Asegurar la seguridad en las reversiones durante las implementaciones

Sandeep Pokkunuri

Uno de los principios referentes sobre como creamos soluciones en Amazon es Evitar pasar por puertas en un solo sentido. Esto significa que nos apartamos de las elecciones que son difíciles de revertir o extender. Aplicamos este principio a todos los pasos del desarrollo de software: desde el diseño de productos, características, API y sistemas de backend hasta las implementaciones. En este artículo, describiré cómo aplicamos este principio a las implementaciones de software.

Una implementación lleva al entorno de software de un estado (versión) a otro. El software puede funcionar perfectamente en ambos estados. Sin embargo, puede que no funcione bien durante o después de la transición futura (actualización o consolidación) o transición reversa (cambio a una versión anterior o reversión). Cuando el software no funciona correctamente, provoca una interrupción del servicio que lo hace poco confiable para los clientes. En este artículo, parto del supuesto de que ambas versiones del software funcionan como se esperaba. Mi enfoque es cómo garantizar que ni las consolidaciones ni las reversiones generen errores durante la implementación.

Antes de lanzar una versión nueva de un software, la probamos en un entorno beta o gamma junto con diversas dimensiones, como su funcionalidad, simultaneidad, rendimiento, escala y administración de fallos de la última etapa. Esta prueba nos ayuda a descubrir cualquier problema en la nueva versión y arreglarla. Sin embargo, puede que esto no siempre sea suficiente para garantizar una implementación exitosa. Es posible que nos encontremos con circunstancias inesperadas o con comportamientos del software subóptimos en los entornos de producción. En Amazon, queremos evitar ponernos en una situación en la que revertir la implementación podría causar errores a nuestros clientes. Para evitar esta situación, nos preparamos completamente para una reversión antes de cada implementación. Cuando una versión de software se puede revertir sin errores o interrupciones a la funcionalidad disponible de la versión anterior, se dice que es compatible con versiones anteriores. En cada una de las revisiones, planeamos y verificamos que nuestro software sea compatible con versiones anteriores.

Antes de entrar en detalles sobre la manera en que Amazon aborda las actualizaciones de software, hablemos sobre las diferencias entre las implementaciones de software independientes y distribuidas.

Implementaciones de software independientes en comparación con las distribuidas

Para el software independiente que funciona como un proceso en un dispositivo, las implementaciones son atómicas. Nunca se ejecutan dos versiones del mismo software al mismo tiempo. Si el software independiente mantiene el estado, la nueva versión debe leer (deserializar) datos escritos (serializados) por la versión antigua, y viceversa. Si esta condición se cumple, la implementación es segura para la consolidación y reversión.

En un sistema distribuido, las implementaciones son más complejas. Estas implementaciones se hacen a través de actualizaciones continuas para que la disponibilidad no se vea afectada. La nueva versión se implementa en un subconjunto de host al mismo tiempo, para que otros puedan continuar ejecutando solicitudes. Normalmente, estos host se comunican entre ellos a través de una llamada a procedimiento remoto (RPC) o un estado permanente compartido (por ejemplo, metadatos o puntos de control). Estas comunicaciones o estados compartidos pueden presentar desafíos adicionales. Tanto el escritor como el lector podrían ejecutar distintas versiones del software. Como resultado, podrían interpretar los datos de forma distinta. El lector podría incluso fallar en la lectura de los datos, lo que podría causar una interrupción.

Problemas con los cambios de protocolo

Hemos descubierto que la razón más común para no poder realizar reversiones es un cambio de protocolo. Por ejemplo, considere un cambio de código que empieza comprimiendo datos mientras permanecen en el disco. Luego de que la nueva versión escriba algunos datos comprimidos, la reversión ya no es una opción. La versión anterior no sabe que tiene que descomprimir datos después de leer el disco. Si los datos se almacenan en un blob o un almacén de documentos, entonces otros servidores no podrán leerlos, incluso cuando la implementación está en curso. Si los datos se transfieren entre dos procesos o servidores, el receptor no podrá leerlos.

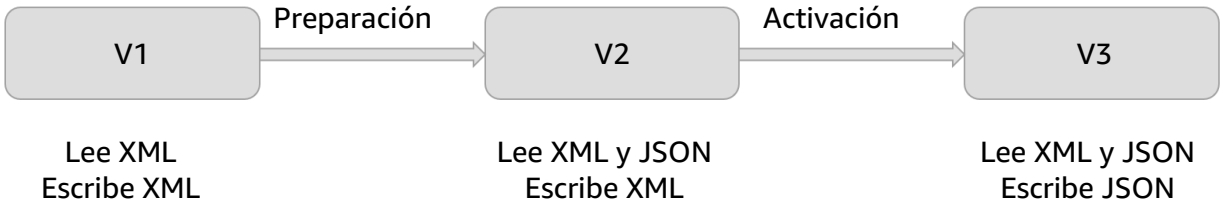
A veces, los cambios de protocolo pueden ser muy sutiles. Por ejemplo, considere dos servidores que se comunican de forma asíncrona durante la conexión. Para que ambos sean conscientes de su existencia, deciden enviar un pulso el uno al otro cada cinco segundos. Si un servidor no siente la pulsación en el tiempo estipulado, este asumirá que el otro servidor no funciona y cerrará la conexión.

Ahora, considere una implementación que aumenta el periodo del latido a 10 segundos. La confirmación de código parece menor, tan solo un cambio de número. Sin embargo, no es seguro para hacer la consolidación y la reversión. Durante la implementación, el servidor que ejecuta la nueva versión envía un latido cada 10 segundos. Por consiguiente, el servidor que ejecuta la versión anterior no siente el latido durante más de cinco segundos y finaliza la conexión con el servidor que ejecuta la nueva versión. En una flota grande, esta situación puede pasar con distintas conexiones, lo que podría provocar una caída en la disponibilidad.

Estos cambios tan sutiles son difíciles de analizar leyendo el código o el diseño de documentos. Por este motivo, verificamos expresamente que cada implementación sea segura para la consolidación y la reversión.

Técnica de implementación de dos fases

Una de las formas de asegurarnos de que podemos hacer la reversión de manera segura es usando una técnica que se conoce como implementación de dos fases. Imagínese el siguiente escenario hipotético con un servicio que gestiona datos (escritura y lectura) en Amazon Simple Storage Service (Amazon S3). El servicio se ejecuta en una flota de servidores en múltiples zonas de disponibilidad para escalado y disponibilidad.
Actualmente, el servicio usa el formato XML para mantener los datos. Tal y como se muestra en el siguiente diagrama en la versión V1, todos los servidores escriben y leen XML. Por razones comerciales, queremos mantener los datos en el formato JSON. Si hacemos este cambio en una implementación, los servidores que incorporaron el cambio escribirán en JSON. Pero los otros servidores no sabrán como leer JSON aún. Esta situación provoca errores. Por lo tanto, dividimos este cambio en dos partes y hacemos una implementación en dos fases.



Como se muestra en el anterior diagrama, llamamos a la primera fase Preparación. En esta fase, preparamos todos los servidores para que lean JSON (además de XML), pero que continúen escribiendo XML, implementando la versión V2. Este cambio no altera nada desde el punto de vista operativo. Todos los servidores aún podrán leer XML, y todos los datos estarán escritos en dicho formato. Si decidimos revertir este cambio, los servidores volverán a una condición en la que no podrán leer JSON. Esto no es un problema, ya que ninguno de los datos se ha escrito en JSON todavía.

Como se muestra en el anterior diagrama, llamamos a la segunda fase Activación. En esta fase, activamos los servidores para que usen el formato de escritura JSON, implementando la versión V3. A medida que cada servidor incorpore este cambio, empezará a escribir en JSON. Los servidores que todavía tienen que incorporar este cambio aún puede leer JSON, porque se prepararon en la primera fase. Si decidimos revertir este cambio, todos los datos escritos por los servidores que estuvieron temporalmente en la fase Activación están en JSON. Los datos escritos por los servidores que no estaban en la fase Activación están en XML. Está situación está bien, ya que, como se muestra en V2, los servidores todavía pueden leer tanto XML como JSON después de la reversión.
Aunque el diagrama anterior muestra el cambio de formato de serialización de XML a JSON, la técnica general se aplica a todas las situaciones descritas en la previa sección Cambios de protocolo. Por ejemplo, recuerde el escenario anterior en el que el tiempo de los latidos entre los servidores se tuvo que aumentar de 5 a 10 segundos. En la fase de Preparación, podemos hacer que todos servidores dilaten el período esperado de latido a 10 segundos, aunque continúen enviando el latido una vez cada cinco segundos. En la fase de Activación, modificamos la frecuencia a una vez cada 10 segundos.

Precauciones con las implementaciones de dos fases

Ahora describiré las precauciones que debemos tomar mientras seguimos la técnica de implementación en dos fases. A pesar de que hago referencia al escenario descrito en la sección anterior, estas precauciones se aplican a la mayoría de las implementaciones en dos fases.

Muchas herramientas de implementación permiten a los usuarios considerar una implementación exitosa si un número mínimo de host incorporan el cambio y lo reportan como correcto. Por ejemplo, AWS CodeDeploy tiene una configuración de implementación llamada minimumHealthyHosts.

Una suposición crítica en el ejemplo de la implementación en dos fases es que, al final de la primera fase, todos los servidores se han actualizados para que lean XML y JSON. Si la actualización de uno o más servidores falla durante la primera fase, entonces habrá un error al leer los datos durante y después de la segunda fase. Por lo tanto, verificaremos puntualmente que todos los servidores hayan incorporado el cambio en la fase de Preparación.

Cuando estaba trabajando en Amazon DynamoDB, decidimos cambiar el protocolo de comunicación entre una cantidad masiva de servidores que abarcaban múltiples microservicios. Coordiné las implementaciones entre todos los microservicios para que todos los servidores llegaran a la fase Preparación y luego procedieran a la fase Activación. Como precaución, verifiqué de forma inequívoca que la implementación se realizara con éxito en cada servidor al final de cada fase.

Si bien es seguro revertir cada una de las dos fases, no podemos revertir ambos cambios. En el ejemplo anterior, al final de la fase Activación, los servidores escriben datos en JSON. La versión de software utilizada antes de los cambios de Preparación y Activación no sabe leer JSON. Por lo tanto, como precaución, dejamos que pase un tiempo prudente y significativo entre las fases de Preparación

y Activación. Llamamos a este tiempo el período de horneado, y su duración suele ser de unos días. Esperamos para asegurarnos de no tener que restaurar a una versión anterior.

Luego de la fase de Activación, no podemos quitar de manera segura la capacidad de lectura de XML del software. No es seguro quitarla porque todos los datos escritos antes de la fase Preparación están en XML. Solo podemos quitar esta capacidad de lectura de XML luego de asegurarnos de que cada objeto se ha reescrito en JSON. Llamamos a este proceso reposición. Es posible que se requieran herramientas que puedan ejecutarse en simultáneo mientras el servicio escribe y lee datos.

Mejores prácticas para la serialización

La mayoría del software incluye la serialización de datos, ya sea para mantenerlos o transferirlos a una red. A medida que evoluciona, es común que la lógica de la serialización cambie. Los cambios pueden variar de agregar un nuevo campo a cambiar el formato por completo. A lo largo de los años, hemos llegado a las mejores prácticas que seguimos para la serialización:

- Generalmente evitamos desarrollar formatos de serialización personalizados.

  La lógica inicial de la serialización personalizada puede parecer insignificante y hasta ofrecer mejor rendimiento. Sin embargo, las iteraciones subsiguientes del formato presentan retos que ya se han resuelto por parte de marcos de trabajo bien establecidos, como JSON, Protocol Buffers, Cap'n Proto y FlatBuffers. Cuando se utilizan correctamente, estos marcos de trabajo proporcionan funciones de seguridad como escape, compatibilidad retrospectiva, y el seguimiento de estado de atributos (es decir, si un campo se configuró como el valor predeterminado de manera explícita o implícita).

- Con cada cambio, asignamos explícitamente una versión distinta a los serializadores.

  Hacemos esto independientemente del código fuente o la versión de compilación. También almacenamos la versión del serializador con los datos serializados o en los metadatos. Las versiones anteriores de serializador siguen funcionando en el nuevo software. Hemos descubierto que suele ser útil para obtener una métrica de la versión de datos escritos o leídos. Brinda a los operadores visibilidad e información sobre solución de problemas en caso de errores. Todo esto aplica para las versiones de RPC y API también.
- Evitamos serializar las estructuras de datos que no podemos controlar.

  Por ejemplo, podríamos serializar los objetos de recopilación de Java utilizando la reflexión. Pero, cuando intentamos actualizar el JDK, la implementación subyacente de dichas clases puede cambiar, lo que hace que falle la deserialización. Este riesgo aplica asimismo a las clases de bibliotecas que se comparten entre los equipos.
- Normalmente, diseñamos serializadores que permitan la presencia de atributos desconocidos.

  Siempre que sea posible, los serializadores conservan atributos desconocidos mientras reescriben los datos. Con este alojamiento, incluso si un servidor que ejecuta la versión nueva de software incluye nuevos atributos en los datos durante la serialización, los servidores que ejecutan la versión anterior no borrarán los atributos mientras actualizan los mismos datos. De esta manera, no es necesaria una implementación de dos fases.

Al igual que con muchas de nuestras mejoras prácticas, las compartimos con la precaución de que nuestras guías no son pertinentes para todas las aplicaciones y escenarios.

Verificación de seguridad de reversión de un cambio

Generalmente, verificamos puntualmente que sea seguro realizar un cambio de software o revertir la implementación, a través de un proceso que llamamos prueba de actualización-devolución. Para este proceso, configuramos un entorno de prueba que represente los entornos de producción. Con el paso de los años, hemos identificado algunos patrones que evitamos al configurar entornos de prueba.

Han ocurrido situaciones en las que implementar un cambio en la producción generó errores a pesar de que el cambio había aprobado todas las pruebas en el entorno de prueba. En una ocasión, los servicios en el entorno de prueba solo tenían un servidor cada uno. Por esta razón, todas las implementaciones fueron atómicas, lo cual impidió la posibilidad de ejecutar distintas versiones del software en forma simultánea. Ahora, incluso si los entornos de prueba no tienen tanto tráfico como los entornos de producción, utilizamos múltiples servidores de distintas zonas de disponibilidad para cada servicio, al igual que se haría en producción. En Amazon, reducir costos es una de las metas favoritas, pero no cuando se refiere a garantizar la calidad.

En otra ocasión, el entorno de prueba tenía múltiples servidores. Sin embargo, la implementación se realizó en todos los servidores al mismo tiempo para agilizar las pruebas. Este enfoque también evitó que las versiones antiguas y más recientes de software se ejecuten al mismo tiempo. No se detectó el problema con la consolidación. Actualmente, utilizamos la misma configuración de implementación en todos los entornos de prueba y producción.

Para los cambios que implican coordinar entre microservicios, mantenemos el mismo orden de implementación en todos los microservicios para los entornos de prueba y producción. Sin embargo, el orden de consolidación y reversión puede ser diferente. Por ejemplo, solemos seguir un orden específico en el contexto de la serialización. En otras palabras, los lectores están antes que los escritores durante la consolidación, mientras que el orden es a la inversa para la reversión. Normalmente se sigue el orden adecuado en los entornos de prueba y producción.

Cuando la configuración de un entorno de prueba es similar a los entornos de producción, simulamos el tráfico de producción con la mayor precisión posible. Por ejemplo, creamos y hacemos la lectura de varios registros (o mensajes) en rápida sucesión. Todas las API se ejecutan en forma continua. Luego, tomamos el entorno a través de tres etapas, cada una de las cuales tiene una duración razonable para identificar errores potenciales. La duración es lo suficientemente larga como para que todas las API, flujos de trabajo de backend y tareas por lotes se ejecuten al menos una vez.
Primero, implementamos el cambio aproximadamente a la mitad de la flota, para asegurar la coexistencia de las versiones de software. Luego, completamos la implementación. En tercer lugar, iniciamos la implementación de la reversión y seguimos los mismos pasos hasta que todos los servidores se ejecuten con la versión anterior de software. Si no hay errores o comportamiento no contemplado durante estas etapas, se considera que la prueba fue exitosa.

Conclusión

Asegurarnos de que podemos revertir una implementación sin interrupciones para nuestros clientes es fundamental para que el servicio sea confiable. Realizar pruebas específicas de seguridad de reversión elimina la necesidad de depender de análisis manuales, los cuales son propensos a los errores. Cuando descubrimos que no es seguro consolidar y revertir un cambio, podemos dividirlo en dos cambios, cada uno de los cuales se puede consolidar y revertir de manera segura.