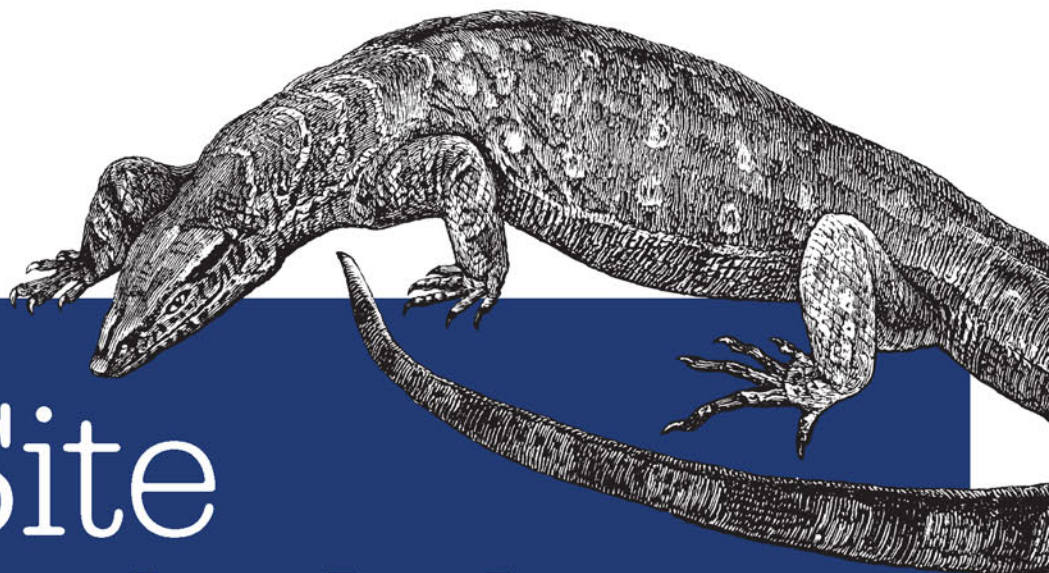




Site Reliability Engineering

JAK GOOGLE ZARZĄDZA SYSTEMAMI PRODUKCYJNYMI

Tytuł oryginału: Site Reliability Engineering: How Google Runs Production Systems

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-3730-5

© 2017 Helion SA

Authorized Polish translation of the English edition of Site Reliability Engineering ISBN 9781491929124 © 2016 Google, Inc.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/sireen>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Przedmowa	13
-----------------	----

Wstęp	15
-------------	----

Część I. Wprowadzenie 23

1. Wprowadzenie	25
-----------------------	----

Zarządzanie usługami przez administratorów systemów	25
---	----

Podejście do zarządzania usługami w Google’u — Site Reliability Engineering	26
---	----

Założenia podejścia SRE	29
-------------------------	----

Koniec początku	34
-----------------	----

2. Środowisko produkcyjne w Google’u z perspektywy SRE	35
--	----

Sprzęt	35
--------	----

Oprogramowanie „organizujące” pracę sprzętu	37
---	----

Inne systemy oprogramowania	40
-----------------------------	----

Infrastruktura dla oprogramowania	40
-----------------------------------	----

Nasze środowisko programistyczne	41
----------------------------------	----

Shakespeare — przykładowa usługa	42
----------------------------------	----

Część II. Zasady 45

3. Akceptowanie ryzyka	47
------------------------------	----

Zarządzanie ryzykiem	47
----------------------	----

Pomiar ryzyka związanego z usługą	48
-----------------------------------	----

Tolerancja ryzyka dla usługi	50
------------------------------	----

Uzasadnienie stosowania budżetu błędów	54
--	----

4. Poziomy SLO	59
Terminologia związana z poziomem usług	59
Wskaźniki w praktyce	62
Cele w praktyce	65
Umowy w praktyce	68
5. Eliminowanie harówki	69
Definicja harówki	69
Dlaczego ograniczenie harówki jest korzystne?	71
Co kwalifikuje się jako prace inżynierskie?	72
Czy harówka zawsze jest czymś złym?	72
Wniosek	74
6. Monitorowanie systemów rozproszonych	75
Definicje	75
Po co monitorować?	76
Wyznaczanie rozsądnych oczekiwań względem monitorowania	77
Symptomy a przyczyny	78
Monitorowanie czarnoskrzynkowe i białoskrzynkowe	79
Cztery złote sygnały	79
Uwzględnianie wartości skrajnych (lub narzędzia pomiarowe i sprawność)	81
Określanie odpowiedniej szczegółowości pomiarów	81
Tak proste jak to możliwe, ale nie prostsze	82
Łączenie opisanych zasad	82
Monitorowanie długoterminowe	83
Wnioski	86
7. Ewolucja automatyzacji w Google’u	87
Wartość automatyzacji	87
Wartość dla zespołów SRE w Google’u	89
Przypadki zastosowania automatyzacji	90
Wyautomatyzuj się z pracy — automatyzuj WSZYSTKO!	93
Łagodzenie problemów — stosowanie automatyzacji do uruchamiania klastrów	95
Borg — narodziny komputera na skalę hurtowni	101
Podstawową cechą jest niezawodność	102
Zalecenia	103
8. Inżynieria udostępniania	105
Rola inżyniera udostępniania	105
Filozofia	106
Ciągłe budowanie i wdrażanie	108
Zarządzanie konfiguracją	111
Wnioski	112

9. Prostota	115
Stabilność a elastyczność systemu	115
Cnota nudy	116
Nie oddam mojego kodu!	116
Wskaźnik „negatywne wiersze kodu”	117
Minimalne interfejsy API	117
Modułowość	117
Udostępnianie prostych zmian	118
Prosty wniosek	118

Część III. Praktyki 119

10. Praktyczne alarmy na podstawie szeregów czasowych	123
Powstanie systemu Borgmon	124
Narzędzia pomiarowe w aplikacji	125
Zbieranie eksportowanych danych	126
Przechowywanie danych w obszarze szeregów czasowych	126
Sprawdzanie reguł	129
Alarmy	132
Podział systemu monitorowania	133
Monitorowanie czarnoskrzynkowe	134
Zarządzanie konfiguracją	135
Dziesięć lat później...	136
11. Dyżury na wezwanie	139
Wprowadzenie	139
Życie inżyniera dyżurnego	140
Zrównoważone dyżury	141
Poczucie bezpieczeństwa	142
Unikanie nieodpowiedniego obciążenia operacyjnego	144
Wnioski	145
12. Skuteczne rozwiązywanie problemów	147
Teoria	148
Praktyka	150
Wyniki negatywne to magia	156
Studium przypadku	158
Ułatwianie rozwiązywania problemów	162
Wnioski	162

13. Reagowanie kryzysowe	163
Co robić, gdy w systemie wystąpi awaria?	163
Kryzys wywołany testami	164
Sytuacje kryzysowe spowodowane zmianami	165
Sytuacje kryzysowe spowodowane procesem	167
Dla każdego problemu istnieje rozwiązanie	169
Wyciągaj wnioski z przeszłości i nie powtarzaj tych samych błędów	170
Wnioski	170
14. Zarządzanie incydentami	173
Niezarządzane incydenty	173
Anatomia niezarządzanego incydentu	174
Aspekty procesu zarządzania incydentami	175
Zarządzany incydent	176
Kiedy ogłaszać incydent?	177
Podsumowanie	178
15. Kultura analizy zdarzeń — wyciąganie wniosków z niepowodzeń	179
Filozofia analizy zdarzeń w Google’u	179
Współpracuj i dziel się wiedzą	181
Wprowadzanie kultury analizy zdarzeń	182
Wnioski i wprowadzane usprawnienia	184
16. Śledzenie przestojów	185
Escalator	185
Outalator	186
17. Testowanie niezawodności	191
Rodzaje testów oprogramowania	192
Tworzenie środowiska testowania i środowiska budowania	198
Testowanie w dużej skali	199
Wnioski	210
18. Inżynieria oprogramowania w SRE	211
Dlaczego inżynieria oprogramowania w zespołach SRE ma znaczenie?	211
Studium przypadku. Auxon — wprowadzenie do projektu i przestrzeń problemowa	213
Planowanie przepustowości na podstawie celów	215
Wspomaganie inżynierii oprogramowania w SRE	223
Wnioski	227

19. Równoważenie obciążenia na poziomie frontonu	229
Moc nie jest rozwiązaniem	229
Równoważenie obciążenia z użyciem systemu DNS	230
Równoważenie obciążenia na poziomie wirtualnych adresów IP	232
20. Równoważenie obciążenia w centrum danych	235
Scenariusz idealny	236
Identyfikowanie problematycznych zadań — kontrola przepływu i „kulawe kaczkę”	237
Ograniczanie puli połączeń za pomocą tworzenia podzbiorów	239
Reguły równoważenia obciążenia	244
21. Obsługa przeciążenia	251
Pułapki związane z „liczbą zapytań na sekundę”	251
Limity na klienta	252
Ograniczanie liczby żądań po stronie klienta	253
Poziom krytyczności	255
Sygnały poziomu wykorzystania	256
Obsługa błędów przeciążenia	257
Obciążenie wynikające z połączeń	260
Wnioski	261
22. Radzenie sobie z awariami kaskadowymi	263
Przyczyny awarii kaskadowych i projektowanie z myślą o ich uniknięciu	264
Zapobieganie przeciążeniu serwerów	268
Powolny rozruch i pusta pamięć podręczna	276
Warunki wywołujące awarie kaskadowe	279
Testowanie pod kątem awarii kaskadowych	280
Pierwsze kroki w obliczu awarii kaskadowych	283
Uwagi końcowe	285
23. Zarządzanie krytycznym stanem — zapewnianie niezawodności za pomocą konsensusu w środowisku rozproszonym	287
Uzasadnienie uzgadniania konsensusu — niepowodzenie koordynacji systemów rozproszonych	289
Jak działa konsensus w środowisku rozproszonym?	291
Wzorce architektury systemu związane z konsensem w środowisku rozproszonym	292
Wydajność uzgadniania konsensusu w środowisku rozproszonym	297
Wdrażanie rozproszonego systemu opartego na konsensusie	305
Monitorowanie rozproszonych systemów uzgadniania konsensusu	312
Wnioski	314

24. Okresowe szeregowanie prac w środowisku rozproszonym za pomocą crona	315
cron	315
Prace crona a idempotencja	316
cron w dużej skali	317
Budowanie crona w Google'u	318
Podsumowanie	325
25. Potoki przetwarzania danych	327
Początki wzorca projektowego „potok danych”	327
Początkowy wpływ big data na prosty wzorec potoku danych	327
Wyzwania związane ze wzorcem „okresowo uruchamiany potok danych”	328
Problemy powodowane przez nierównomierny podział pracy	328
Wady okresowo uruchamianych potoków w środowiskach rozproszonych	329
Wprowadzenie do systemu Workflow Google'a	332
Etapy wykonywania w systemie Workflow	334
Zapewnianie ciągłości biznesowej	335
Podsumowanie i uwagi końcowe	336
26. Integralność danych — wczytywanie tego, co zostało zapisane	337
Ścisłe wymagania z zakresu integralności danych	338
Cele zespołów SRE w Google'u w zakresie integralności i dostępności danych	342
Jak zespoły SRE Google'a radzą sobie z problemami z integralnością danych?	346
Studia przypadków	357
Ogólne zasady SRE stosowane w obszarze integralności danych	363
Wnioski	365
27. Niezawodne udostępnianie produktów w dużej skali	367
Inżynieria koordynowania udostępniania	368
Konfigurowanie procesu udostępniania	370
Tworzenie listy kontrolnej udostępniania	373
Wybrane techniki niezawodnego udostępniania	377
Powstawanie zespołu LCE	381
Wnioski	384

Część IV. Zarządzanie 385

28. Szybkie przygotowywanie inżynierów SRE do dyżurów i innych zadań	387
Zatrudniłeś nowych inżynierów SRE. Co dalej?	387
Początkowe pouczające doświadczenia — argument na rzecz przewagi struktury nad chaosem	389
Rozwój świetnych ekspertów od inżynierii odwrotnej i improwizatorów	393

Pięć praktyk dla przyszłych dyżurnych	395
Dyżury i inne zadania — rytuał przejścia i ciągłe uczenie się	400
Końcowe myśli	401
29. Radzenie sobie z zakłóceniami	403
Zarządzanie obciążeniem operacyjnym	404
Czynniki wpływające na sposób zarządzania zakłóceniami	404
Niedoskonałe maszyny	405
30. Angażowanie inżyniera SRE w celu wyeliminowania przeciążenia operacyjnego	411
Etap 1. Poznaj usługę i kontekst	412
Etap 2. Przedstawianie kontekstu	414
Etap 3. Motywowanie do zmian	415
Wnioski	417
31. Komunikacja i współpraca w zespołach SRE	419
Komunikacja — spotkania produkcyjne	420
Współpraca w ramach zespołów SRE	423
Studium przypadku z obszaru współpracy w zespołach SRE — Viceroy	425
Współpraca z zespołami innymi niż SRE	429
Studium przypadku — przeniesienie DFP do F1	430
Wnioski	432
32. Zmiany w modelu angażowania się zespołów SRE	433
Zaangażowanie zespołów SRE — co, jak i dlaczego?	433
Model PGP	434
Model angażowania się zespołów SRE	434
Przeglądy gotowości produkcyjnej — prosty model oparty na PGP	436
Ewolucja prostego modelu opartego na PGP — wczesne zaangażowanie	439
Zmiany w rozwoju usług — frameworki i platforma SRE	441
Wnioski	446
Część V. Wnioski	447
33. Lekcje z innych branż	449
Poznaj naszych branżowych weteranów	450
Testowanie gotowości i odporności na katastrofy	451
Kultura analizy zdarzeń	454
Eliminowanie powtarzalnej pracy i kosztów operacyjnych dzięki automatyzacji	456
Ustrukturyzowane i racjonalne podejmowanie decyzji	457
Wnioski	459
34. Podsumowanie	461

A	Tabela dostępności	465
B	Zbiór dobrych praktyk dotyczących usług produkcyjnych	467
C	Przykładowy dokument ze stanem incydentu	473
D	Przykładowa analiza zdarzenia	475
E	Lista kontrolna LCE	479
F	Przykładowe notatki ze spotkania produkcyjnego	481
	Bibliografia	483
	Skorowidz	494

Obsługa przeciążenia

Autor: Alejandro Forero Cuervo

Redakcja: Sarah Chavis

Unikanie przeciążenia to cel reguł równoważenia obciążenia. Jednak niezależnie od tego, jak wydajne są te reguły, *ostatecznie* niektóre części systemu staną się przeciążone. Płynna obsługa takich sytuacji jest nieodzowna, jeśli chcesz zarządzać niezawodnym systemem serwerowym.

Jedną z możliwości obsługi przeciążenia jest zwracanie odpowiedzi awaryjnych, które nie są tak precyzyjne jak zwykle lub zawierają mniej danych, ale są łatwiejsze do obliczenia. Oto przykład:

- Zamiast przeszukiwać cały zbiór danych w celu zapewnienia najlepszych dostępnych wyników dla zapytania do wyszukiwarki, można uwzględnić tylko niewielki procent potencjalnie odpowiednich danych.
- Można korzystać z lokalnej kopii wyników, która nie zawsze jest w pełni aktualna, ale jest mniej kosztowna w użyciu niż posługiwanie się standardowym magazynem danych.

Jednak przy skrajnym przeciążeniu usługa może nie radzić sobie nawet z wyznaczaniem i udostępnianiem odpowiedzi awaryjnych. Na takim etapie jedyną możliwością do natychmiastowego zastosowania jest zwracanie błędów. Jednym ze sposobów na złagodzenie skutków tej sytuacji jest równoważenie ruchu między centrami danych w taki sposób, aby żadne centrum nie otrzymywało więcej żądań, niż jest w stanie przetworzyć. Na przykład jeśli w centrum danych działa 100 zadań zaplecza i każde z nich potrafi przetworzyć do 500 żądań na sekundę, to algorytm równoważenia obciążenia nie powinien pozwalać na przekazywanie do tego centrum więcej niż 50 tys. zapytań na sekundę. Jednak nawet to ograniczenie może okazać się niewystarczające do uniknięcia przeciążenia, jeśli system działa na dużą skalę. Ostatecznie najlepiej jest tak budować klienty i zadania zaplecza, by płynnie radziły sobie z ograniczeniami zasobów. Gdy to możliwe, należy przekierowywać ruch, w razie konieczności zwracać wyniki awaryjne, a gdy wszystko inne zawiedzie, bezpośrednio obsługiwać błędy związane z zasobami.

Pułapki związane z „liczbą zapytań na sekundę”

Różne zapytania mogą mieć bardzo zróżnicowane wymagania co do zasobów. Koszt zapytania może się różnić w zależności od arbitralnych czynników, takich jak kod klienta, który je zgłasza (w usługach

o wielu klientach), a nawet pora dnia (np. użytkownicy firmowi i prywatni lub interaktywny ruch generowany przez użytkowników końcowych i ruch z operacji wsadowych).

Przekonał się o tym na własnej skórze. „Zapytania na sekundę” lub statyczne cechy żądań uważane za przybliżenie ilości zużywanych zasobów (np. „ile kluczy wczytuje żądanie”) to często mało przydatne miary do modelowania przepustowości. Nawet jeśli w danym okresie używana miara daje dobre efekty, może się to zmienić. Czasem zmiany następują stopniowo, jednak nieraz zachodzą gwałtownie (np. nowa wersja oprogramowania nagle sprawia, że niektóre funkcje z pewnych żądań wymagają znacznie mniej zasobów). Taki „ruchomy cel” jest niezbyt przydatną miarą do użytku w ramach projektowania i implementowania równoważenia obciążenia.

Lepszym rozwiązaniem jest pomiar przepustowości bezpośrednio na poziomie dostępnych zasobów. Załóżmy, że dla danej usługi w określonym centrum danych zarezerwowanych jest 500 rdzeni procesora i 1 TB pamięci. Oczywiście znacznie lepiej jest modelować przepustowość centrum danych bezpośrednio za pomocą tych liczb. Często mówimy o *koszcie* żądania, posługując się znormalizowaną miarą zajmowanego czasu procesora (z uwzględnieniem różnic w sprawności różnych architektur procesorów).

W większości sytuacji (choć zdecydowanie nie we wszystkich) dobrze sprawdza się proste używanie obciążenia procesorów jako wskaźnika zapotrzebowania na zasoby. Wynika to z następujących powodów:

- W platformach z odzyskiwaniem pamięci duże obciążenie pamięci w naturalny sposób przekłada się na większe wykorzystanie procesorów.
- W innych platformach można udostępniać pozostałe zasoby w taki sposób, że bardzo mało prawdopodobne jest wyczerpanie się ich przed zajęciem całej mocy procesorów.

W sytuacjach, gdy zapewnienie nadmiarowej ilości zasobów innych niż procesor jest zbyt kosztowne, przyglądamy się osobno poszczególnym zasobom w trakcie analizowania ich zużycia.

Limity na klienta

Jednym z aspektów radzenia sobie z przeciążeniem jest określanie, co zrobić w przypadku *globalnego* przeciążenia. W idealnym świecie, gdzie zespoły starannie koordynują udostępnianie swoich produktów z właścicielami powiązanych usług zaplecza, globalne przeciążenie nigdy nie następuje, a usługi zaplecza zawsze zapewniają przepustowość wystarczającą do obsługi klientów. Niestety, nie żyjemy w idealnym świecie. W praktyce globalne przeciążenie zdarza się stosunkowo często (zwłaszcza w usługach wewnętrznych, z których nieraz korzystają liczne klienty wielu zespołów).

Gdy globalne przeciążenie *wystąpi*, bardzo ważne jest, by usługa zwracała odpowiedzi o błędach tylko do niewłaściwie pracujących klientów. Nie powinno to wpływać na pozostałe klienty. Aby to osiągnąć, właściele usługi zapewniają przepustowość na podstawie poziomu zużycia zasobów wynegocjowanego z klientami i według tego definiują limity dla klientów.

Na przykład jeśli usłudze zaplecza przydzielono na świecie 10 tys. procesorów (w różnych centrach danych), limity dla klientów mogą wyglądać tak:

- Gmail może zużywać do 4000 s czasu procesora na sekundę.
- Kalendarz może zużywać do 4000 s czasu procesora na sekundę.

- Android może zużywać do 3000 s czasu procesora na sekundę.
- Google+ może zużywać do 2000 s czasu procesora na sekundę.
- Wszyscy pozostali użytkownicy mogą zużywać do 500 s czasu procesora na sekundę.

Zauważ, że te liczby mogą w sumie przekroczyć 10 tys. procesorów przydzielonych tej usłudze zaplecza. Właściciel usługi opiera się na tym, że mało prawdopodobne jest, by wszyscy klienci jednocześnie osiągnęli limit zużycia zasobów.

Globalne informacje o poziomie zużycia zasobów agregujemy globalnie w czasie rzeczywistym dla wszystkich zadań zaplecza. Na podstawie tych danych zmieniamy limity dla poszczególnych zadań zaplecza. Dokładne omawianie systemu wykonującego to zadanie wykracza poza zakres rozdziału. Napisaliśmy jednak dużo kodu do realizowania tego procesu w zadaniach zaplecza. Ciekawą częścią układanki jest obliczanie w czasie rzeczywistym zasobów (zwłaszcza czasu procesora) wykorzystywanych przez każde żądanie. Jest to skomplikowane przede wszystkim w serwerach, w których nie ma zaimplementowanego modelu „jeden wątek na żądanie”, gdzie pula wątków obsługuje przychodzące różne części wszystkich żądań, używając nieblokujących interfejsów API.

Ograniczanie liczby żądań po stronie klienta

Gdy klient przekroczy limit, zadanie zaplecza powinno szybko odrzucać żądania z założeniem, że zwrócenie błędu „klient przekroczył limit” wymaga znacznie mniej zasobów niż przetworzenie żądania i zwrócenie poprawnej odpowiedzi. Jednak nie we wszystkich usługach jest to prawdą. Na przykład prawie równie kosztowne jest odrzucenie żądania wymagającego prostego wyszukania danych w pamięci RAM (kiedy to koszt obsługi protokołu żądań i odpowiedzi jest znacznie wyższy niż koszt wygenerowania odpowiedzi) co przyjęcie i obsłużenie go. Nawet w sytuacjach, gdy odrzucenie żądań pozwala zaoszczędzić dużo zasobów, żądania *i tak* zużywają zasoby. W takich sytuacjach zadanie zaplecza może stać się przeciążone również wtedy, gdy większość czasu procesora jest przeznaczana na odrzucanie żądań!

Rozwiązaniem problemu jest ograniczanie żądań po stronie klienta¹. Gdy klient wykrywa, że duża część żądań z ostatniego czasu została odrzucona z powodu błędów „przekroczenia limitu”, zaczyna autoregulację i ogranicza ilość generowanego wychodzącego ruchu. Żądania powodujące przekroczenie ograniczenia są blokowane lokalnie i nawet nie trafiają do sieci.

Ograniczanie żądań po stronie klienta zaimplementowaliśmy za pomocą techniki, którą nazwaliśmy *adaptacyjnym ograniczaniem*. Każde zadanie klienckie przechowuje następujące informacje na temat ostatnich dwóch minut pracy:

requests

Liczba żądań, które próbowała zgłosić warstwa aplikacji (po stronie klienta, powyżej systemu adaptacyjnego ograniczania).

accepts

Liczba żądań zaakceptowanych przez zadanie zaplecza.

¹ Zob. np. narzędzie Doorman (<https://github.com/youtube/doorman>) — kooperatywny rozproszony system ograniczania żądań po stronie klienta.

W normalnych warunkach obie te wartości są sobie równe. Gdy zadanie zaplecza zaczyna odrzucać żądania, wartość *accepts* spada poniżej wartości *requests*. Klienci mogą kontynuować zgłaszanie żądań do zadania zaplecza do momentu, gdy *requests* jest *K* razy większe niż *accepts*. Po dojściu do tego poziomu klient zaczyna autoregulację i nowe żądania są odrzucane na poziomie lokalnym (czyli po stronie klienta) z prawdopodobieństwem wyrażonym w równaniu 21.1.

$$\max(0, \frac{requests - K \times accepts}{requests + 1})$$

Równanie 21.1. Prawdopodobieństwo odrzucenia żądania klienta

Gdy klient sam zaczyna odrzucać żądania, wartość *requests* stale przekracza wartość *accepts*. Choć wydaje się to sprzeczne z intuicją, to ponieważ lokalnie odrzucane żądania nie są przekazywane do zadania zaplecza, jest to rozwiązanie preferowane. Gdy szybkość przekazywania żądań przez aplikację do klienta rośnie (względem szybkości, z jaką zadanie zaplecza je akceptuje), chcemy zwiększać prawdopodobieństwo odrzucania nowych żądań.

W usługach, w których koszt przetwarzania żądań jest bardzo bliski kosztowi ich odrzucenia, pozwalanie na zużywanie połowy zasobów tego zadania na odrzucanie żądań może być nieakceptowalne. W takiej sytuacji rozwiązanie jest proste — należy zmodyfikować stosowany do wartości *accepts* mnożnik *K* (równy 2) we wzorze na prawdopodobieństwo odrzucenia żądania klienta (równanie 21.1). Dzięki temu:

- zmniejszenie mnożnika powoduje bardziej agresywne adaptacyjne ograniczanie;
- zwiększenie mnożnika zmniejsza agresywność adaptacyjnego ograniczania.

Na przykład zamiast włączania autoregulacji klienta, gdy *requests* = 2 * *accepts*, można zastosować autoregulację na poziomie *requests* = 1.1 * *accepts*. Obniżenie wartości modyfikatora do 1,1 oznacza, że tylko jedno żądanie zostanie przez zadanie zaplecza odrzucone na każdych 10 zaakceptowanych.

Ogólnie preferujemy stosowanie mnożnika 2. Pozwalając na dotarcie do zadania zaplecza większej liczby żądań, niż zgodnie z oczekiwaniami jest to dopuszczalne, marnujemy więcej zasobów tego zadania, ale też przyspieszamy przekazywanie informacji o stanie z zadania zaplecza do klientów. Jeśli zadanie zaplecza zdecyduje się zakończyć odrzucanie żądań od zadań klienckich, czas do wykrycia tej zmiany stanu przez wszystkie zadania klienckie będzie wtedy krótszy.

Odkryliśmy, że adaptacyjne ograniczanie dobrze się sprawdza w praktyce i prowadzi do stabilnego poziomu liczby żądań na ogólnym poziomie. Nawet w sytuacji wysokiego przeciążenia zadania zaplecza odrzucają tylko jedno żądanie na każde żądanie obsłużone. Ważną zaletą tego podejścia jest to, że decyzja jest podejmowana przez zadanie klienckie wyłącznie na podstawie lokalnych informacji i stosunkowo prostej implementacji. Nie występują tu dodatkowe zależności ani kary w postaci wzrostu opóźnienia.

Dodatkową kwestią jest to, że ograniczanie żądań po stronie klienta może się nie sprawdzać w klientach, które bardzo rzadko kierują żądania do zadań zaplecza. Wtedy wgląd każdego klienta w stan zadań zaplecza jest znacznie słabszy, a próby zwiększenia go są zwykle kosztowne.

Poziom krytyczności

Poziom krytyczności to następne pojęcie, które uznaliśmy za niezwykle przydatne w kontekście globalnych limitów i ograniczania żądań. Żądanie skierowane do zadania zaplecza otrzymuje jeden z czterech możliwych poziomów krytyczności (od najbardziej do najmniej krytycznego):

CRITICAL_PLUS

Zarezerwowany dla najbardziej krytycznych żądań, których niepowodzenie skutkuje poważnymi problemami widocznymi dla użytkowników.

CRITICAL

Jest to poziom domyślny dla żądań zgłaszanych przez prace produkcyjne. Te żądania powodują skutki widoczne dla użytkowników, ale ich wpływ jest mniej poważny niż na poziomie CRITICAL_PLUS. Usługi powinny zapewniać wystarczającą przepustowość dla całego oczekiwanego ruchu z poziomów CRITICAL i CRITICAL_PLUS.

SHEDDABLE_PLUS

Dotyczy ruchu, dla którego oczekiwana jest częściowa niedostępność. Jest to poziom domyślny dla prac wsadowych, które mogą ponawiać żądania po upływie minut, a nawet godzin.

SHEDDABLE

Dotyczy ruchu, dla którego oczekiwana jest częsta częściowa niedostępność i okazjonalna pełna niedostępność.

Stwierdziliśmy, że te cztery wartości są wystarczające do uwzględnienia prawie każdej usługi. Wielokrotnie dyskutowaliśmy nad propozycjami dodatkowych wartości, ponieważ pozwoliłyby one precyzyjniej klasyfikować żądania. Jednak definiowanie dodatkowych wartości wymagałoby więcej zasobów do obsługi różnych systemów uwzględniających poziomy krytyczności.

Krytyczność potraktowaliśmy jako pełnoprawny aspekt systemu RPC i ciężko pracowaliśmy, aby zintegrować ją z wieloma mechanizmami kontrolnymi, tak by można było uwzględniać ją w ramach reagowania na przeciążenie. Oto przykłady:

- Gdy klient przekroczy globalny limit, zadanie zaplecza odrzuca żądania z danego poziomu krytyczności tylko wtedy, jeśli odrzuca już żądania z wszystkich niższych poziomów (opisane wcześniej obsługiwane przez nasz system limity dla klientów można też ustawiać dla poziomów krytyczności).
- Gdy samo zadanie jest przeciążone, szybciej zaczyna odrzucać żądania o niższym poziomie krytyczności.
- System adaptacyjnego ograniczania przechowuje statystyki osobno dla każdego poziomu krytyczności.

Poziom krytyczności żądania jest niezależny od wymagań związanych z opóźnieniem, a tym samym od obowiązującego dla sieci poziomu jakości usług (ang. *Quality of Service* — **QoS**). Na przykład gdy system wyświetla wyniki wyszukiwania lub sugestie w trakcie wpisywania zapytania w wyszukiwarce, z obsługi żądań można łatwo zrezygnować (jeśli system jest przeciążony, niewyświetlanie wyników jest akceptowalne), natomiast wymagania związane z opóźnieniem są tu zwykle wysokie.

Znacznie rozbudowaliśmy też nasz system RPC, aby poziom krytyczności był przekazywany automatycznie. Jeśli zadanie zaplecza otrzymuje żądanie *A* i w ramach jego przetwarzania zgłasza żądania *B* i *C* do innych zadań zaplecza, żądania *B* i *C* będą domyślnie miały ten sam poziom krytyczności co żądanie *A*.

W przeszłości w wielu systemach w Google’u powstały doraźne poziomy krytyczności, często niekompatybilne między usługami. Dzięki ustandaryzowaniu i przekazaniu poziomów krytyczności w ramach systemu RPC obecnie możemy w spójny sposób ustawiać je w określonych punktach. To daje nam pewność, że przeciążone zależne zadania będą respektować poziom krytyczności, odrzucając ruch, niezależnie od tego, jak głęboko w stosie wywołań RPC działają. Dlatego staramy się ustawiać poziom krytyczności jak najbliżej przeglądarki lub klientów mobilnych (zwykle we frontonach HTTP, które generują zwracany kod w HTML-u) i zmieniać go tylko w specjalnych sytuacjach, gdy ma to sens w określonym punkcie stosu.

Sygnały poziomu wykorzystania

Nasza implementacja ochrony przed przeciążeniem na poziomie zadania jest oparta na *poziomie wykorzystania*. W wielu sytuacjach jest to tylko miara stopnia obciążenia procesorów (czyli aktualne obciążenie procesorów podzielone przez łączną moc procesorów zarezerwowaną dla danego zadania). Czasem jednak uwzględniamy też wskaźniki takie jak to, jaka część zarezerwowanej pamięci jest obecnie używana. Gdy poziom wykorzystania zbliża się do ustalonego progu, zaczynamy odrzucać żądania na podstawie ich poziomu krytyczności (dla wyższej krytyczności próg wykorzystania jest wyższy).

Używane przez nas sygnały poziomu wykorzystania są oparte na lokalnym stanie zadania (ponieważ celem sygnałów jest ochrona zadań). Stosujemy różne sygnały. Najczęściej przydatny sygnał jest oparty na „obciążeniu” w procesie, określanym przy użyciu systemu nazywanego przez nas *średnim obciążeniem wątków wykonawczych*.

Aby ustalić średnie obciążenie wątków wykonawczych, zliczamy aktywne wątki procesu. Określenie „aktywne” oznacza tu wątki, które aktualnie działają lub są gotowe do pracy i czekają na wolny procesor. Wartości „wyglądzamy”, stosując wykładniczą funkcję zaniku, i zaczynamy odrzucać żądania, gdy liczba aktywnych wątków przekracza liczbę procesorów dostępnych dla zadania. To oznacza, że wszystkie przychodzące żądania o bardzo wysokim stopniu rozgałęzienia (generują one bardzo dużą liczbę krótkich operacji) powodują bardzo krótki skok obciążenia, który jednak zostaje w większości ukryty dzięki wygładzaniu. Jeśli jednak operacje nie są krótkie (obciążenie wtedy rośnie i pozostaje wysokie przez dłuższy czas), zadanie zacznie odrzucać żądania.

Choć średnie obciążenie wątków wykonawczych okazało się bardzo przydatnym sygnałem, nasz system potrafi uwzględnić dowolne sygnały poziomu wykorzystania potrzebne dla konkretnego zadania zaplecza. Możemy np. posłużyć się szczytowym wykorzystaniem pamięci (określa on, że zużycie pamięci w zadaniu zaplecza wzrosło ponad standardowe parametry operacyjne) jako innym sygnałem poziomu wykorzystania. System można też skonfigurować tak, by łączył wiele sygnałów i odrzucał żądania, które skutkują przekroczeniem sumarycznego (lub częściowego) docelowego progu.

Obsługa błędów przeciążenia

Oprócz zapewnienia płynnej obsługi obciążenia długo analizowaliśmy też to, jak klienci powinny reagować po otrzymaniu odpowiedzi o błędzie związanym z obciążeniem. W przypadku takich błędów rozróżniamy dwie sytuacje.

Przeciążony jest duży podzbiór zadań zaplecza w centrum danych

Jeśli system równoważenia obciążenia między centrami danych działa w idealny sposób (czyli potrafi przekazywać informacje o stanie i natychmiast reagować na zmiany w ruchu), ta sytuacja nigdy nie występuje.

Przeciążony jest niewielki podzbiór zadań zaplecza w centrum danych

Ta sytuacja jest zwykle powodowana niedoskonałością mechanizmów równoważenia obciążenia w centrum danych. Na przykład zadanie mogło niedawno otrzymać bardzo kosztowne żądanie. W takiej sytuacji wysoce prawdopodobne jest, że centrum danych dzięki innym zadaniom ma możliwość obsługiwaną zadań.

Gdy przeciążony jest duży podzbiór zadań zaplecza, nie należy ponawiać zgłoszeń, a błędy powinny być przekazywane aż do nadawcy (błąd należy więc zwrócić użytkownikowi końcowemu). Jednak znacznie częściej przeciążona jest tylko niewielka część zadań. Wtedy preferowaną reakcją jest natychmiastowe ponowienie żądania. Nasz system równoważenia obciążenia między centrami danych zwykle próbuje przekierować ruch od klientów do najbliższych dostępnych centrów danych z zadaniami zaplecza. Czasem najbliższe centrum danych jest znacznie oddalone (najbliższe dostępne zadanie zaplecza dla klienta może się znajdować na innym kontynencie), jednak przeważnie udaje nam się usytuować klienty blisko powiązanych zadań zaplecza. Dzięki temu dodatkowe opóźnienie wynikające z ponowienia żądania (to tylko kilka wymian komunikatów w sieci) jest pomijalne.

Z perspektywy reguł równoważenia obciążenia próby ponawiania żądań są nieodróżnialne od nowych żądań. Oznacza to, że nie stosujemy bezpośrednio kodu do upewniania się, że ponowione żądanie trafi do innego zadania zaplecza. Polegamy na wysokim prawdopodobieństwie, że ponowna próba trafi do innego zadania z powodu dużej liczby zadań zaplecza w podzbiórce. Upewnianie się, że wszystkie ponawiane żądania trafiają do innych zadań, zwiększałoby złożoność interfejsów API w większym stopniu, niż jest to tego warte.

Nawet gdy zadanie zaplecza jest tylko w niewielkim stopniu przeciążone, żądania klienta są często lepiej obsługiwane, jeśli zadanie zaplecza na równych zasadach i szybko odrzuca ponawiane i nowe żądania. Odrzucone żądanie można natychmiast ponowić, kierując je do innego zadania zaplecza, w którym dostępne mogą być wolne zasoby. Efekt traktowania w zadaniach zaplecza ponawianych i nowych żądań w ten sam sposób jest taki, że ponawianie żądań w innych zadaniach pozwala w naturalny sposób równoważyć obciążenie. Jest ono przekierowywane do zadań, które mogą być lepiej dostosowane do ich obsługi.

Decydowanie o ponowieniu żądania

Gdy klient otrzymuje odpowiedź z błędem „przeciążone zadanie”, musi zdecydować, czy ponowić żądanie. Stosujemy kilka mechanizmów pozwalających uniknąć ponawiania żądań, gdy znaczna część zadań w klastrze jest przeciążona.

Po pierwsze, stosujemy *budżet ponowień dla żądania* na poziomie trzech prób. Jeśli żądanie już trzy razy spowodowało błąd, pozwalamy na zwrócenie błędu do nadawcy żądania. Wynika to z następującego rozumowania: jeśli żądanie już trzykrotnie trafiło do przeciążonego zadania, jest stosunkowo mało prawdopodobne, że następna próba pomoże; w takiej sytuacji zapewne całe centrum danych jest przeciążone.

Po drugie, stosujemy *budżet ponowień dla klienta*. Każdy klient śledzi odsetek ponowień żądań. Żądanie jest ponawiane tylko wtedy, gdy ta wartość wynosi poniżej 10%. Wynika to z tego, że gdy przeciążony jest stosunkowo niewielki podzbiór zadań, ponawianie powinno być potrzebne stosunkowo rzadko.

W ramach konkretnego przykładu (najgorszego scenariusza) załóżmy, że centrum danych akceptuje niewielką ilość żądań, a dużą ich część odrzuca. Niech X będzie łączną liczbą żądań przekazanych do centrum danych przez kod kliencki. Z powodu liczby potrzebnych ponowień liczba żądań znacznie wzrośnie — do poziomu niewiele niższego niż $3X$. Choć ograniczyliśmy liczbę żądań spowodowanych ponowieniami, trzykrotny wzrost jest znaczący (zwłaszcza gdy koszt odrzucenia żądania jest wysoki w porównaniu z kosztem jego przetworzenia). Jednak uwzględnienie budżetu ponowień dla klienta (odsetek ponowień na poziomie 10%) w ogólnym przypadku zmniejsza ten wzrost do $1,1X$, co stanowi znaczną poprawę.

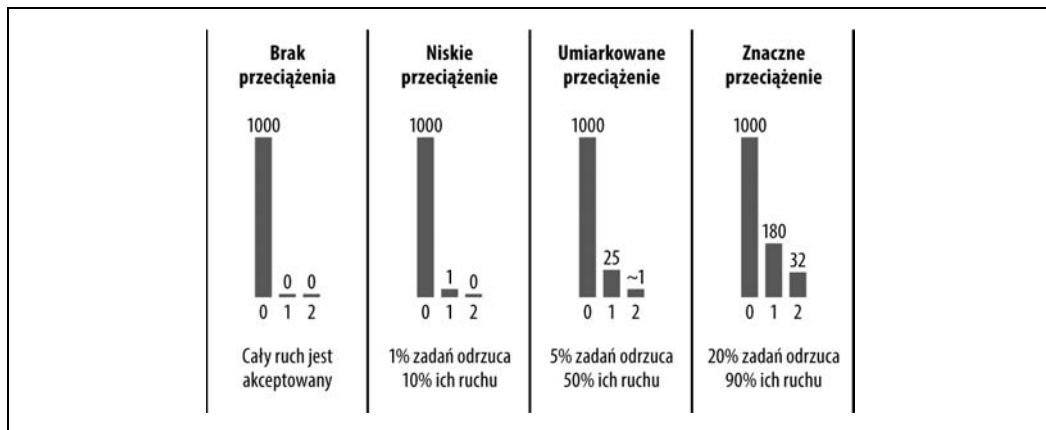
Trzecie podejście polega na tym, że klienci umieszczają w metadanych żądania liczbę prób. Przy pierwszej próbie licznik ma wartość 0 i jest zwiększany po każdym ponowieniu żądania aż do osiągnięcia wartości 2, kiedy to wyczerpanie budżetu ponowień dla żądania powoduje zaprzestanie dalszych prób. Zadania zaplecza przechowują histogramy tych wartości z niedawnej historii. Gdy zadanie zaplecza musi odrzucić żądanie, sprawdza te histogramy, aby ustalić prawdopodobieństwo, że inne zadania zaplecza też są przeciążone. Jeśli histogramy wskazują na dużą liczbę ponowień (co oznacza, że inne zadania zaplecza też zapewne są przeciążone), zwracany jest błąd „przeciążone, nie ponawiać” zamiast standardowego błędu „zadanie jest przeciążone”, który skutkuje ponowną próbą.

Rysunek 21.1 przedstawia liczbę prób dla każdego żądania w określonym zadaniu zaplecza w różnych przykładowych sytuacjach. Wartości te dotyczą przesuwanego okna obejmującego 1000 początkowych żądań (nie licząc ponowień). Dla uproszczenia budżet ponowień dla klienta jest tu ignorowany (w wynikach zakładamy, że uwzględniany jest tylko budżet ponowień w wysokości trzech prób na żądanie). Ponadto zastosowanie podzbiorów mogłoby nieco zmienić uzyskane wartości.

Większe z naszych usług mają zwykle postać dużych stosów systemów, które z kolei mogą być zależne od siebie. W takiej architekturze żądania powinny być ponawiane tylko w warstwie bezpośrednio nad warstwą, która je odrzuca. Gdy uznajemy, że danego żądania nie można obsłużyć i nie należy go ponawiać, zgłaszamy błąd „przeciążone, nie ponawiać” i unikamy w ten sposób kombinatorycznej eksplozji ponawianych prób.

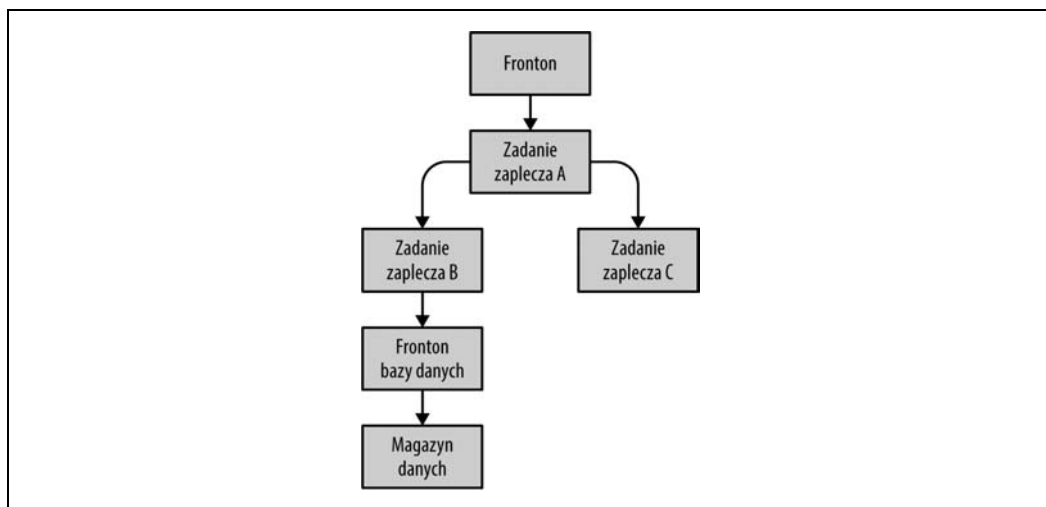
Przyjrzyj się przykładowi z rysunku 21.2 (w praktyce nasze stopy są często dużo bardziej złożone). Wyobraź sobie, że fronton bazy danych jest obecnie przeciążony i odrzuca żądania. W tej sytuacji:

- Zadanie zaplecza B spróbuje ponowić żądanie zgodnie z opisanymi wskazówkami.



Rysunek 21.1. Histogramy liczby prób w różnych warunkach

- Jednak gdy zadanie zaplecza B wykryje, że żądanie do frontonu bazy danych nie może zostać obsłużone (np. w wyniku trzykrotnego odrzucenia prób), musi zwrócić do zadania zaplecza A albo błąd „przeciążone, nie ponawiać”, albo odpowiedź awaryjną (przy założeniu, że można wygenerować częściowo przydatną odpowiedź nawet po nieudanym żądaniu do frontonu bazy danych).
- Zadanie zaplecza A ma dokładnie te same możliwości w stosunku do żądania otrzymanego od frontonu i wykonuje analogiczne operacje.



Rysunek 21.2. Stos zależności

Ważne jest to, że nieudane żądanie do frontonu bazy danych powinno być ponawiane tylko przez zadanie zaplecza B, czyli warstwę bezpośrednio nad tym frontonem. Ponawianie prób przez wiele warstw prowadzi do eksplozji kombinatorycznej.

Obciążenie wynikające z połączeń

Obciążenie wynikające z połączeń to ostatni czynnik, o którym warto wspomnieć. Czasem uwzględniamy tylko obciążenie zadań zaplecza spowodowane bezpośrednio otrzymywanymi żądaniami (jest to jeden z problemów z podejściami, w których model obciążenia jest oparty na zapytaniach na sekundę). Jednak skutkuje to ignorowaniem kosztów procesora i pamięci związanych z dużą liczbą połączeń. Nieuwzględniany jest też koszt szybkiego przełączania połączeń. W małych systemach te problemy są pomijalne, jednak szybko stają się problematyczne w działających na bardzo dużą skalę systemach wywołań RPC.

Wcześniej wspomnieliśmy już, że nasz protokół RPC wymaga okresowego sprawdzania stanu przez nieaktywne klienty. Gdy połączenie jest nieużywane przez ustalony czas, klient zamyka połączenie TCP i nawiązuje połączenie UDP na potrzeby sprawdzania stanu. Niestety, to rozwiązanie jest problematyczne, gdy bardzo duża liczba zadań klienckich zgłasza bardzo niewielką liczbę żądań. Sprawdzanie stanu za pomocą połączeń może wtedy wymagać więcej zasobów niż sama obsługa żądań. Techniki takie jak staranne dostosowanie parametrów połączeń (np. znaczne zmniejszenie częstotliwości sprawdzania stanu) lub nawet dynamiczne tworzenie i zamykanie połączeń pozwalają znacznie poprawić sytuację.

Obsługa skoków liczby żądań nawiązania nowych połączeń to drugi problem. Widzieliśmy tego typu skoki w rozbudowanych pracach wsadowych, które tworzyły jednocześnie bardzo dużą liczbę roboczych zadań klienckich. Konieczność jednoczesnego nawiązywania i utrzymywania dużej liczby nowych połączeń może łatwo przeciążyć grupę zadań zaplecza. Według naszych doświadczeń istnieje kilka strategii pomagających zmniejszyć to obciążenie:

- Przekazanie obciążenia do algorytmu równoważenia obciążenia między centrami danych (odpowiedzialnego za podstawowe równoważenie obciążenia na bazie wykorzystania klastrów, a nie według samej liczby żądań). W takiej sytuacji obciążenie z żądań jest przekazywane do innych centrów danych o wolnej przepustowości.
- Zobowiązanie klienckich prac wsadowych do stosowania odrębnego zestawu zadań zaplecza pełniących funkcję *pośredników prac wsadowych*. Takie zadania jedynie przekazują w kontrolowany sposób żądania do podstawowych zadań zaplecza i dostarczają odpowiedzi od nich do klientów. Dlatego zamiast komunikacji „klient wsadowy → zadanie zaplecza” mamy sekwencję „klient wsadowy → pośrednik prac wsadowych → zadanie zaplecza”. Wtedy gdy bardzo duża praca rozpoczyna działanie, wpływa to tylko na pośrednika prac wsadowych, który chroni podstawowe zadania zaplecza (i klienty o wyższym priorytecie). Pośrednik pełni tu funkcję bezpiecznika. Inna zaleta stosowania pośrednika jest taka, że zwykle ogranicza to liczbę połączeń z zadaniem zaplecza. Może to skutkować usprawnieniem równoważenia obciążenia dla zadań zaplecza (zadania pośredniczące mogą korzystać z większych podzbiorów i często nawet mieć lepszy wgląd w stan zadań zaplecza).

Wnioski

W rozdziałach tym i 20. omówiliśmy, jak za pomocą różnych technik (deterministycznego określania podzbiorów, algorytmu rotacyjnego z wagami, ograniczania żądań po stronie klienta, limitów dla klientów itd.) rozdzielać w stosunkowo równomierny sposób obciążenie między zadania w centrum danych. Jednak te mechanizmy wymagają przekazywania stanu w systemie rozproszonym. Choć w ogólnych sytuacjach rozwiązania te działają całkiem dobrze, w rzeczywistych aplikacjach zdarzają się scenariusze, w których te mechanizmy okazują się niedoskonałe.

Dlatego uważamy za bardzo istotne zapewnianie ochrony poszczególnych zadań przed przeciążeniem. Ujmijmy to prosto: zadanie zaplecza z zasobami pozwalającymi na obsługę określonego poziomu ruchu powinno radzić sobie z ruchem na tym poziomie bez istotnych zmian opóźnień niezależnie od tego, jak dużo dodatkowego ruchu jest do niego kierowane. Wynika z tego, że zadanie zaplecza nie może zawodzić i ulegać awarii z powodu obciążenia. Te stwierdzenia powinny być prawdziwe dla określonego poziomu ruchu — w przybliżeniu większego niż dwukrotność, a nawet do dziesięciokrotności obciążenia, jakie zadanie potrafi obsłużyć za pomocą przydzielonych mu zasobów. Akceptujemy to, że istnieje poziom, powyżej którego system zaczyna się załamywać, i że dalsze podnoszenie tego progu może być stosunkowo trudne.

Najważniejsze jest, by poważnie traktować warunki prowadzące do pogorszenia pracy systemu. Jeśli te warunki są ignorowane, wiele systemów zaczyna działać bardzo źle. Gdy praca się nawarstwia, a zadania zaczynają wyczerpywać pamięć i ulegać awarii (lub zużywają prawie cały czas procesora na migotanie pamięci), opóźnienie rośnie, ponieważ żądania są odrzucane, a zadania współzawodniczą o zasoby. Jeśli się tym nie zajmiesz, awaria części systemu (np. jednego zadania zaplecza) może doprowadzić do awarii innych komponentów. Grozi to wyłączeniem całego systemu lub dużej jego części. Wpływ tego rodzaju awarii kaskadowych może być tak poważny, że w każdym systemie działającym w dużej skali niezwykle istotna jest ochrona przed taką sytuacją (zob. rozdział 22.).

Często popełnianym błędem jest zakładanie, że przeciążone zadanie zaplecza powinno odrzucać i przestać przyjmować cały ruch. Takie podejście byłoby niezgodne z celem, jakim jest stabilne równoważenie obciążenia. W praktyce chcemy, by zadanie zaplecza nadal obsługiwało jak największą część ruchu, ale tylko po uwolnieniu przepustowości. Dobrze działające zadanie zaplecza wspomagane przez solidne reguły równoważenia obciążenia powinno akceptować tylko te żądania, które potrafi obsłużyć; pozostałe żądania należy odrzucać.

Choć posiadamy wiele narzędzi do implementowania skutecznego równoważenia obciążenia i ochrony przed przeciążeniem, uniwersalne rozwiązanie nie istnieje. Równoważenie obciążenia często wymaga dogłębnego zrozumienia systemu i semantyki używanych w nim żądań. Techniki opisane w tym rozdziale ewoluowały wraz z potrzebami stawianymi przez wiele systemów Google'a i zapewne nadal będą się zmieniać wraz z naturą naszych systemów.

A

adekwatność, 98
administrator systemu
 zarządzanie usługami, 25
agregowanie
 alarmów, 187
 pomiarów, 63
akceptowanie ryzyka, 47
aktualizacje procesu, 279
alarmy, 83, 132
alerty, 40
algorytm
 kontrolowanego opóźnienia, 270
 rotacyjny, 244
 z wagami, 248
 z wyborem jednostki, 246
wyboru
 podzbiory deterministyczne, 242
 podzbiory losowe, 240
analiza
 danych, 188
 przyczyn źródłowych, 120
 wydajności, 302
zdarzeń, 120, 179, 395, 454, 475
 chronologia incydentu, 477
 usprawnienia, 184
 w Google'u, 179
 wnioski, 476
 wprowadzanie kultury, 182
 zgłoszeń, 409
anatomia niezarządzanego incydentu, 174
archiwa, 341
automatyzacja, 87, 456
 awarie na wielką skalę, 104
 hierarchia klas, 92
 oszczędność czasu, 89

 platforma, 88
 skalowanie, 87
 system zarządzania klastrem, 101
 szybsze działania, 89
 szybsze naprawy, 88
 uruchamianie klastrów, 95
 zastosowania, 90
Auxon, 217
 implementacja, 219
 przyciąganie użytkowników, 220
awarie, 163, 470
 częściowe, 321
 kaskadowe, 263
 czynniki, 279
 przeciążenie serwerów, 268
 przyczyny, 264
 rozwiązywanie problemu, 283
 testowanie, 280
 powodowane sprawdzaniem stanu, 283

B

bezpieczeństwo, 142, 452
big data, 327
błędna interpretacja statystyk, 64
błędy
 przeciążenia, 257
 w konfiguracji, 96
Borgmon
 alarmy, 132
 powstanie systemu, 124
 przechowywanie danych, 126
 reguły, 129
 system monitorowania, 124
 zarządzanie konfiguracją, 135
zbieranie danych, 126

budowanie, 108
 crona, 318
budżet błędów, 54, 57, 468
 kształtowanie, 55
 usługi, 30

C

cele
 definiowanie, 65
 wybór, 66
centrum danych, 235
changelist, CL, 41
chronologia incydentu, 477
ciągle
 budowanie i wdrażanie, 108
 uczenie się, 400
ciągłość biznesowa, 335
cron, 315
 rozbudowana infrastruktura, 317
 uruchamianie dużej instalacji, 324
 w Google'u, 318
 zwiększone wymagania, 318
czas wymiany komunikatów, 238

D

debugowanie Shakespeare'a, 152
decydowanie o niepowodzeniu, 467
definicja
 celów, 65
 celów SLO, 468
 harówki, 69
deskryptory plików, 266
DevOps, 29
DiRT, Disaster and Recovery Testing, 145, 451
DNS, 230
dobre praktyki, 467
dokumentacja, 398
 stanu incydentu, 176, 473
dostęp do dysku, 304
dostępność, 29
 danych, 342
DSR, Direct Server Return, 234
duża liczbą odczytów, 300
dystrakcje, 406
dyżurny, 395
dyżury, 141, 400, 408
dzienniki, 152
dzierżawa dla kworum, 300

E

eksplozja oprogramowania, 117
eksportowanie zmiennych, 125
elastyczność systemu, 115
eliminowanie
 harówki, 69
 obciążenia, 284
 powtarzalnej pracy, 456
 przeciążenia operacyjnego, 411
 szkodliwego ruchu, 285
Escalator, 185
etykiety, 127
ewolucja prostego modelu opartego na PGP, 439

F

Fast Paxos
 analizowanie wydajności, 302
FIFO, first-in, first-out, 270
format BLOB, 340
formy wsparcia, 435
frameworki, 442
fronton, 229

G

generowanie reguł, 125
GFS, Google File System, 38, 96
Gmail, 84
GRE, Generic Routing Encapsulation, 234
GTape, 357

H

harówka, 45, 69
 definicja, 69
 obliczanie, 71
 ograniczenie, 71
hierarchia testów tradycyjnych, 193

I

ICS, Incident Command System, 175
idempotencja, 316
idempotentne eliminowanie niespójności, 97
identyfikowanie
 problematicznych zadań, 237
 źródeł problemów, 412
 źródła stresu, 412

- improwizacja, 394
- incydent, 177
- informacje
 - o anulowaniu, 275
 - o limicie czasu, 274
- infrastruktura dla oprogramowania, 40
- instalacja duża crona, 324
- integracja, 373
- integralność danych, 337
 - dostępność danych, 342
 - dwadzieścia cztery rodzaje błędów, 346
 - trudności z utrzymywaniem, 345
 - wybór strategii, 339
 - wymagania, 338
 - zasady SRE, 363
- interfejsy API, 117
- interpretacja statystyk, 64
- inżynier dyżurny, 140
- inżynieria, 29
 - koordynowania udostępniania, 368
 - odwrotna, 393
 - odwrotna usługi produkcyjnej, 394
 - oprogramowania, 211
 - budowanie kultury, 224
 - docieranie do celu, 225
 - udostępniania, 45, 105
 - ciągle budowanie i wdrażanie, 108
 - hermetyczny proces budowania, 107
 - model samoobsługowy, 106
 - wymuszanie polityk i procedur, 107
 - wysoka szybkość, 106
 - zarządzanie konfiguracją, 111
- inżynierowie SRE
 - angażowanie, 411
 - branżowi weterani, 450
 - dyżurni, 395
 - dyżury, 389
 - kolejność poznawania systemu, 390
 - komunikacja, 420
 - rozwój, 393
 - shadowing dyżurnych, 399
 - skład zespołu, 424
 - techniki szkoleniowe, 388
 - ukierunkowanie pracy, 392
 - współpraca, 423
 - współpraca z zespołami, 429
 - zadania, 389

J

- JIT, Just-in-Time, 277
- JVM, Java Virtual Machine, 381

K

- kolejki, 269
- kompetencja, 98
- komunikacja, 420
- konfiguracja
 - Borgmona, 135
 - procesu udostępniania, 370
- konsensus, 297
 - w środowisku rozproszonym, 291
- kontrola
 - przepływu, 237
 - wyników pomiarów, 67
- konwergencja, 371
- kopie
 - pełne, 345
 - przyrostowe, 345
 - zapasowe, 341, 349, 352
- koszty
 - obsługi zapytań, 245
 - operacyjne, 444
- kryzys wywołany testami, 164
- kształtowanie budżetu błędów, 55
- kulawe kaczki, 237
- kultura analizy
 - bez obwiniania, 30
 - zdarzeń, 182, 454

L

- liczba
 - replik, 305
 - zapytań na sekundę, 251
- lider, 320
- LIFO, last-in, first-out, 270
- limity
 - czasu, 274
 - na klienta, 252
- lista
 - kontrolna LCE, 479
 - kontrolna udostępniania, 371
 - zmian, 41
- lokalizacja replik, 306

Ł

łączenie żądań w porcje, 303

M

maszyny RSM, 293

metody

kolejkowania, 270

przywracania, 349

miękkie usuwanie, 347

minimalne interfejsy API, 117

model

ACID, 288

angażowania się zespołów, 434

oparty na PGP, 436

PGP, 434

wczesnego zaangażowania, 440

budowanie i implementacja, 440

etap projektowania, 440

udostępnianie, 440

zaangażowania oparty na wspólnej

odpowiedzialności, 445

modułowość, 117

monitorowanie, 29, 31, 40, 119, 469

alarmy, 132

białoskrzynkowe, 79

czarnoskrzynkowe, 79, 134

długoterminowe, 83

podział systemu, 133

problemów w okresowo uruchamianych

potokach, 330

reguły, 83

rozproszonych systemów, 312

system Borgmon, 124

systemów rozproszonych, 75

szczegółowość pomiarów, 81

wartości skrajne, 81

wykorzystanie szeregów czasowych, 124

złote sygnały, 79

motywowanie do zmian, 415

MTBF, Mean Time Between Failure, 192, 205

MTTF, Mean Time to Failure, 31

MTTR, Mean Time to Repair, 31, 88, 192

MTU, Maximum Transmission Unit, 234

Multi-Paxos, 298

MVC, Model-View-Controller, 332

MVP, Minimum Viable Product, 221

myślenie statystyczne i porównawcze, 393

N

narzędzia pomiarowe, 125

narzędzie Auxon, 217

niebezpieczne oprogramowanie, 201

niedostępność usługi, 267

niekontrolowane usuwanie danych, 359

nieliniowe skalowanie, 412

niepowodzenia, 179

nierównomierny podział pracy, 328

niezarządzane incydenty, 173

niezawodna obsługa

kolejek, 296

komunikatów, 296

niezawodne

maszyny RSM, 293

replikowane magazyny danych, 294

udostępnianie, 377

udostępnianie produktów, 367

niezawodność, 15, 102, 191, 316

notatki ze spotkania produkcyjnego, 481

O

obciążenie, 230, 235, 308

operacyjne, 144

wynikające z połączeń, 260

obliczanie harówki, 71

obserwator, 321

obsługa

blokad, 40

błędów przeciążenia, 257

przeciążenia, 251

zapytań, 245

żądań, 42

ochrona przed niebezpiecznym oprogramowaniem,
201

odciążanie, 270

odrzućcie ruchu, 284

ogłaszanie incydentu, 177

ograniczanie

harówki, 71

liczby żądań, 253

puli połączeń, 239

zakłóceń, 409

okresowe szeregowanie prac, 315

okresowo uruchamiany potok, 329

określanie

nazw, 126

wymagań, 454

opóźnienie, 29, 98, 274

dwumodalne, 276

w sieci, 301

Outalator, 186

agregowanie alarmów, 187

tagi, 188

widok incydentu, 187

widok kolejek, 186

P

pamięć, 38, 266

Paxos, 292

zastosowanie, 319

planowane zmiany, 280

planowanie

przepustowości, 29, 32, 121, 213, 374, 470

na podstawie celów, 215

wprowadzania produktu, 376

platformy flag funkcji, 378

podjmowanie decyzji, 457

podejście SRE, 29

podzbiory, 239

deterministyczne, 242

losowe, 240

podział

pracy, 328

systemu monitorowania, 133

pomiar

kontrolowanie wyników, 67

określanie szczegółowości, 81

ryzyka, 48

ponawianie

prób, 271

żądania, 257

poprzedniki celu, 216

potoki

okresowo uruchamiane, 329

przetwarzania danych, 327

wieloetapowe, 328

powtarzalność, 87

powtarzanie błędów, 170

poziom

krytyczności, 255

wykorzystania, 256

poziomy SLO, 59

poznawanie usługi i kontekstu, 412

prace inżynierskie, 72

problem

„pędzącego stada”, 330

„rozszczipienia mózgu”, 290

problemy, 147, *Patrz także* rozwiązywanie

problemów

proces

rozwiazywania problemów, 148

rozwoju oprogramowania, 376

udostępniania, 370

zarządzania incydentami, 175

procesor, 265

procesy certyfikacji, 453

produkt, 122

prognozowanie zapotrzebowania, 32

progresywne wprowadzanie usług, 467

protokół

SNMP, 126

TFTP, 169

przechowywanie

danych, 126

historii przestojoów, 170

stanu, 323

przeciążenie, 251, 257, 380, 470

operacyjne, 144, 411

serwera, 264, 268

przedstawianie kontekstu, 414

przeglądy gotowości produkcyjnej, 436

przełączanie awaryjne, 290

przenośna przepustowość, 452

przepływ komunikatów, 298

przepustowość, 213, 470

obciążenia, 308

przestoje, 185

usługi, 61

przywracanie

danych, 356

z systemu GTape, 357

publiczne nagradzanie ludzi, 183

pusta pamięć podręczna, 276

Q

QoS, Quality of Service, 169

QPS, queries per second, 43, 60, 264

R

Rapid, 109

raporty, 189

- reagowanie w przypadku awarii, 29, 31, 120
- reguły, 129
 - równoważenia obciążenia, 244
 - zgłaszania alarmów, 132
- rejestrowanie wskaźników, 62
- rekurencyjny podział obowiązków, 175
- replikacja, 351
 - liczba, 305
 - lokalizacja, 306
- replikowane magazyny danych, 294
- restartowanie serwerów, 283
- retencja, 346
- rola inżynierów LCE, 369
- rozgałęzienia, 108
- rozkład obciążenia zadań, 236
- rozproszony przepływ danych i procesów, 336
- rozruch, 276
- rozwiązywanie problemów, 147, 169
 - awarie kaskadowe, 283
 - badanie, 151
 - diagnoza, 153
 - dzienniki, 152
 - ocena sytuacji, 150
 - typowe pułapki, 149
 - upraszczanie i przyspieszanie, 162
 - ustalanie przyczyn symptomu, 154
 - w praktyce, 150
- rozwój produktów, 121
- równoważenie obciążenia, 308
 - reguły, 244
 - system DNS, 230
 - w centrum danych, 235
 - wirtualne adresy IP, 232
- RSM, Replicated State Machine, 293
- RTT, round-trip time, 230, 238
- ryzyko, 47
 - określanie tolerancji, 50, 53
 - pomiar, 48
 - zarządzanie, 47

S

- serwery pośredniczące, 302
- shadowing dyżurnych, 399
- sieci, 39
- skalowanie, 87, 345
 - obciążenia roboczego, 300
- skład zespołu, 424
- SLA, service level agreement, 45, 61
- SLI, service level indicators, 59
- SLO, service level objective, 45, 60, 67
- SLO usługi, 30
- SNMP, Simple Network Management Protocol, 126
- SOA, Service-Oriented Architecture, 100
- spotkanie produkcyjne, 420
 - notatki, 481
- sprawdzanie
 - reguł, 129
 - stanu, 283
- sprawność, 29, 33, 246
- sprzęt, 35
 - oprogramowanie, 37
- SRE, Site Reliability Engineering, 15
- stabilność systemu, 115
- stabilny lider, 303
- stan
 - incydentu, 473
 - po awarii, 145
 - przepływu, 405, 406
- standaryzacja wskaźników, 64
- stopniowa degradacja, 270
- stos, 278
- studium przypadku
 - App Engine, 158
 - Auxon, 213
 - błędne algorytmy, 291
 - niekontrolowane usuwanie danych, 359
 - problem „rozszczipienia mózgu”, 290
 - przełączanie awaryjne, 290
 - przeniesienie DFP do F1, 430
 - przywracanie z systemu GTape, 357
 - Viceroy, 425
- sygnały poziomu wykorzystania, 256
- symulacje, 453
- system
 - Borgmon, 124
 - DNS, 230
 - Escalator, 185
 - GTape, 357
 - Rapid, 109
 - Workflow, 332
- systemy
 - oprogramowania, 40
 - elastyczność, 115
 - stabilność, 115
 - przywracania danych, 343
 - rozproszone, 75
 - tworzenia kopii zapasowych, 343
 - zarządzania klastrami, 101

- sytuacje kryzysowe
 - spowodowane procesem, 167
 - spowodowane zmianami, 165
- szereg czasowy, 123, 126
- szkolenia, 453
- szybkie
 - udostępnianie produktu, 46
 - wprowadzanie zmian, 30

Ś

- śledzenie
 - przestojów, 185
 - analizy danych, 188
 - Escalator, 185
 - Outalator, 186
 - stanu prac crona, 319
- średni czas
 - do wystąpienia awarii, 31
 - między awariami, 192
 - naprawy, 31, 192
- środowisko
 - budowania, 198
 - produkcyjne, 35
 - próbkowanie, 208
 - zarządzanie konfiguracją, 204
- programistyczne, 41
- rozproszone, 287
 - cron, 315
 - konsensus, 289, 291, 305
 - monitorowanie systemów uzgadniania konsensusu, 312
 - obsługa kolejek i komunikatów, 296
 - okresowe szeregowanie prac, 315
 - okresowo uruchamiany potok, 329
 - usługi koordynacji i blokowania, 294
 - wrażanie systemu, 305
 - wybór lidera, 294
 - wydajność uzgadniania konsensusu, 297, 301
- testowania, 198

T

- tabela dostępności, 465
- tagi, 188
- techniki
 - skutecznej pracy, 424
 - szkoleniowe, 388
- terminy zakończenia testów, 204

- test produkcyjny usługi DNS, 96
- testowanie
 - gotowości, 451
 - klientów, 282
 - niekrytycznych zadań zaplecza, 282
 - niezawodności, 191
 - odporności na katastrofy, 451
 - pod kątem awarii kaskadowych, 280
 - skalowalnych narzędzi, 200
 - systemu, 193
 - zarządzania katastrofami, 202

- testy, 108, 121
 - dymne, 194
 - integracyjne, 193, 207
 - jednostkowe, 193
 - kanarkowe, 196
 - konfiguracji, 195
 - obciążeniowe, 196, 380
 - proaktywne, 170
 - produkcyjne, 194
 - regresyjne, 194
 - sprawności, 194
 - statystyczne, 203
 - w dużej skali, 199
 - w środowisku produkcyjnym, 96
- TFTP, Trivial File Transfer Protocol, 169
- tolerancja ryzyka, 53
 - dla usługi, 50
- tryb
 - awaryjny, 284
 - operacyjny, 412
- twierdzenie CAP, 288
- tworzenie
 - dokumentacji, 398
 - listy kontrolnej udostępniania, 373
 - pakietów, 109
 - podzbiorów, 239
 - środowiska testowania, 198

U

- udostępnianie, 105
 - działania klientów, 375
 - konfigurowanie procesu, 370
 - lista kontrolna, 371, 373
 - nowości, 372
 - procesy i automatyzacja, 375
 - produktów w dużej skali, 367
 - prostych zmian, 118

- rodzaje błędów, 374
- techniki, 377
- zależności zewnętrzne, 376
- uniwersalne wsparcie, 445
- uporządkowanie danych, 43
- upraszczanie, 371
- uruchamianie klastrów, 95, 100
- usługa
 - Bigtable, 84
 - Shakespeare, 42, 285
- usługi
 - budżet błędów, 30, 54
 - cele, 60
 - dla konsumentów, 50
 - docelowy poziom dostępności, 50, 53
 - globalny planowany przestój, 61
 - infrastrukturalne, 53
 - kandydujące do wczesnego zaangażowania, 439
 - koordynacji i blokowania, 294
 - koszty, 51, 53
 - kształtowanie budżetu błędów, 55
 - obliczanie ryzyka, 48
 - obsługi blokad, 40
 - poziom SLO, 56
 - produkcyjne
 - dobre praktyki, 467
 - rodzaje awarii, 53
 - rodzaje niepowodzeń, 51
 - tolerancja ryzyka, 50, 53
 - umowy, 61
 - wskaźniki, 59, 62
 - wyznaczone oczekiwania, 67
 - zmiany w rozwoju, 441
- ustalanie przyczyn symptomu, 154
- utrata danych, 343
- utrzymywanie integralności danych, 345
- uzgadnianie konsensusu, 297

V

- Viceroy, 425
- VIP, virtuae IP address, 232

W

- wartość automatyzacji, 87
- wątki, 266
- wczesne
 - wykrywanie, 353
 - zaangażowanie, 439

- wdrażanie, 110
- rozproszony system, 305
- wektory, 127
- wirtualne adresy IP, 232
- wnioski, 447
- Workflow, 332
 - etapy wykonywania, 334
 - gwarancje poprawności, 334
- wprowadzanie zmian, 279
- wskaźnik
 - MTTR, 192
 - poziomu usługi, 59
- wskaźniki, 62
 - agregowanie pomiarów, 63
 - rejestrwanie, 62
 - standaryzacja, 64
- wspomaganie inżynierii oprogramowania, 223
- wybór
 - celów, 66
 - lidera, 294
 - podzbioru, 239
- wyczerpanie zasobów, 265
- wydajność, 29, 33, 302
 - uzgadniania konsensusu, 301
- wykrywanie niespójności, 96
- wymagania w chmurze, 341
- wyniki negatywne eksperymentu, 156
- wzorce architektury systemu, 292
- wzorzec projektowy
 - model – widok – kontroler, 332
 - obciążenie w kształcie pasków, 331
 - okresowo uruchamiany potok danych, 328
 - potok danych, 327
- wzrost organiczny, 280

Z

- zaangażowanie zespołów, 433
- zakłócenia, 403
- zależności między zasobami, 267
- zamknięcie procesu, 279
- zapewnianie
 - ciągłości biznesowej, 335
 - zasobów, 33
- zapobieganie przeciążeniu serwerów, 268
- zapytania na sekundę, 43
- zarządzanie, 385
 - incydentami, 173
 - anatomia niezarządzanego incydentu, 174
 - aspekty procesu, 175

zarządzanie

- centrum dowodzenia, 176
- dokumentacja, 176
- incydenty niezarządzane, 173
- ogłaszanie, 177
- przekazanie zadania, 176
- rekurencyjny podział obowiązków, 175
- katastrofami, 202
- kolejkami, 269
- konfiguracją, 111
 - środowiska produkcyjnego, 204
- krytycznym stanem, 287
- maszynami, 37
- niezawodnością usługi, 57
- obciążeniem operacyjnym, 404
- ryzykiem, 47
- usługami, 25, 26
- zakłóceniami, 404
- zmianą, 29, 32

zarządzany incydent, 176

zasady, 45

zastosowanie

- automatyzacji, 90
- Paxosa, 319

zbieranie eksportowanych danych, 126

zespół

- eksploatacji, 25
- rozwoju produktu, 25
- LCE, 381
 - lista kontrolna, 382
 - problemy, 383
- SRE, 419, 433, 463, 471
 - zadania, 29

zgłoszenie problemu, 150, 408

zmiany w rozwoju usług, 441

zreplikowany dziennik, 306

zwiększanie ilości zasobów, 283

Ż

żądania na sekundę, 264

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA



Helion SA

SRE — niezbędne, gdy podstawą sukcesu jest niezawodność systemu!

Historia firmy Google może być analizowana na wiele sposobów. Można podziwiać jej błyskawiczny rozwój i niesamowitą umiejętność skalowania usług. Na uwagę zasługuje jej innowacyjność i zaangażowanie w rozwijanie technologii, które jeszcze wczoraj wydawały się fikcją. Nie możemy traktować tradycji jako autorytetu, musimy nauczyć się myśleć od nowa i nie mamy czasu na czekanie — tak brzmi filozofia firmy, które to podejście przyswieca jej ogromnemu sukcesowi. W ten sposób narodziły się praktyki z obszarów rozwoju oprogramowania, ale i zarządzania zwane SRE: Site Reliability Engineering. Każda firma może je zastosować, ale tylko Google mogła je wymyślić.

Jeśli chcesz zrozumieć filozofię SRE, trzymasz w ręku właściwą, choć nietypową książkę. Jest to zbiór najciekawszych esejów i artykułów autorstwa osób odpowiedzialnych za SRE w Google. Z lektury tych esejów dowiesz się, w jaki sposób zaangażowanie w cały cykl życia oprogramowania umożliwiło skuteczne budowanie, wdrażanie, monitorowanie i konserwowanie największych systemów informatycznych świata. Poznasz zasady i praktyki, które pozwalają inżynierom z Google tworzyć bardziej skalowalne i niezawodne oraz wydajniejsze systemy. Zaprezentowane tu podejście SRE możesz naturalnie bezpośrednio wdrożyć w swojej organizacji.

W tej książce:

- wyjaśniono, czym jest Site Reliability Engineering (SRE) i dlaczego podejście to różni się od tradycyjnych praktyk z branży IT
- opisano wzorce, operacje i obszary zainteresowania wpływające na pracę inżynierów SRE
- przedstawiono zasady codziennej pracy inżynierów SRE
- pokazano, jak budować duże rozproszone systemy informatyczne i jak nimi zarządzać

Betsy Beyer — pisze dokumentację techniczną dla Google. Specjalizuje się w podejściu SRE. Kilka lat temu była wykładowczynią na Uniwersytecie Stanforda.

Chris Jones — jest inżynierem SRE odpowiedzialnym za Google App Engine. Wcześniej odpowiadał za statystyki reklam, hurtownie danych i system pomocy technicznej w Google.

Jennifer Petoff — jest menedżerem programu w zespole SRE w Google. Zarządza dużymi globalnymi projektami z wielu dziedzin, takich jak badania naukowe, inżynieria czy kadry.

Niall Murphy — kieruje zespołem SRE odpowiedzialnym za reklamy w Google. Przewodniczy organizacji INEX — irlandzki hub internetowy. Jest też autorem oraz współautorem wielu prac i książek technicznych.

Helion

księgarnia internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

Sprawdź najnowsze promocje:
● <http://helion.pl/promocje>
Książki najchętniej czytane:
● <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
● <http://helion.pl/novosci>

ISBN 978-83-283-3730-5



9 788328 337305

Informatyka w najlepszym wydaniu

cena: 79,00 zł

ślęgnij po WIĘCEJ



KOD KORZYŚCI