Cómo evitar demoras de colas insuperables

David Yanacek

Algoritmos que imitan la vida real

Desde mi primer curso de ciencias informáticas en la universidad, me ha interesado el papel que desempeñan los algoritmos en el mundo real. Cuando pensamos en algunas de las cosas que ocurren en el mundo real, podemos crear algoritmos que imitan esas cosas. Yo hago esto particularmente cuando estoy atorado haciendo cola, como en el supermercado, en el tránsito o en el aeropuerto. Descubrí que estar aburrido mientras uno hace fila es una gran oportunidad para reflexionar sobre la teoría de las colas.

Hace alrededor de una década, pasé un día trabajando en el centro de logística de Amazon. Me guiaba mediante un algoritmo, tomaba artículos de los estantes, movía artículos de una caja a otra, trasladaba contenedores. Al trabajar en paralelo con tanta gente, descubrí que me gustaba ser parte de lo que es, en esencia, un sorprendente ordenamiento por mezcla físicamente orquestado.

En la teoría de las colas, el comportamiento de las filas cuando son cortas es relativamente poco interesante. Después de todo, cuando la cola es corta, todos están felices. Es solo cuando se demora la cola, cuando la fila para un evento sale por la puerta y da vuelta la esquina, que la gente comienza a pensar en el rendimiento y la priorización.

En este artículo, analizo las estrategias que se usan en Amazon para lidiar con escenarios de demoras en las colas, y diseño los enfoques que tomamos para vaciar las colas rápidamente y dar prioridad a las cargas de trabajo. Lo que es más importante, describo cómo evitar que se acumulen demoras en las cosas en primera instancia. En la primera mitad, describo escenarios que llevan a las demoras, y en la segunda mitad, describo muchos de los enfoques utilizados en Amazon para evitar retrasos o para lidiar con ellos fácilmente.

La naturaleza engañosa de las colas

Las colas son herramientas poderosas para crear sistemas asíncronos confiables. Las colas permiten que un sistema acepte un mensaje de otro sistema, y el mensaje persiste hasta estar completamente procesado, incluso frente a prolongadas interrupciones, fallas de servidor o problemas con los sistemas dependientes. En lugar de cancelar los mensajes cuando hay una falla, la cola redirecciona los mensajes hasta que procesarlos satisfactoriamente. Al final, una cola aumenta la durabilidad y disponibilidad de un sistema, a expensas de un ocasional aumento de latencia debido a los reintentos.

En Amazon, creamos muchos sistemas asíncronos que aprovechan las colas. Algunos de estos sistemas procesan los flujos de trabajo que podrían tomar mucho tiempo y que involucran el traslado de cosas físicas por el mundo, como cumplir con los pedidos realizados en amazon.com. Otros sistemas coordinan los pasos que podrían tomar una cantidad de tiempo no menor. Por ejemplo, Amazon RDS solicita instancias EC2, espera a que se inicien y luego configura sus bases de datos para usted. Otros sistemas aprovechan el procesamiento por lotes. Por ejemplo, los sistemas que introducen métricas y registros a CloudWatch extraen un paquete de datos y luego lo agregan y lo «aplanan» en fragmentos.

Si bien es fácil ver los beneficios de una cola para procesar mensajes de manera asíncrona, los riesgos de utilizar una cola con más sutiles. Con los años, hemos notado que las colas destinadas a mejorar la disponibilidad pueden producir el efecto contrario. De hecho, puede aumentar considerablemente el tiempo de recuperación luego de una interrupción.

En un sistema basado en colas, cuando se detiene el procesamiento pero los mensajes siguen llegando, la deuda de mensajes puede acumularse y producir una gran demora, lo que aumenta el tiempo de procesamiento. Posiblemente el trabajo se complete demasiado tarde y los resultados ya no serán útiles, lo que afectaría la disponibilidad que se suponía la cola debía proteger.

En otras palabras, un sistema basado en colas tiene dos modos de operación o una conducta bimodal. Cuando no hay demoras en la cola, la latencia del sistema es baja y el sistema está en modo rápido. Pero si una falla o un patrón de carga imprevisto hacen que el índice de llegada exceda el índice de procesamiento, cambia rápidamente a un modo de funcionamiento más siniestro. En este modo, la latencia de un extremo al otro aumenta cada vez más, y puede pasar mucho tiempo hasta que se solucione la demora y se vuelva al modo rápido.

Sistemas basados en colas

Para ilustrar los sistemas basados en colas en este artículo, menciono cómo funcionan por dentro dos servicios de AWS: AWS Lambda, un servicio que ejecuta su código en respuesta a eventos sin que tenga que preocuparse por la infraestructura en la que se ejecuta; y AWS IoT Core, un servicio administrado que permite que los dispositivos conectados interactúen en forma fácil y segura con las aplicaciones en nube y otros dispositivos.

Con AWS Lambda, usted carga su código de función y luego invoca sus funciona de una de dos formas:

- De manera sincrónica: donde el resultado de su función regresa a usted en la respuesta HTTP
- De manera asincrónica: donde la respuesta HTTP regresa de inmediato, y su función se ejecuta y se debe reintentar detrás de escena

Lambda asegura que su función se ejecute, incluso frente a fallas del servidor, por lo que debe ser una cola duradera en la cual almacenar su solicitud. Con una cola duradera, se puede redireccionar su solicitud en caso de que su función falle la primera vez.

Con AWS IoT Core, sus dispositivos y aplicaciones se conectan y puede suscribirse a los temas de mensajes PubSub. Cuando un dispositivo o aplicación publica un mensaje, las aplicaciones con suscripciones coincidentes reciben su propia copia del mensaje. Gran parte de esta mensajería PubSub ocurre de manera asíncrona, dado que un dispositivo IoT limitado no desea gastar sus limitados recursos esperando para asegurarse de que todos los dispositivos, aplicaciones y sistemas suscritos reciban una copia. Esto es particularmente importante ya que un dispositivo suscrito podría no tener conexión cuando otro dispositivo publica un mensaje en que está interesado. Cuando el dispositivo sin conexión se vuelve a conectar, se espera que vuelva a aparecer para acelerar primero, y además para que luego se le envíen sus mensajes (para obtener información sobre el cifrado de su sistema para administrar la entrega de mensajes después de volver a conectarse, consulte las Sesiones Persistentes de MQTT en la *Guía para desarrolladores de AWS IoT*). Hay una variedad de sabores de persistencia y procesamiento asíncrono que va más allá de las escenas para que esto ocurra.

Los sistemas basados en colas como estos suelen implementarse con una cola duradera. SQS ofrece una semántica de entrega de mensajes al menos una vez, duradera y escalable, por lo que los equipos de Amazon, incluidos Lambda y IoT, la usan regularmente cuando crean sus sistemas asíncronos escalables. En los sistemas basados en colas, un componente *produce* datos poniendo mensajes en la cola, y otro componente *consuma* esos datos periódicamente pidiendo mensajes, procesando mensajes y por último eliminándolos una vez terminados.

Fallas en sistemas asincrónicos

En AWS Lambda, si la invocación de su función es más lenta de lo normal (por ejemplo, debido a una dependencia), o si falla transitoriamente, no se pierden datos y Lambda reintenta su función. Lambda pone sus llamadas evocadas en cola, y cuando la función comienza a andar nuevamente, Lambda trabaja con los trabajos pendientes de su función. Pero consideremos el tiempo que lleva trabajar con la demora y regresar al estado normal.

Imagine un sistema que experimenta una interrupción de una hora mientras procesaba mensajes. Independientemente de determinada capacidad de calificación y procesamiento, recuperarse de una interrupción requiere el doble de la capacidad del sistema durante otra hora después de la recuperación. En la práctica, el sistema podría tener más del doble de la capacidad disponible, especialmente con servicios elásticos como Lambda, y la recuperación podría ocurrir más rápidamente. Por otro lado, otros sistemas con los que interactúa su función podrían no estar preparados para lidiar con un gran aumento en el procesamiento a medida que trabajo con los trabajos pendientes. Cuando esto ocurre, ponerse al corriente puede llevar incluso más tiempo. Los servicios asíncronos acumulan demora durante las interrupciones, lo que lleva a largos tiempos de recuperación, a diferencia de los servicios sincrónicos, que disminuyen las solicitudes durante las interrupciones pero tienen tiempos de recuperación más rápidos.

Con los años, cuando pensamos en las colas, nos hemos visto tentados a pensar que la latencia no es importante en los sistemas asíncronos. Los sistemas asíncronos suelen estar construidos para lograr durabilidad, o para aislar al intermediario de la latencia. Sin embargo, en la práctica, hemos visto que ese tiempo de procesamiento sí importa, y a menudo incluso se espera que los sistemas asíncronos tengan tiempos inferiores o una mejor latencia. Cuando se introducen colas para mayor durabilidad, es fácil perder el intercambio que provoca dicha latencia de alto procesamiento frente a una demora. El riesgo oculto con los sistemas asíncronos es lidiar con grandes demoras.

Cómo medimos la disponibilidad y la latencia

Este análisis del intercambio de latencia por disponibilidad presenta una pregunta muy interesante: ¿cómo medimos y establecemos los objetivos de latencia y disponibilidad para un servicio asíncrono? Medir los índices de error desde la perspectiva del *productor* nos brinda parte del panorama de disponibilidad, pero no mucho. La disponibilidad del productor es proporcional a la disponibilidad de la cola del sistema que estamos usando. Por lo tanto, cuando creamos en SQS, la disponibilidad de nuestro productor coincide con la disponibilidad de SQS.

Por otro lado, si medimos la disponibilidad del lado del *consumidor*, puede hacer que la disponibilidad del sistema se vea peor de lo que realmente es, ya que se pueden reintentar las fallas y luego tener éxito en el siguiente intento.

También obtenemos mediciones de disponibilidad de colas de mensajes fallidos (DLQ). Si un mensaje se queda sin reintentos, se desecha o se coloca en una DLQ. Una DLQ es simplemente una cola separada que se utiliza para almacenar mensajes que no pueden ser procesados para su posterior investigación e intervención. El índice de mensajes eliminados o DLQ es una buena medición de disponibilidad, pero posiblemente detecte los problemas demasiado tarde. Si bien es buena idea alarmar sobre los volúmenes de las DLQ, dicha información nos llegaría demasiado tarde como para depende exclusivamente de ella para detectar problemas.

¿Qué pasa con la latencia? Nuevamente, la latencia observada por el *productor* refleja la latencia de nuestro servicio de colas. Por lo tanto, nos centramos más en medir la antigüedad de los mensajes que están en la cola. Esto rápidamente capta casos en los que los sistemas están retrasados, o que con frecuencia envían errores y provocan reintentos. Algunos servicios como SQS proporcionan una marca temporal de cuándo cada uno de los mensajes llega a la cola. Con la información de la marca temporal, cada vez que sacamos un mensaje de la cola, podemos registrar y producir métricas sobre qué tan retrasados están nuestros sistemas.

El problema de la latencia puede ser un poco más molesto. Después de todo, se espera que haya demoras, y de hecho está bien con ciertos mensajes. Por ejemplo, en AWS IoT, hay veces en que se espera que un dispositivo se desconecte o se ralentice para leer sus mensajes. Esto se debe a que muchos dispositivos IoT son de baja potencia y tienen una conexión a Internet defectuosa. Como operadores de AWS IoT Core, debemos ser capaces de distinguir la diferencia entre una pequeña demora prevista provocada por dispositivos que están desconectados o por elegir leer los mensajes lentamente, y una demora imprevista de todo el sistema.

En AWS IoT, equipamos el servicio con otra métrica: *AgeOfFirstAttempt*. Esta medición registra el momento *actual* menos el *tiempo que estuvo el mensaje en la cola*, pero solo si fue la primera vez que AWS IoT intentó entregar un mensaje a un dispositivo. De este modo cuando se hace una copia de respaldo de los dispositivos, contamos con una métrica limpia que no se contamina con los dispositivos que intentan nuevamente con los mensajes o con las colas. Para que la métrica sea aún más clara, emitimos una segunda métrica: *AgeOfFirstSubscriberFirstAttempt*. En un sistema PubSub como AWS IoT, no hay un límite práctico de cuántos dispositivos o aplicaciones pueden suscribirse a un tema en particular, por lo que la latencia es mayor cuando se envía el mensaje a un millón de dispositivos que cuando se envía a uno solo. A fin de obtener una métrica estable, emitimos una métrica de temporizador en el primer intento de publicar un mensaje al primer suscriptor de ese tema. Luego tenemos otras métricas para medir el progreso del sistema al publicar los mensajes restantes.

La métrica AgeOfFirstAttempt sirve como advertencia temprana para un problema de todo el sistema, en gran medida porque filtra el ruido de los dispositivos que se eligen leer sus mensajes de manera más lenta. Vale la pena mencionar que los sistemas como AWS IoT vienen equipados con muchas más métricas además de esta. Pero con todas las métricas relacionadas con la latencia disponibles, comúnmente se utiliza la estrategia de categorizar la latencia de los primeros intentos separada de la latencia de los reintentos en todo Amazon.

Medir la latencia y la disponibilidad de los sistemas asíncronos es un desafío, y depurar también puede ser complicado, dado que las solicitudes rebotan entre los diferentes servidores y se pueden demorar en lugares externos a cada sistema. Para colaborar con el rastreo distribuido, propagamos un *ID de solicitud* por nuestros mensajes en cola de manera que podamos unir las piezas. Normalmente usamos sistemas como X-Ray para ayudar con esto también.

Demoras en los sistemas asíncronos multiinquilinos

Muchos sistemas asíncronos son multiinquilinos, y gestionan el trabajo en nombre de muchos clientes diferentes. Esto añade una dimensión complicada a la gestión de latencia y disponibilidad. El beneficio de los sistemas multiinquilino es que nos ahorra los gastos generales operativos de tener que operar por separado múltiples flotas, y nos permite ejecutar cargas de trabajo combinadas con una utilización de recursos mucho más alta. Sin embargo, los clientes esperan que tenga su propio

sistema de un solo inquilino, con latencia predecible y alta disponibilidad, independientemente de las cargas de trabajo de otros clientes.

Los servicios de AWS no exponen sus colas internas directamente para que los intermediarios ingresen mensajes. En cambio, implementan API ligeras para autenticar a los intermediarios y agregar información del intermediario en cada mensaje antes de ponerlos en cola. Esto es similar a la arquitectura de Lambda que se describió anteriormente: cuando evoca una función de manera asincrónica, Lambda coloca su mensaje en una cola propiedad de Lambda y regresa de inmediato, en lugar de exponer las colas internas de Lambda directamente ante usted.

Estas API ligeras también nos permiten agregar límites equitativos. La equidad en un sistema multiinquilino es importante para que la carga de trabajo de un cliente no afecte a otro. Una manera bastante común que tiene AWS para implementar la equidad es estableciendo límites basados en índices por clientes, con cierta flexibilidad para las ráfagas. En muchos de nuestros sistemas, por ejemplo en el mismo SQS, aumentamos los límites por cliente a medida que los clientes crecen de manera orgánica. Los límites sirven como barandales para picos imprevistos, lo que nos da tiempo para realizar ajuste de suministro detrás de escena.

En cierto modo, la equidad en los sistemas asíncronos funciona tal como los límites en los sistemas sincrónicos. Sin embargo, creemos que es incluso más importante pensar en los sistemas asíncronos dadas las grandes demoras que pueden acumularse tan rápidamente.

A modo de demostración, piense en qué pasaría si un sistema asíncrono no tuviera suficientes protecciones contra vecinos ruidosos integradas. Si un cliente del sistema de repente tuviera un pico en el tráfico ilimitado y generara una demora en todo el sistema, el operador demoraría unos 30 minutos en conectarse, descubrir qué ocurre y solucionar el problema. Durante esos 30 minutos, el lado del sistema que pertenece al productor podría haber escalado y puesto en cola todos los mensajes. Pero si el volumen de mensajes en cola era 10 veces la capacidad a la que escaló el lado del cliente, esto significa que el sistema demoraría 300 minutos en lidiar con la demora y recuperarse. Incluso los picos de carga corta pueden producir tiempos de recuperación de varias horas, y por lo tanto, provocar interrupciones de varias horas.

En la práctica, los sistemas de AWS cuentan con numerosos factores de compensación para minimizar o prevenir el impacto negativo de las demoras en las colas. Por ejemplo, el escalado automático ayuda a mitigar los problemas cuando aumenta la carga. Pero es útil para observar los efectos de la cola únicamente, sin tener en cuenta los factores de compensación, ya que esto ayuda a diseñar sistemas confiables en múltiples capas. Estos son algunos patrones de diseño que notamos que pueden evitar las grandes demoras en las colas y los prolongados tiempos de recuperación:

- **La protección en cada capa es importante en los sistemas asíncronos.** Dado que los sistemas sincrónicos no tienden a acumular demoras, los protegemos con limitación de la puerta frontal y control de admisión. En los sistemas asíncronos, cada componente de nuestros sistemas debe protegerse a sí mismo de la sobrecarga, y evitar que una carga de trabajo consuma una parte no equitativa de los recursos. Habrá cierta carga de trabajo que pase por el control de admisión de la puerta frontal, por lo que necesitamos cinturón, tiradores y un protector de bolsillo para evitar que se sobrecarguen los servicios.
- **El uso de más de una cola ayuda a moldear el tráfico.** De cierta forma, una cola única y la multitenencia entran en conflicto entre sí. Para cuando se acumula trabajo en una cola compartida, es difícil separar una carga de trabajo de otra.
- **Los sistemas en tiempo real a menudo se implementan con colas estilo FIFO, pero prefieren una conducta estilo LIFO.** Nuestros clientes dicen que cuando se enfrentan a una

demora, prefieren ver los datos actualizados procesados de inmediato. Cualquier dato acumulado durante una interrupción o sobretensión se puede procesar entonces como capacidad disponible.

Las estrategias de Amazon para crear sistemas asíncronos multiinquilino resilientes

Existen muchos patrones que los sistemas de Amazon usan para hacer que sus sistemas asíncronos multiinquilinos sean resilientes a los cambios en las cargas de trabajo. Existen muchas técnicas, pero también hay muchos sistemas que se usan en todo Amazon, cada uno con su propio conjunto de requisitos de durabilidad y actividad. En la siguiente sección, describo algunos de los patrones que usamos y que los clientes de AWS nos cuentan que usan en sus sistemas.

Separación de las cargas de trabajo en colas separadas

En lugar de compartir una cola entre todos los clientes, en algunos sistemas le damos a cada cliente su propia cola. Agregar una cola para cada cliente o carga de trabajo no siempre es algo rentable, ya que los servicios deberán emplear recursos para agrupar todas las colas. Pero en los sistemas con un puñado de clientes y sistemas adyacentes, esta solución simple puede ser útil. Por otro lado, si un sistema tiene incluso decenas o cientos de clientes, separar las colas puede empezar a ser algo inmanejable. Por ejemplo, AWS IoT no usa una cola separada para cada dispositivo IoT del universo. Los costos de agrupación no escalarán bien en ese caso.

Fragmentación aleatoria

AWS Lambda es un ejemplo de un sistema en que la agrupación de una cola separada para cada cliente de Lambda sería demasiado costoso. Sin embargo, tener una sola cola podría generar algunos de los problemas que se describen en este artículo. Por lo tanto, en lugar de una cola, AWS Lambda suministra una cantidad fija de colas, y divide a cada cliente en una pequeña cantidad de colas. Antes de poner en cola un mensaje, verifica cuál de esas colas objetivos contiene menos mensajes, y pone el mensaje en esa cola. Cuando la carga de trabajo de un cliente aumenta, provocará una demora en sus colas asignadas, pero automáticamente se alejan otras cargas de trabajo de esas colas. No se requiere una gran cantidad de colas para acumular cierto aislamiento de recursos mágicos. Esta es solo una de las muchas protecciones integradas en Lambda, pero es una técnica que también se utiliza en otros servicios de Amazon.

Cómo dejar a un lado el exceso de tráfico en una cola separada

En cierta manera, cuando se ha acumulado demora en una cola, es demasiado tarde como para priorizar el tráfico. Sin embargo, si el procesamiento del mensaje es relativamente costoso o demanda demasiado tiempo, posiblemente aún valga la pena trasladar los mensajes a una cola indirecta separada. En algunos sistemas de Amazon, el servicio al cliente implementa la limitación distribuida, y cuando retiran de la cola los mensajes de un cliente que ha pasado por un índice configurado, colocan en cola esos mensajes excedentes en colas indirectas separadas y eliminan los mensajes de la cola primaria. El sistema sigue funcionando con los mensajes de la cola indirecta

apenas hay recursos disponibles. En esencia, esto se aproxima a una cola de prioridad. En ocasiones se usa una lógica similar del lado del productor. De este modo, si un sistema acepta un gran volumen de solicitudes de una sola carga de trabajo, esa carga no desplaza las demás cargas de trabajo de la cola de ruta de acceso activa.

Cómo dejar a un lado el tráfico antiguo en una cola separada

Parecido a como dejamos a un lado el exceso de tráfico, también podemos hacerlo con el tráfico antiguo. Cuando quitamos de la cola un mensaje, podemos verificar su antigüedad. En lugar de tan solo registrar su antigüedad, podemos usar la información para decidir si debemos mover el mensaje a una cola demorada en la que trabajamos únicamente después de quedar atrapados en una cola en vivo. Si hay un pico de carga en la que introducimos muchos datos, y nos atrasamos, podemos dejar a un lado esa onda de tráfico en una cola diferente tan rápido como podamos quitar y volver a poner en cola el tráfico. Esto libera los recursos del cliente para que trabajen en los mensajes actualizados más rápidamente que si hubiéramos simplemente trabajado la demora en orden. Este es un modo de aproximarse a la ordenación LIFO.

Desecho de mensajes antiguos (tiempo de vida del mensaje)

Algunos sistemas pueden tolerar que se desechen los mensajes muy antiguos. Por ejemplo, algunos sistemas procesan diferencias de los sistemas rápidamente, pero también realizan una sincronización completa en forma periódica. A menudo estos sistemas de sincronización periódica se denominan barrenderos antientropía. En estos casos, en lugar de dejar de lado el tráfico antiguo que está en cola, podemos eliminarlo en forma menos costosa si entró antes de la barrida más reciente.

Limitación de subprocesos (y otros recursos) por carga de trabajo

Al igual que en nuestros servicios sincrónicos, diseñamos nuestros sistemas asíncronos para evitar que una carga de trabajo utilice más que su parte equitativa de subprocesos. Un aspecto de AWS IoT del que aún no hablamos son los motores de reglas. Los clientes pueden configurar AWS IoT para dirigir mensajes de sus dispositivos a un clúster Amazon Elasticsearch propiedad del cliente, Kinesis Stream, y así sucesivamente. Si la latencia de esos recursos propiedad del cliente se torna lenta, pero el índice de mensajes entrantes sigue siendo constante, la cantidad de concurrencia en el sistema aumenta. Y dado que la cantidad de concurrencia que puede manejar un sistema se limita en cualquier momento, el motor de reglas evita que cualquier carga de trabajo consuma más que su parte equitativa de recursos relacionados con la concurrencia.

La fuerza en el trabajo se describe mediante Ley de Little, que dicta que la concurrencia de un sistema es igual al índice de llegada multiplicado por la latencia promedio de cada solicitud. Por ejemplo, si un servidor procesaba 100 mensajes/seg a 100 ms promedio, consumiría 10 subprocesos en promedio. Si la latencia de repente llegara al pico máximo de 10 segundos, utilizaría 1000 subprocesos (en promedio, por lo que podría ser más en la práctica), que podría fácilmente agotar un grupo de subprocesos.

El motor de reglas utiliza varias técnicas para evitar que esto ocurra. Utiliza E/S sin bloqueo para evitar que se agote el subproceso, aunque aún hay otros límites en relación con cuánto trabajo tiene determinado servidor (por ejemplo, memoria y descriptores de archivos cuando el cliente está procesando mediante conexiones y la dependencia se está agotando). Una segunda protección de

concurrencia que se puede usar es un semáforo que mide y limita la cantidad de concurrencia que puede usarse para cualquier carga de trabajo única en cualquier momento. El motor de reglas también utiliza la limitación equitativa basada en el índice. Sin embargo, dado que es perfectamente normal para que las cargas de trabajo cambien con el tiempo, el motor de reglas también escala automáticamente los límites con el tiempo para adaptarse a los cambios de las cargas de trabajo. Y dado que el motor de reglas se basa en la cola, sirve como amortiguador entre los dispositivos IoT y la escalación automática de recursos y protege los límites detrás de escena.

En los servicios de Amazon, usamos grupos de subprocesos separados para cada carga de trabajo para evitar que una carga de trabajo consuma todos los subprocesos disponibles. También usamos un *AtomicInteger* para cada carga de trabajo para limitar la concurrencia permitida para cada una, y enfoques de limitación basados en los índice para aislar los recursos basados en los índices.

Cómo enviar la contrapresión hacia arriba

Si una carga de trabajo genera una demora irracional a la que el cliente no puede seguirle el ritmo, muchos de nuestros sistemas automáticamente comienzan a rechazar trabajo de manera más agresiva en el productor. Es fácil acumular una demora de un día para una carga de trabajo. Incluso si se aísla esa carga de trabajo, podría ser accidental y costoso de procesar. La implementación de este enfoque podría ser simplemente medir la profundidad de la cola de una carga de trabajo en forma ocasional (asumiendo que la carga de trabajo está en su propia cola), y escalar un límite interno (de manera inversa) proporcional al tamaño de la demora.

En caso en que compartamos una cola SQS para múltiples cargas de trabajo, este enfoque se complica. Si bien hay una API de SQS que regresa la cantidad de mensajes que hay en la cola, no hay API que pueda regresar la cantidad de mensajes que hay en la cola con un atributo particular. Aún podemos medir la profundidad de la cola y aplicar contrapresión en consecuencia, pero sería injusto ejercer contrapresión en cargas de trabajo inocentes que comparten la misma cola. Otros sistemas como Amazon MQ tienen una visibilidad de demoras más precisa.

La contrapresión no es apta para todos los sistemas de Amazon. Por ejemplo, en los sistemas que realizan procesamiento de orden para amazon.com, tendemos a preferir aceptar los pedidos incluso si se acumula demora, en lugar de evitar que se acepten nuevos pedidos. Pero por supuesto, esto viene acompañado con mucha priorización detrás de escena, por lo que se lidia primero con los pedidos más urgentes.

Uso de colas demoradas para posponer trabajo para más tarde

Cuando los sistemas tienen la sensación de que se debe reducir el desempeño de una carga de trabajo particular, intentamos usar una estrategia de *interrupción* de esa carga de trabajo. Para implementar esto, solemos usar una función de SQS que retrasa la entrega de un mensaje hasta más tarde. Cuando procesamos un mensaje y decidimos guardarlo para más tarde, en ocasiones volvemos a poner ese mensaje en una *cola en aumento* separada, pero configuramos el parámetro de retraso de manera que el mensaje permanezca oculto en la cola retrasada durante algunos minutos. Esto le da al sistema la posibilidad de trabajar con datos más actualizados.

Cómo evitar que haya demasiados mensajes en tránsito

Algunos servicios de cola como SQS tienen límites en relación con cuántos mensajes en tránsito se pueden enviar al cliente de la cola. Es diferente a la cantidad de mensajes que puede haber en la cola (para los cuales no hay límite práctico), pero en cambio es la cantidad de mensajes en los que está trabajando la flota del consumidor en ese momento. Esta cantidad se puede inflar si un sistema quita de la cola los mensajes, pero luego no los elimina. Por ejemplo, hemos visto errores cuando el código no logra captar una excepción mientras procesa un mensaje y se olvida de eliminar el mensaje. En estos casos, el mensaje permanece en tránsito desde la perspectiva de SQS para la *VisibilityTimeout* del mensaje. Cuando diseñamos nuestra estrategia de sobrecarga y manejo de errores, tenemos presentes estos límites, y tendemos a favorecer el traslado de exceso de mensajes a una cola diferente en lugar de dejarlos permanecer visibles.

Las colas FIFO de SQS tienen un límite similar, pero sutil. Con las colas FIFO de SQS, los sistemas consumen sus mensajes en orden para un *grupo de mensajes*, pero los mensajes de diferentes grupos se procesan en cualquier orden. Por lo tanto, si desarrollamos una demora pequeña en un grupo de mensajes, seguimos procesando los mensajes en otros grupos. Sin embargo, FIFO de SQS solo agrupa los 20.000 mensajes sin procesar más recientes. Por lo tanto, si hay más de 20.000 mensajes sin procesar en un subconjunto de grupos de mensajes, se agotarán otros grupos con mensajes actualizados.

Uso de colas de mensajes fallidos para mensajes que no se pueden procesar

Los mensajes que no se pueden procesar pueden contribuir con la sobrecarga del sistema. Si un sistema pone en cola un mensaje que no se puede procesar (quizás porque dispara un caso de borde de validación de entrada), SQS puede ayudar trasladando estos mensajes automáticamente a una cola separada con la función de cola de mensajes fallidos (DLQ). Accionamos una alarma si hay mensajes en esta cola, ya que significa que se debe reparar un error. El beneficio del DLQ es que nos permite volver a procesar los mensajes una vez que se repara el error.

Cómo asegurar amortiguación adicional en los subprocesos de agrupación por carga de trabajo

Si una carga de trabajo está impulsando suficiente desempeño a tal punto que los subprocesos de agrupación están ocupados todo el tiempo, incluso durante el estado continuo, el sistema habrá llegado a un punto en que no hay amortiguación para absorber un aumento en el tráfico. En este estado, un pequeño pico en el tráfico entrante, llevará a una cantidad sostenida de demora sin procesar, lo que resultará en una mayor latencia. Planificamos una amortiguación adicional en los subprocesos de agrupación a fin de absorber dichos aumentos. Una medida es rastrear la cantidad de intentos de agrupación que resulta en respuestas vacías. Si cada intento de agrupación recupera un mensaje más, entonces tenemos la cantidad justa de subprocesos de agrupación, o posiblemente no tenga la cantidad necesaria para mantener el tráfico entrante.

Latidos de mensajes de ejecución larga

Cuando un sistema procesa un mensaje de SQS, este le da a ese sistema cierta cantidad de tiempo para terminar de procesar el mensaje antes de suponer que el sistema colapsó, y de entregar el mensaje a otro cliente para volver a intentarlo. Si el código sigue funcionando y olvida su fecha límite, se puede entregar el mismo mensaje varias veces en paralelo. Si bien el primer procesador aún está procesando un mensaje luego de acabarse el tiempo, un segundo procesador tomará la tarea y lo procesará de modo similar luego de terminado el tiempo, y luego un tercero, y así sucesivamente. Este potencial para pasar las interrupciones en cascada es el por qué implementamos nuestra lógica de procesamiento de mensajes para que deje de trabajar cuando caduque el mensaje, o para que siga latiendo para recordarle a SQS que aún está trabajando en él. Este concepto es similar a concesiones en la elección de líder.

Este es un problema insidioso, ya que vemos que es probable que aumente la latencia en un sistema durante una sobrecarga, quizás por consultas a una base de datos que demora demasiado, o por servidores que simplemente requieren más trabajo del que pueden manejar. Cuando la latencia del sistema cruza ese umbral de VisibilityTimeout, provoca un servicio ya sobrecargado para prácticamente hacer una bomba fork sobre sí mismo.

Plan para la depuración entre anfitriones

Comprender las fallas en un sistema distribuido ya es difícil. El artículo relacionado sobre instrumentación describe varios de nuestros enfoques para instrumentar los sistemas asíncronos, para registrar las profundidades de las colas en forma periódica, propagar «ID de rastreo» e integrar con X-Ray. O, cuando nuestros sistemas tengan un flujo de trabajo asíncrono complicado más allá de una cola SQS trivial, solemos usar un servicio de flujo de trabajo asíncrono como Step Functions, que proporciona visibilidad en el flujo de trabajo y simplifica la depuración distribuida.

Conclusión

En un sistema asíncrono, es fácil olvidar la importancia de pensar en la latencia. Después de todo, los sistemas asíncronos suelen tomar más tiempo, ya que se enfrentan a una cola para realizar reintentos confiables. Sin embargo, los escenarios de sobrecarga y falla pueden acumular enormes demoras insuperables de las que un servicio no puede recuperarse en una cantidad de tiempo razonable. Estas demoras pueden provenir de una carga de trabajo o de un cliente que forma colas a un índice alto imprevisto, de cargas de trabajo que se hicieron más costosas de lo que se previó que se iba a procesar, o de la latencia o las fallas de una dependencia.

Cuando creamos un sistema asíncrono, debemos centrarnos en estos escenarios de demora y anticiparlos, y minimizarlos utilizando técnicas como la priorización, la marginalización y la contrapresión.

Más información

- Teoría de las colas
- Ley de Little
- Ley de Amdahl

- Little A Proof for the Queuing Formula: L = λW, Case Western, 1961
- McKenney, Stochastic Fairness Queuing, IBM, 1990
- Nichols and Jacobson, Controlling Queue Delay, PARC, 2011