Engineer

Exponentiell backoff

⏳ en paus mellan omförsök som multipliceras efter varje fel

Vad det är

Exponentiell backoff är en regel för hur länge man ska vänta innan man försöker igen. Efter varje misslyckat försök multipliceras väntetiden med en fast faktor, oftast två, tills den når ett tak: en sekund, sedan två, fyra, åtta och så vidare. Regeln bor i klienten, men den finns till för det som klienten anropar. En server, en nätverksväg eller en databas som fallerar under för hög last återhämtar sig inte fortare av att bli tillfrågad igen direkt. Den återhämtar sig när de som anropar den lättar på trycket, och en väntetid som växer med varje fel får dem att lätta mer ju längre problemet varar.

Det äldsta och mest spridda exemplet finns i TCP. RFC 6298, dokumentet i Standards Track från juni 2011 som specificerar hur TCP beräknar sin tidsgräns för omsändning (retransmission timeout), sätter den tidsgränsen till en sekund innan den första tur och retur-tiden (round-trip time) har mätts, och säger att ett beräknat värde under en sekund bör avrundas uppåt. Varje gång timern löper ut och ett segment måste skickas igen, måste värden fördubbla tidsgränsen. En övre gräns är tillåten, så länge den är minst 60 sekunder. När en ny mätning av tur och retur-tiden kommer in, beräknas tidsgränsen från mätningarna igen, vilket kan föra ner den direkt efter en rad fördubblingar. Algoritmen går tillbaka till Van Jacobsons artikel Congestion Avoidance and Control från 1988, och avsnittet om säkerhet i RFC:n säger vad som står på spel: internet är beroende av en korrekt timer för omsändning för att hålla nätet stabilt och undvika kollaps på grund av överbelastning (congestion collapse).

I applikationskod kommer samma idé med en handfull inställningar: en inledande väntetid, en multiplikator, en maximal väntetid där tillväxten stannar och en gräns för antalet försök eller för den sammanlagda tiden som läggs på omförsök. Skrivet som formel är väntetiden efter det n-te felet det minsta av två värden: taket, och grundväntetiden multiplicerad med multiplikatorn upphöjd till n, där n räknas från noll. AWS-analysen som diskuteras nedan kallar detta capped

exponential backoff, alltså exponentiell backoff med tak. Taket betyder mer än det ser ut. Fördubblar man en väntetid på en sekund tio gånger blir det 1 024 sekunder, mer än 17 minuter, och det är längre än de flesta som anropar något är beredda att vänta.

Den sista inställningen är jitter, ett slumpmässigt inslag i varje väntetid. Protokollet för connection backoff i gRPC, som beskriver hur gRPC-klienter ansluter på nytt efter en misslyckad anslutning, börjar på en sekund, multiplicerar med 1,6, slutar växa vid 120 sekunder och använder en jitter på 0,2, så att varje väntetid flyttas slumpmässigt upp eller ner med upp till 20 %. Det ger också varje anslutningsförsök minst 20 sekunder att bli klart, och det återställer backoff till startvärdet så snart servern har accepterat en anslutning, vilket det räknar som ögonblicket då SETTINGS-ramen kommer fram. Protokollet tillåter andra implementationer, till exempel på en mobiltelefon som bör vakna så sällan som möjligt, men kräver att backoff-perioder som började samtidigt sprids ut och att ingen implementation försöker ansluta väsentligt oftare.

Använd begreppet i ett beslut

Använd ”Vad är exponentiell backoff?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Övervakning och larm som säger till medan ett fel fortfarande är litet, och siffrorna som visar hur ofta det förblir så.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Byggde företagets första observerbarhetsdashboard — företagets första överblick över de egna systemen

Byggde företagets första observerbarhetsdashboard Hör av er →

Varför det spelar roll

Ett omförsök är extra last, och det kommer i sämsta tänkbara ögonblick: precis när det som anropas fallerar. Om en tjänst blir långsammare för att den är överbelastad, och alla som anropar den svarar med att försöka igen direkt, stiger lasten medan kapaciteten sjunker. Därför räknar dokumentationen för Google Cloud Storage omförsök utan backoff som ett antimönster: omedelbara eller mycket korta omförsök kan leda till kaskadfel, där ett fel utlöser andra. Med backoff blir omförsöket en broms i stället för en förstärkare: ju längre problemet varar, desto mindre lägger varje klient till.

Backoff räcker inte ensam, för klienter som fallerar tillsammans väntar också tillsammans. I Exponential Backoff And Jitter, publicerad av AWS i mars 2015, simulerade Marc Brooker många klienter som använde optimistisk

samtidighetskontroll (optimistic concurrency control) för att uppdatera samma rad i en fjärrdatabas. När varje klient konkurrerade i varje runda växte det sammanlagda arbetet med kvadraten på antalet klienter. Exponentiell backoff med tak hjälpte bara lite. Anropen kom fortfarande i klungor, med tysta luckor emellan, så konkurrensen minskade knappt. Att lägga till jitter, genom att dra varje väntetid slumpmässigt mellan noll och det exponentiella värdet med tak, tog bort luckorna, gav en ungefär jämn takt av anrop och minskade antalet anrop med mer än hälften i fallet med 100 konkurrerande klienter. Av varianterna han jämförde var backoff utan jitter med Brookers ord ”den klara förloraren”. Han var också tydlig med gränsen: ingen av metoderna ändrar det faktum att arbetet växer med kvadraten på konkurrensen. Jitter minskar slöseriet; den tar inte bort konflikten. I en uppdatering från maj 2023 skriver inlägget att de flesta AWS SDK:er nu har exponentiell backoff och jitter i sina standard- och adaptiva lägen för omförsök.

Backoff avgör när man ska försöka igen, inte om det är säkert att göra det. Dokumentationen för Cloud Storage ställer två frågor först. Den ena är om svaret pekar på ett övergående problem: HTTP-statuskoderna 408, 429 och 5xx, timeouter på socketen och brutna TCP-anslutningar. Den andra är om förfrågan är idempotent. En listoperation kan upprepas fritt. Att skapa en Pub/Sub-avisering kan inte det, eftersom varje lyckat anrop skapar en ny avisering. Mellan de två ligger villkorligt idempotenta operationer, som blir säkra att försöka igen när de har ett förhandsvillkor, till exempel en uppladdning som bara lyckas om objektets generationsnummer fortfarande är det som klienten förväntar sig.

Ibland vet servern bättre än klientens formel. Headerfältet Retry-After, definierat i RFC 9110, talar om för klienten hur länge den bör vänta innan en uppföljande förfrågan, antingen som ett datum eller som ett antal sekunder, och tillsammans med svaret 503 Service Unavailable säger det hur länge tjänsten räknar med att vara otillgänglig. Statuskoden 429 Too Many Requests från RFC 6585, som signalerar hastighetsbegränsning, får bära samma fält. En klient med en backoff-loop kan använda det värdet i stället för sin egen beräknade väntetid, eftersom det kommer från den sida som kan se sin egen last.

Så ser det ut i praktiken

En loop för omförsök med exponentiell backoff med tak och full jitter får plats på några få rader Python:

```
import random
import time

class TransientError(Exception):
    """Timeout, ett 408-, 429- eller 5xx-svar eller en bruten anslutning."""

def call_with_backoff(op, attempts=5, base=1.0, cap=30.0):
    for attempt in range(attempts):
        try:
            return op()
        except TransientError:
            if attempt == attempts - 1:
                raise
            ceiling = min(cap, base * 2**attempt)
            time.sleep(random.uniform(0, ceiling))
```

Med de här värdena skiljs fem försök åt av fyra väntetider med tak på 1, 2, 4 och 8 sekunder. Loopen sover därför i högst 15 sekunder innan den ger upp, och i genomsnitt i ungefär 7,5 sekunder, eftersom varje väntetid dras jämnt mellan noll och sitt tak. Bara `TransientError` leder till ett nytt försök; ett auktoriseringsfel eller en felformad förfrågan kastar undantaget direkt. Och loopen upprepar allt som `op` gör, så `op` måste tåla att upprepas: en läsning, eller en skrivning som servern kan känna igen som en upprepning.

Klientbibliotek har egna versioner av den här loopen, och deras standardvärden skiljer sig åt, även för samma tjänst. För Cloud Storage börjar Python-klienten på en sekund, fördubblar varje väntetid upp till 60 sekunder och ger upp efter en standardtidsfrist på 120 sekunder. C++-klienten börjar också på en sekund och fördubblar, men väntar upp till fem minuter mellan försöken och fortsätter med omförsök i upp till 15 minuter, mer än sju gånger så länge. Den försöker också igen med operationer som aldrig är idempotenta, om inte programmet väljer en strikt idempotenspolicy. Kommandoradsverktyget gcloud använder det som Google kallar truncated binary exponential backoff, med upp till 32 omförsök som standard. Två program som anropar samma API med samma fel kan alltså rapportera det efter två minuter eller efter en kvart, beroende på vilket språk de är skrivna i.

Väntan behöver inte blockera en tråd. I bakgrundsarbete kan ett jobb som misslyckas läggas tillbaka i en kö med en senare tidpunkt för körning, och då blir backoff en egenskap hos jobbet och inte hos en process som sover. Det är den naturliga platsen för den när ingen väntar på svaret.

I e-post med failover och leveransloggning, från 2025, sitter backoff på den asynkrona sidan av systemet. Meddelandena skickas genom tre leverantörer, med SendGrid som primär och SMTP2GO och Azure Communication Services i reserv, och failover mellan dem sker automatiskt. Själva sändningen går genom en klient med en timeout på 30 sekunder, och den asynkrona leveransvägen gör nya försök med backoff, så att en kort störning inte kostar ett meddelande. Portföljsidan sammanfattar resultatet: när en leverantör har en dålig eftermiddag fortsätter e-posten i begränsad drift i stället för att drabbas av ett avbrott. Den nämner också priset, som är tre integrationer att underhålla i stället för en.

Var det går fel

Alla klienter väntar lika länge. Utan jitter försöker klienter som fallerade tillsammans igen tillsammans, och servern ser samma topp igen efter en, två, fyra och åtta sekunder. Brookers simulering visade att vanlig exponentiell backoff av det skälet knappt minskade konkurrensen. Lägg till jitter: i hans jämförelse minskade både Full Jitter och Decorrelated Jitter klientarbetet och serverlasten avsevärt, och båda är ett vettigt standardval.

Väntetiden har inget tak, eller omförsöken tar aldrig slut. En fördubbling utan tak passerar 17 minuter efter tio fel, och en loop utan gräns håller en förfrågan vid liv långt efter att den som väntade på svaret har gett upp. Google räknar det att ignorera gränser för omförsök som ett antimönster: att försöka igen i all oändlighet kan tömma applikationens resurser eller fortsätta skicka förfrågningar till en tjänst som inte återhämtar sig utan ingrepp. Sätt en maximal väntetid och en sammanlagd tidsfrist som passar den som anropar: korta där latens spelar roll och längre för batcharbete.

Permanenta fel får nya försök. Ett auktoriseringsfel eller en ogiltig förfrågan misslyckas på samma sätt varje gång. Ett nytt försök skjuter bara upp felet, och utan en gräns kan klienten hamna i en oändlig loop, som dokumentationen för Cloud Storage varnar för. Dela in felen i övergående och permanenta före loopen, försök bara igen med de övergående, som 408, 429, 5xx och brutna anslutningar, och logga resten med orsaken.

En operation som inte är idempotent upprepas i blindo. Om en förfrågan fick timeout efter att servern redan hade utfört den, upprepar ett nytt försök handlingen, något som Googles lista över antimönster beskriver som oavsiktliga

överskrivningar eller raderingar av data. Försök bara igen med idempotenta operationer, gör skrivningar villkorliga där API:et tillåter det och se annars till att den mottagande sidan kan upptäcka en dubblett.

Omförsök ligger i flera lager. Om applikationen gör tre försök och klientbiblioteket under den gör tre försök för vart och ett av de försöken, blir ett fel nio försök, vilket är Googles eget exempel, och varje ytterligare lager som försöker igen på egen hand multiplicerar antalet igen. Dokumentationen tillägger att så många omförsök kan leda till strypning av förfrågningar (throttling) och högre latens utan att andelen lyckade förfrågningar blir bättre. Försök igen i ett lager, oftast i biblioteket, och stäng av eller skruva ner omförsöken i de andra.

Backoff återställs aldrig, eller ingen ser den. En klient som behåller sin senaste, långa väntetid efter att allt har återhämtat sig sitter av hela maxtiden vid nästa lilla fel; gRPC går tillbaka till den inledande backoff-tiden så snart en anslutning har accepterats, och TCP beräknar sin timer på nytt från färska mätningar. Återställ efter ett lyckat anrop. Och eftersom omförsök döljer problem för användarna döljer de dem också för teamet. Räkna dem i övervakningen och låt ett larm gå när andelen omförsök stiger, för ett beroende som behöver tre försök per anrop fallerar redan.

Läs mer

- RFC 6298: Computing TCP’s Retransmission Timer — reglerna i Standards Track för tidsgränsen för omsändning i TCP, inklusive fördubblingen varje gång timern löper ut.
- gRPC Connection Backoff Protocol — algoritmen för återanslutning, dess fem parametrar med värden och när backoff återställs.
- Exponential Backoff And Jitter — Marc Brookers simulering från 2015 av konkurrerande klienter, som jämför backoff utan jitter med Full, Equal och Decorrelated Jitter.
- Retry strategy — vad Cloud Storage räknar som säkert att försöka igen, standardvärdena i varje klientbibliotek och fem antimönster för omförsök.
- RFC 9110: HTTP Semantics — fältet Retry-After, med vilket en server talar om för en klient hur länge den ska vänta före nästa förfrågan.



Ordlista

Välj utifrån era faktiska förutsättningar

Ta med driftsättningsvägen, driftsriskerna och dem som har jour. Vi kan avgränsa en förändring med kontroller, en väg tillbaka och tydligt ansvar.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är exponentiell backoff?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

welcome@engineer.company +45 50 10 61 84 Köpenhamn, Danmark

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/exponential-backoff/