Uso de la eliminación de carga para evitar la sobrecarga

David Yanacek

Durante algunos años, trabajé en el equipo Marcos de servicio de Amazon. Nuestro equipo escribió herramientas que ayudaron a los propietarios de los servicios de AWS, tales como Amazon Route 53 y Elastic Load Balancing, a crear sus servicios con mayor rapidez y brindar servicio a los clientes que piden esos servicios de manera más fácil. Otros equipos de Amazon les proporcionan a los propietarios de servicios cierta funcionalidad, tal como medición, autenticación, monitoreo, generación de biblioteca del cliente y generación de documentación. En lugar de que cada equipo de servicio tenga que integrar dichas funciones a sus servicios en forma manual, el equipo de Marcos de servicios realizó dicha integración una vez y expuso la funcionalidad a cada servicio mediante la configuración.

Un desafío que enfrentamos fue determinar cómo proporcionar valores predeterminados sensibles, especialmente para funciones relacionadas con el rendimiento o la disponibilidad. Por ejemplo, no podíamos establecer fácilmente el tiempo de espera del lado del cliente, ya que nuestro marco no tenía idea de cuáles podían ser las características de latencia de una llamada de API. Esto no hubiera sido más fácil de descifrar para los mismos propietarios o clientes del servicio, así que seguimos intentando y obtuvimos cierta información en el camino.

Una cuestión común con la que lidiamos fue determinar la cantidad predeterminada de conexiones que el servidor permitiría que se abrieran para los clientes al mismo tiempo. Esta configuración se diseñó para evitar que un servidor demore demasiado tiempo y se sobrecargue. Más específicamente, deseábamos configurar las conexiones máximas para el servidor en proporción con las conexiones máximas para el balanceador de carga. Esto fue antes de los días de Elastic Load Balancing, por lo que los balanceadores de carga estaban en pleno uso.

Nos dispusimos a ayudar a los propietarios y clientes de los servicios de Amazon a descifrar el valor ideal de las conexiones máximas para configurar el balanceador de carga, y el valor correspondiente para configurar los marcos que proporcionábamos. Decidimos de que si podíamos descifrar cómo usar el juicio humano para tomar decisiones, podríamos escribir un software para simular ese juicio.

Determinar el valor ideal terminó siendo un verdadero desafío. Cuando las conexiones máximas eran demasiado bajas, el balanceador de carga podía reducir los aumentos de la cantidad de solicitudes, incluso cuando el servicio tenía mucha capacidad. Cuando las conexiones máximas eran demasiado altas, los servidores podían volverse lentos y dejar de responder. Cuando las conexiones máximas eran las adecuadas para una carga de trabajo, dicha carga cambiaría o cambiaría el rendimiento de dependencia. Entonces, los valores estarían mal nuevamente, lo que resultaría en cortes o sobrecargas innecesarios.

Al final, notamos que el concepto de conexiones máximas era demasiado impreciso como para proporcionar una respuesta completa al desafío. En este artículo, describiremos otros enfoques, como el desbordamiento de carga, que notamos que funcionaron bien.

La anatomía de la sobrecarga

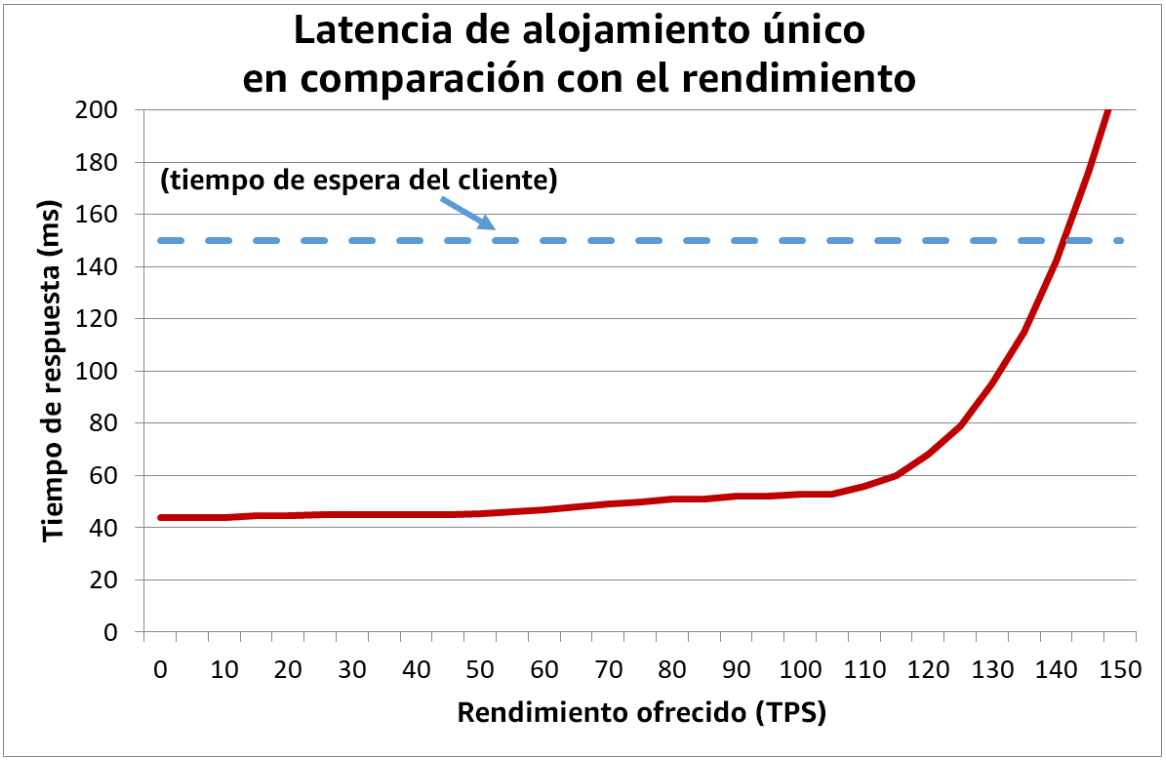
En Amazon, evitamos la sobrecarga diseñando nuestros sistemas para que escalen de manera proactiva, antes de encontrarse con situaciones de sobrecarga. Sin embargo, la protección de los sistemas implica la protección en capas. Esto comienza con la escalación automática, pero también incluye mecanismos para desbordar el exceso de carga de manera simple, la capacidad de monitorear dichos mecanismos y, lo más importante, las pruebas continuas.

Cuando realizamos pruebas de carga de nuestros servicios, encontramos que la latencia de un servidor con bajo nivel de uso es inferior a su latencia a un nivel de uso mayor. Baja una carga pesada, la contención de los subprocesos, el cambio de contexto, la acumulación de basura y la contención de E/S se vuelven más pronunciados. Con el tiempo, los servicios alcanzan un punto de inflexión en el que su rendimiento comienza a degradarse incluso más rápidamente.

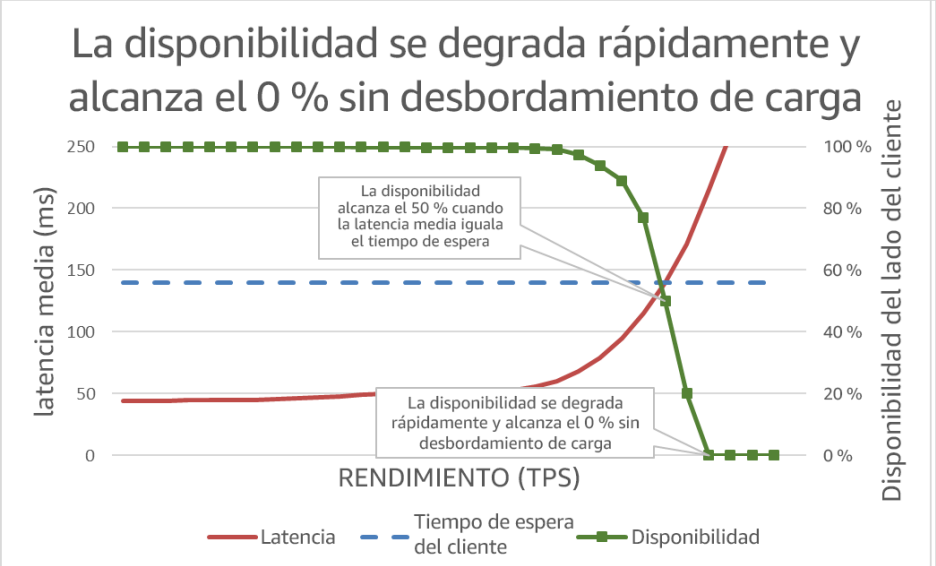
La teoría detrás de esta observación se conoce como laley de escalabilidad universal, que deriva de la ley de Amdahl. Esta teoría especifica que si bien se puede aumentar el rendimiento de un sistema utilizando la paralelización, queda limitado en última instancia por el rendimiento de los puntos de serialización (es decir, por las tareas que no se pueden hacer paralelizar).

Desafortunadamente, el rendimiento no solo está vinculado a los recursos del sistema, sino que generalmente se degrada cuando el sistema está sobrecargado. Cuando se le otorga más trabajo a un sistema de los sus recursos pueden soportar, se hace lento. Las computadoras toman trabajo incluso cuando están sobrecargadas, pero pasan más tiempo cambiando de contexto y se ralentizan demasiado como para ser útiles.

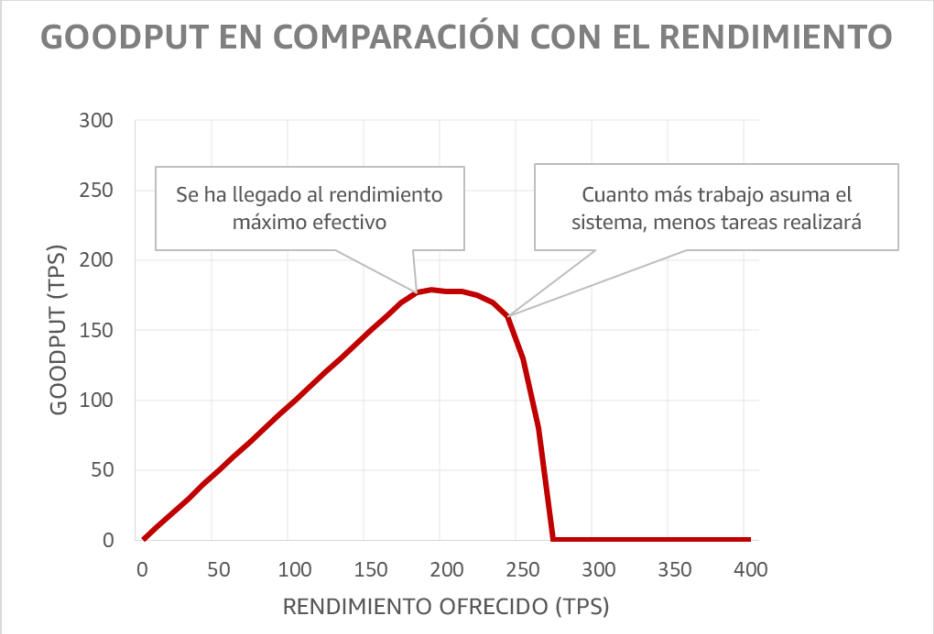
En un sistema distribuido donde un cliente habla con un servidor, el cliente generalmente se impacienta y deja de esperar a que el servidor le responda después de cierto tiempo. Esta duración se conoce como *tiempo de espera.* Cuando un servidor está tan sobrecargado que su latencia excede el tiempo de espera del cliente, las solicitudes comienzan a fallar. El siguiente gráfico muestra cómo aumenta el tiempo de respuesta del servidor a medida que aumenta la oferta de rendimiento (en transacciones por segundo), y el tiempo de respuesta con el tiempo alcanza un punto de inflexión en el que las cosas se deterioran rápidamente.



En el gráfico anterior, cuando el tiempo de respuesta excede el tiempo de espera del cliente, queda claro que las cosas están mal, pero el gráfico no muestra qué tan mal. Para ilustrar esto, podemos mostrar la disponibilidad percibida por el cliente junto con la latencia. En lugar de usar una medición del tiempo de respuesta genérico, podemos cambiar y usar el tiempo de respuesta medio. El tiempo de respuesta medio significa que el 50 % de las solicitudes fueron más rápidas que el valor medio. Si la latencia mediana del servicio equivale al tiempo de espera del cliente, la mitad de las solicitudes se agotarán, por lo que la disponibilidad es del 50 %. Aquí es donde un aumento en la latencia transforma un problema de latencia en un problema de disponibilidad. Este es un gráfico de lo que ocurre:



Desafortunadamente, este gráfico es difícil de leer. Una forma simple de describir el problema de disponibilidad es distinguir entre *goodput* y *rendimiento.* El rendimiento es el número total de solicitudes por segundo que se envían al servidor. Goodput es el subconjunto de rendimiento que se maneja sin errores y con una latencia suficientemente baja como para que el cliente haga uso de la respuesta.



Bucles de retroalimentación positivos

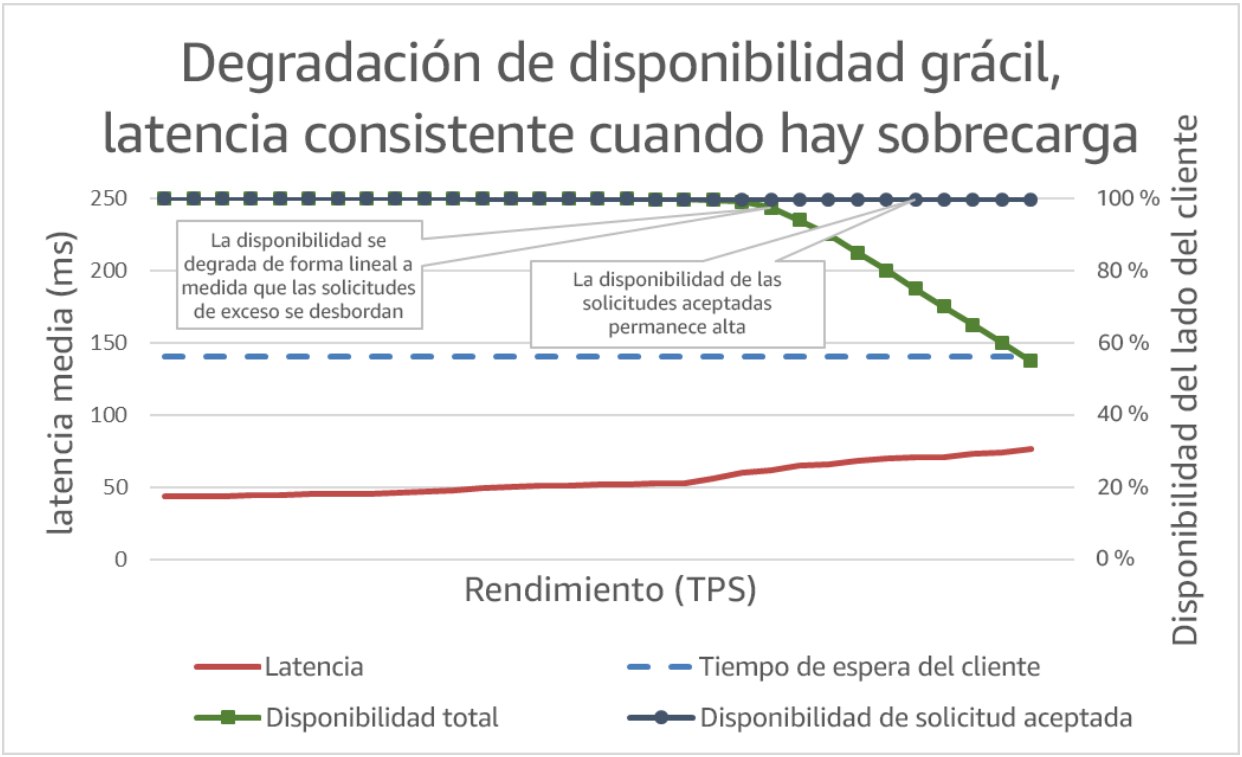
La parte insidiosa de una situación de sobrecarga es cómo se amplifica en un bucle de retroalimentación. Cuando el tiempo de espera de un cliente se agota, es muy probable que reciba un error. Lo peor es que todo el progreso que ha realizado el servidor hasta el momento en esa solicitud se desperdicia. Y lo último que debería hacer un sistema en una situación de sobrecarga, donde la capacidad está limitada, es desperdiciar trabajo.

Para empeorar más las cosas, los clientes suelen reintentar su solicitud. Esto multiplica la carga que se ofrece en el sistema. Y si hay un gráfico de llamadas suficientemente profundo en una arquitectura orientada al servicio (es decir, el cliente llama a un servicio, que llama a otros servicios, que llaman a otros servicios), y si cada capa realiza una cantidad de reintentos, la sobrecarga del nivel inferior provoca reintentos en cascada que amplifican la carga ofrecida de manera exponencial.

Cuando estos factores se combinan, una sobrecarga crea su propio bucle de retroalimentación que resulta en una sobrecarga como un estado continuo.

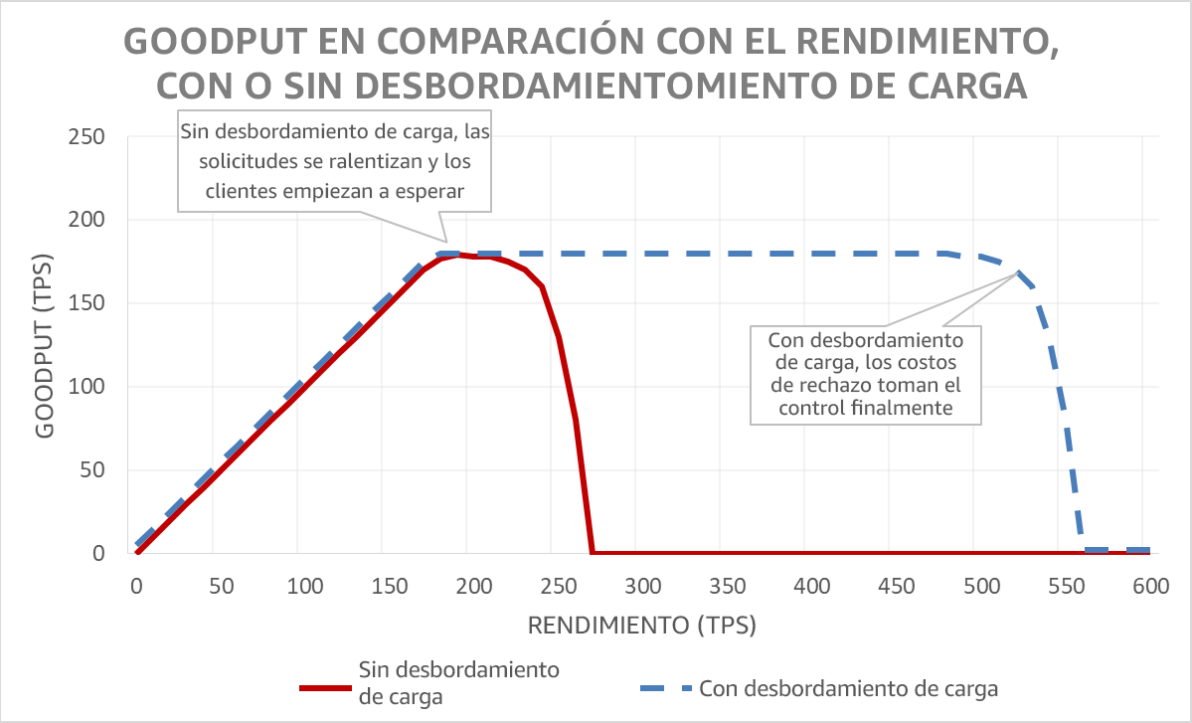
Cómo evitar que se desperdicie trabajo

En la superficie, el desbordamiento de carga es simple. Cuando un servidor aborda una sobrecarga, debería comenzar a rechazar el exceso de solicitudes de manera que se pueda centrar en las solicitudes a las que decide dejar ingresar. El objetivo del desbordamiento de carga es mantener la latencia baja para las solicitudes que el servidor decide aceptar, de manera que el servicio responda antes del tiempo de espera del cliente. Con este enfoque, el servidor mantiene una alta disponibilidad para las solicitudes que acepta y únicamente se ve afectada la disponibilidad del tráfico excesivo.



Al mantener vigilada la latencia desbordando el exceso de carga, el sistema está más disponible. pero los beneficios de este enfoque son difíciles de visualizar en el gráfico anterior. La línea de disponibilidad total baja, y eso no se ve bien. La clave es que las solicitudes que el servidor decidió aceptar permanecen disponibles ya que reciben servicio rápidamente.

El desbordamiento de carga permite que el servidor mantenga su goodput y complete tantas solicitudes como pueda, incluso a medida que el rendimiento ofrecido aumenta. Sin embargo, el acto de desbordar la carga no es gratuito, por lo que con el tiempo el servidor cae presa de la ley de Amdahl y de las caídas de goodput.



Pruebas

Cuando hablo con otros ingenieros sobre el desbordamiento de carga, quiero señalar que si no hubieran probado la carga de su servicio hasta el punto en que se quiebra, y más allá del punto de quiebre, supondrían que el servicio va a caer del modo menos deseable posible. En Amazon, empleamos mucho tiempo en la prueba de carga de nuestros servicios. Al generar gráficos como los de este artículo, establecemos una referencia en el rendimiento de sobrecarga y rastreamos nuestro desempeño en el tiempo a medida que realizamos cambios en nuestros servicios.

Existen múltiples tipos de pruebas de carga. Algunas pruebas de carga garantizan la escalación automática a medida que la carga aumenta, mientras que otras usan un tamaño de flota fija. Si durante una prueba de sobrecarga, la disponibilidad de un servicio se degrada rápidamente hasta cero por el aumento del rendimiento, es una buena señal de que el servicio requiere mecanismos adicionales de desbordamiento de carga. El resultado de una prueba de carga ideal es que el goodput esté estable cuando el servicio está cerca de estar siendo utilizado en su totalidad, y de permanecer plano incluso cuando se aplica más rendimiento.

Algunas herramientas, como Chaos Monkey, ayudan a realizar pruebas de ingeniería del caos en servicios. Por ejemplo, pueden sobrecargar la CPU o introducir pérdida de paquete para simular condiciones que ocurren durante una sobrecarga. Otra técnica de prueba que usamos es tomar un valor controlado o una prueba de generación de carga existente, impulsar una carga sostenida (en lugar de aumentar la carga) hacia un entorno de prueba, pero comenzar a quitar servidores desde ese entorno de prueba. Esto aumenta el rendimiento que se ofrece por instancia, por lo que se puede probar el rendimiento de la instancia. Esta técnica de una carga que aumenta en forma artificial al

disminuir los tamaños de la flota, es útil para probar un servicio aislado, pero no es un sustituto completo para las pruebas de carga completas. Una prueba de carga completa e integral también ayudará a aumentar la carga hasta las dependencias de esos servicios, lo que podría develar otros obstáculos.

Durante las pruebas, nos aseguramos de medir la disponibilidad y la latencia percibidas por el cliente además de a disponibilidad y la latencia del lado del servidor. Cuando la disponibilidad del lado del cliente comienza a disminuir, forzamos la carga más allá de ese punto. Si el desbordamiento de la carga funciona, el goodput permanecerá estable incluso si el rendimiento ofrecido aumenta más allá de las capacidades escaladas del servicio.

Las pruebas de sobrecarga son cruciales antes de explorar los mecanismos a fin de evitar dicha sobrecarga. Cada mecanismo presenta cierta complejidad. Por ejemplo, piense en todas las opciones de configuración de los marcos del servicio que mencioné al comienzo del artículo, y lo difícil que fue obtener valores predeterminados correctos. Cada mecanismo para evitar sobrecargas añade diferentes protecciones y tiene una efectividad limitada. Mediante las pruebas, un equipo puede detectar los obstáculos de su sistema y determinar la combinación de las protecciones que necesitan para manejar la sobrecarga.

Visibilidad

En Amazon, independientemente de qué técnicas usemos para proteger nuestros servidores de la sobrecarga, pensamos cuidadosamente en las métricas y en la visibilidad que necesitamos cuando entran en vigencia estas contramedidas para la sobrecarga.

Cuando la protección contra la caída de tensión rechaza una solicitud, dicho rechazo reduce la disponibilidad de un servicio. Cuando el servicio se equivoca y rechaza una solicitud a pesar de tener dicha capacidad (por ejemplo, cuando el número máximo de conexiones se establece demasiado bajo), genera un falso positivo. Nosotros nos esforzamos por mantener el índice de falsos positivos de un servicio en cero. Si un equipo nota que el índice de falsos positivos de un servicio suele no ser cero, el servicio puede estar configurado de manera muy sensible, o los hosts individuales se sobrecargan de manera constante y legítima, y pueden tener un problema de escalado o de balance de carga. En estos casos, es posible que debamos ajustar el rendimiento de la aplicación, o podemos cambiar a tipos de instancias más grandes que puedan manejar desequilibrios en la carga de manera más satisfactoria.

En términos de visibilidad, cuando el desbordamiento de la carga rechaza las solicitudes, nos aseguramos de contar con la instrumentación adecuada para saber quién fue el cliente, qué operación estaba llamando y cualquier otra información que nos ayude a ajustar las medidas de protección. También usamos alarmas para detectar si las contramedidas están rechazando algún volumen significativo de tráfico. Cuando hay una caída en la tensión, nuestra prioridad es añadir capacidad y abordar el obstáculo actual.

Hay otra consideración, sutil pero importante, en torno a la visibilidad del desbordamiento de carga. Notamos que es importante no contaminar las métricas de latencia de nuestros servicios con latencia de solicitudes fallidas. Después de todo, la latencia del desbordamiento de carga de una solicitud debería ser extremadamente baja en comparación con otras solicitudes. Por ejemplo, si el desbordamiento de carga de un servicio es del 60 % de su tráfico, la latencia mediana del servicio podría verse increíble incluso si la latencia de solicitud satisfactoria es terrible, ya que no se informa como resultados de las solicitudes que fallan rápidamente.

El desbordamiento de carga afecta el escalado automático y la falla de la zona de disponibilidad

Si se configura mal, el desbordamiento de carga puede desactivar el escalado automático reactivo. Considere el siguiente ejemplo: se configura un servicio para un escalado reactivo basado en CPU y también se configura el desbordamiento de carga para que rechace solicitudes de un CPU objetivo similar. En este caso, el sistema de desbordamiento de carga reducirá la cantidad de solicitudes para mantener la carga de la CPU baja, y el escalado reactivo nunca recibirá la señal demorada para lanzar nuevas instancias.

También tenemos cuidado de considerar la lógica de desbordamiento cuando establecemos los límites de escalado automático para manejar las fallas de la zona de disponibilidad. Los servicios escalan hasta un punto en que el valor de la capacidad de una zona de disponibilidad puede quedar no disponible a la vez que preserva nuestros objetivos de latencia. Los equipos de Amazon suelen ver las métricas del sistema como CPU para calcular qué tan cerca está un servicio de alcanzar el límite de su capacidad. Sin embargo, con el desbordamiento de carga, una flota podría funcionar mucho más cerca del punto en el que se rechazarían las solicitudes de lo que indican las métricas del sistema, y podrían no tener el exceso de capacidad para lidiar con una falla de zona de disponibilidad. Con el desbordamiento de carga, debemos estar muy seguros de probar el quiebre de nuestros servicios a fin de comprender la capacidad y el margen de nuestra flota en cualquier momento.

De hecho, podemos usar el desbordamiento de carga para ahorrar costos formando un tráfico sin picos y no crítico. Por ejemplo, si una flota lidia con tráfico del sitio web de amazon.com, podría decidir que no vale la pena de escalar hasta una redundancia de zona de disponibilidad por buscar tráfico rastreador. Sin embargo, somos muy cuidadosos con este enfoque. No todas las solicitudes cuestan lo mismo, y considerando que un servicio debería proporcionar una redundancia de zona de disponibilidad para tráfico humano y para tráfico rastreador de exceso desbordado a la vez, se requiere un diseño cuidadoso, pruebas continuas y la aceptación del negocio. Y si los clientes de un servicio no saben que hay un servicio configurado de este modo, su conducta durante una falla de zona de disponibilidad podría verse como una caída masiva crítica en la disponibilidad en lugar de como un desbordamiento de carga no crítico. Por este motivo, en una arquitectura orientada al servicio, intentamos presionar este tipo de formación lo antes posible (como en el servicio que recibe la solicitud inicial del cliente) en lugar de intentar tomar decisiones de priorización globales durante el apilamiento.

Decanismos de debordamiento de carga

Cuando se habla sobre desbordamiento de cargas y posibles escenarios, también es importante centrarse en las muchas condiciones *predecibles* que llevan a una bajada de tensión. En Amazon, los servicios mantienen suficiente exceso de capacidad como para lidiar con las fallas de la zona de disponibilidad sin tener que añadir más capacidad. Usan la limitación controlada par asegurar la equidad entre clientes.

Sin embargo, a pesar de estas protecciones y prácticas operativas, un servicio tiene cierta cantidad de capacidad en un momento determinado, y por lo tanto, puede sobrecargarse por diversos motivos. Estos motivos incluyen aumentos imprevistos de tráfico, pérdida repentina de capacidad de flota (por malas implementaciones u otro motivo), clientes que cambian de hacer solicitudes económicas (como lecturas con caché) a solicitudes costosas (como pérdidas o escrituras de caché). Cuando un servicio se sobrecarga, debe terminar las solicitudes que ha tomado; es decir, los servicios se deben proteger

a sí mismos de las caídas de tensión. En el resto de esta sección, analizaremos algunas consideraciones y técnicas que hemos usado durante estos años para lidiar con la sobrecarga.

Comprensión del costo de eliminar las solicitudes

Nos aseguramos de hacer una prueba de carga de nuestros servicios *mucho* más allá de dónde el goodput se estanca. Uno de los motivos clave de este enfoque es garantizar que cuando eliminamos solicitudes durante el desbordamiento de carga, el costo de esa eliminación sea lo menor posible. Hemos visto que es fácil perder una declaración de registro accidental o una configuración de socket, lo que haría que la eliminación de una solicitud fuera más costosa de lo que debe ser.

Muy pocas veces, eliminar rápidamente una solicitud puede ser más costoso que mantenerla. En estos casos, ralentizamos las solicitudes rechazadas para que coincidan (como mínimo) con la latencia de las respuestas satisfactorias. Sin embargo, es importante hacerlo cuando el costo de mantener las solicitudes es lo más bajo posible; por ejemplo, cuando no están vinculadas con el subproceso de una aplicación.

Priorización de solicitudes

Cuando se sobrecarga un servidor, tiene la oportunidad de evaluar las solicitudes entrantes para decidir cuáles debe aceptar y cuales rechazar. La solicitud más importante que recibirá un servidor es una solicitud de ping del balanceador de carga. Si el servidor no responde a tiempo a las solicitudes de ping, el balanceador de carga se detendrá y enviará nuevas solicitudes a ese servidor durante un período de tiempo, y el servidor quedará inactivo. Y en caso de una caída de tensión, lo último que deseamos hacer es reducir el tamaño de nuestras flotas. Más allá de las solicitudes de ping, las opciones de priorización varían de servicio en servicio.

Considere un servicio web que proporciona datos para renderizar amazon.com. Una llamada de servicio que admite la renderización de la página web para un rastreador de índice de búsqueda posiblemente sea menos crítica de servir que una solicitud que se origina a partir de un ser humano. Las solicitudes de rastreo son importantes de servir, pero idealmente se pueden cambiar para un momento que no sea el pico. Sin embargo, en un entorno complejo como el de amazon.com, donde se encuentran muchos servicios corporativos, si los servicios usan heurísticas de priorización en conflicto, la disponibilidad de todo el sistema podría verse afectada y se podría desperdiciar trabajo.

La priorización y la limitación controlada se pueden usar juntas para evitar techos de limitación controlada estríctos a la vez que se protege un servicio de la sobrecarga. En Amazon, en casos en los que los clientes permiten ráfagas por encima de sus límites controlados configurados, el exceso de solicitudes de estos clientes puede tener una prioridad inferior que las solicitudes que están dentro de la cuota de los demás clientes. Pasamos mucho tiempo centrándonos en colocar algoritmos para minimizar la probabilidad de que la capacidad de ráfagas no esté disponible, pero dadas las compensaciones, favorecemos la carga de trabajo suministradas predecibles por encima de las imprevistas.

Con un ojo en el reloj

Si un servicio quedó a medio camino entre servir una solicitud y nota que el tiempo de espera del cliente se agotó, puede omitir el resto del trabajo y no enviar la solicitud en ese momento. De otro modo, el servidor sigue trabajando en la solicitud, y su respuesta tardía es como un árbol que cae en el bosque. Desde la perspectiva del servidor, ha brindado una respuesta satisfactoria. Pero desde la perspectiva del cliente que agotó su tiempo de espera, fue un error.

Un modo de evitar este desperdicio de trabajo es que los clientes incluyan sugerencias de tiempo de espera en cada solicitud, lo que le informa al servidor cuánto tiempo estarán dispuestos a esperar. El servidor puede evaluar estas sugerencias y eliminar las solicitudes que están destinadas a ser eliminadas con muy bajo costo.

Esta sugerencia de tiempo de espera puede expresarse como un tiempo absoluto o como una duración. Desafortunadamente, los servidores de los sistemas distribuidos son notablemente peores para ponerse de acuerdo acerca de cuál es el tiempo actual exacto. El Servicio de sintronización de tiempo de Amazon compensa sincronizando los relojes de sus instancias Amazon Elastic Compute Cloud (Amazon EC2) con una flota de relojes redundantes atómicos y controlados por satélite en cada región de AWS. Los relojes bien sincronizados son importantes en Amazon también para los fines de registro. Al comparar dos archivos de registro con relojes que no están sincronizados, se dificulta aún más la resolución de problemas.

El otro modo de "observar el reloj" es medir una duración en una sola máquina. Los servidores se desempeñan bien al medir las duraciones transcurridas en forma local, ya que no requieren consenso de otros servidores. Desafortunadamente, expresar los tiempos de espera en términos de duración también tiene sus problemas. Primero, el temporizador que use debe ser monotónico y no retroceder cuando el servidor se sincronice con el Protocolo de tiempo de redes (NTP). Un problema mucho más difícil es que a fin de medir una duración, el servidor debe saber cuándo comenzar un cronómetro. En ciertos escenarios de sobrecarga extremos, se pueden poner en cola enormes volúmenes de solicitudes en los búferes del Protocolo de control de transmisión (TCP), para que en el momento en que el servidor lea las solicitudes de sus búferes, el cliente ya haya agotado su tiempo de espera.

Cuando los sistemas de Amazon expresan sugerencias de tiempo de espera de los clientes, intentamos aplicarlas de manera transitiva. En lugares en los que la arquitectura orientada al servicio incluye múltiples saltos, propagamos el límite de "tiempo restante" entre cada salto, de manera que un servicio posterior al final de una cadena de llamadas pueda saber cuánto tiempo tiene para que la respuesta sea útil.

Una vez que un servidor conoce el tiempo límite del cliente, está la cuestión de dónde hacer cumplir ese límite en la implementación del servicio. Si un servicio tiene una cola de solicitudes, usamos esa oportunidad para evaluar el tiempo de espera después que quitar cada solicitud de la cola. Pero esto sigue siendo demasiado complicado, ya que no sabemos cuánto tiempo demorará la solicitud. Algunos sistemas llevan un cálculo de cuánto tiempo toman las solicitudes de API, y eliminan solicitudes en forma temprana si la fecha límite informada por el cliente excede un cálculo de latencia. Sin embargo, las cosas no suelen ser tan simples. Por ejemplo, los aciertos de caché son más rápidas que las pérdidas de caché, y el calculador no sabe si se trata de un acierto o de una pérdida. O puede que los recursos de backend del servicio estén partidos, y solo algunas partes sean lentas. Esta es una buena oportunidad para usar el ingenio, pero también es posible que el ingenio resulte en una situación impredecible.

Según nuestra experiencia, cumplir con los tiempos de espera del cliente en el servidor es mejor que la otra alternativa, a pesar de las complejidades y compensaciones. En lugar de apilar solicitudes y que el servidor trabaje en solicitudes que posiblemente ya no le interesan a nadie, notamos que es útil hacer cumplir un "tiempo de vida por solicitud" y eliminar las solicitudes destinadas a fallar.

Cómo terminar lo que empezamos

No queremos desperdiciar el trabajo, especialmente en una sobrecarga. La eliminación del trabajo crea un bucle de retroalimentación positiva que aumenta la sobrecarga, ya que los clientes suelen

volver a intentar con una solicitud si un servicio no responde a tiempo. Cuando esto ocurre, una solicitud que consume recursos se convierte en muchas solicitudes que consumen recursos, lo que multiplica la carga del servicio. Cuando los clientes agotan su tiempo de espera y vuelven a intentarlo, suelen dejar de esperar una respuesta en su primera conexión mientras realizan una nueva solicitud en otra conexión. Si el servidor termina la primera solicitud y responde, el cliente podría no estar escuchando, ya que ahora está esperando una respuesta de su nuevo intento de solicitud.

El problema del trabajo desperdiciado es porque intentamos diseñar servicios para realizar *trabajo delimitado*. En lugares en los que exponemos una API que puede regresar un gran conjunto de datos (o cualquier lista en realidad), la exponemos como una API que admite paginación. Estas API regresan resultados parciales y un token que el cliente puede usar para solicitar más datos. Notamos que es más fácil calcular la carga adicional de un servicio cuando el servidor maneja una solicitud con una limitación superior a la cantidad de memoria, CPU y ancho de banda. Es muy difícil realizar un control de admisión cuando un servidor no tiene idea de lo que tomará procesar una solicitud.

Una oportunidad más sutil de priorizar las solicitudes es la de cómo usar las API de un servicio. Por ejemplo, digamos que un servicio tiene dos API: **inicio()** y **final()**. A fin de terminar su trabajo, los clientes deben poder llamar ambas API. En este caso, el servicio debería priorizar las solicitudes de **final()** por encima de las de **inicio()**. Si se da prioridad a las de **inicio()**, los clientes no podrían terminar el trabajo que comenzaron, y como resultado habría caídas de tensión.

La paginación es otro lugar al que demos estar atentos en cuanto a trabajo desperdiciado. Si un cliente debe hacer varias solicitudes en forma secuencial para paginar los resultados de un servicio, y observa una falla luego de la página N-1 y elimina los resultados, está desperdiciando las llamadas de servicio N-2 y cualquier reintento realizado en el camino. Esto sugiere que al igual que las solicitudes de **final()**, las solicitudes de primera página deberían tener prioridad detrás de las solicitudes de paginación de páginas posteriores. También subraya por qué diseñamos servicios para que realicen trabajo delimitado y que no paginen de manera infinita en un servicio al que llaman durante una operación sincrónica.

Cuidado con las colas

También ayuda ver la duración de la solicitud cuando lidiamos con colas internas. Muchas arquitecturas de servicios modernos usan colas en memoria para conectar grupos de subprocesos a fin de procesar solicitudes durante varias etapas del trabajo. Es posible que un marco de servicio web con un ejecutor tenga una cola configurada en frente. Con cualquier servicio basado en TCP, el sistema operativo mantiene un búfer para cada socket, y esos búferes pueden contener un gran volumen de solicitudes de contenido.

Cuando quitamos trabajo de las colas, usamos esa oportunidad para evaluar cuánto tiempo el trabajo estuvo en la cola. Como mínimo, intentamos registrar esa duración en las métricas de nuestros servicios. Además de delimitar el tamaño de las colas, nos parece extremadamente importante colocar una delimitación superior en la cantidad de tiempo que permanece una solicitud entrante en una cola, y la desechamos si es demasiado antigua. Esto libera al servidor para trabajar en solicitudes más nuevas con mayores posibilidades de tener éxito. Como versión extrema de este enfoque, buscamos los modos de usar una cola en que el último en ingresar es el último en salir (LIFO), si el protocolo la admite. (la canalización HTTP/1.1 de solicitudes de una determinada conexión TCP no admite las colas LIFO, pero HTTP/2 por lo general sí lo hace).

Los balanceadores de carga podrían también hacer cola de solicitudes o conexiones cuando los servicios estén sobrecargados usando una función que se llama *cola en aumento*. Estas colas pueden

provocar caídas de tensión, ya que cuando un servidor finalmente obtiene una solicitud, no tiene idea de cuánto tiempo la solicitud estuvo en la cola. Un valor predeterminado generalmente seguro es usar una configuración indirecta, que falla rápidamente en lugar de poner en cola un exceso de solicitudes. En Amazon, este aprendizaje se integró a la próxima generación de servicio de Elastic Load Balancing (ELB). El balanceador de carga clásico usó una cola en aumento, pero el balanceador de carga de aplicaciones rechaza tráfico de exceso. Independientemente de la configuración, los equipos de Amazon controlan las métricas de balanceador de carga pertinentes, como el aumento de la profundidad de la cola o el recuento indirecto, para sus servicios.

En nuestra experiencia, la importancia de controlar las colas no se debe sobrestimar. A menudo me sorprende encontrar colas en la memoria donde no creí que las encontraría, en sistemas y en bibliotecas en las que confío. Cuando busco en los sistemas, me resulta útil suponer que hay colas en algún lugar que aún no conozco. Por supuesto, las pruebas de sobrecarga proporcionan más información útil que explorar un código, siempre que pueda obtener los casos de prueba realistas adecuados.

Protección contra la sobrecarga en las capas inferiores

Los servicios cuentan con varias capas(desde balanceadores de carga y sistemas operativos con capacidades *netfilter* e *iptables,* hasta marcos de servicio y código) y cada capa proporciona alguna capacidad para proteger el servicio.

Proxies HTTP, como NGINX, a menudo soportan una función de conexiones máximas (*max_conns*) para limitar el número de solicitudes activas o conexiones que pasarán al servidor back-end. Este puede ser un mecanismo útil, pero aprendimos a usarlo como último recurso en lugar de como opción de protección predeterminada. Con los proxies, es difícil establecer una prioridad en el tráfico importante, y el rastreo del conteo total de solicitudes en tránsito suele brindar información poco precisa sobre si un servicio está actualmente sobrecargado.

Al comienzo de este artículo, describí un desafío de mis tiempos en el equipo de Marcos de servicio. Intentábamos proporcionarles a los equipos de Amazon un valor predeterminado recomendado de las máximas conexiones para configurar en sus balanceadores de carga. Al final, sugerimos que los equipos establezcan las conexiones máximas para su balanceador de carga y alto de proxy, y permitimos que el servidor implemente algoritmos de desbordamiento de carga más precisos con información local. Sin embargo, también fue importante para el valor de conexiones máximas no exceder el número de subprocesos de escucha, procesos de escucha o descriptores de archivos en un servidor, de manera que el servidor tuviera recursos para lidiar con solicitudes críticas de verificación de estado del balanceador de carga.

Las funciones del sistema operativo para limitar el uso de recursos del servidor son potentes y puede ser útiles en caso de emergencia. Y dado que sabemos que las sobrecargas ocurren, nos aseguramos de prepararnos utilizando los runbooks adecuados con los comandos específicos listos. La utilidad de los *iptables* puede poner un límite en el número de conexiones que un servidor es capaz de aceptar; y puede rechazar las conexiones de exceso por un costo menor que cualquier proceso de servidor. También se puede configurar con controles más sofisticados, como permitiendo nuevas conexiones en un índice delimitado, o incluso permitiendo un índice de conexión limitado o un recuento por dirección de IP de origen. Los filtros del IP de origen son potentes, pero no se aplican a los balanceadores de carga tradicionales. Sin embargo, un balanceador de carga de red ELB preserva el IP de origen del intermediario incluso en la capa del sistema operativo a través de la virtualización de la red, lo que hace que las reglas de los *iptables,* como los filtros de IP de origen, trabajen como se esperan.

Protección en capas

En ciertos casos, un servidor se queda sin recursos incluso para rechazar solicitudes sin ralentizarse. Con esta realidad en mente, vemos todos los saltos entre un servidor y su cliente para verificar cómo pueden cooperar y ayudar a eliminar el exceso de carga. Por ejemplo, varios servicios de AWS incluyen opciones de eliminación de cargas de manera predeterminada. Cuando enfrentamos un servicio con Amazon API Gateway, podemos configurar un índice de solicitud máximo en cualquier API que acepte. Cuando nuestros servicios se enfrentan con API Gateway, un balanceador de carga de aplicación, o Amazon CloudFront, podemos configurar AWS WAF para desbordar el exceso de tráfico en una cantidad de dimensiones.

La visibilidad crea una tensión difícil. El rechazo temprano es importante ya que es el lugar más económico para eliminar el exceso de tráfico, pero es un costo para la visibilidad. Es por esto que protegemos en capas: para permitir que un servidor tome más de lo que puede trabajar y elimine el exceso, y registre suficiente información para que sepamos qué tráfico se elimina. Como hay una cierta cantidad de tráfico que un servidor puede omitir, confiamos en la capa que se encuentra por delante para protegerlo de volúmenes extremos de tráfico.

Piense diferente acerca de la sobrecarga

En este artículo, tratamos cómo la necesidad de evitar las cargas surge del hecho que los sistemas se vuelven más lentos al realizar más trabajo simultáneo, a medida que las fuerzas como los recursos se ven limitados y la conexión surte efecto. El ciclo de retroalimentación de sobrecarga es impulsado por la latencia, lo que, en definitiva, resulta en un trabajo perdido, amplificación de la tasa de solicitud e incluso más sobrecarga. Esta fuerza, impulsada por la Ley universal de escalabilidad y la Ley de Amdahl, es importante que se evite **desbordando cargas de exceso y manteniendo un rendimiento constante y predecible a pesar de la sobrecarga.** Enfocarse en un rendimiento predecible y consistente es un principio de diseño clave sobre el que se construyen los servicios en Amazon.

Por ejemplo, Amazon DynamoDB es un servicio de base de datos que ofrece un rendimiento predecible y disponibilidad a escala. Incluso si una carga de trabajo incrementa rápidamente y supera los recursos aprovisionados, DynamoDB mantiene una latencia de rendimiento predecible para esa carga de trabajo. Los factores como DynamoDB Auto Scaling, la capacidad de adaptación y bajo demanda reaccionan rápidamente para aumentar las buenas tasas de trabajo para adaptarse al aumento en la carga de trabajo. Durante ese tiempo, el buen rendimiento se mantiene estable, manteniendo un servicio en las capas por sobre DynamoDB con un rendimiento predecible y mejorando la estabilidad de todo el sistema.

AWS Lambda ofrece un ejemplo incluso más amplio del enfoque sobre el rendimiento predecible. Cuando usamos Lambda para implementar un servicio, cada llamada de API se ejecuta en su propio entorno con una cantidad consistente de recursos informáticos asignados, y este entorno de ejecución funciona solo en una solicitud a la vez. Esto difiere de un paradigma basado en servidor, donde un servidor determinado funciona en múltiples API.

Aislar cada llamada API en sus propios recursos independientes (informática, memoria, disco, red) eludirá la ley de Amdahl de alguna manera, porque los recursos de una llamada API no competirán con los recursos de otra llamada API. Por lo tanto, si el rendimiento excede el rendimiento bueno, este permanecerá invariable en lugar de desplegarse como lo hace en un entorno basado en un servidor más tradicional. Esto no es la panacea, ya que las dependencias pueden ralentizarse y

provocar que las simultaneidades se incrementen. Sin embargo, en este escenario, al menos los tipos de conexiones de recursos en host que tratamos en este artículo no aplican.

Este aislamiento de recursos es un beneficio algo delicado pero importante de los entornos informáticos modernos sin servidor como AWS Fargate, Amazon Elastic Container Service (Amazon ECS) y AWS Lambda. En Amazon descubrimos que lleva mucho tiempo implementar el desbordamiento de carga, desde ajustar los grupos de subprocesos hasta elegir la configuración perfecta para las conexiones de balanceador de carga máxima. Los valores predeterminados sensibles para estos tipos de configuraciones son difíciles o imposibles de encontrar porque dependen de las características operativas únicas de cada sistema. Estos nuevos entornos informáticos sin servidor proporcionan aislamiento de recursos de nivel inferior y exponen botones de nivel superior, como controles de aceleración y concurrencia, para proteger contra la sobrecarga. De cierto modo, en lugar de perseguir el valor de configuración predeterminado perfecto, podemos aislar dicha configuración por completo y proteger por categorías de sobrecarga sin la necesidad de ningún tipo de configuración.

Más información

- Ley de escalabilidad universal
- Ley de Amdahl
- Arquitecturas basadas en eventos en secuencia (SEDA)
- Ley de Little (describe la concurrencia en un sistema y cómo determinar la capacidad de los sistemas distribuidos)
- Contando historias sobre la Ley de Little, *Marc's Blog*
- Información detallada de Elastic Load Balancing y prácticas recomendadas, presentación en re:Invent 2016 (describe la evolución de Elastic Load Balancing para detener el exceso de solicitudes en cola)
- Burgess, Pensar en las promesas: Diseño de sistemas para la cooperación, O'Reilly Media, 2015