis.

8.3. Algoritmo RAFT

El algoritmo RAFT es un algoritmo de consenso utilizado para elegir un nodo líder sobre un conjunto de nodos que comparten información. Una vez elegido, este nodo se encargará de recibir todas las solicitudes para realizar las operaciones para las que está configurado y se encargará de realizar la sincronización entre el resto de nodos.

Un ejemplo de operativa sería un conjunto de nodos que comparten la misma BBDD de forma replicada, y sólo un nodo debe de poder leer y escribir. Para ello se utiliza RAFT en la elección de un nodo líder que se encargue de recibir todas las operaciones de lectura y escritura y de replicar la nueva información escrita en el resto de los nodos.

Los nodos pueden tener 3 tipos de roles:

- **Líder**: es el nodo que se encarga de recibir todas las solicitudes y donde se realizan primero las operaciones
- **Seguidores**: son los nodos que se encargan de replicar las operaciones que ocurren en el nodo líder
- **Candidatos**: son los nodos que están esperando a que se realice la elección de un nuevo nodo líder

El funcionamiento de una votación es simple: cuando uno de los nodos no recibe respuesta del nodo líder, éste comienza la votación comunicándose con el resto de nodos para pedir que se le elija a él como nuevo nodo líder. Si la mayoría de nodos del clúster está de acuerdo, se completará la votación y el nodo que pidió ser líder se convertirá en el nuevo líder.

La votación puede ocurrir la primera vez que se inicializan los nodos del clúster, en cuyo caso no existe líder todavía, o en el caso de que el líder anterior deje de estar disponible por cualquier razón, ya sea porque ha tenido un fallo de red y se queda incomunicado con el resto de nodos, porque ocurre un failover, etc.

Por otro lado, cuando se recomienda que un clúster de nodos que utilizan RAFT para la elección de líder tenga un mínimo de 3 nodos, esto es porque el algoritmo sería capaz de seguir funcionando con 2 nodos, ya que siguen siendo mayoría para realizar una votación en el caso de la caída del líder.

8.4. Redis

8.4.1. Definición de las pruebas

Tanto la prueba sin TLS como la prueba con TLS tienen una estructura similar. En cuanto a la prueba sin TLS, algunos de sus puntos más relevantes son los siguientes.

```
370 @test "3.10 Redis6-ha - Delete the leader and a replica" {
371     # Try to delete the redis leader pod and a replica pod. They should degrade correctly, choosing
372     # a new master (the only alive) and next, they should get up correctly and join to the new master
373
374     [[ ! -f "${PROJECT_ROOT}/deploy-successful.tmp" ]] && skip "Redis helm charts have not been deployed correctly"
375     # Get Master ip from sentinel svc so we can delete it
376     LEADERIP="$(${KUBECTL} exec -i client -- redis-cli -D '' -h redis-headless \
377       -p 26379 sentinel get-master-addr-by-name primary |head -n1)"
378     # Get the Leader pod
379     LEADERPOD="$(${KUBECTL} get pods -o custom-columns=:metadata.name --no-headers=true \
380      --field-selector status.podIP="$LEADERIP")"
381     # Get a replica pod
382     REPLICAPOD="$(${KUBECTL} get pods -o custom-columns=:metadata.name --no-headers=true \
383      --field-selector=status.podIP!="$LEADERIP" -l app="redis"|head -n1)"
384     # Delete the leader pod and the replica pod at the same time
385     run ${KUBECTL} delete pod $LEADERPOD $REPLICAPOD
386     assert_success
387     # Verify that the ex leader pod is up and running again
388     run try "at most 20 times every 15s to get pods named '$LEADERPOD' and verify that '.status.containerStatuses[?(@.name==\"redis\")].ready' is 'true'"
389     assert_success
390     run try "at most 20 times every 15s to get pods named '$LEADERPOD' and verify that '.status.containerStatuses[?(@.name==\"sentinel\")].ready' is 'true'"
391     assert_success
392     # Verify that the ex replica pod is up and running again
393     run try "at most 20 times every 15s to get pods named '$REPLICAPOD' and verify that '.status.containerStatuses[?(@.name==\"redis\")].ready' is 'true'"
394     assert_success
395     run try "at most 20 times every 15s to get pods named '$REPLICAPOD' and verify that '.status.containerStatuses[?(@.name==\"sentinel\")].ready' is 'true'"
396     assert_success
397     # Check if all the sentinel agree on leader/master ip
398     assert_equal $(sentinel_get_master redis-0) $(sentinel_get_master redis-1)
399     assert_equal $(sentinel_get_master redis-1) $(sentinel_get_master redis-2)
400 }
```

Figura 9: Código de la prueba de Redis sin TLS

En cada una de nuestras pruebas con Redis se comprueba que el proceso de despliegue anterior se ha efectuado con éxito. Esto se hace comprobando que se generó durante el proceso el archivo *deploy-successful.tmp*. Si esto no es así, la prueba se omitirá con el comando *skip*. Esto se puede observar en la línea 374 de la Figura 9.

Para eliminar los dos nodos de la prueba se necesita el nombre de los 2 pods. Para ello se utiliza la IP del nodo maestro obtenida en la línea 376 y se guarda en *LEADERIP*. Esta dirección IP se obtiene desde el cliente instalado en el clúster de kind, con la ejecución del comando especificado de la *CLI* de Redis. Sabiendo la IP del nodo maestro se pueden obtener fácilmente los nombres del nodo primario y de un secundario, según se ve en las líneas 379 y 382 de la Figura 9.

Por otro lado tenemos varias ocurrencias de *assert_success* que nos sirve para comprobar que el comando que le precede devuelve un código de error igual a 0, es decir, que la ejecución del comando ha terminado correctamente. Este comando se utiliza para chequear que la eliminación de los pods ha sido correcta y que se han levantado correctamente tras un tiempo (líneas 386, 389, 391, 394 y 396).

Finalmente, para realizar la comparación entre las direcciones IP para comprobar que la IP del maestro es la misma para los 3 nodos del clúster cuando se recuperan los 2 nodos caídos, se utiliza el comando *assert_equal* en la línea 431 y 432.

```
@test "3.10 Redistls - Delete the leader and a replica" {
    # Try to delete the redis leader pod and a replica pod. They should degrade correctly, choosing
    # a new master (the only alive) and next, they should get up correctly and join to the new master

    [[ ! -f "${PROJECT_ROOT}/deploy-successful.tmp" ]] && skip "Redis helm charts have not been deployed correctly"
    # Get Master ip from sentinel svc so we can delete it
    LEADERIP="$(${KUBECTL} exec -i client -- redis-cli -D '' -h redistls-headless -a "$RANDOMPW" \
        --tls --cacert /opt/bitnami/redis/certs/ca.crt --no-auth-warning \
        -p 26379 sentinel get-master-addr-by-name primary |head -n1)"
    # Get the Leader pod
    LEADERPOD="$(${KUBECTL} get pods -o custom-columns=:metadata.name --no-headers=true \
        --field-selector status.podIP="$LEADERIP")"
    # Get a replica pod
    REPLICAPOD="$(${KUBECTL} get pods -o custom-columns=:metadata.name --no-headers=true \
        --field-selector=status.podIP!="$LEADERIP" -l app.kubernetes.io/name="redistls"|head -n1)"
    # Delete the leader pod and the replica pod at the same time
    run ${KUBECTL} delete pod $LEADERPOD $REPLICAPOD
    assert_success
    # Verify that the ex leader pod is up and running again
    run try "at most 20 times every 15s to get pods named '$LEADERPOD' and verify that '.status.containerStatuses[?(@.name==\"redis\")].ready' is 'true'"
    assert_success
    run try "at most 20 times every 15s to get pods named '$LEADERPOD' and verify that '.status.containerStatuses[?(@.name==\"sentinel\")].ready' is 'true'"
    assert_success
    # Verify that the ex replica pod is up and running again
    run try "at most 20 times every 15s to get pods named '$REPLICAPOD' and verify that '.status.containerStatuses[?(@.name==\"redis\")].ready' is 'true'"
    assert_success
    run try "at most 20 times every 15s to get pods named '$REPLICAPOD' and verify that '.status.containerStatuses[?(@.name==\"sentinel\")].ready' is 'true'"
    assert_success
    # Check if all the sentinel agree on leader/master ip
    assert_equal $(sentinel_get_master_tls redistls-node-0) $(sentinel_get_master_tls redistls-node-1)
    assert_equal $(sentinel_get_master_tls redistls-node-1) $(sentinel_get_master_tls redistls-node-2)
}
```

Figura 10: Código de la prueba de Redis con TLS

Por otro lado tenemos la prueba de la Figura 10, que corresponde a la prueba de Redis con TLS. Su estructura es exactamente la misma que la versión sin TLS, pero con la diferencia de que se eliminan los dos nodos de Redis con TLS y la comprobación de la IP del maestro se hace con los nodos del clúster con TLS.

8.4.2. Ejecución de las pruebas

Como ya se ha visto en la Figura 4, con kind se lanza un clúster de K8s de forma local, seguidamente se instala el clúster de Redis con el gestor de paquetes de K8s Helm y se ejecutan las pruebas con Bats. Aquí entra el comando *task*[11], que no es más que un ejecutor de tareas, análogo a *GNU Make*, que a través del varios *Taskfiles* (archivos donde se guardan las instrucciones para cada tarea) se ejecutan las tareas del proceso.

Estas tareas van desde la creación del entorno hasta la ejecución de las pruebas. En la Figura 11 podemos ver una ejecución del comando *task* con todas las posibles acciones que se pueden tomar dentro del repositorio de la infraestructura de Redis en Sysdig.

[11] https://taskfile.dev/

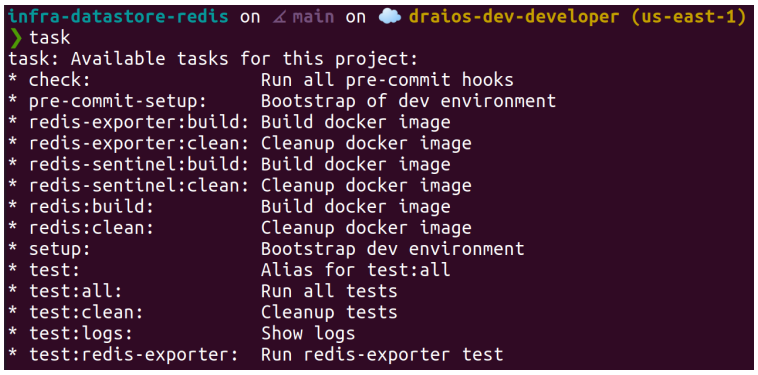


Figura 11: Ejecución del comando *task* de Redis

El procedimiento para ejecutar las pruebas es el siguiente.

```
# Setup local env tasks
task setup
# Run all tests
task test
```

Este último comando se encargará de comprobar una serie de requerimientos antes de ejecutar las pruebas de bats, como que se tenga instalado bats y sus librerías (*bats_core*, *bats_assert* y *bats_detik*), que deben de ser instaladas manualmente, así como comprobar que se han inicializado las variables de entorno e instalar el clúster de K8s kind donde se desplegará el clúster de Redis si no se ha hecho ya.

Una vez comprobados los requerimientos, *bats* ejecuta 3 ficheros. El primero despliega el clúster de Redis en kind junto con comprobaciones para asegurar que todo ha ido bien, el segundo instala el cliente de Redis junto con más pruebas, y por último, el tercer archivo implementa la ejecución de las pruebas de integración de Redis. Las pruebas que se han creado en este proyecto forman parte de este tercer archivo. Un ejemplo de ejecución de algunas pruebas se puede ver en la Figura 12.

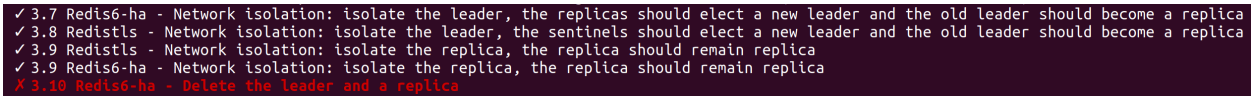


Figura 12: Parte de la ejecución de las pruebas de Redis

8.5. NATS Streaming

8.5.1. Implementación de los módulos

Los experimentos de NATS Streaming utilizan un conjunto de módulos para implementar las pruebas, acciones y controles de *Chaos Toolkit Sysdig*. En esta sección se detallan sus funciones y cómo se han implementado.

```
10  def before_experiment_control(
11      context: Experiment, configuration: Configuration = None, secrets: Secrets = None, **kwargs
12  ):
13      """
14      Run NATS BOX client (deployment) in the Kubernetes cluster
15      """
16
17      yaml_file = (
18          "../../../chaostoolkit-sysdig/src/sysdigchaos/nats_streaming/k8s/nats-box-deployment-with-sysdig.yaml"
19      )
20
21      try:
22          create_deployment(spec_path=yaml_file, ns=NAMESPACE)
23          deployment_fully_available(name=NAME, ns=NAMESPACE)
24      except Exception:
25          logger.debug("Error creating nats-box deployment")
26
27      pass
```

Figura 13: Despliegue de NATS Box en K8s

El despliegue de NATS Box[12] de la Figura 13 es el primer paso del experimento, ya que se necesita para simular un cliente de NATS Streaming en el clúster de Sysdig y así poder comprobar la lectura/escritura del *datastore*. Para ello se utiliza un fichero YAML, en la línea 17, que especifica la creación del *deployment* en K8s junto con las credenciales de Sysdig para su conexión con NATS Streaming. Para la creación del *deployment* se ejecuta la función de la línea 22 y después en la línea 23 se espera a que la creación haya terminado satisfactoriamente. Estas funciones forman parte del módulo de Kubernetes de Chaos Toolkit *chaosk8s*. Si hay un error, saltará una excepción con el mensaje de la línea 25.

```
30  def after_experiment_control(
31      context: Experiment, configuration: Configuration = None, secrets: Secrets = None, **kwargs
32  ):
33      """
34      Terminate NATS BOX client (deployment) in the Kubernetes cluster
35      """
36
37      delete_deployment(name=NAME, ns=NAMESPACE)
38
39      pass
```

Figura 14: Eliminación de NATS Box en K8s

Cuando el experimento termina, el framework se encarga de ejecutar automáticamente la función de la Figura 14. Ésta ejecuta la función de la línea 37 para eliminar el *deployment* creado anteriormente de NATS Box.

[12]https://github.com/nats-io/nats-box

```python
def check_cluster(ns: str = "sysdigcloud", configuration: Configuration = None, secrets: Secrets = None):
    """
    Check if all NATS Streaming nodes are healthy
    """

    results = exec_in_pods(
        ns=ns,
        name_pattern="sysdigcloud-nats-streaming",
        container_name="nats-streaming",
        all=True,
        secrets=secrets,
        cmd=["/bin/sh", "-c", "curl localhost:8222/varz"],
    )

    for key, result in results.items():
        logger.debug(f"{key} - {result['stdout']}")
        json_data = json.loads(result["stdout"])
        if result["exit_code"] != 0:
            raise ActivityFailed("some NATS nodes are unhealthy")

    return len(json_data["connect_urls"])
```

Figura 15: Comprobación de la salud del clúster de NATS Streaming

Para comprobar que el clúster está saludable y que hay 3 nodos conectados, se implementa el módulo de la Figura 15. La función *exec_in_pods* ejecuta el comando *curl localhost:8222/varz/* en cada uno de los 3 contenedores de los servidores de NATS Streaming, recuperando la salida del comando. La dirección anterior es un endpoint de monitorización de NATS que, ente otros datos, devuelve las IPs de los nodos conectados al clúster.

Teniendo esta información, en las siguientes líneas se filtran los datos obtenidos del endpoint para devolver sólo el número de nodos en el clúster, contando el número de IPs activas.

```python
def read_write(ns: str = "sysdigcloud", configuration: Configuration = None, secrets: Secrets = None):
    """
    Check if NATS Streaming can read and write data.
    """

    try:
        for attempt in Retrying(
            wait=wait_exponential(multiplier=2, min=2, max=8), stop=stop_after_attempt(MAX_ATTEMPS)
        ):
            with attempt:
                benchmark_test()
                if not check_result_of_benchmark():
                    raise ActivityFailed("NATS cannot read and write data")
    except RetryError as e:
        e.reraise()

    return True
```

Figura 16: Comprobación de la lectura/escritura de NATS Streaming

Para verificar la lectura/escritura de NATS Streaming se implementa el código de la Figura 16. Esta es la función principal del módulo. Su estructura implementa una política de reintentos con *Tenacity* para realizar 6 reintentos con backoff exponencial, es decir, que cada vez que se realiza un nuevo intento el tiempo hasta el siguiente aumenta. Para cada uno de estos se ejecuta la función *benchmark_test()* que ejecuta el siguiente comando en nuestro NATS Box a través de la función *exec_in_pods*:

```
timeout 10s stan-bench -s $NATS_URL -c secure-cluster \
-u $NATS_USERNAME -pw $NATS_PASS -np 1 -ns 1 -n 1 chaostest
```

Este comando lanza un *benchmark* con un productor que envía un mensaje y un consumidor que lo recibe de forma concurrente. El resultado se valida posteriormente con la función *check_result_of_benchmark* (línea 23), que comprueba que la salida del comando no correcta. Si no lo es y quedan reintentos, se iterará al siguiente. Si es el último reintento, saltará la excepción definida en la línea 24 y ejecutada en la línea 26.

```
14  def terminate_node(nodeType, qty):
15      """
16      Shoot n NATS Streaming nodes (in a cluster of 3 nodes: primary, secondary, secondary)
17      nodeType: {primary, secondary, both}
18      qty: {1-3}
19      """
20
21      results = get_nats_cluster_info()
22      roles = get_roles(results)
23      sort_nodes(roles)
24      terminate_specific_pods(roles, nodeType, qty)
25
26      return True
```

Figura 17: Función para terminar nodos de NATS Streaming

En cuanto al módulo para terminar nodos de NATS Streaming, se define en la Figura 17 su función principal. En la línea 21, con la función *get_nats_cluster_info* se obtiene cierta información relevante sobre el clúster en cada uno de los nodos, a través del siguiente comando:

```
curl localhost:8222/streaming/serverz
```

```
40  def get_roles(results):
41      roles = {"primary": [], "secondary": [], "both": []}
42
43      for key, result in results.items():
44          if result["exit_code"] != 0:
45              raise ActivityFailed("some NATS nodes are unhealthy")
46
47          logger.debug(f"{key} - {result['stdout']}")
48
49          json_data = json.loads(result["stdout"])
50
51          if json_data["role"] == "Leader":
52              roles["primary"].append(json_data["node_id"])
53          if json_data["role"] == "Follower":
54              roles["secondary"].append(json_data["node_id"])
55          if json_data["role"] == "Candidate":
56              raise ActivityFailed("Some NATS node is with the not expected role Candidate")
57
58      return roles
```

Figura 18: Función para clasificar los nodos de NATS Streaming según sus roles

Después, con la función *get_roles* de la línea 22 se trata la información anterior para obtener los roles de cada uno de los nodos y guardarlos en diccionarios (ver línea 41). Si algún nodo declara que no es ni *Follower* ni *Leader*, saltará una excepción con el mensaje de *Some NATS node is with the not expected role Candidate* y se abortará la ejecución.

Con la función *sort_nodes* de la línea 23 se modificará el diccionario de roles para que la lista con clave *both* contenga los 3 nodos. Finalmente, con la función *terminate_specific_pods* de la línea 24 se eliminarán los pods de K8s según los parámetros *nodeType* y *qty*. Esta función ejecuta la función *terminate_pods* del módulo de K8s de Chaos Toolkit para eliminar cada uno de los nodos.

8.5.2. Definición de las pruebas

Las pruebas o experimentos de NATS Streaming están hechas con el framework de *Chaos Toolkit Sysdig*. La semántica de las pruebas se puede ver en el Anexo 8.7. Aquí se expone cada uno de los experimentos con la definición de que se hace en cada uno de las etapas de ejecución de las pruebas.

```yaml
---
version: 1.0.0
title: Killing Nats Streaming primary node
description: Killing Nats Streaming primary node
configuration:
  sysdig_api_ssl_verify: false
extensions:
  - name: junit-report
    category: nats_streaming
    classname: nats_streaming.kill-primary-node
controls:
  - name: sysdig-infer-config
    provider:
      type: python
      module: sysdigchaos.controls.infer_config
  - name: run-nats-box
    provider:
      type: python
      module: sysdigchaos.nats_streaming.controls.nats_box
  - name: read-write-continuous-hypothesis
    provider:
      type: python
      module: sysdigchaos.controls.continuous_hypothesis
      arguments:
        title: nats-replicas-should-read-and-write-data-continuously
        frequency: 1
        probes:
          - ref: nats-replicas-should-read-and-write-data
steady-state-hypothesis:
  title: NATS Streaming should be ok
  probes:
    - type: probe
      name: statefulset-replicas-should-be-ready
      tolerance: true
      provider:
        type: python
        module: sysdigchaos.k8s.probes.statefulset
        func: statefulset_fully_available
        arguments:
          ns: sysdigcloud
          name: "sysdigcloud-nats-streaming-cluster"
    - type: probe
      name: nats-cluster-should-be-ok
      tolerance: 3
      provider:
        type: python
        module: sysdigchaos.nats_streaming.probes.cluster
        func: check_cluster
    - type: probe
      name: nats-replicas-should-read-and-write-data
      tolerance: true
      provider:
        type: python
        module: sysdigchaos.nats_streaming.probes.pub_sub
        func: read_write
method:
  - type: action
    name: terminate-primary-node-nats
    provider:
      type: python
      module: sysdigchaos.nats_streaming.actions.pod
      func: terminate_node
      arguments:
        nodeType: primary
        qty: 1
    pauses:
      after: 60
```

Figura 19: Código de la prueba de NATS Streaming al terminar el nodo primario

En la Figura 19 se define la prueba de Chaos que mata el nodo primario de NATS Streaming y comprueba los requisitos que se han impuesto para el caso de uso.

Por un lado tenemos la *extension* de *JUnit report* para generar el log de la prueba cada vez que se ejecuta para su depuración posterior y que se explicará con más detalle en el anexo 8.10. Luego tenemos 3 *controls*, *sysdig-infer-config* que carga la configuración y credenciales al inicio de la ejecución para acceder al clúster de Sysdig on-premise creado previamente, *run-nats-box* que inicia *NATS Box* en el clúster para su uso posterior al inicio y se encarga de eliminarlo tras el experimento de forma automática, y *read-write-continuous-hypothesis* que se encarga de ejecutar de forma continua la prueba de lectura/escritura (*nats-replicas-should-read-and-write-data-continuously*) para comprobar la *continuous hypothesis*.

La *steady-state-hypothesis* implementa la comprobación del estado del *StatefulSet* en los nodos, la comprobación de que el clúster está saludable y finalmente se comprueba la lectura/escritura, todo esto se forma secuencial.

Finalmente tenemos el *method*, que se encarga de terminar el nodo primario y después realizar una espera de 60s antes de volver a ejecutar la *steady-state-hypothesis* y finalizar el experimento.

```
56  method:
57    - type: action
58      name: terminate-secondary-node-nats
59      provider:
60        type: python
61        module: sysdigchaos.nats_streaming.actions.pod
62        func: terminate_node
63        arguments:
64            nodeType: secondary
65            qty: 1
66      pauses:
67        after: 60
```

Figura 20: Método para terminar el nodo secundario de NATS Streaming

En cuanto al experimento para terminar el nodo secundario, el único cambio relevante que se necesita resaltar es cambiar los parámetros de la función que termina los nodos de NATS Streaming. En la *action* del *method* *terminate-secondary-node-nats* de la Figura 20 se le indica al parámetro *nodeType* que sólo queremos terminar nodos secundarios, y con el parámetro *qty* le indicamos que sólo queremos terminar uno de ellos.

```
1   ---
2   version: 1.0.0
3   title: Killing two Nats Streaming secondary nodes
4   description: Killing two Nats Streaming secondary nodes
5   configuration:
6     sysdig_api_ssl_verify: false
7   extensions:
8     - name: junit-report
9       category: nats_streaming
10      classname: nats_streaming.kill-two-secondary-nodes
11  controls:
12    - name: sysdig-infer-config
13      provider:
14        type: python
15        module: sysdigchaos.controls.infer_config
16    - name: run-nats-box
17      provider:
18        type: python
19        module: sysdigchaos.nats_streaming.controls.nats_box
20  steady-state-hypothesis:
21    title: NATS Streaming should be ok
22    probes:
23      - type: probe
24        name: statefulset-replicas-should-be-ready
25        tolerance: true
26        provider:
27          type: python
28          module: sysdigchaos.k8s.probes.statefulset
29          func: statefulset_fully_available
31          ns: sysdigcloud
32          name: "sysdigcloud-nats-streaming-cluster"
33      - type: probe
34        name: nats-cluster-should-be-ok
35        tolerance: 3
36        provider:
37          type: python
38          module: sysdigchaos.nats_streaming.probes.cluster
39          func: check_cluster
40      - type: probe
41        name: nats-replicas-should-read-and-write-data
42        tolerance: true
43        provider:
44          type: python
45          module: sysdigchaos.nats_streaming.probes.pub_sub
46          func: read_write
47  method:
48    - type: action
49      name: terminate-two-secondary-nodes-nats
50      provider:
51        type: python
52        module: sysdigchaos.nats_streaming.actions.pod
53        func: terminate_node
54        arguments:
55            nodeType: secondary
56            qty: 2
57      pauses:
58        after: 90
59
```

Figura 21: Código de la prueba de NATS Streaming al terminar 2 nodos secundarios

En la Figura 21 se puede observar el experimento relacionado con la eliminación de los 2 nodos secundarios de NATS Streaming. La diferencia principal con los anteriores es que esta vez no se incluye la *continuous hypothesis* para comprobar la lectura/escritura de forma continua durante el experimento. Y quitando la modificación de los parámetros de la función para terminar los nodos, en la que le pasamos que se quieren eliminar 2 nodos (qty: 2) secundarios (nodeType: secondary), el experimento no tiene más cambios. Además, tras la eliminación de los nodos se aumentará la espera hasta 90s para dar tiempo a que se recuperen los 2 nodos.

```
47  method:
48    - type: action
49      name: terminate-primary-and-secondary-node-nats
50      provider:
51        type: python
52        module: sysdigchaos.nats_streaming.actions.pod
53        func: terminate_node
54        arguments:
55            nodeType: both
56            qty: 2
57      pauses:
58        after: 90
```

Figura 22: Método para terminar el nodo primario y secundario de NATS Streaming

Para el experimento relacionado con la eliminación tanto del nodo primario como de un nodo secundario mostrado en la Figura 22, el cambio principal es la modificación de parámetros de la función *terminate_node* con los parámetros *nodeType: both* para indicar que se quiere eliminar tanto nodo primario como secundarios, y con *qty: 2* para indicar que se elimine primero el primario y luego un secundario, es decir, 2 en total.

```
47  method:
48    - type: action
49      name: terminate-all-cluster-nodes-nats
50      provider:
51        type: python
52        module: sysdigchaos.nats_streaming.actions.pod
53        func: terminate_node
54        arguments:
55            nodeType: both
56            qty: 3
57      pauses:
58        after: 120
```

Figura 23: Método para terminar los 3 nodos del clúster de NATS Streaming

Y finalmente en la Figura 23 se muestra el experimento que termina los 3 nodos del clúster y comprueba los requisitos del caso de uso. Como en el experimento anterior, la diferencia reside en la modificación de los parámetros para que la función se encargue de eliminar correctamente todos los nodos del clúster. Esta vez se esperará 120s para dar tiempo a que los nodos se recuperen y se vuelva a comprobar la *steady-state-hypothesis*.

8.6. Kafka

8.6.1. Zookeeper

Apache Zookeeper es un componente utilizado por Kafka para guardar cierta información sobre el estado del clúster de Kafka y sus nodos, así como también información acerca del estado de las particiones y los topics, su distribución entre los distintos brókers, cuáles son las particiones líderes, etc. Así mismo, Zookeeper puede formar parte de un clúster para aportar HA y que si un nodo deja de estar disponible por alguna razón, cualquiera de los dos nodos restantes puedan seguir dando servicio. Esto es lo que ocurre en la implementación de Sysdig, con un clúster de 3 nodos.

8.6.2. Implementación de los módulos

```
11  def check_cluster_is_healthy():
12
13      check_connection_among_all_brokers()
14      check_active_controllers()
15      check_all_nodes_readiness()
16
17      return True
```

Figura 24: Función principal para comprobar que el clúster de Kafka está saludable

Se ha definido que para comprobar que el clúster de Kafka está saludable se realizan 3 comprobaciones (ver Figura 24): comprobar la conexión entre los 3 nodos, comprobar que sólo hay un nodo con el rol de *ActiveController* y comprobar que no haya particiones *UnderReplicated* para cada uno de los brókers, respectivamente.

```
20  def check_connection_among_all_brokers(
21      ns: str = "sysdigcloud", configuration: Configuration = None, secrets: Secrets = None
22  ):
23      for nodeName in KAFKA_NODES:
24          results = exec_in_pods(
25              ns=ns,
26              name_pattern="cp-kafka",
27              container_name="broker",
28              all=True,
29              secrets=secrets,
30              cmd=["/bin/sh", "-c", f"echo $(nc -vz {nodeName}.cp-kafka-headless.sysdigcloud 9092 2>&1)"],
31          )
32
33          for key, result in results.items():
34              logger.debug(f"{key} - {result['stdout']}")
35              if result["exit_code"] != 0:
36                  raise ActivityFailed("some Kafka nodes are unhealthy")
37              try:
38                  assert "Connected to" in result["stdout"]
39              except AssertionError:
40                  raise ActivityFailed("some Kafka nodes are unhealthy")
41
42      return True
```

Figura 25: Comprobación de la conexión entre todos los nodos del clúster de Kafka

De forma más específica, para la función *check_connection_among_all_brokers* definida en la Figura 25 se comprueba la conexión de cada bróker con el resto. Para ello se utiliza el

comando de la línea 30. Si para alguna de estas pruebas la conexión falla, saltará la excepción de la línea 36 o 40, dependiendo del tipo de error.

```
45  def check_active_controllers(
46      ns: str = "sysdigcloud", configuration: Configuration = None, secrets: Secrets = None
47  ):
48      results = exec_in_pods(
49          ns=ns,
50          name_pattern="cp-kafka",
51          container_name="broker",
52          all=True,
53          secrets=secrets,
54          cmd=[
55              "/bin/sh",
56              "-c",
57              'unset JMX_PORT && kafka-run-class kafka.tools.JmxTool --object-name \
58                  "kafka.controller":name="ActiveControllerCount",type="KafkaController" \
59                  --jmx-url service:jmx:rmi:///jndi/rmi://:9010/jmxrmi --one-time true',
60          ],
61      )
62
63      num_active_controllers = 0
64
65      for key, result in results.items():
66          logger.debug(f"{key} - {result['stdout']}")
67          if result["exit_code"] != 0:
68              raise ActivityFailed("some Kafka nodes are unhealthy")
69          if ",1\n" in result["stdout"]:
70              num_active_controllers = num_active_controllers + 1
71
72      if num_active_controllers > 1:
73          raise ActivityFailed("some Kafka nodes are unhealthy")
74
75      return True
```

Figura 26: Comprobación para saber si hay un *ActiveController* en el clúster de Kafka

En la Figura 26 se observa la definición de la función *check_active_controllers*, de tal forma que se ejecuta el comando de la línea 57 en cada bróker para saber si cada uno de ellos tiene el rol de *ActiveController*. La salida del comando está formada por un *timestamp* seguido de 0 o 1. Si ese bróker tiene el rol, devolverá 1. En las líneas siguientes se tratan los resultados para cada nodo y se realiza un conteo acerca de cuántos se auto consideran *ActiveController*. Si finalmente hay más de 1, saltará la excepción de la línea 73.

```
78  def check_all_nodes_readiness(
79      ns: str = "sysdigcloud", configuration: Configuration = None, secrets: Secrets = None
80  ):
81      results = exec_in_pods(
82          ns=ns,
83          name_pattern="cp-kafka",
84          container_name="broker",
85          all=True,
86          secrets=secrets,
87          cmd=["/bin/sh", "-c", "./readiness-check.sh"],
88      )
89
90      for key, result in results.items():
91          logger.debug(f"{key} - {result['stdout']}")
92          if result["exit_code"] != 0:
93              raise ActivityFailed("some Kafka nodes are unhealthy")
94          try:
95              assert "All partitions in sync for this broker." in result["stdout"]
96          except AssertionError:
97              raise ActivityFailed("some Kafka nodes are unhealthy")
98
99      return True
```

Figura 27: Comprobación de particiones *UnderReplicated* en los nodos de Kafka

La última de las pruebas se especifica en la Figura 27. La función *check_all_nodes_readiness* ejecuta un script que comprueba si existen particiones *UnderReplicated* en el bróker que se ejecuta. Este script se encuentra en cada nodo de Kafka y ya se encuentra disponible en el sistema debido a que se utiliza para realizar pruebas de *Readiness* de K8s periódicamente en Sysdig. Si la ejecución en algún nodo deriva en algún error, se lanzará una excepción *ActivityFailed*, en la línea 93 o 97, dependiendo del tipo de error.

```
9   def get_active_controller(
10      ns: str = "sysdigcloud", configuration: Configuration = None, secrets: Secrets = None
11  ):
12      results = exec_in_pods(
13          ns=ns,
14          name_pattern="zookeeper",
15          container_name="server",
16          secrets=secrets,
17          qty=1,
18          cmd=[
19              "/bin/sh",
20              "-c",
21              "zookeeper-shell localhost:2181 get /brokers/ids/$(zookeeper-shell localhost:2181 \
22                  get /controller | tail -1 | jq .brokerid) | tail -1 | jq .host",
23          ],
24      )
25
26      for key, result in results.items():
27          logger.debug(f"{key} - {result['stdout']}")
28          if result["exit_code"] != 0:
29              raise ActivityFailed("Kafka's controller could not be obtained")
30          kafka_controller = result["stdout"]
31
32      return kafka_controller
```

Figura 28: Función para obtener el nodo de Kafka con el rol de *ActiveController*

Antes de definir el módulo para terminar un nodo de Kafka, se define la función de la Figura 28. Su cometido es obtener el nombre DNS del *host* de Kafka con el rol de *ActiveController*. En la línea 21 se puede ver el comando utilizado ejecutado en un nodo de Zookeeper, a través de *zookeeper-shell*, para obtener la información. En las siguientes líneas se trata el

resultado, mientras que si hay algún error se lanza la excepción *ActivityFailed* de la línea 29.

```
9  def terminate_active_controller():
10     controller: str = get_active_controller().split(".")[0]
11     controller = controller[1:]
12     logger.debug(f"Pod to shoot: {controller}")
13
14     terminate_specific_pod(controller)
15
16     return True
```

Figura 29: Función para terminar el nodo de Kafka con el rol de *ActiveController*

Ahora, para implementar la función que termina un nodo con el rol de *ActiveController*, podemos obtener fácilmente el nombre del pod que contiene el bróker con dicho rol y poder eliminarlo fácilmente como se ve en la función de la Figura 29. Una vez obtenido el nombre del pod se ejecuta la función *terminate_specific_pod* de la línea 14, que ejecuta la función *terminate_pods* del módulo de K8s de Chaos Toolkit para eliminar el pod.

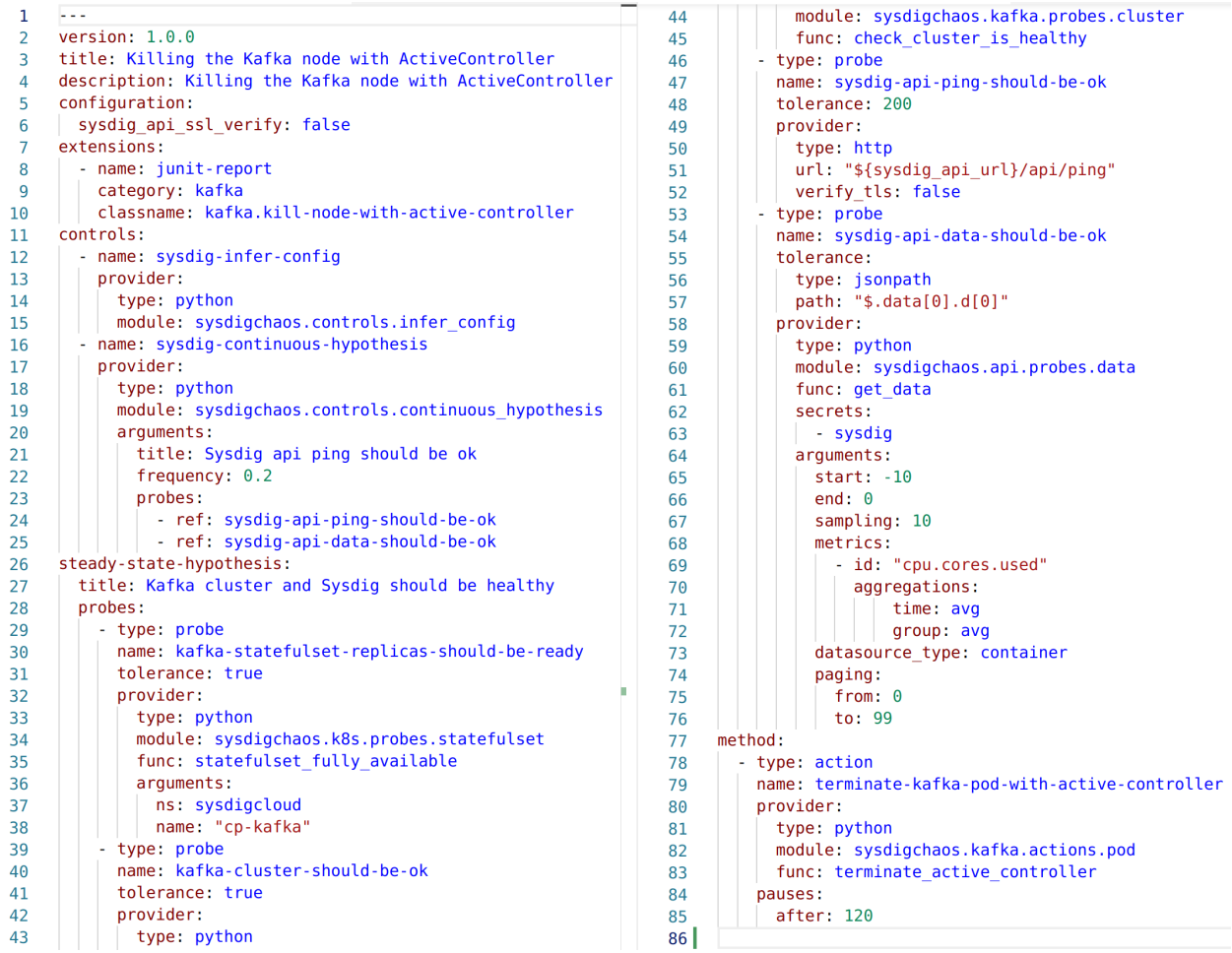
```
19 def terminate_node_without_active_controller(qty: int = 1):
20     CLUSTER_PODS = ["cp-kafka-0", "cp-kafka-1", "cp-kafka-2"]
21
22     controller: str = get_active_controller().split(".")[0]
23     controller = controller[1:]
24     CLUSTER_PODS.remove(controller)
25
26     for _ in range(qty):
27         no_controller = CLUSTER_PODS.pop()
28         logger.debug(f"Pod to shoot: {no_controller}")
29
30         terminate_specific_pod(no_controller)
31
32     return True
```

Figura 30: Función para terminar nodos de Kafka sin el rol de *ActiveController*

Lo mismo ocurre con la función para eliminar un nodo sin dicho rol. En la Figura 30 se detalla la implementación de la función. La única diferencia con la función anterior es que se tiene una lista con los nombres de los pods de los 3 nodos del clúster, y eliminando el nodo *ActiveController* de la lista nos quedamos con los 2 que no lo son. A partir de ahí, y dependiendo de la cantidad de nodos que queremos eliminar según el parámetro *qty*, se eliminan los nodos de la misma forma que en la función de la Figura 29.

8.6.3. Definición de las pruebas

La definición de las pruebas de Chaos de Kafka es similar a las pruebas diseñadas con NATS Streaming, a efectos de la estructura de los experimentos y su semántica.



```yaml
---
version: 1.0.0
title: Killing the Kafka node with ActiveController
description: Killing the Kafka node with ActiveController
configuration:
  sysdig_api_ssl_verify: false
extensions:
  - name: junit-report
    category: kafka
    classname: kafka.kill-node-with-active-controller
controls:
  - name: sysdig-infer-config
    provider:
      type: python
      module: sysdigchaos.controls.infer_config
  - name: sysdig-continuous-hypothesis
    provider:
      type: python
      module: sysdigchaos.controls.continuous_hypothesis
      arguments:
        title: Sysdig api ping should be ok
        frequency: 0.2
        probes:
          - ref: sysdig-api-ping-should-be-ok
          - ref: sysdig-api-data-should-be-ok
steady-state-hypothesis:
  title: Kafka cluster and Sysdig should be healthy
  probes:
    - type: probe
      name: kafka-statefulset-replicas-should-be-ready
      tolerance: true
      provider:
        type: python
        module: sysdigchaos.k8s.probes.statefulset
        func: statefulset_fully_available
        arguments:
          ns: sysdigcloud
          name: "cp-kafka"
    - type: probe
      name: kafka-cluster-should-be-ok
      tolerance: true
      provider:
        type: python
        module: sysdigchaos.kafka.probes.cluster
        func: check_cluster_is_healthy
    - type: probe
      name: sysdig-api-ping-should-be-ok
      tolerance: 200
      provider:
        type: http
        url: "${sysdig_api_url}/api/ping"
        verify_tls: false
    - type: probe
      name: sysdig-api-data-should-be-ok
      tolerance:
        type: jsonpath
        path: "$.data[0].d[0]"
      provider:
        type: python
        module: sysdigchaos.api.probes.data
        func: get_data
        secrets:
          - sysdig
        arguments:
          start: -10
          end: 0
          sampling: 10
          metrics:
            - id: "cpu.cores.used"
              aggregations:
                time: avg
                group: avg
          datasource_type: container
          paging:
            from: 0
            to: 99
method:
  - type: action
    name: terminate-kafka-pod-with-active-controller
    provider:
      type: python
      module: sysdigchaos.kafka.actions.pod
      func: terminate_active_controller
    pauses:
      after: 120
```

Figura 31: Código de la prueba de Kafka al terminar nodo con el rol de *ActiveController*

Como se observa en la Figura 31, se trata del experimento que elimina el nodo con el rol de *ActiveController*. Se usa la misma *extension* para generar logs y el mismo *control* para cargar la configuración y credenciales del clúster de Sysdig que con NATS Streaming.

En esta ocasión para la *continuous hypothesis* se ejecutan dos pruebas cada 0.2s, estas son *sysdig-api-ping-should-be-ok* para comprobar que la API de Sysdig responde y *sysdig-api-data-should-be-ok* para chequear que se reciben datos de la API. En la *steady-state-hypothesis* se realiza de forma cronológica las siguientes pruebas: primero se comprueba el estado del *StatefulSet* de los brókers de Kafka, se comprueba que el clúster está saludable y finalmente se ejecutan las dos pruebas que se han mencionado en la *continuous hypothesis*.

La prueba para comprobar que la API de Sysdig está funcionando definida en la línea 46 espera que el ping devuelva el código HTTP (*type: http*) 200 (*tolerance: 200*). Este *request* se envía a la URL definida en el argumento *url*, en la que *${sysdig_api_url}* es una variable definida con anterioridad en el control *sysdig-infer-config*.

En cuanto a la prueba *sysdig-api-data-should-be-ok* definida en la línea 53 se quiere obtener la métrica *cpu.cores.used* que es una de las métricas mostradas al cliente a través de la interfaz de usuario de Sysdig. No se va a entrar en mucho detalle pero con los argumentos *start: 10* y *end: 0* le estamos diciendo que se obtendrán los datos de los últimos 10 segundos. Esta métrica es una de las métricas que pasan por Kafka, por lo que leer los datos de ésta nos sirve para comprobar que el funcionamiento de Kafka sigue siendo el esperado y funciona con normalidad.

En cuanto al *method* del experimento, se ejecuta la función que elimina el nodo con el rol de *ActiveController* y se realiza una espera de 120s para dar tiempo a que el bróker de Kafka se recupere y se pueda terminar el experimento con la comprobación del estado estable del sistema.

```
77  method:
78    - type: action
79      name: terminate-kafka-pod-without-active-controller
80      provider:
81        type: python
82        module: sysdigchaos.kafka.actions.pod
83        func: terminate_node_without_active_controller
84      pauses:
85        after: 120
```

Figura 32: Método para terminar un nodo de Kafka sin el rol de *ActiveController*

Para el experimento relacionado con eliminar el bróker de Kafka sin el rol de *ActiveController* mostrado en la Figura 32, la diferencia reside en el *method*, ya que todo lo demás es igual que el el experimento anterior. En él se ejecuta la función que elimina dicho nodo y se realiza la espera de 120s antes de la ejecución de la *steady-state-hypothesis*.

8.7. Pruebas de Chaos Toolkit

Tanto Chaos Toolkit como *Chaos Toolkit Sysdig* tienen la misma estructura de pruebas. En todas ellas se utiliza la misma semántica para definir las acciones antes, durante y después del experimento:

- **Probes**: son las pruebas se se usan en el experimento para comprobar una serie de condiciones en el sistema.
- **Actions**: son actividades concretas del experimento. Por ejemplo, la eliminación de pods de un clúster.
- **Steady State Hypothesis**: describe las pruebas y acciones del estado estable del sistema definiendo su comportamiento normal. Éste se ejecuta antes y después del experimento para comprobar si ha habido alguna variación en el estado estable.
- **Method**: es el conjunto de acciones y pruebas que se ejecutan para realizar el chaos del experimento entre las comprobaciones del *Steady State Hypothesis*.
- **Rollbacks**: son el conjunto de acciones y pruebas que se ejecutan al final del experimento para restablecer el sistema de los efectos de las acciones de chaos.
- **Controls**: son el conjunto de controles que se ejecutan con el objetivo de realizar acciones y pruebas que tengan que ver con aspectos de configuración antes, durante o después del experimento. Entre otros aspectos, la *hipótesis continua* de las pruebas para comprobar el estado estable del sistema de forma continua.
- **Extensions**: son módulos instalables que añaden ciertas funcionalidades al experimento. Un ejemplo que se verá en las pruebas de Chaos es la extensión de *JUnit report*, que genera logs para su posterior depuración.

Además de todo esto, cada experimento define su propio título y una descripción acerca de lo que se está haciendo, además de poder definir la versión de la prueba, por si se quiere llevar un control de versiones.

Para cada una de las pruebas, acciones, etc. se define su tipo, de tal forma que se pueden ejecutar funciones de Python con sus parámetros o argumentos, *requests* HTTP, entre otros. Además, en *Chaos Toolkit Sysdig* se usan las *tolerancias* para definir qué es lo que queremos que nos devuelvan las pruebas y acciones. Un ejemplo: cuando lanzamos la prueba para comprobar que existen 3 nodos en el clúster de NATS Streaming, la *tolerancia* se define en 3, ya que ese es el número de nodos que nos debe de devolver la función de Python, en este caso. Las *tolerancias* pueden ser booleanos, cadenas, enteros, etc.

8.8. Ejecución de las pruebas con Chaos Toolkit

Para los experimentos de *Chaos Toolkit Sysdig*, éstos se ejecutan en un clúster de Sysdig. Hay dos formas de lanzar el clúster, a través de Jenkins con un Job que lanza el OSC (*On-Prem Stack Creator*) para crear el clúster on-premise de forma remota en AWS, o a través de la ejecución de OSC localmente (para crear el clúster de forma local). En todas las pruebas, tanto en NATS Streaming como en Kafka, éstas se harán con la instalación de un clúster remoto a través de Jenkins, por facilidad de uso y preferencia de la empresa.

Las pruebas de Chaos de NATS Streaming y Kafka siguen el mismo proceso de ejecución. Cuando queremos ejecutar las pruebas de forma manual en un clúster ya creado con anterioridad y teniendo todo configurado para su conexión con K8s, se siguen los siguientes pasos.

```
# Create a Python virtual environment
python -m venv venv
source venv/bin/activate
pip install --upgrade pip

# Install the required dependencies
pip install -r requirements.txt
make deps

# Run several experiments
make run-experiment BY="<dir>"
```

Después de entrar en el *Virtual Enviroment* de Python y teniendo todas las dependencias instaladas, hay varias formas de ejecutar los experimentos de *Chaos Toolkit Sysdig*. La más sencilla es a través la ejecución del archivo *makefile* que contiene los comandos de ejecución del framework y de monitorización con logs. Como se observa en el bloque de código superior, es suficiente con poner el directorio con los experimentos que queremos lanzar o bien su ruta completa. Un ejemplo de ejecución se muestra en la Figura 33.

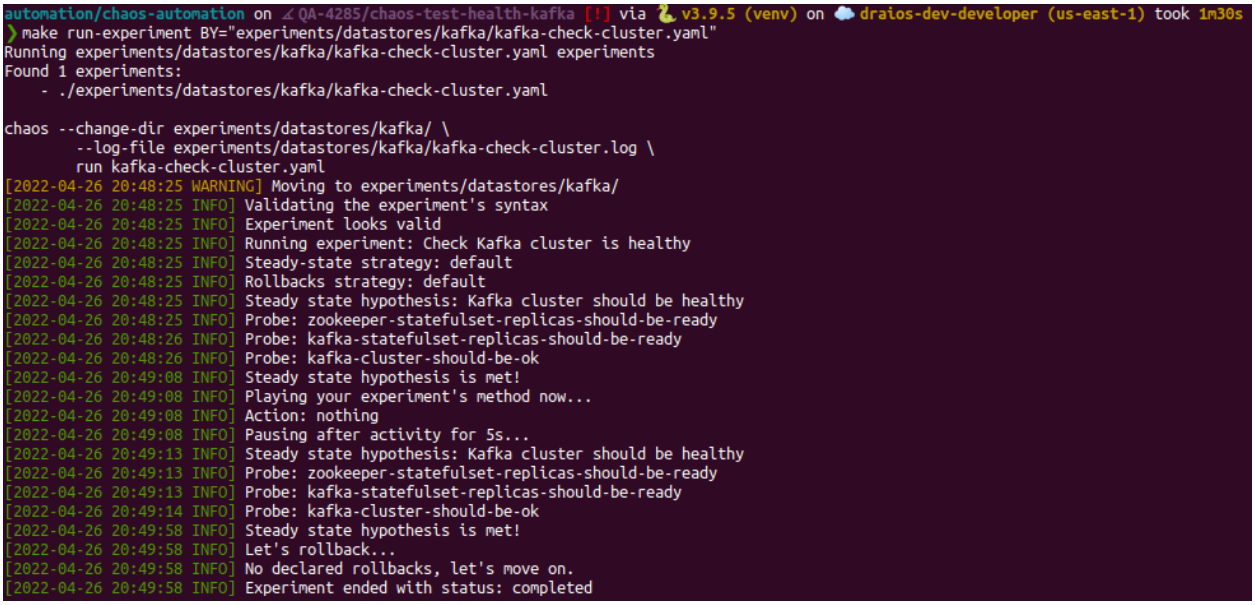


Figura 33: Ejecución de un experimento con *Chaos Toolkit Sysdig*

Como se observa en esta última figura, al ejecutar una prueba de *Chaos Toolkit Sysdig*, se siguen los pasos mencionados en el apartado de diseño de las pruebas con Chaos Toolkit de la Figura 5. Además, se muestra todo el proceso con cada etapa ejecutada junto con una marca de tiempo y, finalmente, el resultado del experimento.

8.9. Pruebas unitarias con Chaos Toolkit

Para realizar las pruebas unitarias del código creado en el framework de *Chaos Toolkit Sysdig* se utiliza lo que se llama como *Funciones Mockeadas*. La idea es simular (*mockear*) la ejecución de partes más pequeñas de código como funciones, definiendo sus resultados de forma previa para poder comprobar que el código restante está bien diseñado y hace lo que nosotros teníamos pensado para los distintos casos de uso posibles.

8.9.1. NATS Streaming

Teniendo en cuenta la implementación de los módulos explicada en el anexo 4.4.2, el diseño de las pruebas unitarias es el siguiente:

- Para el módulo acerca de la salud del clúster:
  - Se verifica que la función principal del módulo devuelve *3* cuando existen 3 direcciones IP configuradas en el clúster.
  - Se verifica que la función principal del módulo arroja una excepción *ActivityFailed* cuando no se puede obtener la información de algún nodo.
  - Se verifica que la función principal del módulo devuelve un número distinto de 3 cuando los nodos declaran conocer a un número distinto de 3 nodos en el clúster.
- Para el módulo de lectura/escritura:
  - Se verifica que la función principal del módulo devuelve *True* cuando la lectura/escritura con NATS Box es correcta.
  - Se verifica que la función principal del módulo arroja una excepción *ActivityFailed* cuando la lectura/escritura no se ha podido realizar.
- Para el módulo de terminar nodos:
  - Se verifica que la función principal del módulo devuelve *True* para cada una de las posibilidades con sus parámetros (eliminar un nodo secundario, eliminar dos nodos secundarios, etc).
  - Se verifica que la función principal del módulo arroja una excepción *ActivityFailed* cuando algún nodo se encuentra con el rol de *Candidate*.

Por poner un ejemplo de forma detallada acerca de la implementación de las pruebas unitarias en NATS Streaming, se detallan las pruebas referentes al módulo que comprueba la salud del clúster del *datastore*. El comportamiento normal de la función principal del módulo, *check_cluster*, debe de devolver 3, que corresponde al número de nodos en el clúster.

```python
def test_nats_streaming_is_healthy(mocker):
    exec_in_pods = mocker.patch("sysdigchaos.nats_streaming.probes.cluster.exec_in_pods")
    exec_in_pods.return_value = {
        "nats-streaming-0": {
            "exit_code": 0,
            "stdout": '{"connect_urls": ["192.168.1.1", "192.168.1.2", "192.168.1.3"]}',
        },
        "nats-streaming-1": {
            "exit_code": 0,
            "stdout": '{"connect_urls": ["192.168.1.1", "192.168.1.2", "192.168.1.3"]}',
        },
        "nats-streaming-2": {
            "exit_code": 0,
            "stdout": '{"connect_urls": ["192.168.1.1", "192.168.1.2", "192.168.1.3"]}',
        },
    }

    assert check_cluster() == 3
```

Figura 34: Prueba unitaria para verificar que hay 3 nodos de NATS Streaming

Como se puede observar en la Figura 34 se definen los datos de salida de la función *exec_in_pods*, que es la que se encarga de obtener la información de las direcciones IP del clúster para cada uno de los nodos. En este caso, para cada nodo, se tienen 3 direcciones IP con el objetivo de simular un comportamiento normal. Al final de la prueba se chequea que la función *check_cluster*, con los datos mockeados, devuelve 3.

```python
def test_nats_streaming_is_not_healthy(mocker):
    exec_in_pods = mocker.patch("sysdigchaos.nats_streaming.probes.cluster.exec_in_pods")
    exec_in_pods.return_value = {
        "nats-streaming-0": {
            "exit_code": 0,
            "stdout": '{"connect_urls": ["192.168.1.1", "192.168.1.2", "192.168.1.3"]}',
        },
        "nats-streaming-1": {
            "exit_code": 0,
            "stdout": '{"connect_urls": ["192.168.1.1", "192.168.1.2", "192.168.1.3"]}',
        },
        "nats-streaming-2": {"exit_code": 1, "stdout": "{}"},
    }

    with pytest.raises(ActivityFailed) as e:
        check_cluster()

    assert "some NATS nodes are unhealthy" in str(e.value)
```

Figura 35: Prueba unitaria para verificar fallo al obtener nodos de NATS Streaming

Sin embargo, si uno de los nodos falla cuando se ejecuta la función *exec_in_pods* y devuelve un código de error 1, debería de fallar la ejecución con una excepción *ActivityFailed*. La Figura 35 muestra la implementación.

```python
def test_nats_streaming_is_not_healthy_2(mocker):
    exec_in_pods = mocker.patch("sysdigchaos.nats_streaming.probes.cluster.exec_in_pods")
    exec_in_pods.return_value = {
        "nats-streaming-0": {"exit_code": 0, "stdout": '{"connect_urls": ["192.168.1.1", "192.168.1.3"]}'},
        "nats-streaming-1": {"exit_code": 0, "stdout": '{"connect_urls": ["192.168.1.1", "192.168.1.3"]}'},
        "nats-streaming-2": {"exit_code": 0, "stdout": '{"connect_urls": ["192.168.1.1", "192.168.1.3"]}'},
    }

    assert check_cluster() != 3
```

Figura 36: Prueba unitaria 2 para verificar fallo al obtener nodos de NATS Streaming

En este último caso de la Figura 36 se verifica que el número de nodos en el clúster que devuelve la función *check_cluster* cuando los nodos devuelven 2 IP es distinto a 3.

8.9.2. Kafka

Teniendo en cuenta la implementación de los módulos explicada en el anexo 4.5.2, el diseño de las pruebas unitarias es el siguiente:

- Para el módulo acerca de la salud del clúster:
  - Se verifica que la función *check_connection_among_all_brokers* devuelve *True* cuando los 3 nodos logran conectarse entre sí.
  - Se verifica que la función *check_connection_among_all_brokers* arroja una excepción *ActivityFailed* cuando no se puede realizar la conexión entre algún nodo.
  - Se verifica que la función *test_check_active_controllers* devuelve *True* cuando sólo un nodo declara ser *ActiveController*.
  - Se verifica que la función *check_active_controllers* arroja una excepción *ActivityFailed* cuando no se puede obtener la información en algún nodo o cuando más de un nodo dice ser *ActiveController*.
  - Se verifica que la función *check_all_nodes_readiness* devuelve *True* cuando no existen particiones *UnderReplicated* en ningún nodo.
  - Se verifica que la función *check_all_nodes_readiness* arroja una excepción *ActivityFailed* cuando no se puede obtener la información en algún nodo o cuando algún nodo tiene particiones *UnderReplicated*.
- Para el módulo que obtiene el nombre del *ActiveController*:
  - Se verifica que la función principal del módulo devuelve *True* cuando se obtiene el nombre DNS de dicho nodo correctamente.
  - Se verifica que la función principal del módulo arroja una excepción *ActivityFailed* cuando no se puede obtener el nombre DNS de dicho nodo.
- Para el módulo de terminar nodos:

- Se verifica que la función *terminate_active_controller* devuelve *True* cuando se ha realizado la eliminación del nodo *ActiveController* correctamente.

- Se verifica que la función *terminate_node_without_active_controller* devuelve *True* tanto si se ha realizado la eliminación de uno como de dos nodos sin el rol de *ActiveController* correctamente.

Por poner un ejemplo de forma detallada acerca de la implementación de las pruebas unitarias en Kafka, se detallan las pruebas referentes al módulo que obtiene el nombre DNS con el rol de *ActiveController*.

```
8  def test_get_active_controller_is_healthy(mocker):
9      stdout = "cp-kafka-0.cp-kafka-headless.sysdigcloud"
10
11     exec_in_pods = mocker.patch("sysdigchaos.kafka.controller.exec_in_pods")
12     exec_in_pods.return_value = {
13         "cp-kafka-0": {
14             "exit_code": 0,
15             "stdout": stdout,
16         },
17     }
18
19     assert stdout in get_active_controller()
```

Figura 37: Prueba unitaria para obtener nombre DNS del *ActiveController*

En la Figura 37 se muestra la prueba unitaria referente a la obtención del nombre del nodo con el rol de *ActiveController*. Para ello se mockea la función *exec_in_pods*, que es la que se encarga de obtener dicha información. Al final, en la línea 19, se realiza la ejecución de la función principal del módulo *get_active_controller* con los datos mockeados para verificar que se obtiene el nombre del nodo.

```
22 def test_get_active_controller_is_not_healthy(mocker):
23     exec_in_pods = mocker.patch("sysdigchaos.kafka.controller.exec_in_pods")
24     exec_in_pods.return_value = {
25         "cp-kafka-0": {
26             "exit_code": 1,
27             "stdout": "",
28         },
29     }
30
31     with pytest.raises(ActivityFailed) as e:
32         get_active_controller()
33     assert "Kafka's controller could not be obtained" in str(e.value)
```

Figura 38: Prueba unitaria para verificar fallo al obtener *ActiveController*

Por el otro lado, según se ve en la Figura 38, para verificar que el módulo falla, se simula un error en la obtención del dato de *exec_in_pods* y se comprueba que se ha desencadenado una excepción *ActivityFailed* con el mensaje de error correspondiente.

8.10. Depuración de errores con Chaos Toolkit y Jenkins

La depuración de los resultados de nuestras pruebas es esencial para poder encontrar las causas de los problemas, además de que nos sea fácil de llevarlo a cabo, aún más si se realizan de forma automatizada a través de Jenkins CI.

Chaos Toolkit Sysdig, durante la ejecución de las pruebas, es capaz de generar logs y journals. Los journals son archivos .json que detallan de forma simplificada las pruebas y acciones que se han lanzado en cada experimento, el tiempo que han tardado, si se han cumplido las tolerancias, si se ha cumplido el *steady-state-hypothesis* o si por el contrario se ha desviado, etc. Los logs (archivos .log), sin embargo, entran en un mayor nivel de detalle donde se muestra, a parte de los mensajes de salida estándar de las ejecuciones de las pruebas, todos los mensajes de debug del propio framework así como del código de Python de nuestros módulos y que podemos insertar de la siguiente forma:

```
logger.debug("mensaje de debug")
```

Por otro lado, el framework incorpora un sistema de reports en formado .md, y junit reports en formato .xml que consisten en el tratamiento de los journals generados anteriormente para generar otros archivos con una sintaxis más legible. Estos archivos son utilizados por Jenkins para mostrar los resultados de las pruebas de forma más interactiva en cada uno de los Job de las pruebas de Chaos Engineering.

Todos los test con fallos

Nombre del test	Duración	Antigüedad
kafka.kill-node-with-active-controller.after.kafka-statefulset-replicas-should-be-ready	15 Min	1
kafka.kill-node-without-active-controller.before.kafka-statefulset-replicas-should-be-ready	15 Min	1

Todos los tests

Paquete	Duración	Fallidos	(diferencias)	Omitir	(diferencias)	Pass	(diferencias)	Total	(diferencias)
cassandra.kill-replicas	2 Min 42 Seg	0		0		734	-32	734	-32
kafka.check-cluster	57 Seg	0		0		4		4	
kafka.kill-node-with-active-controller	15 Min	1	+1	0		1042	-10	1043	-9
kafka.kill-node-without-active-controller	15 Min	1	+1	0		0	-1078	1	-1077
nats_streaming.kill-all-cluster-nodes	3.1 Seg	0		0		6		6	
nats_streaming.kill-primary-and-secondary-node	6 Seg	0		0		6		6	
nats_streaming.kill-primary-node	42 Seg	0		0		32		32	
nats_streaming.kill-secondary-node	42 Seg	0		0		33		33	
nats_streaming.kill-two-secondary-nodes	3 Seg	0		0		6		6	
postgres.kill-replicas	58 Seg	0		0		393	-51	393	-51
redis.kill-replicas	1.6 Seg	0		0		378	-12	378	-12

Figura 39: Resultado de las pruebas de Chaos en Jenkins

En la Figura 39 se puede observar los resultados de las pruebas de Chaos desde la interfaz de usuario de Jenkins para un Job en concreto. En la parte de abajo se pueden ver todos los experimentos o tests ejecutados por Chaos Toolkit con información acerca de la duración del experimento, sus fallos, etc. además de poder ver más información si se decide hacer clic

en la prueba. Por otro lado, en la parte de arriba se puede ver el par de ellos que ha fallado. Toda esta información es extraída de los report generados anteriormente durante la ejecución de las pruebas, como se ha mencionado previamente.

Figura 40: Salida estándar de un error de una prueba en Jenkins

Si queremos obtener más detalles acerca de uno de los fallos, en la Figura 40 se observa la salida estándar de la prueba *kill-node-with-active-controller* al desplegar el + de la primera prueba que ha fallado de la Figura 39. En concreto, se obtiene la salida de la prueba *kafka-replicas-should-be-ready*, que con una tolerancia de *True*, no se ha obtenido nada, lo que quiere decir que los nodos no se han logrado recuperar correctamente (*StatefulSet* no está *Ready*). Podemos obtener más detalles si nos dirigimos a la sección *Detalles del error* o si hacemos clic en la prueba.