Evitar los planes alternativos en los sistemas distribuidos

Jacob Gabrielson

Los errores críticos impiden que el servicio genere resultados útiles. Por ejemplo, en un sitio web de comercio electrónico, si se produce un error en una consulta a la base de datos para obtener información sobre el producto, el sitio no puede mostrar la página del producto correctamente. Los servicios de Amazon deben gestionar la mayor cantidad de errores críticos para ser confiables. Existen cuatro grandes categorías de estrategias para gestionar los errores críticos:

- **Reintento:** volver a realizar la actividad en la que se produjeron errores, ya sea de inmediato o una vez que haya transcurrido un poco de tiempo.
- **Reintento proactivo:** realizar la actividad varias veces en paralelo y utilizar la instancia que se complete primero.
- **Conmutación por error:** realizar la actividad otra vez en una copia diferente del punto de enlace o, preferentemente, realizar varias copias paralelas de la actividad para aumentar la probabilidad de que al menos una de ellas tenga éxito.
- **Plan alternativo:** utilizar un mecanismo diferente para lograr el mismo resultado.

Este artículo describe las estrategias alternativas y explica por qué casi nunca las utilizamos en Amazon. Esto puede sorprenderlo. Después de todo, los ingenieros suelen utilizar el mundo real como punto de partida para sus diseños. Y en el mundo real, las estrategias alternativas se deben planificar con anticipación y utilizar cuando sean necesarias. Supongamos que se apagan las pantallas de un aeropuerto. Se debe implementar un plan de contingencia (como que los personas escriban la información de los vuelos en pizarras) para gestionar la situación, ya que los pasajeros aún necesitan encontrar sus puertas de embarque. Pero considere cuán horrible es el plan de contingencia: la dificultad de leer las pizarras, la dificultad de mantenerlas actualizadas y el riesgo de que las personas agreguen información incorrecta. La estrategia alternativa de la pizarra es necesaria, pero está plagada de problemas.

En el mundo de los sistemas distribuidos, las estrategias alternativas se encuentran entre los desafíos más complicados de gestionar, en especial para los servicios urgentes. Lo que empeora esta dificultad es que puede transcurrir un largo tiempo (incluso años) hasta que se observen las repercusiones de las malas estrategias alternativas, y la diferencia entre una buena y una mala estrategia es sutil. Este artículo se enfocará en cómo las estrategias alternativas pueden causar más problemas de los que solucionan. Incluiremos ejemplos donde las estrategias alternativas han causado problemas en Amazon. Finalmente, analizaremos otras opciones diferentes del plan alternativo que utilizamos en Amazon.

Analizar las estrategias alternativas para los servicios no es un trabajo intuitivo, y sus efectos secundarios son difíciles de prever en los sistemas distribuidos, así que comencemos revisando las estrategias alternativas para la aplicación en una sola máquina.

Plan alternativo de una sola máquina

Tenga en cuenta el siguiente fragmento de código C que ilustra un patrón común para gestionar los errores de asignación de la memoria en muchas aplicaciones. Este código asigna memoria mediante la función `malloc()` y luego copia un búfer de imagen en dicha memoria mientras realiza algún tipo de transformación:

```
pixel_ranges = malloc(image_size); // allocates memory
if (pixel_ranges == NULL) {

  // On error, malloc returns NULL

  exit(1);}

for (i = 0; i < image_size; i++) {
  pixel_ranges[i] = xform(original_image[i]);}
```

El código no se recupera bien en el caso que se produzca un error en la función `malloc`. En la práctica, raramente se producen errores en las llamadas a la función `malloc`, por lo que a menudo los desarrolladores ignoran sus errores en el código. ¿Por qué es tan común esta estrategia? El razonamiento que tiene lugar es que, en una sola máquina, si se produce un error en la función `malloc`, este se deba a que probablemente la máquina ya no tiene espacio en la memoria. Por lo tanto, hay problemas más grandes que los errores en una llamada a la función `malloc`: la posible caída de la máquina en poco tiempo. Y la mayoría de las veces, en una sola máquina, este es un razonamiento acertado. Muchas aplicaciones no son lo suficientemente críticas como para que valga la pena resolver un problema tan espinoso. Pero ¿y si sí quisiera encargarme del error? Es difícil intentar realizar algo útil en esa situación. Supongamos que implementamos un segundo método denominado `malloc2`, que asigna la memoria de una forma diferente y realizamos una llamada a `malloc2` si se produce un error en la implementación de la función `malloc` predeterminada:

```
pixel_ranges = malloc(image_size);
if (pixel_ranges == NULL) {
  pixel_ranges = malloc2(image_size);
}
```

A primera vista, parece que este código podría funcionar, pero presenta algunos problemas, aunque unos sean menos evidentes que otros. Para empezar, la lógica alternativa es **difícil de probar**. Podríamos interceptar la llamada a la función `malloc` e introducir un error, pero es posible que esto no genere una simulación precisa de lo que pasaría en el entorno de producción. En la producción, si se produce un error en la función `malloc`, es muy probable que la máquina no tenga más memoria o tenga poca. ¿Cómo puede hacer una simulación de esos problemas de memoria más amplios? Incluso si pudiera generar un entorno de baja memoria para ejecutar una prueba (por ejemplo, en un contenedor de Docker), ¿cómo programaría la condición de baja memoria para que coincida con la ejecución del código alternativo `malloc2`?

Otro problema es que el **plan alternativo en sí mismo podría fallar**. Los códigos alternativos anteriores no se encargan de los errores de `malloc2`. Por lo tanto, el programa no ofrece tantos beneficios como se podría esperar. Posiblemente, la estrategia alternativa hace que una falla completa sea menos probable, pero no imposible. En Amazon, descubrimos que gastar recursos de ingeniería para mejorar la confiabilidad del código principal (no alternativo) en general a menudo aumenta la probabilidad de tener éxito más que invertir en una estrategia alternativa poco utilizada.

Además, si nuestra máxima prioridad es la disponibilidad, **es posible que no valga la pena arriesgarse por una estrategia alternativa**. ¿Por qué molestarse por la función `malloc` en absoluto si `malloc2` tiene más posibilidades de tener éxito? Lógicamente, `malloc2` debe compensar de alguna manera el mayor nivel disponibilidad. Tal vez asigna la memoria en un almacenamiento basado en SDD de mayor latencia, pero más grande. Pero eso genera una pregunta; ¿por qué está bien que `malloc2` compense los beneficios de esta manera? Consideremos una posible secuencia de eventos que podría ocurrir con esta estrategia alternativa. En primer lugar, el cliente está utilizando la aplicación. De repente (porque falló la función `malloc`), `malloc2` comienza a funcionar, y la aplicación se vuelve más lenta. Esto es malo. ¿Realmente está bien que sea más lenta? Y los problemas no terminan aquí. Tenga en cuenta que es muy probable que la máquina no tenga memoria (o tenga muy poca). Ahora el cliente enfrenta dos problemas (aplicación más lenta y máquina más lenta) en lugar de uno solo. Los efectos secundarios de cambiar a `malloc2` incluso podrían empeorar el problema general. Por ejemplo, otros subsistemas también podrían competir por el mismo almacenamiento basado en SSD.

La lógica alternativa también puede generar una **carga impredecible** en el sistema. Incluso una simple lógica común, como escribir un mensaje de error en un registro con un seguimiento de pila, parece inofensiva, pero si algo llegara a cambiar de repente y provocara ese error a una alta velocidad, es posible que una aplicación limitada por la CPU de pronto se transforme en una aplicación limitada por las operaciones de E/S. Y si no se aprovisionó el disco para que se encargue de escribir a esa velocidad o para que almacene esa cantidad de datos, puede provocar la reducción de la velocidad de la aplicación o su caída.

La estrategia alternativa no solo podría empeorar el problema, sino que probablemente esto ocurra como un **error latente**. Es fácil desarrollar estrategias alternativas que rara vez se activen en la producción. Pueden pasar años antes de que la máquina de siquiera un cliente realmente se quede sin memoria en el momento justo de activar la línea de código específica con el plan alternativo de la función `malloc2` que se mostró anteriormente. Si hay un error en la lógica alternativa o algún tipo de efecto secundario que empeore el problema general, es probable que los ingenieros que escribieron el código hayan olvidado cómo funcionaba en primer lugar, por lo que será más difícil arreglar el código. Para una aplicación de una sola máquina, esta puede ser una compensación comercial aceptable, pero en los sistemas distribuidos las consecuencias son muchos más significativas, como se analiza más adelante.

Todos estos problemas son espinosos, pero en nuestra experiencia, a menudo se pueden ignorar sin problemas en las aplicaciones de una sola máquina. La solución más común es la que mencionamos anteriormente: solo dejar que los errores en la asignación de memoria provoquen la caída de la aplicación. El código que asigna la memoria tiene el mismo destino que el resto de la máquina, y es muy probable que el resto de la máquina esté a punto de fallar en este caso. Incluso si no compartiera el mismo destino, la aplicación estaría ahora en un estado que no se habría anticipado, y es una buena estrategia que el error se produzca rápidamente. La compensación comercial es razonable.

Para las aplicaciones de una sola máquina críticas que deben funcionar en caso de que se produzca un error en la asignación de memoria, una solución es asignar previamente toda la memoria dinámica en el arranque y no volver a depender de la función `malloc`, incluso bajo condiciones de error. Amazon ha implementado esta estrategia varias veces, por ejemplo, a la hora de monitorear daemons que se ejecutan en los servidores de producción y los daemons de Amazon Elastic Compute Cloud (Amazon EC2) que monitorean las ráfagas de la CPU del cliente.

Plan alternativo distribuido

En Amazon, no dejamos que los sistemas distribuidos, en especial los sistemas destinados a responder en tiempo real, lleven a cabo las mismas compensaciones que las aplicaciones de una sola máquina. Una de las razones es la falta de un destino compartido con el cliente. Podemos suponer que las aplicaciones se están ejecutando en la máquina frente al cliente. Si la aplicación se queda sin memoria, es probable que el cliente no espere que se siga ejecutando. Los servicios no se ejecutan directamente en la máquina que está utilizando el cliente, por lo que la expectativa es diferente. Más allá de eso, los clientes suelen utilizar los servicios precisamente porque brindan mayor disponibilidad que la ejecución de una aplicación en un solo servidor. Es por eso que debemos lograr que estén disponibles. En teoría, esto nos llevaría a implementar un plan alternativo como una forma de mejorar la fiabilidad del servicio. Desafortunadamente, los planes alternativos distribuidos presentan los mismos problemas y aún más cuando se trata de los errores críticos del sistema.

Las estrategias de los planes alternativos distribuidos son más difíciles de probar. El plan alternativo del servicio es más complicado que el caso de la aplicación de una sola máquina, ya que varias máquinas y servicios posteriores participan en los errores. Es difícil replicar los propios modos de error, como los casos de sobrecarga, en una prueba, incluso si la organización de la prueba en varias máquinas no tiene problemas de disponibilidad. La combinatoria también aumenta la enorme cantidad de casos que hay que probar, por lo que necesita más pruebas, y es mucho más difícil configurarlas.

Las propias estrategias de los planes alternativos distribuidos pueden fallar. Si bien puede parecer que las estrategias alternativas garantizan el éxito, según nuestra experiencia, por lo general estas solo aumentan la probabilidad de tener éxito.

Las estrategias de los planes alternativos distribuidos muchas veces empeoran la interrupción. En nuestra experiencia, las estrategias alternativas aumentan el alcance del impacto de los errores, así como también los tiempos de recuperación.

Muchas veces no vale la pena arriesgarse por las estrategias de los planes alternativos distribuidos. Al igual que con `malloc2`, la estrategia alternativa suele realizar algún tipo de compensación. Si no fuera este el caso, la utilizaríamos todo el tiempo. ¿Por qué utilizaríamos un plan alternativo que es peor cuando algo ya está fallando?

Las estrategias de los planes alternativos distribuidos suelen tener errores latentes que aparecen solo cuando se produce una serie de coincidencias poco probables, posiblemente meses o años después de su introducción.

Una interrupción importante en el mundo real que se provocó por un mecanismo alternativo en el sitio web de venta minorista de Amazon ejemplifica todos estos problemas. La interrupción se produjo alrededor del año 2001, y su causa fue una característica nueva que ofrecía velocidades de envío actualizadas para todos los productos que se mostraban en el sitio web. La característica nueva se parecía a algo como esto:

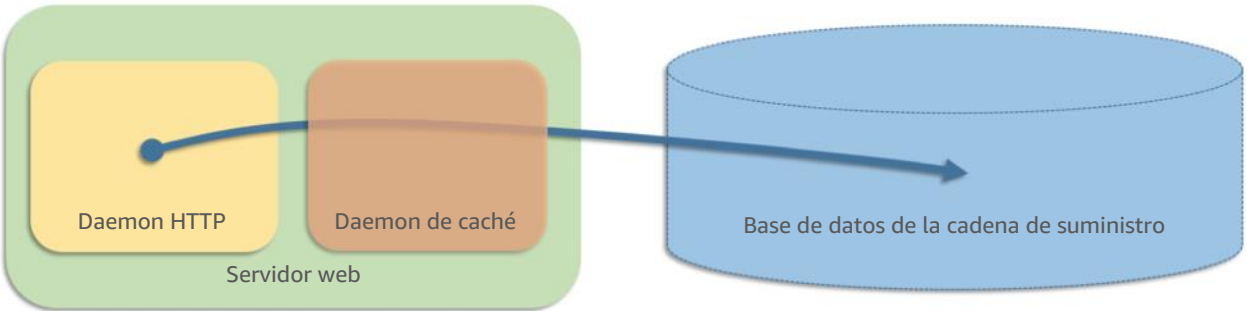
Para acceder a la entrega GRATUITA **hoy mismo**, realice el pedido en los próximos 14 min y elija esta fecha en la salida. Detalles

Se encuentra en existencias.

Amazon Digital Services LLC se encarga del envío y de la venta. Está disponible la opción de envoltorio para regalo.

Color: **Negro**

En aquel momento, la arquitectura del sitio web solo tenía dos capas y, como los datos se almacenaban en una base de datos de la cadena de suministro, los servidores web necesitaban consultar la base de datos directamente. Pero la base de datos no podía seguir el ritmo del volumen de solicitudes del sitio web. El sitio web tenía un gran volumen de tráfico, y algunas páginas mostraban 25 productos o más, con las velocidades de envío de cada producto en exposición directa. Por eso, agregamos una capa de almacenamiento en memoria caché que se ejecutaba como un proceso independiente en cada servidor web (similar a Memcached):



Esto funcionó bien, pero el equipo también intentó gestionar los casos en los que la memoria caché (un proceso independiente) fallaba por alguna razón. En este caso, los servidores web volvieron a consultar la base de datos directamente. Escribimos algo como los siguiente en pesudocódigo:

```
if (cache_healthy) {
  shipping_speed = get_speed_via_cache(sku);
} else {
  shipping_speed = get_speed_from_database(sku);
}
```

Recurrir a la alternativa de realizar las consultas a la base de datos directamente fue una solución intuitiva que funcionó durante varios meses. Pero, finalmente, todas las memorias caché fallaron en la misma época, lo que significó que todos los servidores web tenían que recurrir a la base de datos de manera directa. Esto generó suficiente carga como para bloquear completamente la base de datos. El sitio web en su totalidad dejó de funcionar porque todos los procesos de servidor web estaban bloqueados en la base de datos. Esta base de datos de la cadena de suministro también era fundamental para los centros de distribución, por lo que la interrupción se expandió aún más, y los centros de distribución de todo el mundo se paralizaron hasta que se solucionó el problema.

Todos los problemas que observamos en el escenario de una sola máquina estaban presentes en el caso distribuido con consecuencias más graves. El caso del plan alternativo distribuido era **difícil de probar**. Incluso si hubiéramos simulado los errores en la memoria caché, no habríamos encontrado

el problema, el cual requería que hubiera errores en varias máquinas para la activación. Y, en este caso, la **propia estrategia alternativa amplificó el problema** y fue peor que si no hubiera habido ninguna estrategia en absoluto. El plan alternativo convirtió una interrupción parcial del sitio web (no poder mostrar las velocidades de envío) en una interrupción del sitio completo (ninguna página se cargaba) y acabó con toda la red de distribución de Amazon en el backend.

En este caso, el razonamiento detrás de nuestra estrategia alternativa era **ilógico**. Si recurrir a la base de datos directamente era más confiable que hacerlo a través de la memoria caché, ¿por qué tomarse de la molestia de colocar la memoria caché en primer lugar? Teníamos miedo de que si no usábamos la memoria caché, como resultado, se iba a sobrecargar la base de datos. Pero ¿por qué molestarse en tener un código alternativo si tenía tal capacidad de daño? Podríamos haber detectado nuestra falla en el comienzo, pero el error era **latente**, y la situación que provocó la interrupción apareció meses después del lanzamiento.

Manera de evitar los planes alternativos de Amazon

Debido a estas dificultades que enfrentamos con los planes alternativos distribuidos, ahora casi siempre preferimos otras opciones distintas de los planes alternativos. Estas se resumen a continuación.

Mejorar la confiabilidad de los casos que no son planes alternativos

Como se mencionó anteriormente, las estrategias alternativas solo reducen la probabilidad de que se produzca una falla completa. Un servicio puede tener mucha más disponibilidad si el código principal (no alternativo) es más sólido. Por ejemplo, en lugar de implementar una lógica alternativa entre dos almacenes de datos diferentes, un equipo podría invertir en el uso de una base de datos con mayor disponibilidad inherente, como Amazon DynamoDB. Esta estrategia se suele utilizar con éxito en Amazon. Por ejemplo, esta presentación describe el uso de DynamoDB para respaldar a amazon.com en el Prime Day 2017.

Dejar que el intermediario se encargue de los errores

Una solución a los errores críticos del sistema es no recurrir a planes alternativos, sino dejar que el sistema de llamada se encargue de ellos (mediante reintentos, por ejemplo). Esta es una estrategia que se prefiere para los servicios de AWS, donde las CLI y los SDK ya tienen incorporada la lógica de reintentos. Siempre que sea posible, preferimos esta estrategia, especialmente en situaciones donde se ha puesto el esfuerzo necesario para tener el mismo destino y reducir la probabilidad de que el caso principal tenga errores (además, sería poco probable que la lógica alternativa mejorara la disponibilidad en absoluto).

Enviar los datos de manera proactiva

Otra opción que utilizamos para no tener que recurrir a planes alternativos es reducir la cantidad de partes en movimiento al momento de responder las solicitudes. Si, por ejemplo, un servicio necesita datos para completar una solicitud, y esos datos ya están presentes en las instalaciones

(no es necesario recuperarlos), no hay necesidad de una estrategia de conmutación por error. Se puede encontrar un excelente ejemplo de esto en la implementación de roles de AWS Identity and Access Management (IAM) para Amazon EC2. El servicio de IAM tiene que proporcionar credenciales firmadas y rotadas al código que se ejecuta en las instancias EC2. Para no tener que recurrir a planes alternativos, las credenciales se transmiten de manera proactiva a cada instancia y siguen siendo válidas durante muchas horas. Esto significa que las solicitudes relacionadas con el rol de IAM siguen funcionando en el caso improbable de que haya una interrupción en el mecanismo de envío.

Convertir el plan alternativo en una conmutación por error

Una de las peores características del plan alternativo es que no se utiliza a menudo, y es probable que tenga errores o que aumente el alcance del impacto cuando se active durante una interrupción. Las circunstancias que activan el plan alternativo podrían no ocurrir de forma natural durante meses o incluso años. Para solucionar el problema de los errores latentes en la estrategia alternativa, es importante utilizarla con regularidad en la producción. Un servicio debe ejecutar tanto la lógica alternativa como la no alternativa de manera constante. No debe solo ejecutar el caso de plan alternativo, sino que también debe tratarlo como un origen de datos igualmente válido. Por ejemplo, un servicio puede elegir de forma aleatoria entre las respuestas alternativas y no alternativas (cuando obtiene las dos respuestas) para asegurarse de que ambas funcionen. Pero a esta altura, la estrategia ya no se puede considerar una alternativa y entra sin duda en la categoría de conmutación por error.

Asegurarse de que los reintentos y los tiempos de espera no se conviertan en planes alternativos

Los reintentos y los tiempos de espera se analizan en el artículo Timeouts, Retries, and Backoff with Jitter (Tiempos de espera, reintentos y retardo con fluctuación). El artículo explica que los reintentos son un mecanismo potente para ofrecer alta disponibilidad frente a **errores transitorios y aleatorios**. En otras palabras, los reintentos y los tiempos de espera ofrecen un seguro frente a errores ocasionales generados por problemas menores, como la pérdida de paquetes falsos, los errores no relacionados de una sola máquina y similares. Sin embargo, es fácil que haya equivocaciones con los reintentos y los tiempos de espera. Muchas veces, los servicios pueden pasar meses o más tiempo sin necesitar muchos reintentos, y es posible que estos se produzcan finalmente en casos que su equipo nunca ha probado. Por esta razón, mantenemos las métricas que monitorean las tasas de reintentos generales y las alarmas que avisan a nuestros equipos si los reintentos ocurren con frecuencia.

Otra forma de evitar que los reintentos se conviertan en planes alternativos es ejecutarlos todo el tiempo con **reintentos proactivos** (también conocidos como solicitudes de cobertura o paralelas). Esta técnica está intrínsecamente integrada en los sistemas que realizan lecturas o escrituras de quórum, donde es posible que un sistema requiera una respuesta de dos de tres servidores para responder. El reintento proactivo sigue el patrón de diseño del **trabajo constante**. Debido a que siempre se realizan solicitudes redundantes, no se agrega una carga adicional de reintentos al sistema a medida que aumenta la necesidad de este tipo de solicitudes.

En conclusión

En Amazon, evitamos utilizar los planes alternativos en nuestros sistemas porque son difíciles de probar y evaluar su efectividad también es complicado. Las estrategias alternativas introducen un modo operativo al que el sistema accede solo en los momentos más caóticos donde las cosas comienzan averiarse y cambiar a este modo solo aumenta el caos. A menudo, transcurre un tiempo largo entre el momento en el cual se implementa una estrategia alternativa y en el que se encuentra en un entorno de producción.

En cambio, favorecemos las rutas de códigos que se utilizan en la producción continuamente y no rara vez. Nos enfocamos en mejorar la disponibilidad de nuestros sistemas principales, mediante el uso de patrones, como enviar datos a sistemas que los necesitan, en lugar de extraerlos y arriesgarse a que una llamada remota produzca un error en un momento crítico. Finalmente, estamos atentos al comportamiento sutil en nuestro código que podría cambiarlo a un modo de operación alternativo, como realizar demasiados reintentos.

Si el plan alternativo es esencial en un sistema, lo utilizamos con la mayor frecuencia posible en la producción, de modo que se comporte de manera tan confiable y predecible como el modo de operación principal.