AWS Whitepaper

Ketersediaan dan Selanjutnya: Memahami dan Meningkatkan Ketahanan Sistem Terdistribusi AWS

Ketersediaan dan Selanjutnya: Memahami dan Meningkatkan Ketahanan Sistem Terdistribusi AWS: AWS Whitepaper

Table of Contents

Ketersediaan dan Beyond: Memahami dan Meningkatkan Ketahanan Sistem Terdistribusi AWS

Tanggal penerbitan: 12 November 2021 (Riwayat dokumen)

Saat ini, bisnis mengoperasikan sistem terdistribusi yang kompleks baik di cloud maupun lokal. Mereka ingin beban kerja ini tangguh untuk melayani pelanggan mereka dan mencapai hasil bisnis mereka. Makalah ini menguraikan pemahaman umum tentang ketersediaan sebagai ukuran ketahanan, menetapkan aturan untuk membangun beban kerja yang sangat tersedia, dan menawarkan panduan tentang cara meningkatkan ketersediaan beban kerja.

Pengantar

Apa artinya membangun beban kerja yang sangat tersedia? Bagaimana Anda mengukur ketersediaan? Apa yang dapat saya lakukan untuk meningkatkan ketersediaan beban kerja saya? Dokumen ini akan membantu Anda menjawab pertanyaan semacam ini. Hal ini dibagi menjadi tiga bagian utama. Bagian pertama, Memahami ketersediaan sebagian besar teoritis. Ini menetapkan pemahaman umum tentang definisi ketersediaan dan faktor-faktor yang mempengaruhinya. Bagian kedua, Mengukur ketersediaan, memberikan panduan tentang pengukuran secara empiris ketersediaan beban kerja Anda. Bagian ketiga, Merancang sistem terdistribusi yang sangat tersedia AWS adalah aplikasi praktis dari ide-ide yang disajikan di bagian pertama. Selain itu, di seluruh bagian ini, makalah ini akan menjadi aturan identitas untuk membangun beban kerja yang tangguh. Dokumen ini dimaksudkan untuk mendukung panduan dan praktik terbaik yang disajikan dalam Pilar Keandalan AWS Well-Architected.

Sepanjang makalah ini, Anda akan menemukan banyak matematika aljabar. Takeaways utama adalah konsep matematika ini mendukung, bukan matematika itu sendiri. Konon, makalah ini juga merupakan maksud dari makalah ini untuk menghadirkan tantangan. Ketika Anda mengoperasikan beban kerja yang sangat tersedia, Anda harus dapat membuktikan, secara matematis, bahwa apa yang Anda bangun adalah mencapai apa yang Anda inginkan. Bahkan desain terbaik yang dibangun dengan niat baik mungkin tidak secara konsisten mencapai hasil yang diinginkan. Ini berarti Anda memerlukan mekanisme yang mengukur efektivitas solusi, dan dengan demikian, beberapa tingkat matematika diperlukan dalam membangun dan mengoperasikan sistem terdistribusi yang tangguh dan sangat tersedia.

Memahami ketersediaan

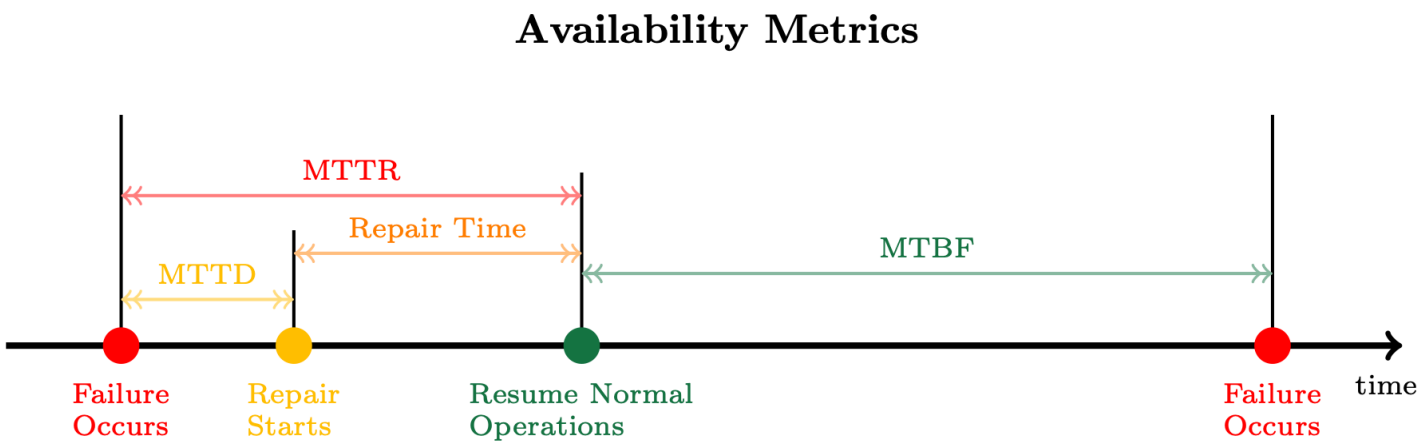
Ketersediaan adalah salah satu cara utama kita dapat mengukur ketahanan secara kuantitatif. Kami mendefinisikan ketersediaan, A, sebagai persentase waktu beban kerja tersedia untuk digunakan. Ini adalah rasio dari “uptime” yang diharapkan (tersedia) dengan total waktu yang diukur (“uptime” yang diharapkan ditambah “downtime” yang diharapkan).

$$A = \frac{uptime}{uptime + downtime}$$

Persamaan 1 - Ketersediaan

Untuk lebih memahami rumus ini, kita akan melihat bagaimana mengukur uptime dan downtime. Pertama, kami ingin tahu berapa lama beban kerja akan berjalan tanpa kegagalan. Kami menyebutnya waktu rata-rata antara kegagalan (MTBF), waktu rata-rata antara ketika beban kerja memulai operasi normal dan kegagalan berikutnya. Kemudian, kami ingin tahu berapa lama waktu yang dibutuhkan untuk pulih setelah gagal.

Kami menyebutnya mean time to repair (atau recovery) (MTTR), periode waktu ketika beban kerja tidak tersedia saat subsistem yang gagal diperbaiki atau dikembalikan ke layanan. Periode waktu penting dalam MTTR adalah waktu rata-rata untuk deteksi (MTTD), jumlah waktu antara kegagalan yang terjadi dan ketika operasi perbaikan dimulai. Diagram berikut menunjukkan bagaimana semua metrik ini terkait.



Hubungan antara MTTD, MTTR, dan MTBF

Dengan demikian kita dapat menyatakan ketersediaan, A, menggunakan MTBF, waktu beban kerja naik, dan MTTR, waktu beban kerja turun.

$$A = \frac{MTBF}{MTBF + MTTR}$$

Persamaan 2 - Hubungan antara MTBF dan MTTR

Dan probabilitas beban kerja “turun” (yaitu, tidak tersedia) adalah probabilitas kegagalan, F.

$$F = 1 - A$$

Persamaan 3 - Probabilitas kegagalan

Keandalan adalah kemampuan beban kerja untuk melakukan hal yang benar, ketika diminta, dalam waktu respons yang ditentukan. Inilah ukuran ketersediaan. Memiliki beban kerja gagal lebih jarang (MTBF lebih lama) atau memiliki waktu perbaikan yang lebih pendek (MTTR yang lebih pendek) meningkatkan ketersediaannya.

> Aturan 1
>
> Kegagalan yang lebih jarang (MTBF lebih lama), waktu deteksi kegagalan yang lebih pendek (MTTD lebih pendek), dan waktu perbaikan yang lebih pendek (MTTR lebih pendek) adalah tiga faktor yang digunakan untuk meningkatkan ketersediaan dalam sistem terdistribusi.

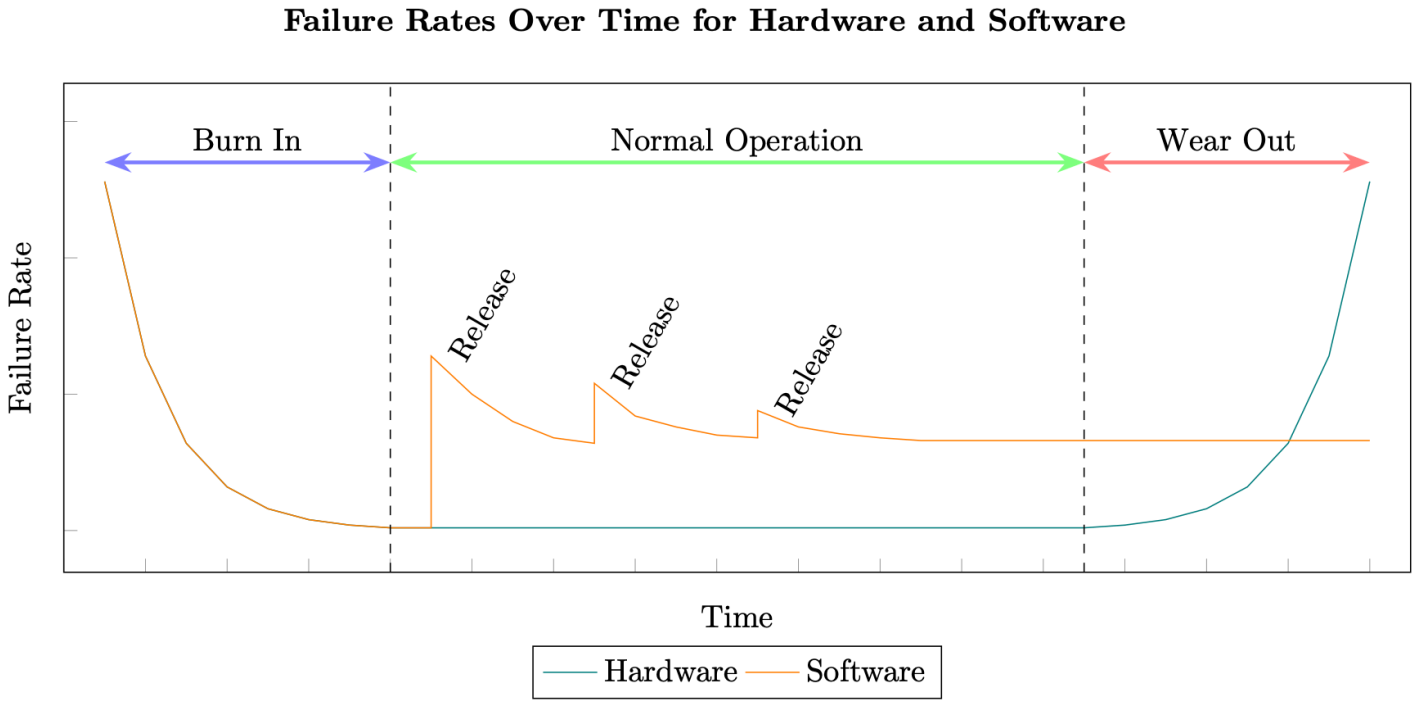
Topik

- Ketersediaan sistem terdistribusi
- Ketersediaan dengan dependensi
- Ketersediaan dengan redundansi
- Teorema CAP
- Toleransi kesalahan dan isolasi kesalahan

Ketersediaan sistem terdistribusi

Sistem terdistribusi terdiri dari komponen perangkat lunak dan komponen perangkat keras. Beberapa komponen perangkat lunak itu sendiri mungkin merupakan sistem terdistribusi lain. Ketersediaan komponen perangkat keras dan perangkat lunak yang mendasarinya memengaruhi ketersediaan beban kerja Anda yang dihasilkan.

Perhitungan ketersediaan menggunakan MTBF dan MTTR berakar pada sistem perangkat keras. Namun, sistem terdistribusi gagal karena alasan yang sangat berbeda dari perangkat keras. Di mana produsen dapat secara konsisten menghitung waktu rata-rata sebelum komponen perangkat keras habis, pengujian yang sama tidak dapat diterapkan pada komponen perangkat lunak dari sistem terdistribusi. Perangkat keras biasanya mengikuti kurva “bak mandi” tingkat kegagalan, sementara perangkat lunak mengikuti kurva terhuyung-huyung yang dihasilkan oleh cacat tambahan yang diperkenalkan dengan setiap rilis baru (lihat Keandalan Perangkat Lunak.)



Tingkat kegagalan perangkat keras dan perangkat lunak

Selain itu, perangkat lunak dalam sistem terdistribusi biasanya berubah pada tingkat yang secara eksponensial lebih tinggi daripada perangkat keras. Misalnya, hard drive magnetik standar mungkin memiliki tingkat kegagalan tahunan rata-rata (AFR) 0,93% yang, dalam praktiknya untuk HDD, dapat berarti umur setidaknya 3-5 tahun sebelum mencapai periode keausan, berpotensi lebih lama (lihat Data dan Statistik Hard Drive Backblaze, 2020.) Hard drive tidak berubah secara material selama

masa hidup itu, di mana, dalam 3-5 tahun, sebagai contoh, Amazon mungkin menyebarkan lebih dari 450 hingga 750 juta perubahan pada sistem perangkat lunaknya. (Lihat Amazon Builders' Library — Mengotomatiskan penerapan yang aman dan lepas tangan.)

Perangkat keras juga tunduk pada konsep usang yang direncanakan, yaitu memiliki masa pakai bawaan, dan perlu diganti setelah jangka waktu tertentu. (Lihat Konspirasi Bola Lampu Hebat.) Perangkat lunak, secara teoritis, tidak tunduk pada kendala ini, tidak memiliki periode aus dan dapat dioperasikan tanpa batas waktu.

Semua ini berarti bahwa model pengujian dan prediksi yang sama yang digunakan untuk perangkat keras untuk menghasilkan nomor MTBF dan MTTR tidak berlaku untuk perangkat lunak. Ada ratusan upaya untuk membangun model untuk memecahkan masalah ini sejak tahun 1970-an, tetapi semuanya umumnya terbagi dalam dua kategori, pemodelan prediksi dan pemodelan estimasi. (Lihat Daftar model keandalan perangkat lunak.)

Dengan demikian, menghitung MTBF dan MTTR berwawasan ke depan untuk sistem terdistribusi, dan dengan demikian ketersediaan berwawasan ke depan, akan selalu diturunkan dari beberapa jenis prediksi atau perkiraan. Mereka dapat dihasilkan melalui pemodelan prediktif, simulasi stokastik, analisis historis, atau pengujian yang ketat, tetapi perhitungan tersebut bukan jaminan uptime atau downtime.

Alasan mengapa sistem terdistribusi gagal di masa lalu mungkin tidak pernah terulang kembali. Alasan kegagalan di masa depan cenderung berbeda dan mungkin tidak dapat diketahui. Mekanisme pemulihan yang diperlukan mungkin juga berbeda untuk kegagalan masa depan daripada yang digunakan di masa lalu dan membutuhkan waktu yang sangat berbeda.

Selain itu, MTBF dan MTTR adalah rata-rata. Akan ada beberapa varians dari nilai rata-rata ke nilai aktual yang terlihat (standar deviasi, σ, mengukur variasi ini). Dengan demikian, beban kerja mungkin mengalami waktu yang lebih pendek atau lebih lama antara kegagalan dan waktu pemulihan dalam penggunaan produksi aktual.

Karena itu, ketersediaan komponen perangkat lunak yang membentuk sistem terdistribusi masih penting. Perangkat lunak dapat gagal karena berbagai alasan (dibahas lebih lanjut di bagian berikutnya) dan berdampak pada ketersediaan beban kerja. Dengan demikian, untuk sistem terdistribusi yang sangat tersedia, fokus yang sama untuk menghitung, mengukur, dan meningkatkan ketersediaan komponen perangkat lunak harus diberikan untuk perangkat keras dan subsistem perangkat lunak eksternal.

> **Aturan 2**
>
> Ketersediaan perangkat lunak dalam beban kerja Anda merupakan faktor penting dari ketersediaan keseluruhan beban kerja Anda dan harus menerima fokus yang sama dengan komponen lainnya.

Penting untuk dicatat bahwa meskipun MTBF dan MTTR sulit diprediksi untuk sistem terdistribusi, mereka masih memberikan wawasan kunci tentang cara meningkatkan ketersediaan. Mengurangi frekuensi kegagalan (MTBF yang lebih tinggi) dan mengurangi waktu untuk pulih setelah kegagalan terjadi (MTTR yang lebih rendah) keduanya akan mengarah pada ketersediaan empiris yang lebih tinggi.

Jenis kegagalan dalam sistem terdistribusi

Umumnya ada dua kelas bug dalam sistem terdistribusi yang memengaruhi ketersediaan, dinamai Bohrbug dan Heisenbug (lihat "A Conversation with Bruce Lindsay", ACM Queue vol. 2, no. 8 — November 2004.)

Bohrbug adalah masalah perangkat lunak fungsional berulang. Dengan input yang sama, bug akan secara konsisten menghasilkan output yang salah yang sama (seperti model atom Bohr deterministik, yang padat dan mudah dideteksi). Jenis bug ini jarang terjadi pada saat beban kerja sampai ke produksi.

Heisenbug adalah bug yang bersifat sementara, artinya hanya terjadi dalam kondisi tertentu dan tidak biasa. Kondisi ini biasanya terkait dengan hal-hal seperti perangkat keras (misalnya, kesalahan perangkat sementara atau spesifik implementasi perangkat keras seperti ukuran register), pengoptimalan kompiler dan implementasi bahasa, kondisi batas (misalnya, sementara di luar penyimpanan), atau kondisi balapan (misalnya, tidak menggunakan semaphore untuk operasi multi-threaded).

Heisenbug merupakan sebagian besar bug dalam produksi dan sulit ditemukan karena sulit dipahami dan tampaknya mengubah perilaku atau menghilang ketika Anda mencoba mengamati atau men-debugnya. Namun, jika Anda me-restart program, operasi yang gagal kemungkinan akan berhasil karena lingkungan operasi sedikit berbeda, menghilangkan kondisi yang memperkenalkan Heisenbug.

Dengan demikian, sebagian besar kegagalan dalam produksi bersifat sementara dan ketika operasi dicoba lagi, tidak mungkin gagal lagi. Agar tangguh, sistem terdistribusi harus toleran terhadap

kesalahan terhadap Heisenbugs. Kami akan mengeksplorasi bagaimana hal ini dapat dicapai di bagian Meningkatkan sistem terdistribusi MTBF.

Ketersediaan dengan dependensi

Di bagian sebelumnya, kami menyebutkan bahwa perangkat keras, perangkat lunak, dan kemungkinan sistem terdistribusi lainnya adalah semua komponen beban kerja Anda. Kami menyebut dependensi komponen ini, hal-hal yang bergantung pada beban kerja Anda untuk menyediakan fungsinya. Ada dependensi keras, yang merupakan hal-hal yang beban kerja Anda tidak dapat berfungsi tanpanya, dan dependensi lunak yang ketidaktersediaannya dapat luput dari perhatian atau ditoleransi selama beberapa periode waktu. Dependensi keras berdampak langsung pada ketersediaan beban kerja Anda.

Kami mungkin ingin mencoba dan menghitung ketersediaan maksimum teoritis dari beban kerja. Ini adalah produk dari ketersediaan semua dependensi, termasuk perangkat lunak itu sendiri, (α_n adalah ketersediaan subsistem tunggal) karena masing-masing harus operasional.

$$A = \alpha_1 \times \alpha_2 \times \ldots \times \alpha_n$$

Persamaan 4 - Ketersediaan maksimum teoritis

Nomor ketersediaan yang digunakan dalam perhitungan ini biasanya dikaitkan dengan hal-hal seperti SLA atau Tujuan Tingkat Layanan (SLO). SLA menentukan tingkat layanan yang diharapkan akan diterima pelanggan, metrik yang digunakan untuk mengukur layanan, dan remediasi atau penalti (biasanya moneter) jika tingkat layanan tidak tercapai.

Dengan menggunakan rumus di atas, kita dapat menyimpulkan bahwa, murni secara matematis, beban kerja tidak dapat lebih tersedia daripada dependensinya. Namun pada kenyataannya, apa yang biasanya kita lihat adalah bahwa ini tidak terjadi. Beban kerja yang dibangun menggunakan dua atau tiga dependensi dengan ketersediaan 99,99% SLA masih dapat mencapai ketersediaan 99,99% itu sendiri, atau lebih tinggi.

Ini karena seperti yang kami uraikan di bagian sebelumnya, angka ketersediaan ini adalah perkiraan. Mereka memperkirakan atau memprediksi seberapa sering kegagalan terjadi dan seberapa cepat itu dapat diperbaiki. Mereka bukan jaminan downtime. Dependensi sering melebihi ketersediaan yang dinyatakan SLA atau SLO.

Dependensi mungkin juga memiliki tujuan ketersediaan internal yang lebih tinggi yang mereka targetkan kinerja dibandingkan dengan angka yang disediakan di SLA publik. Ini memberikan tingkat mitigasi risiko dalam memenuhi SLA ketika hal yang tidak diketahui atau tidak diketahui terjadi.

Akhirnya, beban kerja Anda mungkin memiliki dependensi yang SLA tidak dapat diketahui atau tidak menawarkan SLA atau SLO. Misalnya, perutean internet di seluruh dunia adalah ketergantungan umum untuk banyak beban kerja, tetapi sulit untuk mengetahui penyedia layanan internet mana yang digunakan lalu lintas global Anda, apakah mereka memiliki SLA, dan seberapa konsisten mereka di seluruh penyedia.

Apa yang semua ini katakan kepada kita adalah bahwa menghitung ketersediaan teoretis maksimum hanya mungkin menghasilkan urutan perhitungan besarnya yang kasar, tetapi dengan sendirinya kemungkinan tidak akurat atau memberikan wawasan yang berarti. Apa yang dikatakan matematika kepada kami adalah bahwa semakin sedikit hal yang bergantung pada beban kerja Anda mengurangi kemungkinan kegagalan secara keseluruhan. Semakin sedikit angka kurang dari satu dikalikan bersama, semakin besar hasilnya.

> **Aturan 3**
>
> Mengurangi dependensi dapat berdampak positif pada ketersediaan.

Matematika juga membantu menginformasikan proses pemilihan ketergantungan. Proses seleksi memengaruhi cara Anda mendesain beban kerja Anda, bagaimana Anda memanfaatkan redundansi dalam dependensi untuk meningkatkan ketersediaannya, dan apakah Anda menganggap dependensi tersebut lunak atau keras. Dependensi yang dapat berdampak pada beban kerja Anda harus dipilih dengan cermat. Aturan berikutnya memberikan panduan tentang cara melakukannya.

> **Aturan 4**
>
> Secara umum, pilih dependensi yang tujuan ketersediaannya sama dengan atau lebih besar dari sasaran beban kerja Anda.

Ketersediaan dengan redundansi

Ketika beban kerja menggunakan beberapa subsistem, independen, dan redundan, ia dapat mencapai tingkat ketersediaan teoritis yang lebih tinggi daripada dengan menggunakan subsistem

tunggal. Misalnya, pertimbangkan beban kerja yang terdiri dari dua subsistem yang identik. Ini dapat sepenuhnya operasional jika subsistem satu atau subsistem dua beroperasi. Agar seluruh sistem turun, kedua subsistem harus turun pada saat yang bersamaan.

Jika probabilitas kegagalan satu subsistem adalah 1 α, maka probabilitas bahwa dua subsistem redundan turun pada saat yang sama adalah produk dari probabilitas kegagalan masing-masing subsistem, F = (1− α_1) × (1− α). $_2$ Untuk beban kerja dengan dua subsistem redundan, menggunakan Persamaan (3), ini memberikan ketersediaan yang didefinisikan sebagai:

$$A = 1 - F$$
$$F = (1 - \alpha_1) \times (1 - \alpha_2)$$
$$A = 1 - (1 - \alpha)^2$$

Persamaan 5

Jadi, untuk dua subsistem yang ketersediaannya 99%, probabilitas bahwa satu gagal adalah 1% dan probabilitas bahwa keduanya gagal adalah (1− 99%) × (1− 99%) = 0,01%. Ini membuat ketersediaan menggunakan dua subsistem redundan 99,99%.

Ini dapat digeneralisasi untuk memasukkan suku cadang tambahan yang berlebihan, s, juga. Dalam Persamaan (5) kami hanya mengasumsikan satu cadangan, tetapi beban kerja mungkin memiliki dua, tiga, atau lebih suku cadang sehingga dapat bertahan dari hilangnya beberapa subsistem secara simultan tanpa memengaruhi ketersediaan. Jika beban kerja memiliki tiga subsistem dan dua adalah suku cadang, probabilitas bahwa ketiga subsistem gagal pada saat yang sama adalah (1− α) × (1− α) × (1− α) atau (1− α) 3. Secara umum, beban kerja dengan suku cadang s hanya akan gagal jika subsistem s + 1 gagal.

Untuk beban kerja dengan n subsistem dan suku cadang s, f adalah jumlah mode kegagalan atau cara subsistem s + 1 gagal dari n.

Ini secara efektif adalah teorema binomial, matematika kombinatorial memilih k elemen dari himpunan n, atau “n pilih k”. Dalam hal ini, k adalah s + 1.

$$k = s + 1$$

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

$$\binom{n}{s+1} = \frac{n!}{(s+1)!\,\big(n-(s+1)\big)!}$$

$$f = \frac{n!}{(s+1)!\,(n-s-1)!}$$

Persamaan 6

Kami kemudian dapat menghasilkan perkiraan ketersediaan umum yang menggabungkan jumlah mode kegagalan dan hemat. (Untuk memahami mengapa ini dalam perkiraan, lihat Lampiran 2 dari Highleyman, dkk. Melanggar Penghalang Ketersediaan.)

$$s = \ Number\ of\ spares$$
$$\alpha = Availability\ of\ subcomponent$$
$$f = Number\ of\ failure\ modes$$
$$A = 1 - F \approx 1 - f(1-\alpha)^{s+1}$$

Persamaan 7

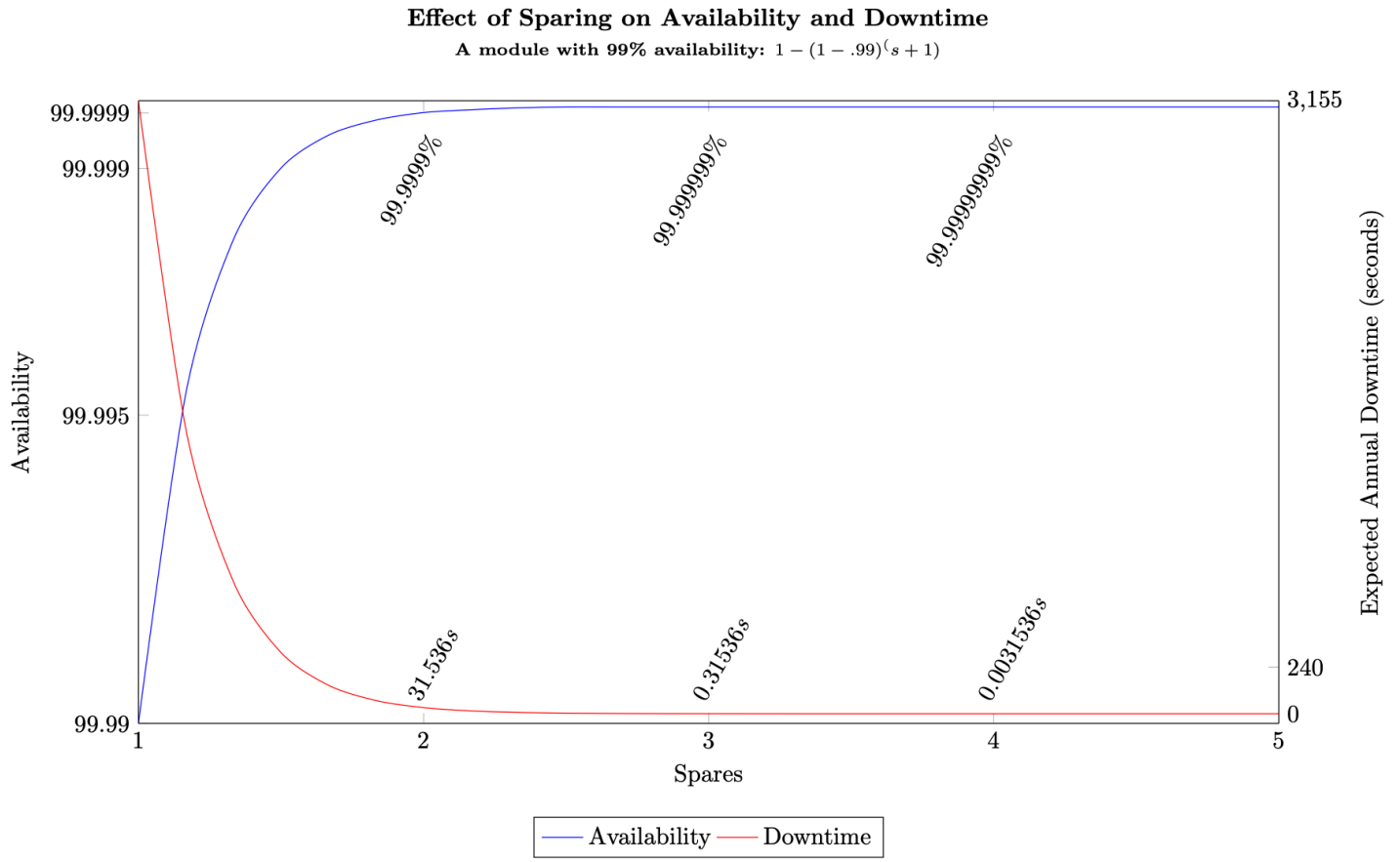
Hemat dapat diterapkan pada ketergantungan apa pun yang menyediakan sumber daya yang gagal secara independen. Instans Amazon EC2 di AZ yang berbeda atau bucket Amazon S3 berbeda adalah contohnya. Wilayah AWS Menggunakan suku cadang membantu ketergantungan mencapai ketersediaan total yang lebih tinggi untuk mendukung tujuan ketersediaan beban kerja.

> **Aturan 5**
>
> Gunakan hemat untuk meningkatkan ketersediaan dependensi dalam beban kerja.

Namun, hemat datang dengan biaya. Setiap biaya cadangan tambahan sama dengan modul asli, biaya mengemudi setidaknya secara linier. Membangun beban kerja yang dapat menggunakan suku cadang juga meningkatkan kompleksitasnya. Itu harus tahu bagaimana mengidentifikasi kegagalan ketergantungan, pekerjaan berat dari itu ke sumber daya yang sehat, dan mengelola kapasitas keseluruhan beban kerja.

Redundansi adalah masalah optimasi. Terlalu sedikit suku cadang, dan beban kerja bisa gagal lebih sering dari yang diinginkan, terlalu banyak suku cadang dan biaya beban kerja terlalu banyak untuk dijalankan. Ada ambang batas di mana menambahkan lebih banyak suku cadang akan lebih mahal daripada ketersediaan tambahan yang mereka capai waran.

Menggunakan ketersediaan umum kami dengan rumus suku cadang, Persamaan (7), untuk subsistem yang memiliki ketersediaan 99,5%, dengan dua suku cadang ketersediaan beban kerja adalah $A \approx 1 - (1)\ (1)\ (1-.995)^{3} = 99,9999875\%$ (sekitar 3,94 detik waktu henti setahun), dan dengan 10 suku cadang kita mendapatkan $A \approx 1 - (1)\ (1-.995)^{11} =$ 25.995 5 9 ′ s (perkiraan waktu henti adalah 1,26252 × 10 15 m s per tahun, efektif 0). Dalam membandingkan dua beban kerja ini, kami telah mengalami peningkatan 5X dalam biaya hemat untuk mencapai waktu henti empat detik lebih sedikit dalam setahun. Untuk sebagian besar beban kerja, kenaikan biaya tidak beralasan untuk peningkatan ketersediaan ini. Gambar berikut menunjukkan hubungan ini.



Mengurangi pengembalian dari peningkatan hemat

Pada tiga suku cadang dan seterusnya, hasilnya adalah sepersekian detik dari waktu henti yang diharapkan dalam setahun, yang berarti bahwa setelah titik ini Anda mencapai area pengembalian yang semakin berkurang. Mungkin ada dorongan untuk “hanya menambahkan lebih banyak” untuk mencapai tingkat ketersediaan yang lebih tinggi, tetapi pada kenyataannya, manfaat biaya menghilang dengan sangat cepat. Menggunakan lebih dari tiga suku cadang tidak memberikan material, keuntungan nyata untuk hampir semua beban kerja ketika subsistem itu sendiri memiliki setidaknya ketersediaan 99%.

> **Aturan 6**
>
> Ada batas atas efisiensi biaya hemat. Memanfaatkan suku cadang paling sedikit yang diperlukan untuk mencapai ketersediaan yang dibutuhkan.

Anda harus mempertimbangkan unit kegagalan saat memilih jumlah suku cadang yang benar. Sebagai contoh, mari kita periksa beban kerja yang memerlukan 10 instans EC2 untuk menangani kapasitas puncak dan mereka digunakan dalam satu AZ.

Karena AZ dirancang untuk menjadi batas isolasi kesalahan, unit kegagalan bukan hanya satu instans EC2, karena seluruh instans EC2 senilai AZ dapat gagal bersama. Dalam hal ini, Anda ingin menambahkan redundansi dengan AZ lain, menerapkan 10 instans EC2 tambahan untuk menangani beban jika terjadi kegagalan AZ, dengan total 20 instans EC2 (mengikuti pola stabilitas statis).

Meskipun ini tampaknya 10 instans EC2 cadangan, ini benar-benar hanya satu AZ cadangan, jadi kami belum melampaui titik pengembalian yang berkurang. Namun, Anda dapat menjadi lebih hemat biaya sekaligus meningkatkan ketersediaan Anda dengan memanfaatkan tiga AZ dan menerapkan lima instans EC2 per AZ.

Ini menyediakan satu AZ cadangan dengan total 15 instans EC2 (dibandingkan dua AZ dengan 20 instans), masih menyediakan 10 instans total yang diperlukan untuk melayani kapasitas puncak selama peristiwa yang memengaruhi satu AZ. Dengan demikian, Anda harus membangun hemat agar toleran terhadap kesalahan di semua batas isolasi kesalahan yang digunakan oleh beban kerja (instance, sel, AZ, dan Wilayah).

Teorema CAP

Cara lain yang mungkin kita pikirkan tentang ketersediaan adalah dalam kaitannya dengan teorema CAP. Teorema menyatakan bahwa sistem terdistribusi, yang terdiri dari beberapa node yang menyimpan data, tidak dapat secara bersamaan memberikan lebih dari dua dari tiga jaminan berikut:

- C onsistency: Setiap permintaan baca menerima penulisan terbaru atau kesalahan ketika konsistensi tidak dapat dijamin.
- Vailability: Setiap permintaan menerima respons non-error, bahkan ketika node sedang down atau tidak tersedia.
- Toleransi seni P: Sistem terus beroperasi meskipun kehilangan sejumlah pesan yang sewenang-wenang antar node.

(Untuk detail lebih lanjut, lihat Seth Gilbert dan Nancy Lynch, dugaan Brewer dan kelayakan layanan web yang konsisten, tersedia, dan toleran partisi, ACM SIGACT News, Volume 33 Edisi 2 (2002), hal. 51—59.)

Sebagian besar sistem terdistribusi harus mentolerir kegagalan jaringan, dan dengan demikian, partisi jaringan harus diizinkan. Ini berarti bahwa beban kerja ini harus membuat pilihan antara konsistensi dan ketersediaan ketika partisi jaringan terjadi. Jika beban kerja memilih ketersediaan, maka itu selalu mengembalikan respons, tetapi dengan data yang berpotensi tidak konsisten. Jika memilih konsistensi, maka selama partisi jaringan itu akan mengembalikan kesalahan karena beban kerja tidak dapat memastikan konsistensi data.

Untuk beban kerja yang tujuannya adalah untuk menyediakan tingkat ketersediaan yang lebih tinggi, mereka mungkin memilih Availability and Partition tolerance (AP) untuk mencegah pengembalian kesalahan (tidak tersedia) selama partisi jaringan. Ini menghasilkan membutuhkan model konsistensi yang lebih santai, seperti konsistensi akhirnya atau konsistensi monotonik.

Toleransi kesalahan dan isolasi kesalahan

Ini adalah dua konsep penting ketika kita berpikir tentang ketersediaan. Toleransi kesalahan adalah kemampuan untuk menahan kegagalan subsistem dan mempertahankan ketersediaan (melakukan hal yang benar dalam SLA yang sudah mapan). Untuk menerapkan toleransi kesalahan, beban kerja menggunakan subsistem cadangan (atau redundan). Ketika salah satu subsistem dalam set redundan gagal, yang lain mengambil pekerjaannya, biasanya hampir mulus. Dalam hal ini, suku cadang benar-benar kapasitas cadangan, mereka tersedia untuk mengasumsikan 100% pekerjaan dari subsistem yang gagal. Dengan suku cadang sejati, beberapa kegagalan subsistem diperlukan untuk menghasilkan dampak buruk pada beban kerja.

Isolasi kesalahan meminimalkan ruang lingkup dampak ketika kegagalan memang terjadi. Ini biasanya diimplementasikan dengan modularisasi. Beban kerja dipecah menjadi subsistem kecil yang gagal secara independen dan dapat diperbaiki secara terpisah. Kegagalan modul tidak menyebar di luar modul. Ide ini mencakup baik secara vertikal, di seluruh fungsionalitas yang berbeda dalam beban kerja, dan horizontal, di beberapa subsistem yang menyediakan fungsionalitas yang sama. Modul-modul ini bertindak sebagai wadah kesalahan yang membatasi ruang lingkup dampak selama suatu peristiwa.

Pola arsitektur bidang kontrol, bidang data, dan stabilitas statis secara langsung mendukung penerapan toleransi kesalahan dan isolasi kesalahan. Artikel Perpustakaan Pembangun Amazon Stabilitas statis menggunakan Availability Zones memberikan definisi yang baik untuk istilah-istilah ini dan bagaimana penerapannya untuk membangun beban kerja yang tangguh dan sangat tersedia. Whitepaper ini menggunakan pola-pola ini di bagian Merancang sistem terdistribusi yang sangat tersediaAWS, dan kami juga merangkum definisinya di sini.

- Control plane — Bagian dari beban kerja yang terlibat dalam membuat perubahan: menambahkan sumber daya, menghapus sumber daya, memodifikasi sumber daya, dan menyebarkan perubahan tersebut ke tempat yang dibutuhkan. Pesawat kontrol biasanya lebih kompleks dan memiliki bagian yang lebih bergerak daripada pesawat data, dan dengan demikian secara statistik lebih mungkin gagal dan memiliki ketersediaan yang lebih rendah.
- Bidang data — Bagian dari beban kerja yang menyediakan fungsionalitas day-to-day bisnis. Pesawat data cenderung lebih sederhana dan beroperasi pada volume yang lebih tinggi daripada pesawat kontrol, yang mengarah ke ketersediaan yang lebih tinggi.
- Stabilitas statis — Kemampuan beban kerja untuk melanjutkan operasi yang benar meskipun ada gangguan ketergantungan. Salah satu metode implementasi adalah menghapus dependensi bidang kontrol dari pesawat data. Metode lain adalah menggabungkan dependensi beban kerja secara longgar. Mungkin beban kerja tidak melihat informasi yang diperbarui (seperti hal-hal baru, hal-hal yang dihapus, atau hal-hal yang dimodifikasi) yang seharusnya disampaikan oleh ketergantungannya. Namun, semua yang dilakukannya sebelum ketergantungan menjadi terganggu terus bekerja.

Ketika kita berpikir tentang penurunan beban kerja, ada dua pendekatan tingkat tinggi yang dapat kita pertimbangkan untuk pemulihan. Metode pertama adalah menanggapi gangguan itu setelah itu terjadi, mungkin menggunakan AWS Auto Scaling untuk menambah kapasitas baru. Metode kedua adalah mempersiapkan gangguan tersebut sebelum terjadi, mungkin dengan menyediakan infrastruktur beban kerja secara berlebihan sehingga dapat terus beroperasi dengan benar tanpa memerlukan sumber daya tambahan.

Sistem yang stabil secara statis menggunakan pendekatan yang terakhir. Ini menyediakan kapasitas cadangan untuk tersedia selama kegagalan. Metode ini menghindari pembuatan ketergantungan pada bidang kontrol di jalur pemulihan beban kerja untuk menyediakan kapasitas baru untuk pulih dari kegagalan. Selain itu, penyediaan kapasitas baru untuk berbagai sumber daya membutuhkan waktu. Sambil menunggu kapasitas baru, beban kerja Anda dapat kelebihan beban oleh permintaan yang ada dan mengalami degradasi lebih lanjut, yang menyebabkan “kecoklatan” atau kehilangan ketersediaan total. Namun, Anda juga harus mempertimbangkan implikasi biaya dari penggunaan kapasitas yang telah disediakan sebelumnya terhadap tujuan ketersediaan Anda.

Stabilitas statis menyediakan dua aturan berikutnya untuk beban kerja ketersediaan tinggi.

Aturan 7

Jangan mengambil dependensi pada pesawat kontrol di bidang data Anda, terutama selama pemulihan.

Aturan 8

Gabungkan dependensi secara longgar sehingga beban kerja Anda dapat beroperasi dengan benar meskipun ada gangguan ketergantungan, jika memungkinkan.

Mengukur ketersediaan

Seperti yang kita lihat sebelumnya, membuat model ketersediaan berwawasan ke depan untuk sistem terdistribusi sulit dilakukan dan mungkin tidak memberikan wawasan yang diinginkan. Apa yang dapat memberikan lebih banyak utilitas adalah mengembangkan cara yang konsisten untuk mengukur ketersediaan beban kerja Anda.

Definisi ketersediaan sebagai uptime dan downtime mewakili kegagalan sebagai opsi biner, baik beban kerja sudah habis atau tidak.

Namun, ini jarang terjadi. Kegagalan memiliki tingkat dampak dan sering dialami dalam beberapa bagian dari beban kerja, mempengaruhi persentase pengguna atau permintaan, persentase lokasi, atau persentil latensi. Ini semua adalah mode kegagalan sebagian.

Dan sementara MTTR dan MTBF berguna dalam memahami apa yang mendorong ketersediaan sistem yang dihasilkan, dan karenanya, bagaimana memperbaikinya, utilitas mereka bukan sebagai ukuran ketersediaan empiris. Selain itu, beban kerja terdiri dari banyak komponen. Misalnya, beban kerja seperti sistem pemrosesan pembayaran terdiri dari banyak antarmuka pemrograman aplikasi (API) dan subsistem. Jadi, ketika kita ingin mengajukan pertanyaan seperti, “apa ketersediaan seluruh beban kerja?” , itu sebenarnya pertanyaan yang kompleks dan bernuansa.

Di bagian ini, kita akan memeriksa tiga cara ketersediaan dapat diukur secara empiris: tingkat keberhasilan permintaan sisi server, tingkat keberhasilan permintaan sisi klien, dan waktu henti tahunan.

Tingkat keberhasilan permintaan sisi server dan sisi klien

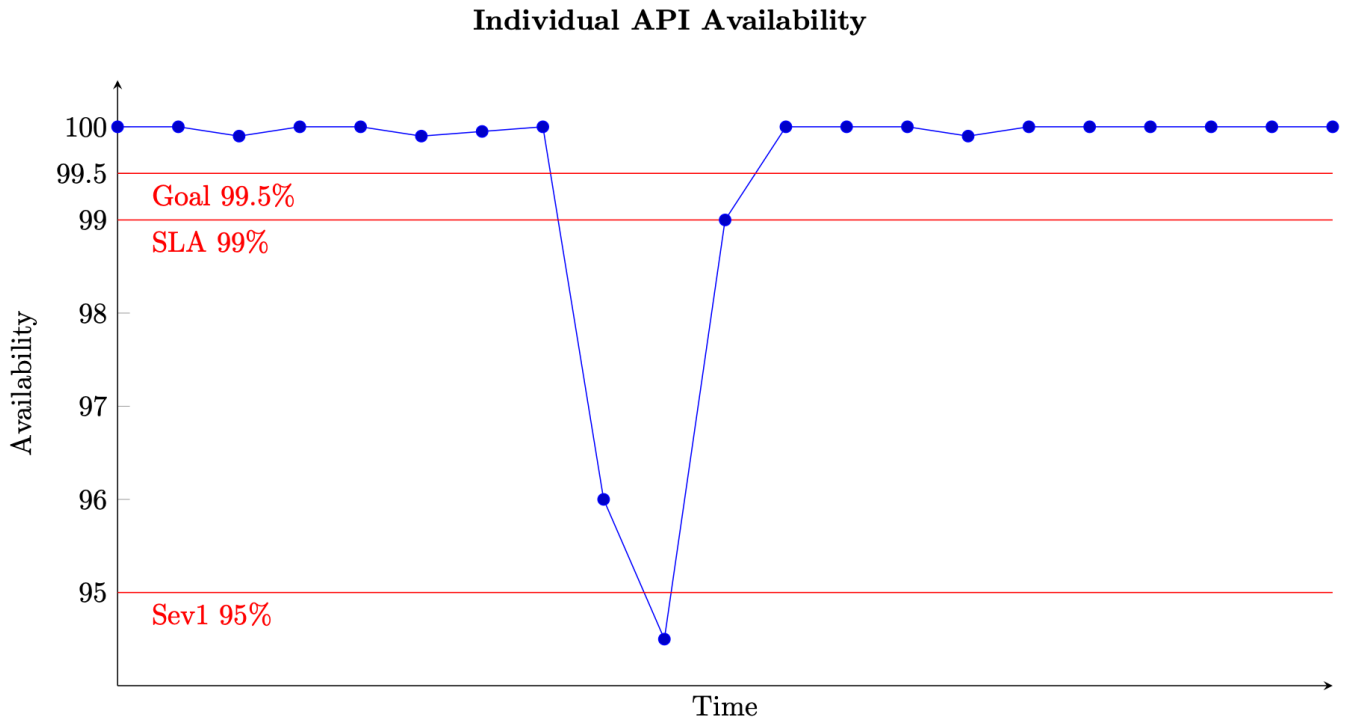
Dua metode pertama sangat mirip, hanya berbeda dari sudut pandang pengukuran yang diambil. Metrik sisi server dapat dikumpulkan dari instrumentasi dalam layanan. Namun, mereka tidak lengkap. Jika klien tidak dapat mencapai layanan, Anda tidak dapat mengumpulkan metrik tersebut. Untuk memahami pengalaman klien, alih-alih mengandalkan telemetri dari klien tentang permintaan yang gagal, cara yang lebih mudah untuk mengumpulkan metrik sisi klien adalah dengan mensimulasikan lalu lintas pelanggan dengan burung kenari, perangkat lunak yang secara teratur menyelidiki layanan Anda dan mencatat metrik.

Kedua metode ini menghitung ketersediaan sebagai fraksi dari total unit kerja valid yang diterima layanan dan yang berhasil diprosesnya (ini mengabaikan unit kerja yang tidak valid, seperti permintaan HTTP yang menghasilkan kesalahan 404).

$$A = \frac{Successfully\ Processed\ Units\ of\ Work}{Total\ Valid\ Units\ of\ Work\ Received}$$

Persamaan 8

Untuk layanan berbasis permintaan, unit kerja adalah permintaan, seperti permintaan HTTP. Untuk layanan berbasis acara atau berbasis tugas, unit pekerjaan adalah acara atau tugas, seperti memproses pesan dari antrian. Ukuran ketersediaan ini bermakna dalam interval waktu singkat, seperti jendela satu menit atau lima menit. Hal ini juga paling cocok pada perspektif granular, seperti pada tingkat per API untuk layanan berbasis permintaan. Gambar berikut memberikan pandangan tentang apa ketersediaan dari waktu ke waktu mungkin terlihat seperti ketika dihitung dengan cara ini. Setiap titik data pada grafik dihitung menggunakan Persamaan (8) selama jendela lima menit (Anda dapat memilih dimensi waktu lain seperti interval satu menit atau sepuluh menit). Misalnya, titik data 10 menunjukkan ketersediaan 94,5%. Itu berarti selama menit t+45 hingga t+50 jika layanan menerima 1.000 permintaan, hanya 945 di antaranya yang berhasil diproses.



Contoh pengukuran ketersediaan dari waktu ke waktu untuk satu API

Grafik juga menunjukkan sasaran ketersediaan API, ketersediaan 99,5%, perjanjian tingkat layanan (SLA) yang ditawarkannya kepada pelanggan, ketersediaan 99%, dan ambang batas untuk alarm

tingkat keparahan tinggi, 95%. Tanpa konteks ambang batas yang berbeda ini, grafik ketersediaan mungkin tidak memberikan wawasan yang signifikan tentang bagaimana layanan Anda beroperasi.

Kami juga ingin dapat melacak dan menggambarkan ketersediaan subsistem yang lebih besar, seperti bidang kontrol, atau seluruh layanan. Salah satu cara untuk melakukan ini adalah dengan mengambil rata-rata setiap titik data lima menit untuk setiap subsistem. Grafik akan terlihat mirip dengan yang sebelumnya, tetapi akan mewakili satu set input yang lebih besar. Ini juga memberikan bobot yang sama untuk semua subsistem yang membentuk layanan Anda. Pendekatan alternatif mungkin untuk menjumlahkan semua permintaan yang diterima dan berhasil diproses dari semua API dalam layanan untuk menghitung ketersediaan dalam interval lima menit.

Namun, metode terakhir ini mungkin menyembunyikan API individual yang memiliki throughput rendah dan ketersediaan buruk. Sebagai contoh sederhana, pertimbangkan layanan dengan dua API.

API pertama menerima 1.000.000 permintaan dalam jendela lima menit dan berhasil memproses 999.000 dari mereka, memberikan ketersediaan 99,9%. API kedua menerima 100 permintaan dalam jendela lima menit yang sama dan hanya berhasil memproses 50 dari mereka, memberikan ketersediaan 50%.

Jika kami menjumlahkan permintaan dari setiap API bersama-sama, ada 1.000.100 total permintaan valid dan 999.050 di antaranya berhasil diproses, memberikan ketersediaan 99.895% untuk layanan secara keseluruhan. Tetapi, jika kita rata-rata ketersediaan dari dua API, metode sebelumnya, kita mendapatkan ketersediaan yang dihasilkan 74.95%, yang mungkin lebih menceritakan pengalaman sebenarnya.

Tidak ada pendekatan yang salah, tetapi ini menunjukkan pentingnya memahami apa yang diberitahukan metrik ketersediaan kepada Anda. Anda dapat memilih untuk mendukung penjumlahan permintaan untuk semua subsistem jika beban kerja Anda menerima volume permintaan serupa di masing-masing subsistem. Pendekatan ini berfokus pada “permintaan” dan keberhasilannya sebagai ukuran ketersediaan dan pengalaman pelanggan. Atau, Anda dapat memilih untuk rata-rata ketersediaan subsistem untuk sama-sama mewakili kekritisan mereka meskipun perbedaan volume permintaan. Pendekatan ini berfokus pada subsistem dan kemampuan masing-masing sebagai proxy untuk pengalaman pelanggan.

Downtime tahunan

Pendekatan ketiga adalah menghitung downtime tahunan. Bentuk metrik ketersediaan ini lebih sesuai untuk penetapan dan peninjauan tujuan jangka panjang. Hal ini membutuhkan mendefinisikan

apa arti downtime untuk beban kerja Anda. Anda kemudian dapat mengukur ketersediaan berdasarkan jumlah menit bahwa beban kerja tidak dalam kondisi “pemadaman” relatif terhadap jumlah menit dalam periode tertentu.

Beberapa beban kerja mungkin dapat menentukan waktu henti sebagai sesuatu seperti penurunan di bawah 95% ketersediaan API tunggal atau fungsi beban kerja untuk interval satu menit atau lima menit (yang terjadi pada grafik ketersediaan sebelumnya). Anda mungkin juga hanya mempertimbangkan waktu henti karena berlaku untuk subset operasi bidang data penting. Misalnya, Perjanjian Tingkat Layanan Amazon Messaging (SQS, SNS) untuk ketersediaan SQS berlaku untuk SQS Send, Receive, dan Delete API.

Beban kerja yang lebih besar dan lebih kompleks mungkin perlu menentukan metrik ketersediaan di seluruh sistem. Untuk situs e-commerce besar, metrik seluruh sistem dapat menjadi sesuatu seperti tingkat pesanan pelanggan. Di sini, penurunan 10% atau lebih dalam pesanan dibandingkan dengan kuantitas yang diperkirakan selama jendela lima menit dapat menentukan waktu henti.

Dalam kedua pendekatan, Anda kemudian dapat menjumlahkan semua periode pemadaman untuk menghitung ketersediaan tahunan. Misalnya, jika selama tahun kalender, ada 27 periode waktu henti lima menit, yang didefinisikan sebagai ketersediaan API bidang data yang turun di bawah 95%, waktu henti keseluruhan adalah 135 menit (beberapa periode lima menit mungkin berturut-turut, yang lain terisolasi), mewakili ketersediaan tahunan 99,97%.

Metode pengukuran ketersediaan tambahan ini dapat memberikan data dan wawasan yang hilang dari sisi klien dan metrik sisi server. Misalnya, pertimbangkan beban kerja yang terganggu dan mengalami peningkatan tingkat kesalahan secara signifikan. Pelanggan beban kerja ini mungkin berhenti melakukan panggilan ke layanannya sama sekali. Mungkin mereka telah mengaktifkan pemutus sirkuit atau mengikuti rencana pemulihan bencana mereka untuk menggunakan layanan di wilayah yang berbeda. Jika kita hanya mengukur respons yang gagal, ketersediaan beban kerja sebenarnya dapat meningkat selama gangguan, tetapi bukan karena gangguan membaik atau hilang, tetapi karena pelanggan berhenti menggunakannya.

Latensi

Akhirnya, penting juga untuk mengukur latensi unit pemrosesan pekerjaan dalam beban kerja Anda. Bagian dari definisi ketersediaan adalah melakukan pekerjaan dalam SLA mapan. Jika mengembalikan respons membutuhkan waktu lebih lama dari batas waktu klien, persepsi dari klien adalah bahwa permintaan gagal dan beban kerja tidak tersedia. Namun, di sisi server, permintaan mungkin tampaknya telah diproses dengan sukses.

Mengukur latensi menyediakan lensa lain yang dapat digunakan untuk mengevaluasi ketersediaan. Menggunakan persentil dan rata-rata yang dipangkas adalah statistik yang baik untuk pengukuran ini. Mereka biasanya diukur pada persentil ke-50 (P50 dan TM50) dan persentil ke-99 (P99 dan TM99). Latensi harus diukur dengan kenari untuk mewakili pengalaman klien serta dengan metrik sisi server. Setiap kali rata-rata beberapa latensi persentil, seperti P99 atau TM99.9, berada di atas target SLA, Anda dapat mempertimbangkan bahwa downtime, yang berkontribusi pada perhitungan downtime tahunan Anda.

Merancang sistem terdistribusi yang sangat tersedia di AWS

Bagian sebelumnya sebagian besar tentang ketersediaan teoritis beban kerja dan apa yang dapat mereka capai. Mereka adalah seperangkat konsep penting yang perlu diingat saat Anda membangun sistem terdistribusi. Mereka akan membantu menginformasikan proses pemilihan ketergantungan Anda dan bagaimana Anda menerapkan redundansi.

Kami juga melihat hubunganMTTD,MTTR, dan MTBF ketersediaan. Bagian ini akan memperkenalkan panduan praktis berdasarkan teori sebelumnya. Singkatnya, beban kerja teknik untuk ketersediaan tinggi bertujuan untuk meningkatkan MTBF dan mengurangi MTTR serta. MTTD

Meskipun menghilangkan semua kegagalan akan ideal, itu tidak realistis. Dalam sistem terdistribusi besar dengan dependensi yang ditumpuk dalam, kegagalan akan terjadi. “Semuanya gagal sepanjang waktu” (lihat Werner Vogels,, Amazon.comCTO, 10 Pelajaran dari 10 Tahun Amazon Web Services.) dan “Anda tidak dapat membuat undang-undang terhadap kegagalan [jadi] fokus pada deteksi dan respons cepat.” (lihat Chris Pinkham, anggota pendiri, EC2 tim Amazon, ARC335Merancang untuk kegagalan: Merancang sistem tangguh aktif) AWS

Apa artinya ini adalah bahwa seringkali Anda tidak memiliki kendali atas apakah kegagalan terjadi. Apa yang dapat Anda kontrol adalah seberapa cepat Anda mendeteksi kegagalan dan melakukan sesuatu tentang hal itu. Jadi, sementara peningkatan masih MTBF merupakan komponen penting dari ketersediaan tinggi, perubahan paling signifikan yang dimiliki pelanggan dalam kendali mereka adalah mengurangi MTTD danMTTR.

Topik

- Mengurangi MTTD
- Mengurangi MTTR
- Meningkat MTBF

Mengurangi MTTD

MTTDMengurangi kegagalan berarti menemukan kegagalan secepat mungkin. Memperpendek MTTD didasarkan pada observabilitas, atau bagaimana Anda telah menginstrumentasi beban kerja Anda untuk memahami statusnya. Pelanggan harus memantau metrik Pengalaman Pelanggan mereka di subsistem kritis beban kerja mereka sebagai cara untuk secara proaktif mengidentifikasi kapan masalah terjadi (lihat Lampiran 1 — MTTD dan metrik MTTR penting untuk informasi lebih lanjut tentang metrik ini.). Pelanggan dapat menggunakan Amazon CloudWatch Synthetics untuk

membuat kenari yang memantau Anda APIs dan konsol untuk mengukur pengalaman pengguna secara proaktif. Ada sejumlah mekanisme pemeriksaan kesehatan lain yang dapat digunakan untuk meminimalkanMTTD, seperti pemeriksaan kesehatan Elastic Load Balancing (ELB), pemeriksaan kesehatan Amazon Route 53, dan banyak lagi. (Lihat Amazon Builders' Library — Menerapkan pemeriksaan kesehatan.)

Pemantauan Anda juga harus dapat mendeteksi kegagalan sebagian dari sistem secara keseluruhan dan dalam subsistem individu Anda. Metrik ketersediaan, kegagalan, dan latensi Anda harus menggunakan dimensi batas isolasi kesalahan Anda sebagai dimensi metrik. CloudWatch Misalnya, pertimbangkan satu EC2 instance yang merupakan bagian dari arsitektur berbasis sel, di use1-az1 AZ, di Wilayah us-east-1, yang merupakan bagian dari pembaruan beban kerja yang merupakan bagian dari subsistem bidang kontrolnya. API Ketika server mendorong metriknya, ia dapat menggunakan id instans, AZ, Wilayah, API nama, dan nama subsistem sebagai dimensi. Ini memungkinkan Anda untuk memiliki observabilitas dan mengatur alarm di masing-masing dimensi ini untuk mendeteksi kegagalan.

Mengurangi MTTR

Setelah kegagalan ditemukan, sisa MTTR waktu adalah perbaikan atau mitigasi dampak yang sebenarnya. Untuk memperbaiki atau mengurangi kegagalan, Anda harus tahu apa yang salah. Ada dua kelompok kunci metrik yang memberikan wawasan selama fase ini: 1/ metrik Penilaian Dampak dan 2/ metrik Kesehatan Operasional. Kelompok pertama memberi tahu Anda ruang lingkup dampak selama kegagalan, mengukur jumlah atau persentase pelanggan, sumber daya, atau beban kerja yang terkena dampak. Kelompok kedua membantu mengidentifikasi mengapa ada dampak. Setelah mengapa ditemukan, operator dan otomatisasi dapat merespons dan menyelesaikan kegagalan. Lihat Lampiran 1 — MTTD dan metrik MTTR penting untuk informasi lebih lanjut tentang metrik ini.

> Aturan 9
>
> Observabilitas dan instrumentasi sangat penting untuk mengurangi MTTD dan. MTTR

Rute di sekitar kegagalan

Pendekatan tercepat untuk mengurangi dampak adalah melalui subsistem cepat gagal yang mengelilingi kegagalan. Pendekatan ini menggunakan redundansi untuk mengurangi MTTR dengan cepat menggeser pekerjaan subsistem yang gagal ke cadangan. Redundansi dapat berkisar dari proses perangkat lunak, EC2 instance, hingga beberapa, hingga beberapa AZs Wilayah.

Subsistem cadangan dapat mengurangi MTTR turun menjadi hampir nol. Waktu pemulihan hanya apa yang diperlukan untuk mengalihkan pekerjaan ke cadangan siaga. Ini sering terjadi dengan latensi minimal dan memungkinkan pekerjaan selesai dalam yang ditentukanSLA, menjaga ketersediaan sistem. Ini menghasilkan MTTRs yang dialami sebagai penundaan ringan, bahkan mungkin tidak terlihat, daripada periode tidak tersedianya yang berkepanjangan.

Misalnya, jika layanan Anda menggunakan EC2 instans di belakang Application Load Balancer ALB (), Anda dapat mengonfigurasi pemeriksaan kesehatan pada interval sekecil lima detik dan hanya memerlukan dua pemeriksaan kesehatan yang gagal sebelum target ditandai sebagai tidak sehat. Ini berarti bahwa dalam 10 detik, Anda dapat mendeteksi kegagalan dan berhenti mengirim lalu lintas ke host yang tidak sehat. Dalam hal ini, MTTR secara efektif sama dengan MTTD karena segera setelah kegagalan terdeteksi, itu juga dikurangi.

Inilah yang coba dicapai oleh beban kerja ketersediaan tinggi atau ketersediaan berkelanjutan. Kami ingin dengan cepat merutekan kegagalan dalam beban kerja dengan cepat mendeteksi subsistem yang gagal, menandainya sebagai gagal, berhenti mengirim lalu lintas ke mereka, dan sebagai gantinya mengirim lalu lintas ke subsistem yang berlebihan.

Perhatikan bahwa menggunakan mekanisme fail-fast semacam ini membuat beban kerja Anda sangat sensitif terhadap kesalahan sementara. Dalam contoh yang diberikan, pastikan bahwa pemeriksaan kesehatan penyeimbang beban Anda melakukan pemeriksaan kesehatan dangkal atau aktif dan lokal hanya pada instance, bukan menguji dependensi atau alur kerja (sering disebut sebagai pemeriksaan kesehatan mendalam). Ini akan membantu mencegah penggantian instance yang tidak perlu selama kesalahan sementara yang memengaruhi beban kerja.

Observabilitas dan kemampuan untuk mendeteksi kegagalan dalam subsistem sangat penting untuk perutean di sekitar kegagalan untuk berhasil. Anda harus mengetahui ruang lingkup dampaknya sehingga sumber daya yang terkena dampak dapat ditandai sebagai tidak sehat atau gagal dan dikeluarkan dari layanan sehingga dapat dialihkan. Misalnya, jika satu AZ mengalami gangguan sebagian layanan, instrumentasi Anda harus dapat mengidentifikasi bahwa ada masalah yang dilokalkan AZ untuk merutekan semua sumber daya di AZ tersebut hingga pulih.

Mampu merutekan kegagalan mungkin juga memerlukan perkakas tambahan tergantung pada lingkungan. Menggunakan contoh sebelumnya dengan EC2 contoh di belakangALB, bayangkan bahwa instance dalam satu AZ mungkin melewati pemeriksaan kesehatan lokal, tetapi gangguan AZ yang terisolasi menyebabkan mereka gagal terhubung ke database mereka di AZ yang berbeda. Dalam hal ini, pemeriksaan kesehatan load balancing tidak akan menghilangkan instans tersebut dari layanan. Mekanisme otomatis yang berbeda akan diperlukan untuk menghapus AZ dari penyeimbang beban atau memaksa instans untuk gagal dalam pemeriksaan kesehatan mereka,

yang bergantung pada identifikasi bahwa ruang lingkup dampaknya adalah AZ. Untuk beban kerja yang tidak menggunakan penyeimbang beban, metode serupa akan diperlukan untuk mencegah sumber daya di AZ tertentu menerima unit kerja atau menghapus kapasitas dari AZ sama sekali.

Dalam beberapa kasus, pergeseran kerja ke subsistem yang berlebihan tidak dapat diotomatisasi, seperti failover database primer ke sekunder di mana teknologi tidak menyediakan pemilihan pemimpinnya sendiri. Ini adalah skenario umum untuk arsitektur AWS Multi-region. Karena jenis kegagalan ini memerlukan sejumlah waktu henti untuk diselesaikan, tidak dapat segera dibalik, dan membiarkan beban kerja tanpa redundansi untuk jangka waktu tertentu, penting untuk memiliki manusia dalam proses pengambilan keputusan.

Beban kerja yang dapat mencakup model konsistensi yang kurang ketat dapat mencapai yang lebih pendek MTTRs dengan menggunakan otomatisasi failover multi-wilayah untuk mengatasi kegagalan. Fitur seperti replikasi lintas wilayah Amazon S3 atau tabel global Amazon DynamoDB memberikan kemampuan Multi-wilayah melalui replikasi yang konsisten. Selanjutnya, menggunakan model konsistensi santai bermanfaat ketika kita mempertimbangkan CAP teorema. Selama kegagalan jaringan yang memengaruhi konektivitas ke subsistem stateful, jika beban kerja memilih ketersediaan daripada konsistensi, itu masih dapat memberikan respons non-kesalahan, cara lain untuk merutekan kegagalan.

Routing seputar kegagalan dapat diimplementasikan dengan dua strategi yang berbeda. Strategi pertama adalah dengan menerapkan stabilitas statis dengan melakukan pra-penyediaan sumber daya yang cukup untuk menangani beban lengkap subsistem yang gagal. Ini bisa berupa satu EC2 contoh atau mungkin kapasitas seluruh AZ. Mencoba menyediakan sumber daya baru selama kegagalan meningkatkan MTTR dan menambahkan ketergantungan ke bidang kontrol di jalur pemulihan Anda. Namun, itu datang dengan biaya tambahan.

Strategi kedua adalah merutekan sebagian lalu lintas dari subsistem yang gagal ke yang lain dan beban menumpahkan kelebihan lalu lintas yang tidak dapat ditangani oleh kapasitas yang tersisa. Selama periode degradasi ini, Anda dapat meningkatkan sumber daya baru untuk menggantikan kapasitas yang gagal. Pendekatan ini memiliki lebih lama MTTR dan menciptakan ketergantungan pada bidang kontrol, tetapi biaya lebih sedikit dalam siaga, kapasitas cadangan.

Kembali ke keadaan baik yang dikenal

Pendekatan umum lainnya untuk mitigasi selama perbaikan adalah mengembalikan beban kerja ke keadaan baik yang diketahui sebelumnya. Jika perubahan baru-baru ini mungkin menyebabkan kegagalan, memutar kembali perubahan itu adalah salah satu cara untuk kembali ke keadaan sebelumnya.

Dalam kasus lain, kondisi sementara mungkin telah menyebabkan kegagalan, dalam hal ini, memulai kembali beban kerja dapat mengurangi dampaknya. Mari kita periksa kedua skenario ini.

Selama penyebaran, meminimalkan MTTD dan MTTR bergantung pada observabilitas dan otomatisasi. Proses penerapan Anda harus terus memperhatikan beban kerja untuk pengenalan tingkat kesalahan yang meningkat, peningkatan latensi, atau anomali. Setelah ini dikenali, itu harus menghentikan proses penerapan.

Ada berbagai strategi penyebaran, seperti penerapan di tempat, penerapan biru/hijau, dan penerapan bergulir. Masing-masing dari ini mungkin menggunakan mekanisme yang berbeda untuk kembali ke keadaan yang diketahui baik. Ini dapat secara otomatis memutar kembali ke keadaan sebelumnya, mengalihkan lalu lintas kembali ke lingkungan biru, atau memerlukan intervensi manual.

CloudFormation menawarkan kemampuan untuk secara otomatis melakukan rollback sebagai bagian dari membuat dan memperbarui operasi tumpukan, seperti halnya. AWS CodeDeploy CodeDeploy juga mendukung penerapan biru/hijau dan bergulir.

Untuk memanfaatkan kemampuan ini dan meminimalkan kemampuan AndaMTTR, pertimbangkan untuk mengotomatiskan semua infrastruktur dan penyebaran kode Anda melalui layanan ini. Dalam skenario di mana Anda tidak dapat menggunakan layanan ini, pertimbangkan untuk menerapkan pola saga dengan AWS Step Functions untuk mengembalikan penerapan yang gagal.

Saat mempertimbangkan restart, ada beberapa pendekatan berbeda. Mulai dari me-reboot server, tugas terpanjang, hingga memulai ulang utas, tugas terpendek. Berikut adalah tabel yang menguraikan beberapa pendekatan restart dan perkiraan waktu untuk menyelesaikan (mewakili urutan perbedaan besarnya, ini tidak tepat).

Mekanisme pemulihan kesalahan	Diperkirakan MTTR
Luncurkan dan konfigurasikan server virtual baru	15 menit
Menerapkan ulang perangkat lunak	10 menit
Reboot server	5 menit
Mulai ulang atau luncurkan wadah	2 detik
Memanggil fungsi tanpa server baru	100 ms

Mekanisme pemulihan kesalahan	Diperkirakan MTTR
Mulai ulang proses	10 ms
Mulai ulang utas	10 µs

Meninjau tabel, ada beberapa manfaat yang jelas untuk MTTR dalam menggunakan wadah dan fungsi tanpa server (seperti). AWS Lambda Mereka MTTR adalah urutan besarnya lebih cepat daripada memulai ulang mesin virtual atau meluncurkan yang baru. Namun, menggunakan isolasi kesalahan melalui modularitas perangkat lunak juga bermanfaat. Jika Anda dapat menahan kegagalan pada satu proses atau utas, memulihkan dari kegagalan itu jauh lebih cepat daripada memulai ulang wadah atau server.

Sebagai pendekatan umum untuk pemulihan, Anda dapat berpindah dari bawah ke atas: 1/Restart, 2/Reboot, 3/Re-image/Redeploy, 4/Replace. Namun, setelah Anda mencapai langkah reboot, merutekan kegagalan biasanya merupakan pendekatan yang lebih cepat (biasanya memakan waktu paling lama 3-4 menit). Jadi, untuk paling cepat mengurangi dampak setelah percobaan restart, rute di sekitar kegagalan, dan kemudian, di latar belakang, lanjutkan proses pemulihan untuk mengembalikan kapasitas ke beban kerja Anda.

> Aturan 10
>
> Fokus pada mitigasi dampak, bukan resolusi masalah. Ambil jalur tercepat kembali ke operasi normal.

Diagnosis kegagalan

Bagian dari proses perbaikan setelah deteksi adalah periode diagnosis. Ini adalah periode waktu di mana operator mencoba menentukan apa yang salah. Proses ini mungkin melibatkan kueri log, meninjau metrik Kesehatan Operasional, atau masuk ke host untuk memecahkan masalah. Semua tindakan ini membutuhkan waktu, jadi membuat alat dan runbook untuk mempercepat tindakan ini dapat membantu mengurangi MTTR juga.

Runbook dan otomatisasi

Demikian pula, setelah Anda menentukan apa yang salah dan tindakan apa yang akan memperbaiki beban kerja, operator biasanya perlu melakukan beberapa set langkah untuk melakukannya.

Misalnya, setelah kegagalan, cara tercepat untuk memperbaiki beban kerja mungkin dengan memulai ulang, yang dapat melibatkan beberapa langkah yang dipesan. Memanfaatkan runbook yang mengotomatiskan langkah-langkah ini atau memberikan arahan khusus kepada operator akan mempercepat proses dan membantu mengurangi risiko tindakan yang tidak disengaja.

Meningkat MTBF

Komponen terakhir untuk meningkatkan ketersediaan adalah meningkatkanMTBF. Ini dapat berlaku untuk perangkat lunak serta AWS layanan yang digunakan untuk menjalankannya.

Meningkatkan sistem terdistribusi MTBF

Salah satu cara untuk meningkatkan MTBF adalah dengan mengurangi cacat pada perangkat lunak. Ada beberapa cara untuk melakukan ini. Pelanggan dapat menggunakan alat seperti Amazon CodeGuru Reviewer untuk menemukan dan memperbaiki kesalahan umum. Anda juga harus melakukan tinjauan kode sejawat yang komprehensif, pengujian unit, tes integrasi, uji regresi, dan uji beban pada perangkat lunak sebelum digunakan untuk produksi. Meningkatkan jumlah cakupan kode dalam pengujian akan membantu memastikan bahwa jalur eksekusi kode yang tidak biasa pun diuji.

Menyebarkan perubahan yang lebih kecil juga dapat membantu mencegah hasil yang tidak terduga dengan mengurangi kompleksitas perubahan. Setiap kegiatan memberikan kesempatan untuk mengidentifikasi dan memperbaiki cacat sebelum mereka dapat dipanggil.

Pendekatan lain untuk mencegah kegagalan adalah pengujian rutin. Menerapkan program rekayasa kekacauan dapat membantu menguji bagaimana beban kerja Anda gagal, memvalidasi prosedur pemulihan, dan membantu menemukan dan memperbaiki mode kegagalan sebelum terjadi dalam produksi. Pelanggan dapat menggunakan AWS Fault Injection Simulator sebagai bagian dari perangkat eksperimen rekayasa kekacauan mereka.

Toleransi kesalahan adalah cara lain untuk mencegah kegagalan dalam sistem terdistribusi. Modul cepat gagal, percobaan ulang dengan backoff dan jitter eksponensial, transaksi, dan idempotensi adalah semua teknik untuk membantu membuat beban kerja toleran terhadap kesalahan.

Transaksi adalah sekelompok operasi yang mematuhi ACID properti. Mereka adalah sebagai berikut:

- Atomisitas — Entah semua tindakan terjadi atau tidak satupun dari mereka akan terjadi.
- Konsistensi — Setiap transaksi meninggalkan beban kerja dalam keadaan valid.
- Isolasi — Transaksi yang dilakukan secara bersamaan meninggalkan beban kerja dalam keadaan yang sama seolah-olah telah dilakukan secara berurutan.

- Daya Tahan — Setelah transaksi dilakukan, semua efeknya dipertahankan bahkan dalam kasus kegagalan beban kerja.

Mencoba lagi dengan backoff eksponensial dan jitter memungkinkan Anda mengatasi kegagalan sementara yang disebabkan oleh Heisenbug, kelebihan beban, atau kondisi lainnya. Ketika transaksi idempoten, mereka dapat dicoba ulang beberapa kali tanpa efek samping.

Jika kita mempertimbangkan efek Heisenbug pada konfigurasi perangkat keras yang toleran terhadap kesalahan, kita akan cukup tidak peduli karena kemungkinan Heisenbug muncul pada subsistem primer dan redundan sangat kecil. (Lihat Jim Gray, "Mengapa Komputer Berhenti dan Apa Yang Dapat Dilakukan Tentang Itu? ", Juni 1985, Laporan Teknis Tandem 85.7.) Dalam sistem terdistribusi, kami ingin mencapai hasil yang sama dengan perangkat lunak kami.

Ketika Heisenbug dipanggil, sangat penting bahwa perangkat lunak dengan cepat mendeteksi operasi yang salah dan gagal sehingga dapat dicoba lagi. Ini dicapai melalui pemrograman defensif, dan memvalidasi input, hasil antara, dan output. Selain itu, proses diisolasi dan tidak berbagi keadaan dengan proses lain.

Pendekatan modular ini memastikan bahwa ruang lingkup dampak selama kegagalan terbatas. Proses gagal secara independen. Ketika suatu proses gagal, perangkat lunak harus menggunakan “pasangan proses” untuk mencoba kembali pekerjaan, yang berarti proses baru dapat mengasumsikan pekerjaan yang gagal. Untuk menjaga keandalan dan integritas beban kerja, setiap operasi harus diperlakukan sebagai ACID transaksi.

Hal ini memungkinkan proses gagal tanpa merusak keadaan beban kerja dengan membatalkan transaksi dan mengembalikan setiap perubahan yang dibuat. Ini memungkinkan proses pemulihan untuk mencoba kembali transaksi dari keadaan yang diketahui baik dan memulai kembali dengan anggun. Ini adalah bagaimana perangkat lunak dapat toleran terhadap kesalahan terhadap Heisenbugs.

Namun, Anda tidak boleh bertujuan untuk membuat perangkat lunak toleran terhadap kesalahan Bohrbugs. Cacat ini harus ditemukan dan dihilangkan sebelum beban kerja memasuki produksi karena tidak ada tingkat redundansi yang akan mencapai hasil yang benar. (Lihat Jim Gray, "Mengapa Komputer Berhenti dan Apa Yang Dapat Dilakukan Tentang Itu? ", Juni 1985, Laporan Teknis Tandem 85.7.)

Cara terakhir untuk meningkatkan MTBF adalah dengan mengurangi ruang lingkup dampak dari kegagalan. Menggunakan isolasi kesalahan melalui modularisasi untuk membuat wadah kesalahan adalah cara utama untuk melakukannya seperti yang diuraikan sebelumnya dalam Toleransi

kesalahan dan isolasi kesalahan. Mengurangi tingkat kegagalan meningkatkan ketersediaan. AWS menggunakan teknik seperti membagi layanan menjadi pesawat kontrol dan pesawat data, Availability Zone Independence (AZI), isolasi Regional, arsitektur berbasis sel, dan shuffle-sharding untuk memberikan isolasi kesalahan. Ini juga pola yang dapat digunakan oleh AWS pelanggan juga.

Sebagai contoh, mari kita tinjau skenario di mana beban kerja menempatkan pelanggan ke dalam wadah kesalahan yang berbeda dari infrastrukturnya yang melayani paling banyak 5% dari total pelanggan. Salah satu wadah kesalahan ini mengalami peristiwa yang meningkatkan latensi di luar batas waktu klien untuk 10% permintaan. Selama acara ini, untuk 95% pelanggan, layanan ini 100% tersedia. Untuk 5% lainnya, layanan tampaknya 90% tersedia. Hal ini menghasilkan ketersediaan 1 − (5% o f c u s t o m e r s × 10% o f t h e i r r e q u e s t s s) = 99,5% bukannya 10% permintaan gagal untuk 100% pelanggan (menghasilkan ketersediaan 90%).

> Aturan 11
>
> Isolasi kesalahan mengurangi cakupan dampak dan meningkatkan beban kerja dengan mengurangi tingkat kegagalan keseluruhan. MTBF

Meningkatkan Ketergantungan MTBF

Metode pertama untuk meningkatkan AWS ketergantungan Anda MTBF adalah dengan menggunakan isolasi kesalahan. Banyak AWS layanan menawarkan tingkat isolasi di AZ, yang berarti kegagalan dalam satu AZ tidak mempengaruhi layanan di AZ yang berbeda.

Menggunakan EC2 instance redundan dalam beberapa AZs meningkatkan ketersediaan subsistem. AZImenyediakan kemampuan hemat dalam satu Wilayah, memungkinkan Anda untuk meningkatkan ketersediaan Anda untuk AZI layanan.

Namun, tidak semua AWS layanan beroperasi di tingkat AZ. Banyak yang menawarkan isolasi regional. Dalam hal ini, di mana ketersediaan layanan regional yang dirancang untuk tidak mendukung ketersediaan keseluruhan yang diperlukan untuk beban kerja Anda, Anda dapat mempertimbangkan pendekatan Multi-wilayah. Setiap Wilayah menawarkan instantiasi layanan yang terisolasi, setara dengan hemat.

Ada berbagai layanan yang membantu membuat membangun layanan Multi-region lebih mudah. Sebagai contoh:

- Basis Data Global Amazon Aurora

- Tabel global Amazon DynamoDB
- Amazon ElastiCache (RedisOSS) - Datastore Global
- AWS Akselerator Global
- Replikasi Lintas Wilayah Amazon S3
- Pengontrol Pemulihan Aplikasi Amazon Route 53

Dokumen ini tidak mempelajari strategi membangun beban kerja Multi-wilayah, tetapi Anda harus mempertimbangkan manfaat ketersediaan arsitektur Multi-wilayah dengan biaya tambahan, kompleksitas, dan praktik operasional yang diperlukan untuk memenuhi tujuan ketersediaan yang Anda inginkan.

Metode selanjutnya untuk meningkatkan ketergantungan MTBF adalah dengan merancang beban kerja Anda agar stabil secara statis. Misalnya, Anda memiliki beban kerja yang menyajikan informasi produk. Ketika pelanggan Anda membuat permintaan untuk suatu produk, layanan Anda membuat permintaan ke layanan metadata eksternal untuk mengambil detail produk. Kemudian beban kerja Anda mengembalikan semua info itu ke pengguna.

Namun, jika layanan metadata tidak tersedia, permintaan yang dibuat oleh pelanggan Anda gagal. Sebagai gantinya, Anda dapat secara asinkron menarik atau mendorong metadata secara lokal ke layanan Anda untuk digunakan untuk menjawab permintaan. Ini menghilangkan panggilan sinkron ke layanan metadata dari jalur kritis Anda.

Selain itu, karena layanan Anda masih tersedia bahkan ketika layanan metadata tidak, Anda dapat menghapusnya sebagai ketergantungan dalam perhitungan ketersediaan Anda. Contoh ini bergantung pada asumsi bahwa metadata tidak sering berubah dan bahwa menyajikan metadata basi lebih baik daripada permintaan yang gagal. Contoh serupa lainnya adalah serve-stale untuk DNS yang memungkinkan data disimpan dalam cache setelah TTL kedaluwarsa dan digunakan untuk tanggapan ketika jawaban yang diperbarui tidak tersedia.

Metode terakhir untuk meningkatkan ketergantungan MTBF adalah dengan mengurangi ruang lingkup dampak dari kegagalan. Seperti dibahas sebelumnya, kegagalan bukanlah peristiwa biner, ada derajat kegagalan. Ini adalah efek modularisasi; kegagalan terkandung hanya pada permintaan atau pengguna yang dilayani oleh wadah itu.

Hal ini mengakibatkan lebih sedikit kegagalan selama suatu peristiwa yang pada akhirnya meningkatkan ketersediaan beban kerja secara keseluruhan dengan membatasi ruang lingkup dampak.

Mengurangi sumber dampak umum

Pada tahun 1985, Jim Gray menemukan, selama studi di Tandem Computers, bahwa kegagalan terutama didorong oleh dua hal: perangkat lunak dan operasi. (Lihat Jim Gray, "Mengapa Komputer Berhenti dan Apa Yang Dapat Dilakukan Tentang Itu? “, Juni 1985, Laporan Teknis Tandem 85.7.) Bahkan setelah 36 tahun kemudian, ini terus menjadi kenyataan. Terlepas dari kemajuan teknologi, tidak ada solusi mudah untuk masalah ini, dan sumber utama kegagalan belum berubah. Mengatasi kegagalan dalam perangkat lunak dibahas di awal bagian ini, jadi fokusnya di sini adalah operasi dan mengurangi frekuensi kegagalan.

Stabilitas dibandingkan dengan fitur

Jika kita merujuk kembali ke tingkat kegagalan untuk grafik perangkat lunak dan perangkat keras di bagian inithe section called “Ketersediaan sistem terdistribusi”, kita dapat melihat bahwa cacat ditambahkan di setiap rilis perangkat lunak. Ini berarti bahwa setiap perubahan pada beban kerja menimbulkan peningkatan risiko kegagalan. Perubahan ini biasanya hal-hal seperti fitur baru, yang memberikan akibat wajar. Beban kerja ketersediaan yang lebih tinggi akan mendukung stabilitas dibandingkan fitur baru. Dengan demikian, salah satu cara paling sederhana untuk meningkatkan ketersediaan adalah dengan menggunakan lebih jarang atau memberikan lebih sedikit fitur. Beban kerja yang diterapkan lebih sering secara inheren akan memiliki ketersediaan yang lebih rendah daripada yang tidak. Namun, beban kerja yang gagal menambahkan fitur tidak sesuai dengan permintaan pelanggan dan dapat menjadi kurang berguna dari waktu ke waktu.

Jadi, bagaimana kita terus berinovasi dan merilis fitur dengan aman? Jawabannya adalah standardisasi. Apa cara yang benar untuk menyebarkan? Bagaimana Anda memesan penerapan? Apa standar untuk pengujian? Berapa lama Anda menunggu di antara tahapan? Apakah pengujian unit Anda cukup mencakup kode perangkat lunak? Ini adalah pertanyaan yang akan dijawab oleh standardisasi dan mencegah masalah yang disebabkan oleh hal-hal seperti tidak memuat pengujian, melewatkan tahapan penerapan, atau menyebarkan terlalu cepat ke terlalu banyak host.

Cara Anda menerapkan standardisasi adalah melalui otomatisasi. Ini mengurangi kemungkinan kesalahan manusia dan memungkinkan komputer melakukan hal yang mereka kuasai, yang melakukan hal yang sama berulang kali dengan cara yang sama setiap saat. Cara Anda menyatukan standardisasi dan otomatisasi adalah dengan menetapkan tujuan. Sasaran seperti tidak ada perubahan manual, akses host hanya melalui sistem otorisasi kontingen, menulis tes beban untuk setiapAPI, dan sebagainya. Keunggulan operasional adalah norma budaya yang membutuhkan perubahan besar. Membangun dan melacak kinerja terhadap suatu tujuan membantu mendorong perubahan budaya yang akan berdampak luas pada ketersediaan beban kerja. Pilar AWS Well-

Architected Operational Excellence memberikan praktik terbaik yang komprehensif untuk keunggulan operasional.

Keamanan operator

Kontributor utama lainnya untuk peristiwa operasional yang menyebabkan kegagalan adalah orang-orang. Manusia membuat kesalahan. Mereka mungkin menggunakan kredensil yang salah, memasukkan perintah yang salah, menekan Enter terlalu cepat, atau melewatkan langkah kritis. Mengambil tindakan manual secara konsisten menghasilkan kesalahan, yang mengakibatkan kegagalan.

Salah satu penyebab utama kesalahan operator adalah antarmuka pengguna yang membingungkan, tidak intuitif, atau tidak konsisten. Jim Gray juga mencatat dalam studinya tahun 1985 bahwa “antarmuka yang meminta informasi kepada operator atau memintanya untuk melakukan beberapa fungsi harus sederhana, konsisten, dan toleran terhadap kesalahan operator.” (Lihat Jim Gray, "Mengapa Komputer Berhenti dan Apa Yang Dapat Dilakukan Tentang Itu? “, Juni 1985, Laporan Teknis Tandem 85.7.) Wawasan ini terus menjadi kenyataan hari ini. Ada banyak contoh selama tiga dekade terakhir di seluruh industri di mana antarmuka pengguna yang membingungkan atau kompleks, kurangnya konfirmasi atau instruksi, atau bahkan hanya bahasa manusia yang tidak ramah menyebabkan operator melakukan hal yang salah.

> Aturan 12
>
> Memudahkan operator untuk melakukan hal yang benar.

Mencegah kelebihan beban

Kontributor dampak umum terakhir adalah pelanggan Anda, pengguna sebenarnya dari beban kerja Anda. Beban kerja yang berhasil cenderung digunakan, banyak, tetapi terkadang penggunaan itu melebihi kemampuan beban kerja untuk skala. Ada banyak hal yang bisa terjadi, disk bisa menjadi penuh, kumpulan thread mungkin habis, bandwidth jaringan mungkin jenuh, atau batas koneksi database dapat dicapai.

Tidak ada metode gagal untuk menghilangkan ini, tetapi pemantauan proaktif kapasitas dan pemanfaatan melalui metrik Kesehatan Operasional akan memberikan peringatan dini ketika kegagalan ini mungkin terjadi. Teknik seperti pelepasan beban, pemutus sirkuit, dan coba lagi dengan backoff eksponensial dan jitter dapat membantu meminimalkan dampak dan meningkatkan tingkat keberhasilan, tetapi situasi ini masih merupakan kegagalan. Penskalaan otomatis

berdasarkan metrik Kesehatan Operasional dapat membantu mengurangi frekuensi kegagalan karena kelebihan beban, tetapi mungkin tidak dapat merespons dengan cukup cepat terhadap perubahan pemanfaatan.

Jika Anda perlu memastikan kapasitas yang tersedia secara terus menerus untuk pelanggan, Anda harus melakukan pengorbanan pada ketersediaan dan biaya. Salah satu cara untuk memastikan kurangnya kapasitas tidak menyebabkan tidak tersedianya adalah dengan menyediakan setiap pelanggan dengan kuota dan memastikan kapasitas beban kerja Anda diskalakan untuk memberikan 100% dari kuota yang dialokasikan. Ketika pelanggan melebihi kuota mereka, mereka terhambat, yang bukan merupakan kegagalan dan tidak dihitung terhadap ketersediaan. Anda juga perlu melacak basis pelanggan Anda dengan cermat dan memperkirakan pemanfaatan masa depan untuk menjaga kapasitas yang cukup disediakan. Ini memastikan beban kerja Anda tidak didorong ke skenario kegagalan melalui konsumsi berlebihan oleh pelanggan Anda.

- Amazon Builders' Library — Menggunakan load shedding untuk menghindari kelebihan beban
- Amazon Builders' Library — Keadilan dalam sistem multi-penyewa

Misalnya, mari kita periksa beban kerja yang menyediakan layanan penyimpanan. Setiap server dalam beban kerja dapat mendukung 100 unduhan per detik, pelanggan diberikan kuota atau 200 unduhan per detik, dan ada 500 pelanggan. Untuk dapat mendukung volume pelanggan ini, layanan perlu menyediakan kapasitas 100.000 unduhan per detik, yang membutuhkan 1.000 server. Jika ada pelanggan yang melebihi kuota mereka, mereka terhambat, yang memastikan kapasitas yang cukup untuk setiap pelanggan lainnya. Ini adalah contoh sederhana dari salah satu cara untuk menghindari kelebihan beban tanpa menolak unit kerja.

Kesimpulan

Kami menetapkan 12 aturan untuk ketersediaan tinggi di seluruh dokumen ini.

- Aturan 1 - Kegagalan yang lebih jarang (MTBF lebih lama), waktu deteksi kegagalan yang lebih pendek (MTTD yang lebih pendek), dan waktu perbaikan yang lebih pendek (MTTR yang lebih pendek) adalah tiga faktor yang digunakan untuk meningkatkan ketersediaan dalam sistem terdistribusi.
- Aturan 2 — Ketersediaan perangkat lunak dalam beban kerja Anda merupakan faktor penting dari ketersediaan beban kerja Anda secara keseluruhan dan harus menerima fokus yang sama seperti komponen lainnya.
- Aturan 3 - Mengurangi dependensi dapat berdampak positif pada ketersediaan.
- Aturan 4 — Secara umum, pilih dependensi yang sasaran ketersediaannya sama atau lebih besar dari tujuan beban kerja Anda.
- Aturan 5 - Gunakan hemat untuk meningkatkan ketersediaan dependensi dalam beban kerja.
- Aturan 6 - Ada batas atas untuk efisiensi biaya hemat. Memanfaatkan suku cadang paling sedikit yang diperlukan untuk mencapai ketersediaan yang diperlukan.
- Aturan 7 - Jangan mengambil dependensi pada bidang kontrol di bidang data Anda, terutama selama pemulihan.
- Aturan 8 - Ketergantungan pasangan longgar sehingga beban kerja Anda dapat beroperasi dengan benar meskipun ada gangguan ketergantungan, jika memungkinkan.
- Aturan 9 — Observabilitas dan instrumentasi sangat penting untuk mengurangi MTTD dan MTTR.
- Aturan 10 — Fokus pada mitigasi dampak, bukan penyelesaian masalah. Ambil jalur tercepat kembali ke operasi normal.
- Aturan 11 - Isolasi kesalahan mengurangi ruang lingkup dampak dan meningkatkan MTBF beban kerja dengan mengurangi tingkat kegagalan keseluruhan.
- Aturan 12 - Memudahkan operator untuk melakukan hal yang benar.

Meningkatkan ketersediaan beban kerja didorong melalui pengurangan MTTD dan MTTR, dan meningkatkan MTBF. Singkatnya, kami membahas cara-cara berikut untuk meningkatkan ketersediaan yang mencakup teknologi, orang, dan proses.

- MTTD
  - Kurangi MTTD melalui pemantauan proaktif metrik Pengalaman Pelanggan Anda.

    - Manfaatkan pemeriksaan kesehatan granular untuk failover cepat.
- MTTR
    - Memantau Lingkup Dampak dan metrik Kesehatan Operasional.
    - Kurangi MTTR dengan mengikuti 1/Restart, 2/Reboot, 3/Re-image/Redeploy, dan 4/Replace.
    - Rute sekitar kegagalan dengan memahami lingkup dampak.
    - Memanfaatkan layanan yang memiliki waktu restart lebih cepat, seperti kontainer dan fungsi tanpa server melalui mesin virtual atau host fisik.
    - Secara otomatis rollback penyebaran gagal bila memungkinkan.
    - Menetapkan runbook dan alat operasional untuk operasi diagnosis dan prosedur restart.
- MTBF
    - Hilangkan bug dan cacat pada perangkat lunak melalui pengujian yang ketat sebelum dilepaskan ke produksi.
    - Menerapkan rekayasa kekacauan dan injeksi kesalahan.
    - Memanfaatkan jumlah yang tepat hemat dalam dependensi untuk mentolerir kegagalan.
    - Minimalkan ruang lingkup dampak selama kegagalan melalui kontainer kesalahan.
    - Menerapkan standar untuk penyebaran dan perubahan.
    - Desain antarmuka operator yang sederhana, intuitif, konsisten, dan terdokumentasi dengan baik.
    - Tetapkan tujuan untuk keunggulan operasional.
    - Mendukung stabilitas atas rilis fitur baru ketika ketersediaan adalah dimensi penting dari beban kerja Anda.
    - Menerapkan kuota penggunaan dengan throttling atau load shedding atau keduanya untuk menghindari kelebihan beban.

Ingatlah bahwa kita tidak akan pernah berhasil sepenuhnya dalam mencegah kegagalan. Fokus pada desain perangkat lunak dengan isolasi kegagalan sebaik mungkin yang membatasi ruang lingkup dan besarnya dampak, idealnya menjaga dampak di bawah ambang “downtime” DAN berinvestasi dalam deteksi dan mitigasi yang sangat cepat, sangat andal. Sistem terdistribusi modern masih perlu merangkul kegagalan sebagai tak terelakkan dan dirancang di semua tingkatan untuk ketersediaan tinggi.

Lampiran 1 - Metrik kritis MTTD dan MTTR

Berikut ini adalah kerangka kerja untuk standardisasi dalam instrumentasi dan observability yang dapat membantu mengurangi MTTD dan MTTR selama suatu acara.

Metrik Pengalaman Pelanggan. Metrik ini mencerminkan bahwa layanan responsif dan tersedia untuk melayani permintaan pelanggan. Misalnya, latensi bidang kontrol. Metrik ini mengukur tingkat kesalahan, ketersediaan, latensi, volume, dan laju throttle.

Metrik Penilaian Dampak. Metrik ini memberikan wawasan tentang ruang lingkup dampak selama peristiwa. Misalnya, jumlah atau persentase pelanggan yang terkena dampak peristiwa bidang data. Mengukur jumlah atau persentase hal yang terkena dampak.

Metrik Kesehatan Operasional. Metrik ini mencerminkan bahwa layanan responsif dan tersedia untuk melayani permintaan pelanggan, tetapi berfokus pada subsistem infrastruktur umum dan sumber daya. Misalnya, persentase pemanfaatan CPU armada EC2 Anda. Metrik ini harus mengukur pemanfaatan, kapasitas, throughput, tingkat kesalahan, ketersediaan, dan latensi.

Kontributor

Kontributor untuk dokumen ini meliputi:

- Michael Haken, Arsitek Solusi Utama, Layanan Web Amazon

Bacaan lebih lanjut

Untuk informasi tambahan, lihat:

- Pilar Keandalan yang Dirancang dengan Baik
- Pilar Keunggulan Operasional yang Didesain dengan Baik
- Perpustakaan Amazon Builders — Memastikan keamanan rollback selama penerapan
- Perpustakaan Amazon Builders — Lebih dari lima 9 d: Pelajaran dari bidang data tertinggi kami yang tersedia
- Perpustakaan Amazon Builders — Mengotomatiskan penerapan hands-off yang aman
- Perpustakaan Amazon Builders — Merancang dan mengoperasikan sistem tanpa server yang tangguh dalam skala besar
- Perpustakaan Amazon Builders — Pendekatan Amazon terhadap penerapan ketersediaan tinggi
- Amazon Builders 'Library — Pendekatan Amazon untuk membangun layanan tangguh
- Perpustakaan Amazon Builders — Pendekatan Amazon untuk gagal berhasil
- AWSPusat Arsitektur

Riwayat dokumen

Untuk diberi tahu tentang pembaruan pada whitepaper ini, berlangganan umpan RSS.

Perubahan	Deskripsi	Tanggal
Publikasi awal	Whitepaper pertama kali diterbitkan.	November 12, 2021

> **Note**
>
> Untuk berlangganan pembaruan RSS, Anda harus mengaktifkan plug-in RSS untuk browser yang Anda gunakan.

Pemberitahuan

Pelanggan bertanggung jawab untuk membuat penilaian independen mereka sendiri terhadap informasi dalam dokumen ini. Dokumen ini: (a) hanya untuk tujuan informasi, (b) mewakili penawaran dan praktik AWS produk saat ini, yang dapat berubah tanpa pemberitahuan, dan (c) tidak menciptakan komitmen atau jaminan apa pun dari AWS dan afiliasinya, pemasok atau pemberi lisensinya. AWSproduk atau layanan disediakan “sebagaimana adanya” tanpa jaminan, pernyataan, atau kondisi apa pun, baik tersurat maupun tersirat. Tanggung jawab dan kewajiban AWS kepada pelanggannya dikendalikan oleh AWS perjanjian, dan dokumen ini bukan bagian dari, juga tidak memodifikasi, perjanjian apa pun antara AWS dan pelanggannya.

AWSGlosarium

Untuk AWS terminologi terbaru, lihat AWSglosarium di Referensi. Glosarium AWS

Terjemahan disediakan oleh mesin penerjemah. Jika konten terjemahan yang diberikan bertentangan dengan versi bahasa Inggris aslinya, utamakan versi bahasa Inggris.