Menggunakan pelepasan muatan untuk mencegah kelebihan muatan

David Yanacek

Saya bekerja di tim Service Frameworks di Amazon selama beberapa tahun. Tim kami menulis alat yang membantu pemilik layanan AWS, seperti Amazon Route 53 dan Elastic Load Balancing mengembangkan layanan mereka dengan lebih cepat, dan melayani klien yang memanggil layanan tersebut dengan lebih mudah. Tim Amazon lainnya memberi pemilik layanan fungsionalitas seperti pengukuran, autentikasi, pemantauan, pembuatan pustaka klien, dan pembuatan dokumentasi. Sebagai ganti mengharuskan setiap anggota tim mengintegrasikan berbagai fitur tersebut ke dalam layanan mereka secara manual, tim Service Framework melakukan integrasi tersebut satu kali dan mengekspos fungsionalitas ke setiap layanan melalui konfigurasi.

Satu tantangan yang kami hadapi adalah menentukan cara untuk memberikan default yang masuk akal, terutama untuk fitur yang terkait dengan kinerja atau ketersediaan. Misalnya, kami tidak dapat mengatur batas waktu sisi klien dengan mudah, karena kerangka kerja kami tidak mengetahui kemungkinan karakteristik latensi panggilan API. Tidak akan lebih mudah bagi pemilik klien atau pemilik untuk mencari tahu sendiri, jadi kami terus mencoba, dan mendapatkan sejumlah wawasan berguna dalam prosesnya.

Satu pertanyaan umum yang kami coba jawab adalah menentukan jumlah koneksi default yang diizinkan dibuka oleh server untuk klien pada saat yang bersamaan. Pengaturan ini dirancang untuk mencegah server melakukan terlalu banyak pekerjaan dan mengalami kelebihan muatan. Lebih spesifiknya lagi, kami ingin mengonfigurasikan pengaturan koneksi maksimum untuk server yang seimbang dengan koneksi maksimum untuk penyeimbang muatan. Ini sebelum adanya Elastic Load Balancing, jadi penyeimbang muatan perangkat keras masih digunakan secara luas.

Kami bertekad ingin membantu pemilik layanan dan klien layanan Amazon mencari tahu nilai ideal untuk koneksi maksimum yang akan diatur pada penyeimbang muatan, dan nilai yang sesuai untuk diatur dalam kerangka kerja yang kami sediakan. Kami memutuskan bahwa jika kami dapat mencari tahu cara menggunakan penilaian manusia untuk mengambil keputusan, kami dapat menulis perangkat lunak untuk mengemulasi penilaian tersebut.

Menentukan nilai yang ideal ternyata sangat menantang. Jika koneksi maksimum diatur terlalu rendah, penyeimbang muatan dapat memangkas peningkatan dalam jumlah permintaan, bahkan saat layanan memiliki kapasitas dalam jumlah besar. Jika koneksi maksimum diatur terlalu tinggi, server akan menjadi lamban dan tidak responsif. Jika koneksi maksimum diatur dengan tepat untuk beban kerja, beban kerja akan berubah atau kinerja dependensi akan berubah. Kemudian nilai akan kembali salah, yang menyebabkan pemadaman atau kelebihan muatan yang tidak perlu.

Pada akhirnya, kami menemukan bahwa konsep koneksi maksimum terlalu kurang akurat untuk memberikan jawaban lengkap bagi puzzle. Dalam artikel ini, kami akan menjelaskan pendekatan lainnya, seperti pelepasan muatan yang menurut kami bekerja dengan baik.

Anatomi kelebihan muatan

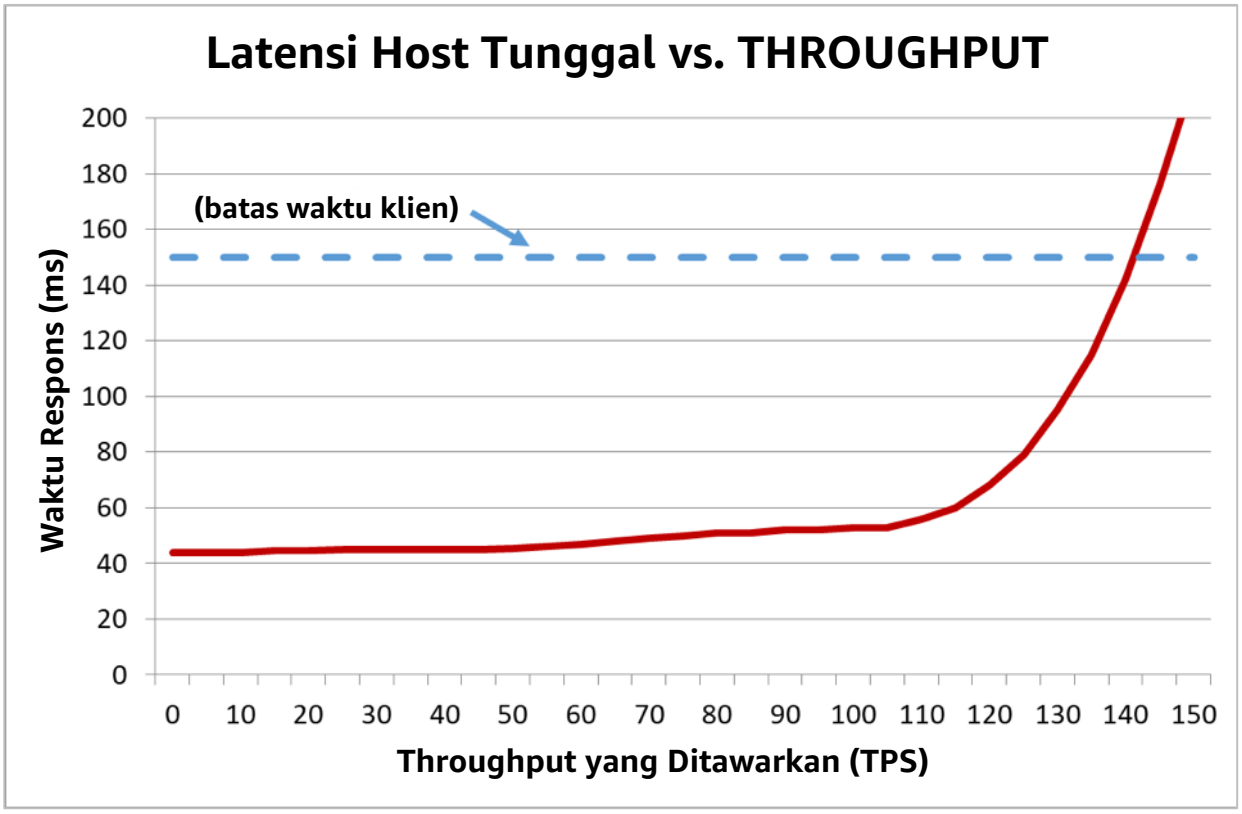
Di Amazon, kami menghindari kelebihan muatan dengan merancang sistem kami untuk menskalakan secara proaktif, sebelum menghadapi situasi kelebihan muatan. Meski demikian, melindungi sistem membutuhkan beberapa lapisan perlindungan. Perlindungan ini diawali dengan auto scaling, tetapi juga meliputi mekanisme untuk melepaskan kelebihan muatan dengan mulus, kemampuan untuk memantau mekanisme tersebut, dan yang terpenting, pengujian berkelanjutan.

Saat kami melakukan uji muatan pada layanan kami, kami menemukan bahwa latensi server pada penggunaan rendah lebih rendah daripada latensi server pada penggunaan tinggi. Di bawah muatan berat, perbedaan thread, peralihan konteks, pengumpulan sampah, dan I/O perbedaan menjadi lebih terlihat jelas. Pada akhirnya, layanan akan mencapai titik infleksi dengan kinerja mereka menurun jauh lebih cepat.

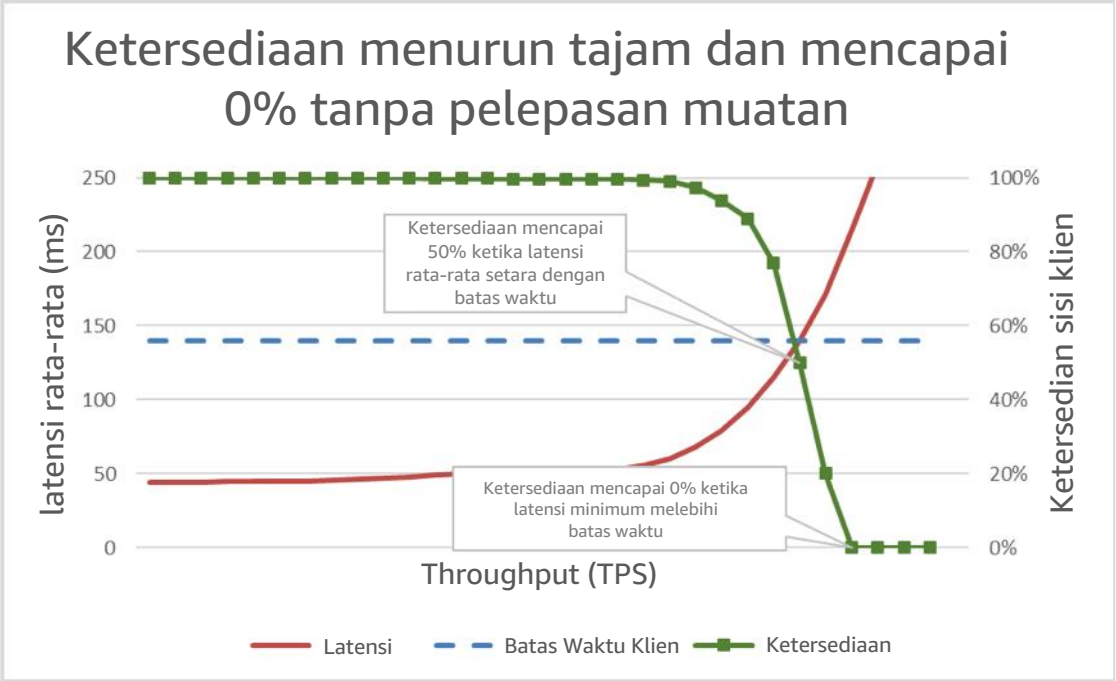
Teori di balik pengamatan ini dikenal sebagai Universal Scalability Law, yang merupakan turunan dari Amdahl's Law. Teori ini menetapkan bahwa meskipun throughput sistem dapat ditingkatkan dengan menggunakan paralelisasi, pada akhirnya sistem akan dibatasi oleh throughput titik serialisasi (yakni oleh tugas yang tidak dapat diparalelisasi).

Sayangnya, throughput tidak hanya dibatasi oleh sumber daya sistem, tetapi secara umum juga mengalami penurunan saat sistem mengalami kelebihan muatan. Saat sistem diberi lebih banyak pekerjaan daripada yang dapat didukung oleh sumber dayanya, sistem akan menjadi lamban. Komputer melakukan pekerjaan bahkan saat mengalami kelebihan muatan, tetapi komputer menghabiskan waktu lebih lama untuk melakukan peralihan konteks dan menjadi terlalu lamban dalam melakukan pekerjaannya.

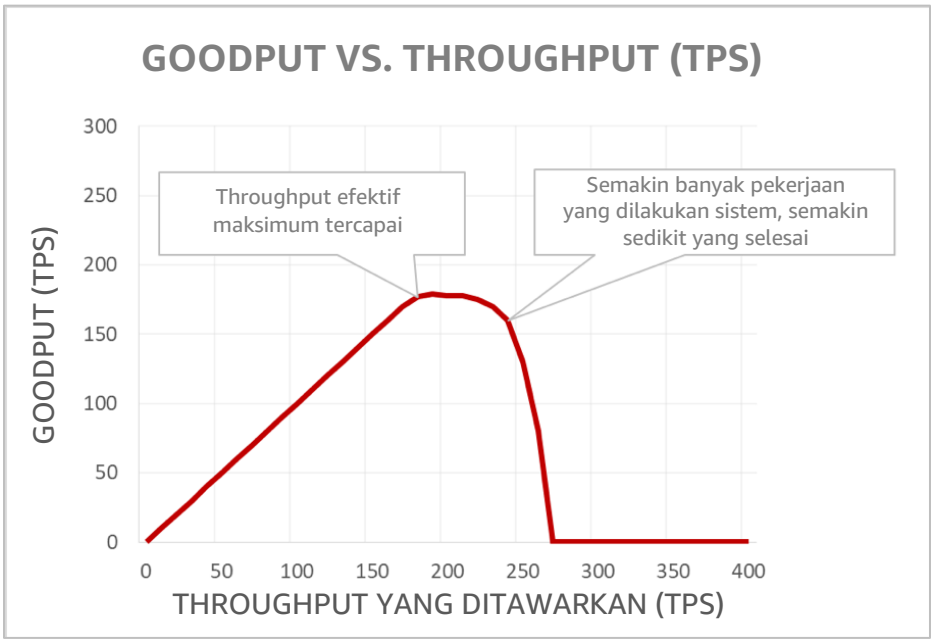
Dalam sebuah sistem terdistribusi tempat klien bicara dengan server, klien umumnya menjadi tidak sabar dan berhenti menunggu server merespons setelah beberapa waktu. Durasi ini dikenal sebagai *batas waktu*. Saat server menjadi terlalu kelebihan muatan hingga latensinya melampuai batas waktu klien, permintaan akan mulai mengalami kegagalan. Grafik berikut akan menunjukkan bagaimana waktu respons server meningkat seiring meningkatnya throughput yang ditawarkan (dalam transaksi per detik), dan waktu respons pada akhirnya akan mencapai titik infleksi dan banyak hal memburuk dengan cepat.



Dalam grafik sebelumnya, saat waktu respons melampaui batas waktu klien, jelas ini buruk, tetapi grafik tidak menunjukkan seberapa buruknya hal tersebut. Untuk mengilustrasikannya, kami dapat meletakkan ketersediaan yang dirasakan klien di samping latensi. Sebagai ganti menggunakan pengukuran waktu respons generik, kami dapat beralih ke menggunakan waktu respons rata-rata. Waktu respons rata-rata berarti bahwa 50 persen permintaan lebih cepat daripada nilai rata-rata. Jika latensi rata-rata layanan setara dengan batas waktu klien, setengah dari permintaan melebihi batas waktu, jadi ketersediaannya adalah 50 persen. Di sinilah kenaikan latensi mentransformasi masalah latensi menjadi masalah ketersediaan. Berikut grafik mengenai apa yang sedang terjadi:



Sayangnya, grafik ini sedikit sulit untuk dibaca. Cara yang lebih sederhana untuk menjelaskan masalah ketersediaan adalah dengan membedakan *goodput* dari *throughput*. Throughput adalah total jumlah permintaan per detik yang dikirim ke server. Goodput merupakan subset throughput yang ditangani tanpa kesalahan dan dengan latensi yang cukup rendah untuk klien dapat menggunakan respons.



Loop umpan balik positif

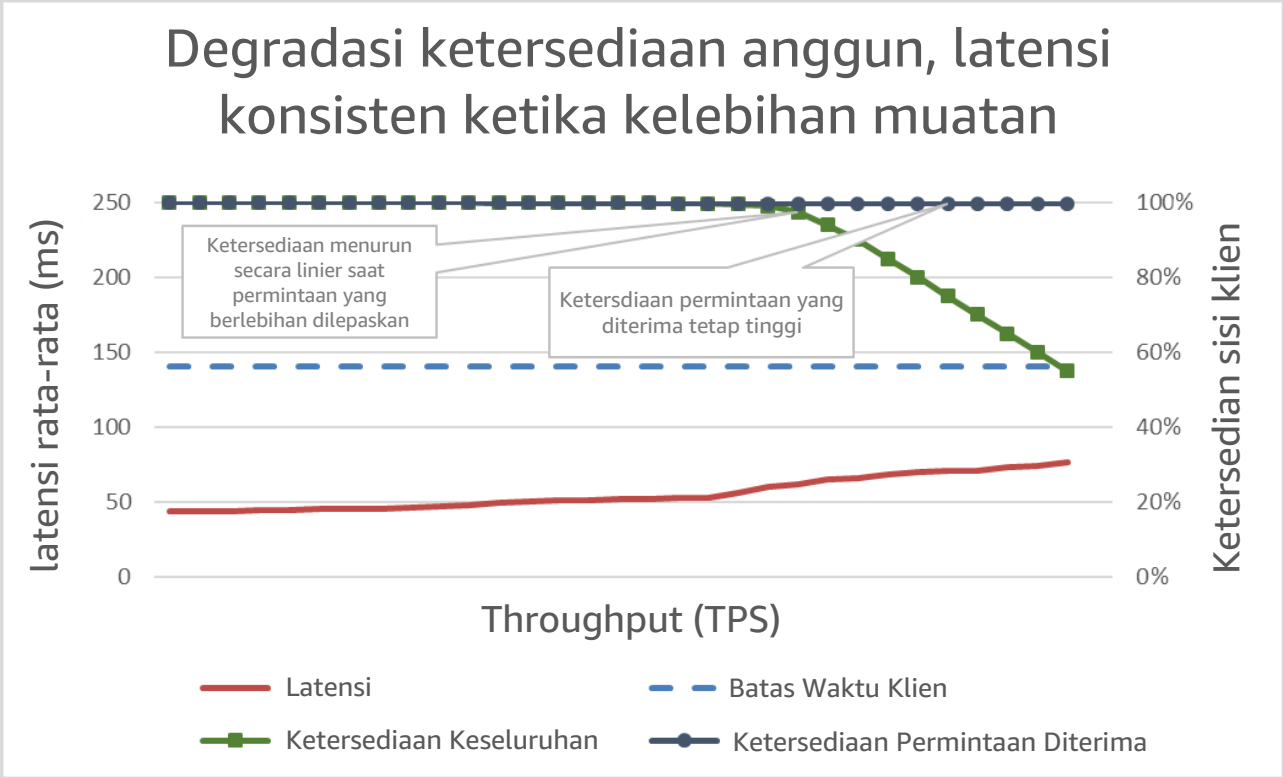
Bagian yang berbahaya dari situasi kelebihan muatan adalah bagaimana situasi ini semakin jelas terlihat dalam loop umpan balik. Saat klien mencapai batas waktu, sudah cukup buruk klien mendapatkan kesalahan. Yang lebih buruknya lagi adalah semua kemajuan yang telah dibuat oleh server sejauh ini pada permintaan tersebut akan menjadi sia-sia. Hal terakhir yang sebaiknya dilakukan oleh sistem dalam situasi kelebihan muatan, dengan kapasitas yang dibatasi, adalah menyia-nyiakan pekerjaan.

Yang membuat situasi semakin buruk adalah klien sering mencoba kembali permintaan mereka. Ini menggandakan muatan yang ditawarkan pada sistem. Jika terdapat grafik panggilan yang cukup dalam di arsitektur berorientasi layanan (yakni klien memanggil layanan, yang memanggil layanan lainnya, yang memanggil layanan lainnya), dan jika setiap lapisan menjalankan beberapa kali percobaan ulang, kelebihan muatan di lapisan bagian bawah akan menyebabkan percobaan ulang bertingkat yang memperkuat muatan yang ditawarkan secara eksponensial.

Saat berbagai faktor tersebut digabungkan, kelebihan muatan akan membuat loop umpan baliknya sendiri, yang menyebabkan kelebihan muatan sebagai status yang tetap.

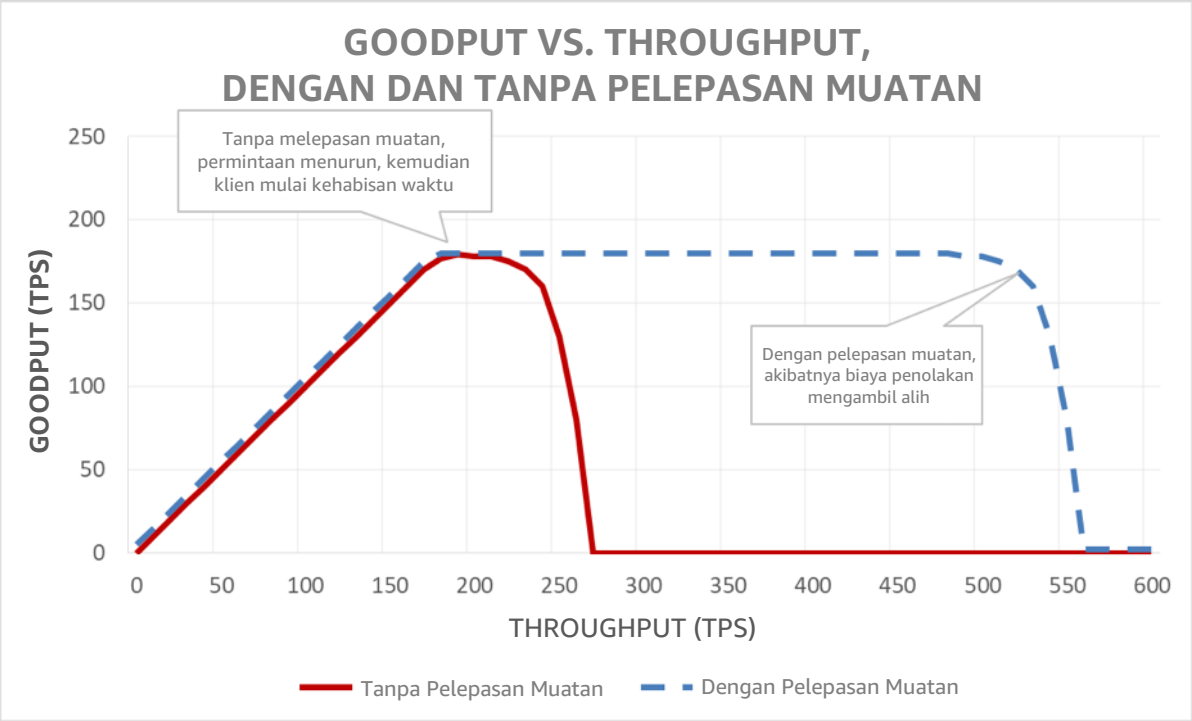
Mencegah pekerjaan menjadi sia-sia

Pelepasan muatan sekilas terlihat sederhana. Saat sebuah server sudah hampir mengalami kelebihan muatan, server akan mulai menolak kelebihan permintaan agar dapat berfokus pada permintaan yang diputuskannya untuk diterima. Tujuan pelepasan muatan adalah menjaga latensi tetap rendah untuk permintaan yang diputuskan oleh server untuk diterima agar permintaan tersebut membalas sebelum klien mencapai batas waktu. Dengan pendekatan ini, server akan mempertahankan ketersediaaan tinggi untuk permintaan yang diterimanya, dan hanya ketersediaan lalu lintas berlebih yang terpengaruh.



Memastikan latensi selalu diperiksa dengan melepaskan muatan berlebih akan meningkatkan ketersediaan sistem. Tetapi manfaat dari pendekatan ini sulit untuk divisualisasikan dalam grafik sebelumnya. Baris ketersediaan keseluruhan bergerak turun secara perlahan, yang terlihat buruk. Kuncinya adalah bahwa permintaan yang diputuskan oleh server untuk diterima tetap tersedia karena permintaan tersebut dilayani dengan cepat.

Pelepasan muatan memungkinkan server mempertahankan goodput-nya dan menyelesaikan sebanyak mungkin permintaan, bahkan saat throughput yang ditawarkan meningkat. Meski demikian, tindakan melepaskan muatan bukannya tanpa risiko, sehingga pada akhirnya server akan terpengaruh dengan Amdahl's Law dan penurunan goodput.



Pengujian

Saat saya bicara dengan teknisi lainnya mengenai pelepasan muatan, saya ingin menekankan bahwa jika mereka belum melakukan uji muatan pada layanan mereka hingga ke titik layanan meluap, dan jauh melampaui titik layanan meluap, mereka sebaiknya berasumsi bahwa layanan akan mengalami kegagalan dalam kemungkinan cara yang lebih tidak menyenangkan. Di Amazon, kami menghabiskan banyak waktu melakukan pengujian muatan pada layanan kami. Menghasilkan grafik seperti grafik sebelumnya dalam artikel ini membantu kami membuat dasar kinerja kelebihan muatan dan melacak kinerja kami dari waktu ke waktu saat kami membuat perubahan pada layanan.

Terdapat beberapa jenis uji muatan. Beberapa uji muatan memastikan armada secara otomatis diskalakan saat muatan mengalami peningkatan, sedangkan beberapa pengujian lainnya menggunakan ukuran armada tetap. Jika, dalam uji muatan, ketersediaan layanan menurun dengan cepat ke nol seiring peningkatan throughput, ini adalah pertanda baik bahwa layanan membutuhkan mekanisme pelepasan muatan tambahan. Hasil uji muatan yang ideal adalah goodput memasuki fase stabil saat layanan hampir digunakan sepenuhnya, dan tidak berubah bahkan saat throughput diterapkan.

Alat seperti Chaos Monkey membantu menjalankan uji rekayasa chaos pada layanan. Sebagai contoh, alat tersebut dapat menyalurkan terlalu banyak muatan pada CPU atau memperkenalkan paket hilang untuk menyimulasikan kondisi yang terjadi selama kelebihan muatan. Teknik pengujian lainnya yang kami gunakan adalah melakukan uji pembuatan muatan atau canary, mendorong muatan berkelanjutan (bukan muatan yang terus bertambah) ke lingkungan uji, tetapi mulai menghapus server dari lingkungan uji. Ini akan meningkatkan throughput yang ditawarkan per instans, sehingga dapat menguji throughput instans. Teknik meningkatkan muatan secara artifisial dengan mengurangi ukuran armada berguna untuk menguji layanan dalam isolasi, tetapi bukan pengganti yang lengkap untuk uji muatan lengkap. Uji muatan lengkap dan menyeluruh juga akan meningkatkan muatan ke dependensi layanan tersebut, yang akan membuka kemacetan lainnya.

Selama pengujian, kami memastikan untuk mengukur ketersediaan dan latensi yang dirasakan klien selain ketersediaan dan latensi sisi server. Saat ketersediaan sisi klien mulai berkurang, kami mendorong muatan jauh melampaui titik tersebut. Jika pelepasan muatan bekerja, goodput akan tetap stabil bahkan saat throughput yang ditawarkan meningkat jauh melampaui kemampuan layanan yang diskalakan.

Pengujian kelebihan muatan sangat penting sebelum menjelajahi mekanisme untuk menghindari kelebihan muatan. Setiap mekanisme membawa kompleksitas. Sebagai contoh, pertimbangkan semua opsi konfigurasi dalam kerangka kerja layanan yang saya sebutkan di awal artikel, dan seberapa sulitnya mendapatkan default yang tepat. Setiap mekanisme untuk menghindari kelebihan muatan akan menambahkan perlindungan berbeda dan memiliki efektivitas yang terbatas. Melalui pengujian, sebuah tim dapat mendeteksi kemacetan pada sistem mereka dan menentukan kombinasi perlindungan yang mereka butuhkan untuk menangani kelebihan muatan.

Visibilitas

Di Amazon, apa pun teknik yang kami gunakan untuk melindungi layanan kami dari kelebihan muatan, kami memikirkan dengan matang metrik dan visibilitas yang kami butuhkan saat penanganan kelebihan muatan ini berlaku.

Jika perlindungan pembatasan daya menolak permintaan, penolakan tersebut akan mengurangi ketersediaan layanan. Saat layanan membuat kesalahan dan menolak permintaan meskipun memiliki kapasitas (misalnya, jika jumlah koneksi maksimum diatur terlalu rendah), layanan akan menghasilkan positif palsu. Kami berusaha menjaga tingkat positif palsu di angka nol. Jika suatu tim menemukan bahwa tingkat positif palsu mereka rutin berada di angka selain nol, layanan disetel terlalu sensitif, atau host individu secara terus-menerus dan meyakinkan mengalami kelebihan muatan, dan mungkin terjadi masalah penskalaan atau penyeimbang muatan. Dalam kasus semacam ini, kami mungkin harus melakukan penyetelan pada kinerja aplikasi, atau kami mungkin beralih ke jenis instans yang lebih besar yang dapat menangani ketidakseimbangan muatan dengan lebih mulus.

Dalam hal visibilitas, saat pelepasan muatan menolak permintaan, kami memastikan kami memiliki instrumen yang tepat untuk mengetahui siapa kliennya, operasi apa yang mereka panggil, dan informasi lainnya yang akan membantu kami menyetel tindakan perlindungan lainnya. Kami juga menggunakan alarm untuk mendeteksi apakan tindakan penanggulangan menolak lalu lintas dalam volume yang signifikan. Jika terjadi pembatasan daya, prioritas kami adalah menambahkan kapasitas dan mengatasi kemacetan.

Ada pertimbangan lainnya yang subtil namun penting yang terkait dengan visibilitas dalam pelepasan muatan. Kami menemukan bahwa penting untuk tidak mengotori metrik latensi layanan kami dengan latensi permintaan gagal. Lagipula, latensi pelepasan muatan permintaan seharusnya sangat rendah jika dibandingkan dengan permintaan lainnya. Sebagai contoh, jika sebuah layanan melepaskan muatan 60 persen dari lalu lintasnya, latensi rata-rata layanan mungkin terlihat cukup luar biasa bahkan jika latensi permintaan berhasilnya tidak mengesankan, karena dilaporkan sebagai hasil permintaan laporan cepat.

Efek pemuatan otomatis pada kegagalan auto scaling dan Availability Zone

Jika keliru dikonfigurasi, pelepasan muatan dapat menonaktifkan auto scaling yang diaktifkan kembali. Pertimbangkan contoh berikut: sebuah layanan dikonfigurasikan untuk penskalaan yang diaktifkan kembali berbasis CPU dan mengonfigurasikan pelepasan muatan untuk menolak permintaan pada target CPU yang serupa. Dalam kasus ini, sistem pelepasan muatan akan

mengurangi jumlah permintaan untuk memastikan muatan CPU tetap rendah, dan penskalaan yang diaktifkan kembali tidak akan pernah menerima atau mendapatkan penundaan sinyal untuk meluncurkan instans baru.

Kami juga mempertimbangkan logika pelepasan muatan secara hati-hati saat kami mengatur batas auto scaling untuk menangani kegagalan Availability Zone. Layanan diskalakan ke titik yang melihat kelayakan Availability Zone atas kapasitas mereka dapat menjadi tidak tersedia saat mempertahankan sasaran latensi kita. Tim Amazon sering melihat metrik sistem, seperti CPU untuk memperkirakan seberapa dekatnya sebuah layanan mencapai batas kapasitasnya. Meski demikian, dengan pelepasan muatan, sebuah armada mungkin berjalan jauh lebih dekat dengan titik saat permintaan akan ditolak daripada yang diindikasikan oleh metrik sistem, dan mungkin tidak memiliki kelebihan kapasitas yang disediakan untuk menangani kegagalan Availability Zone. Dengan pelepasan muatan, kami harus ekstra yakin saat menguji layanan kami hingga rusak untuk memahami kapasitas dan jarak armada setiap saat.

Faktanya, kami dapat menggunakan pelepasan muatan untuk menghemat biaya dengan membentuk lalu lintas di luar jam sibuk dan tidak bersifat kritis. Sebagai contoh, jika sebuah armada menangani lalu lintas situs web untuk amazon.com, armada tersebut dapat memutuskan apakah lalu lintas crawler pencarian sepadan dengan biaya diskalakan untuk redundansi Availability Zone. Meski demikian, kami sangat berhati-hati dengan pendekatan ini. Tidak semua permintaan memiliki biaya yang sama, dan mengingat bahwa sebuah layanan harus memberikan redundansi Availability Zone untuk lalu lintas manusia serta untuk melepaskan kelebihan lalu lintas crawler pada saat yang bersamaan memerlukan rancangan, pengujian berkelanjutan, dan pembelian dari bisnis. Jika layanan klien tidak mengetahui apakah layanan dikonfigurasikan seperti ini, perilakunya selama kegagalan Availability Zone dapat terlihat seperti penurunan ketersediaan kritis besar, bukan pelepasan muatan yang tidak bersifat kritis. Karena alasan ini, dalam arsitektur berorientasi layanan, kami mencoba untuk mendorong pembentukan semacam ini sesegera mungkin (misalnya dalam layanan yang menerima permintaan awal dari klien) sebagai ganti mencoba membuat keputusan prioritisasi global di seluruh tumpukan.

Mekanisme pelepasan muatan

Saat mendiskusikan pelepasan muatan dan skenario yang tidak dapat diprediksi, juga penting untuk berfokus pada banyak kondisi *yang dapat diprediksi* yang menyebabkan pembatasan daya. Di Amazon, layanan mempertahankan kelebihan kapasitas yang cukup untuk menangani kegagalan Availability Zone tanpa perlu menambahkan kapasitas lainnya. Layanan menggunakan pembatasan untuk memastikan kesetaraan di antara klien.

Meski demikian, terlepas dari praktik perlindungan dan operasional ini, sebuah layanan memiliki sejumlah kapasitas pada setiap waktu, dan karenanya dapat mengalami kelebihan muatan karena berbagai alasan. Alasan-alasan tersebut meliputi lonjakan tidak terduga dalam lalu lintas, kapasitas armada yang hilang tiba-tiba (akibat penerapan yang buruk atau semacamnya), klien berubah dari membuat permintaan murah (seperti hasil pembacaan yang di-cache) ke permintaan mahal (seperti kegagalan atau hasil penulisan cache). Saat sebuah layanan mengalami kelebihan muatan, layanan tersebut harus menyelesaikan permintaan yang telah diambilnya; yang artinya, layanan harus melindungi dirinya dari pembatasan daya. Dalam sisa bagian ini, kita akan mendiskusikan sejumlah pertimbangan dan teknik yang telah kami gunakan selama bertahun-tahun untuk mengatasi kelebihan muatan.

Memahami biaya membuat permintaan

Kami memastikan kami melakukan uji muatan pada layanan kami *jauh* melampaui titik terjadinya plateau goodput. Salah satu alasan utama untuk pendekatan ini adalah untuk memastikan bahwa saat kami membuat permintaan selama pelepasan muatan, biaya membuat permintaannya sekecil mungkin. Kami telah melihat bahwa melewatkan pernyataan log atau pengaturan soket tidak disengaja mudah terjadi, yang dapat menyebabkan pembuatan permintaan jauh lebih mahal daripada yang seharusnya.

Dalam kasus yang jarang terjadi, membuat permintaan dengan cepat dapat lebih mahal daripada bertahan pada permintaan tersebut. Dalam kasus ini, kami memperlambat permintaan yang ditolak untuk disesuaikan (paling tidak) dengan latensi respons yang berhasil. Meski demikian, penting untuk melakukannya saat biaya mempertahankan permintaan semurah mungkin; misalnya, saat tidak mengikat thread aplikasi.

Memprioritaskan permintaan

Saat sebuah server mangalami kelebihan muatan, server memiliki kesempatan untuk mengatasi permintaan masuk untuk memutuskan permintaan mana yang harus diterima dan mana yang harus diabaikan. Permintaan yang paling penting yang akan diterima server adalah permintaan ping dari penyeimbang muatan. Jika server tidak merespons permintaan ping tepat waktu, penyeimbang muatan akan berhenti mengirim permintaan baru ke server tersebut selama jangka waktu tertentu, dan server akan menjadi idle. Dalam skenario pembatasan daya, hal terakhir yang ingin kami lakukan adalah mengurangi ukuran armada. Di luar permintaan ping, opsi prioritisasi permintaan dapat berbeda-beda dari layanan ke layanan.

Pertimbangkan layanan web yang menyediakan data untuk merender amazon.com. Panggilan layanan yang mendukung rendering halaman web untuk crawler indeks pencarian cenderung kurang kritis untuk melayani daripada permintaan yang berasal dari manusia. Permintaan crawler penting untuk dilayani, tetapi idealnya dapat diubah ke luar jam sibuk. Meski demikian, dalam lingkungan yang kompleks seperti amazon.com, tempat banyak layanan bekerja sama, jika layanan menggunakan heuristik prioritisasi yang bertentangan, ketersediaan di seluruh sistem dapat terpengaruh dan pekerjaan dapat menjadi sia-sia.

Prioritisasi dan throttling dapat digunakan bersamaan untuk menghindari batasan throttling yang terlalu tegas sambil tetap melindungi layanan dari kelebihan muatan. Di Amazon, dalam kasus saat kami mengizinkan klien melonjak melebihi batas throttle yang mereka konfigurasikan, kelebihan permintaan dari klien tersebut dapat diprioritaskan lebih rendah daripada permintaan dalam kuota dari klien lain. Kami menghabiskan banyak waktu berfokus pada algoritme penempatan untuk meminimalkan kemungkinkan lonjakan kapasitas menjadi tidak tersedia, tetapi mengingat pertukaran, kami lebih memilih beban kerja yang disediakan dan dapat diprediksi daripada beban kerja yang tidak dapat diprediksi.

Mewaspadai jam

Jika layanan sudah setengah jalan untuk melayani permintaan dan menyadari bahwa klien telah mencapai batas waktu, layanan dapat melewati melakukan sisa pekerjaan dan memutuskan permintaan gagal pada titik tersebut. Atau, server akan terus mengerjakan permintaan, dan balasan terlambatnya seperti pohon yang tumbang di hutan. Dari perspektif server, layanan telah menghasilkan respons yang sukses. Tetapi dari perspektif klien telah mencapai batas waktu, ini adalah kesalahan.

Satu cara untuk menghindari pekerjaan yang sia-sia ini adalah dengan klien menyertakan petunjuk batas waktu dalam setiap permintaan, yang memberi tahu server berapa lama mereka bersedia untuk menunggu. Server dapat mengevaluasi petunjuk ini dan menjatuhkan permintaan tidak berguna dengan sedikit biaya.

Petunjuk batas waktu dapat dinyatakan sebagai waktu absolut atau sebagai durasi. Sayangnya, server dalam sistem terdistribusi sangat buruk dalam menyetujui kapan waktu saat ini yang pasti. Amazon Time Sync Service mengompensasi dengn menyinkronkan jam instans Amazon Elastic Compute Cloud (Amazon EC2) Anda dengan armada jam atomik dan berkontrol satelit di setiap Wilayah AWS. Jam yang disinkronkan dengan baik juga sangat penting di Amazon untuk tujuan pencatatan. Membandingkan dua file log pada server yang memiliki jam yang tidak disinkronkan membuat pemecahan masalah menjadi lebih sulit daripada sebenarnya.

Cara lain untuk "mengawasi jam" adalah dengan mengukur durasi pada satu mesin. Server mengukur durasi yang telah berlalu secara lokal dengan baik karena tidak perlu mendapatkan konsensus dengan server lain. Sayangnya, menyatakan batas waktu dalam bentuk durasi memiliki masalahnya sendiri. Sebagai contoh, Anda harus monoton dan tidak berbalik arah saat server disinkronkan dengan Network Time Protocol (NTP). Masalah yang jauh lebih sulit adalah bahwa untuk mengukur durasi, server harus mengetahui kapan harus memulai stopwatch. Dalam sejumlah skenario kelebihan muatan yang ekstrem, volume permintaan yang besar dapat menyebabkan antrean pada buffer Transmission Control Protocol (TCP), sehingga pada saat server membaca permintaan dari buffernya, klien telah mencapai batas waktu.

Kapan pun sistem di Amazon menyatakan petunjuk batas waktu klien, kami mencoba untuk menerapkannya secara transitif. Di tempat-tempat arsitektur berorientasi layanan meliputi beberapa lompatan, kami menyebarkan batas akhir "waktu yang tersisa" antara setiap lompatan sehingga layanan downstream pada akhir rantai panggilan dapat menyadari berapa lama waktu yang dimilikinya agar respons dapat berguna.

Setelah server mengetahui batas akhir kliennya, muncul pertanyaan di mana harus memberlakukan batas akhir dalam implementasi layanan. Jika layanan memiliki antrean permintaan, kami menggunakan peluang tersebut untuk mengevaluasi batas waktu setelah menghapus antrean setiap perangkat. Tetapi hal ini masih sedikit rumit, karena kami tidak tahu berapa lama waktu yang dibutuhkan oleh permintaan. Beberapa sistem menyimpan perkiraan berapa lama waktu yang dibutuhkan oleh permintaan API, dan mereka meluncurkan permintaan lebih awal jika batas akhir yang dilaporkan oleh klien melampaui perkiraan latensi. Meski demikian, masalah sering tidak sesederhana itu. Misalnya, keberhasilan cache lebih cepat daripada kegagalan cache, dan pembuat perkiraan tidak mengetahui sebelumnya apakah cache berhasil atau gagal. Atau, sumber daya backend layanan mungkin dipartisi, dan mungkin hanya beberapa partisi yang lamban. Kepandaian memberi banyak peluang, tetapi kepandaian tersebut juga dapat menjadi bumerang dalam situasi yang tidak dapat diprediksi.

Dalam pengalaman kami, memberlakukan batas waktu klien pada server tetap lebih baik daripada alternatifnya, terlepas dari kompleksitas dan pertukarannya. Agar permintaan tidak menumpuk dan server kemungkinan bekerja pada permintaan yang tidak penting lagi bagi setiap orang, kami menemukan bahwa memberlakukan "per-request time-to-live" dan membuang permintaan yang tidak berguna akan membantu.

Menyelesaikan apa yang sudah dimulai

Kami tidak ingin pekerjaan yang berguna menjadi sia-sia, terutama dalam kelebihan muatan. Membuang pekerjaan akan menciptakan loop umpan balik positif, yang akan meningkatkan kelebihan muatan, karena klien sering mencoba kembali permintaan jika layanan tidak merespons tepat waktu. Saat hal itu terjadi, satu permintaan yang memakan sumber daya akan berubah menjadi banyak permintaan yang memakan sumber daya, melipatgandakan muatan pada layanan. Saat klien mencapai batas waktu dan mencoba kembali, mereka sering menunggu balasan pada koneksi pertama mereka saat mereka membuat permintaan baru pada koneksi terpisah. Jika server menyelesaikan permintaan pertama dan membalas, klien mungkin tidak mendengarkan, karena saat ini menunggu balasan dari permintaan yang dicoba kembali.

Masalah pekerjaan yang sia-sia ini menjadi alasan kami mencoba merancang layanan untuk menjalankan *pekerjaan yang dibatasi*. Di tempat-tempat kami mengekspos API yang dapat menghasilkan kumpulan data besar (atau daftar apa pun), kami mengeksposnya sebagai API yang mendukung paginasi. API ini menghasilkan hasil sebagian dan token yang dapat digunakan oleh klien untuk meminta lebih banyak data. Menurut kami, lebih mudah untuk memperkirakan muatan tambahan pada layanan saat server menangani permintaan yang memiliki batas atas untuk jumlah memori, CPU, dan bandwidth jaringan. Sangat sulit untuk menjalankan kontrol penerimaan jika server tidak tahu apa yang dibutuhkan untuk memproses permintaan.

Peluang yang lebih subtil untuk memprioritaskan permintaan terkait dengan bagaimana klien menggunakan API layanan. Contoh, sebuah layanan yang memiliki dua API: **start()** dan **end()**. Agar dapat menyelesaikan pekerjaan mereka, klien harus dapat memanggil kedua API. Dalam kasus ini, layanan harus memprioritaskan permintaan **end()** di atas permintaan **start()**. Jika layanan memprioritaskan **start()**, klien tidak akan dapat menyelesaikan pekerjaan yang mereka mulai, yang menyebabkan penurunan tegangan.

Paginasi adalah contoh lainnya untuk pekerjaan yang sia-sia. Jika klien harus membuat beberapa permintaan berurutan untuk melakukan paginasi pada hasil dari layanan, dan terjadi kegagalan setelah halaman N-1 kemudian membuang hasilnya, ini menyia-nyiakan panggilan layanan N-2 dan setiap percobaan ulang yang dijalankan dalam proses. Ini menandakan bahwa seperti halnya permintaan **end()**, permintaan halaman pertama seharusnya diprioritaskan di belakang permintaan paginasi halaman berikutnya. Ini juga menggarisbawahi alasan kami merancang layanan untuk menjalankan pekerjaan yang dibatasi dan tidak melakukan paginasi tanpa akhir melalui layanan yang mereka panggil saat operasi.

Mewaspadai antrean

Melihat durasi permintaan saat sedang mengelola antrean internal juga dapat membantu. Banyak arsitektur layanan modern menggunakan antrean dalam memori untuk menghubungkan kumpulan thread guna memproses permintaan selama beberapa tahapan pekerjaan. Kerangka kerja layanan web dengan eksekutor cenderung memiliki antrean yang dikonfigurasi di depannya. Dengan layanan berbasis TCP, sistem operasi mempertahankan buffer untuk setiap soket, dan buffer tersebut dapat berisi permintaan terkunci dan volume besar.

Saat kami menarik pekerjaan keluar dari antrean, kami menggunakan peluang tersebut untuk memeriksa berapa lama pekerjaan bertahan dalam antrean. Minimal, kami mencoba mencatat durasi tersebut dalam metrik layanan kami. Selain membatasi ukuran antrean, kami menemukan bahwa sangat penting untuk memberlakukan batas atas pada lamanya waktu permintaan masuk dapat bertahan dalam antrean, dan kami membuangnya jika terlalu usang. Ini akan mengosongkan ruang pada server untuk bekerja pada permintaan yang lebih baru dan memiliki peluang lebih besar untuk

berhasil. Sebagai versi ekstrem dari pendekatan ini, kami mencari cara untuk menggunakan antrean pertama masuk, pertama keluar (LIFO), jika protokol mendukungnya. (Saluran permintaan HTTP/1.1 pada koneksi TCP tertentu tidak mendukung antrean LIFO, tetapi HTTP/2 secara umum mendukungnya.)

Penyeimbang muatan juga dapat memasukkan permintaan atau koneksi masuk ke dalam antrean saat layanan mengalami kelebihan muatan, dengan menggunakan fitur yang disebut *antrean lonjakan*. Antrean ini dapat menyebabkan pembatasan daya karena saat server akhirnya mendapatkan permintaan, server tidak tahu berapa lama permintaan tersebut berada dalam antrean. Default yang secara umum aman adalah menggunakan konfigurasi spillover, yang memberikan laporan cepat sebagai ganti memasukkan kelebihan permintaan ke dalam antrean. Di Amazon, pembelajaran ini diterapkan dalam generasi layanan Elastic Load Balancing (ELB) berikutnya. Classic Load Balancer menggunakan antrean lonjakan, tetapi Application Load Balancer menolak kelebihan lalu lintas. Apa pun konfigurasinya, tim di Amazon memantau metrik penyeimbang muatan yang relevan, seperti kedalaman antrean lonjakan atau jumlah spillover, untuk layanan mereka.

Dalam pengalaman kami, mewaspadai antrean tidak dapat diremehkan dan sangat penting. Saya sering terkejut saat menemukan antrean dalam memori di tempat saya tidak mencarinya secara intuitif, dalam sistem dan pustaka yang saya andalkan. Saat saya menelusuri sistem, menurut saya berasumsi bahwa ada antrean di suatu tempat yang belum saya ketahui dapat membantu. Tentu saja pengujian kelebihan muatan akan memberikan informasi yang lebih berguna daripada menelusuri kode, asalkan saya dapat membuat kasus uji realistis yang tepat.

Perlindungan terhadap kelebihan muatan di lapisan yang lebih rendah

Layanan terdiri dari beberapa lapisan—dari penyeimbang muatan hingga sistem operasi dengan kemampuan *netfilter* dan *iptables*, hingga kerangka kerja layanan, hingga kode—dan setiap lapisan memberikan kemampuan untuk melindungi layanan.

Proxy HTTP seperti NGINX sering mendukung fitur koneksi maksimum (*max_conns*) untuk membatasi jumlah permintaan atau koneksi maksimum yang akan diteruskannya ke server backend. Ini dapat menjadi mekanisme yang berguna, tetapi kami belajar untuk menggunakannya sebagai upaya terakhir, sebagai ganti opsi perlindungan default. Dengan proxy, sulit untuk memprioritaskan lalu lintas penting, dan pelacakan jumlah permintaan in-flight mentah sering memberikan informasi yang tidak akurat mengenai apakah layanan memang benar mengalami kelebihan muatan.

Di awal artikel ini, saya telah menjelaskan tantangan dari saat saya bergabung di tim Service Frameworks. Kami mencoba memberi tim Amazon default yang direkomendasikan untuk koneksi maksimum guna dikonfigurasikan pada penyeimbang muatan mereka. Pada akhirnya, kami menyarankan kepada tim agar mengatur koneksi maksimum untuk penyeimbang muatan mereka dan proxy tinggi, serta memungkinkan server mengimplementasikan algoritme pelepasan muatan yang lebih akurat dengan informasi lokal. Meski demikian, juga penting untuk nilai koneksi maksimum tidak melampaui jumlah thread pendengar, proses pendengar, atau deskriptor file pada server, sehingga server memiliki sumber daya untuk menangani permintaan pemeriksaan kesehatan penting dari penyeimbang muatan.

Fitur sistem operasi untuk membatasi penggunaan sumber daya server sangat andal dan dapat berguna untuk digunakan dalam keadaan darurat. Karena kami mengetahui bahwa kelebihan muatan dapat terjadi lagi, kami memastikan kami bersiap untuk itu dengan menggunakan buku panduan yang tepat dengan perintah yang spesifik selalu tersedia. Utilitas *iptables* dapat

menempatkan batas atas pada jumlah koneksi yang akan diterima oleh server, dan dapat menolak kelebihan koneksi dengan jauh lebih murah dibandingkan proses server mana pun. Ini juga dapat dikonfigurasikan dengan kontrol yang lebih canggih, seperti mengizinkan koneksi baru pada laju terbatas, atau bahkan mengizinkan laju maupun jumlah koneksi terbatas per alamat IP sumber. Filter IP sumber sangat tangguh, tetapi tidak berlaku untuk penyeimbang muatan tradisional. Meski demikian, ELB Network Load Balancer menyimpan IP sumber pemanggil bahkan pada lapisan sistem operasi melalui virtualisasi jaringan, membuat aturan *iptables* seperti filter IP sumber bekerja seperti seharusnya.

Melindungi dalam lapisan

Dalam beberapa kasus, sebuah server kehabisan sumber daya untuk sekadar menolak permintaan tanpa menjadi lambat. Dengan mempertimbangkan hal ini, kami melihat semua lompatan antara server dan kliennya untuk melihat apakah mereka dapat bekerja sama dan membantu melepaskan kelebihan muatan. Sebagai contoh, beberapa layanan AWS menyertakan opsi pelepasan muatan secara default. Saat kami menyediakan layanan dengan Amazon API Gateway, kami dapat mengonfigurasikan laju permintaan maksimum yang akan diterima oleh API mana pun. Saat layanan kami dihadapkan pada API Gateway, Application Load Balancer, atau Amazon CloudFront, kami dapat mengonfigurasikan AWS WAF untuk melepaskan kelebihan lalu lintas pada beberapa dimensi.

Visibilitas menciptakan tensi yang sulit. Penolakan awal penting karena merupakan tempat termurah untuk meletakkan kelebihan lalu lintas, tetapi akan mengorbankan visibilitas. Inilah mengapa kami melindungi dalam beberapa lapisan; untuk memungkinkan server mengambil lebih daripada yang dapat dikerjakannya dan meletakkan kelebihan, serta mencatat cukup informasi untuk mengetahui lalu lintas apa yang dijatuhkannya. Karena hanya ada beberapa lalu lintas yang dapat dijatuhkan oleh server, kami mengandalkan lapisan di bagian depan untuk melindungi server dari volume lalu lintas ekstra.

Berpikir tentang kelebihan muatan secara berbeda

Dalam artikel ini, kita mendiskusikan bagaimana kebutuhan untuk melepaskan muatan muncul karena fakta bahwa sistem menjadi lebih lamban saat diberikan lebih banyak pekerjaan konkuren karena dorongan seperti batasan sumber daya dan persaingan akan menjadi penghambat. Loop umpan balik kelebihan muatan didorong oleh latensi, yang pada akhirnya akan menyebabkan pekerjaan yang sia-sia, penguatan laju permintaan, dan bahkan kelebihan muatan yang semakin besar. Dorongan ini, didorong oleh Universal Scalability Law dan Amdahl's Law, penting untuk dihindari dengan **melepaskan kelebihan muatan dan mempertahankan kinerja yang konsisten dan dapat diprediksi dalam menghadapi kelebihan muatan**. Berfokus pada kinerja yang konsisten dan dapat diprediksi menjadi prinsip desain utama yang mendasari layanan di Amazon.

Sebagai contoh, Amazon DynamoDB adalah layanan database yang menawarkan kinerja yang dapat diprediksi dan ketersediaan skala besar. Bahkan jika beban kerja melonjak dengan tajam dan melampaui sumber daya yang disediakan, DynamoDB akan mempertahankan latensi goodput yang dapat diprediksi untuk beban kerja tersebut. Faktor seperti auto scaling DynamoDB, kapasitas adaptif, dan sesuai permintaan akan bereaksi cepat untuk meningkatkan laju goodput untuk beradaptasi dengan peningkatan beban kerja. Selama itu, goodput akan tetap stabil, memastikan layanan dalam lapisan di atas DynamoDB dengan kinerja yang juga dapat dipastikan, dan meningkatkan stabilitas keseluruhan sistem.

AWS Lambda memberi contoh yang lebih luas mengenai fokus pada kinerja yang dapat diprediksi. Saat kami menggunakan Lambda untuk mengimplementasikan sebuah layanan, setiap panggilan API berjalan dalam lingkungan eksekusinya sendiri dengan jumlah sumber daya komputasi yang konsisten dialokasikan ke panggilan, dan lingkungan eksekusi tersebut hanya bekerja pada satu permintaan dalam satu waktu. Ini berbeda dari paradigma berbasis server, dengan server tertentu bekerja pada beberapa API.

Mengisolasi setiap panggilan API ke sumber daya independennya sendiri (komputasi, memori, disk, jaringan) akan memangkas Amdahl's law dalam beberapa cara, karena sumber daya satu panggilan API tidak akan bersaing dengan sumber daya panggilan API lainnya. Karenanya, jika throughput melebihi goodput, goodput akan tetap sama, dan tidak menurun seperti dalam lingkungan berbasis server yang lebih tradisional. Ini adalah panacea, karena dependensi dapat memperlambat dan menyebabkan kenaikan konkurensi. Meski demikian, dalam skenario ini, setidaknya satu jenis persaingan sumber daya pada host yang kita diskusikan dalam artikel ini tidak akan berlaku.

Isolasi sumber daya ini cenderung subtil, tetapi merupakan manfaat penting dari lingkungan komputasi tanpa server modern seperti AWS Fargate, Amazon Elastic Container Service (Amazon ECS), dan AWS Lambda. Di Amazon, kami telah menemukan bahwa butuh banyak usaha untuk mengimplementasikan pelepasan muatan, dari penyetelan kumpulan thread, hingga memilih konfigurasi yang sempurna untuk koneksi penyeimbang muatan maksimum. Default yang masuk akal untuk jenis konfigurasi ini sulit atau mustahil untuk ditemukan, karena bergantung pada karakteristik operasional yang unik dari setiap sistem. Lingkungan komputasi tanpa server yang lebih baru ini memberikan isolasi sumber daya level lebih rendah dan memaparkan kenop level lebih tinggi, seperti throttling dan kontrol konkurensi, untuk melindungi dari kelebihan muatan. Terkadang, sebagai ganti mengejar nilai konfigurasi default, kami dapat benar-benar menghindari konfigurasi tersebut dan melindungi dari kategori kelebihan muatan tanpa konfigurasi sama sekali.

Baca lebih lanjut

- Universal Scalability Law
- Amdahl's Law
- (SEDA)
- Little's Law (menjelaskan konkurensi dalam sistem dan cara menentukan kapasitas sistem yang didistribusikan)
- Telling Stories About Little's Law, *Marc's Blog*
- Panduan dan Praktik Terbaik Elastic Load Balancing, presentasi re:Invent 2016 (menjelaskan evolusi Elastic Load Balancing untuk menghentikan atrean kelebihan permintaan)
- Burgess, Thinking in Promises: Designing Systems for Cooperation, O'Reilly Media, 2015