

Local

Confiabilidade Engenharia

COMO O GOOGLE EXECUTA SISTEMAS DE PRODUÇÃO

Editado por Betsy Beyer, Chris Jones,
Jennifer Petoff e Niall Richard Murphy

Site Reliability Engineering

The overwhelming majority of a software system's lifespan is spent in use, not in design or implementation. So, why does conventional wisdom insist that software engineers focus primarily on the design and development phases of large-scale computing systems?

In this collection of essays and articles, key members of Google's Site Reliability Engineering team explain how and why their commitment to the *entire* lifecycle has enabled the company to successfully build, deploy, monitor, and maintain some of the largest software systems in the world. You'll learn principles and practices that enable Google engineers to make systems more scalable, reliable, and efficient—lessons directly applicable to your organization.

This book is divided into four sections:

- **Introduction**—Learn what Site Reliability Engineering is and why it differs from conventional IT industry practices
- **Principles**—Examine the patterns, behaviors, and areas of concern that influence the work of a Site Reliability Engineer (SRE)
- **Practices**—Understand the theory and practice of an SRE's day-to-day work: building and operating large distributed computing systems
- **Management**—Explore Google best practices for training, communication, and meetings that your organization can use

Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy are members of Google's Site Reliability Engineering organization, which is responsible for the care and feeding of Google's production systems.

“In-depth technical and managerial practices that every company can use, but only Google could invent.”

—Thomas A. Limoncelli
ex-Google SRE and coauthor of
The Practice of Cloud System Administration (Addison-Wesley)

“Essential reading for anyone running highly available web services at scale.”

—Adrian Cockcroft
Battery Ventures,
former Netflix Cloud Architect

“You owe it to yourself and your organization to read this book and try out these ideas for yourself.”

—Jez Humble
coauthor of *Continuous Delivery* (Addison-Wesley) and *Lean Enterprise* (O'Reilly)

SYSTEM ADMINISTRATION

US \$44.99

CAN \$51.99

ISBN: 978-1-491-92912-4



5 4 4 9 9

9 781491 929124



Twitter: @oreillymedia
facebook.com/oreilly

Elogios à Engenharia de Confiabilidade do Site

Os SREs do Google prestaram um enorme serviço ao nosso setor ao redigir os princípios, práticas e padrões—arquitetônicos e culturais—que permitem que suas equipes combinem entrega contínua com confiabilidade de classe mundial em escala absurda. Você deve a si mesmo e à sua organização ler este livro e experimentar essas ideias por si mesmo.

—Jez Humble, coautor de Continuous Delivery e Lean Enterprise

Lembro-me de quando o Google começou a falar em conferências de administração de sistemas.

Era como ouvir uma palestra em um show de répteis por um especialista em monstros de Gila. Claro, foi divertido ouvir sobre um mundo muito diferente, mas no final o público voltaria para suas lagartixas.

Agora vivemos em um universo alterado onde as práticas operacionais do Google não são tão distantes daquelas que trabalham em menor escala. De repente, as melhores práticas de SRE que foram aperfeiçoadas ao longo dos anos agora são de grande interesse para o resto de nós. Para aqueles de nós que enfrentam desafios de escala, confiabilidade e operações, este livro não chega muito cedo.

—David N. Blank-Edelman, Diretor, Conselho de Administração da USENIX e co-organizador fundador da SREcon

Estou esperando por este livro desde que saí do castelo encantado do Google.

É o evangelho que estou pregando aos meus colegas de trabalho.

—Björn Rabenstein, líder de equipe de engenharia de produção no SoundCloud, desenvolvedor do Prometheus e Google SRE até 2013

Uma discussão completa da Engenharia de Confiabilidade do Site da empresa que inventou o conceito. Inclui não apenas os detalhes técnicos, mas também o processo de pensamento, objetivos, princípios e lições aprendidas ao longo do tempo. Se você quiser aprender o que SRE realmente significa, comece aqui.

—Russ Allbery, SRE e Engenheiro de Segurança

Com este livro, os funcionários do Google compartilharam os processos que tomaram, incluindo os erros, que permitiram que os serviços do Google se expandissem em grande escala e grande confiabilidade. Eu recomendo que qualquer pessoa que queira criar um conjunto de serviços integrados que eles esperam escalar leia este livro. O livro fornece um guia de informações privilegiadas para a criação de serviços sustentáveis.

—Rik Farrow, USENIX

Escrever serviços de grande escala como o Gmail é difícil. Executá-los com alta confiabilidade é ainda mais difícil, especialmente quando você os troca todos os dias. Este abrangente “livro de receitas” mostra como o Google faz isso, e você achará muito mais barato aprender com nossos erros do que cometê-los você mesmo.

—Urs Hözle, vice-presidente sênior de infraestrutura técnica, Google

Engenharia de confiabilidade do site

Como o Google executa os sistemas de produção

**Editado por Betsy Beyer, Chris Jones, ,
Jennifer Peto e Niall Richard Murphy**

Pequim Boston Farnham Sebastopol Tóquio

O'REILLY®

Engenharia de confiabilidade do site

Editado por Betsy Beyer, Chris Jones, Jennifer Petoff e Niall Richard Murphy

Copyright © 2016 Google, Inc. Todos os direitos reservados.

Impresso nos Estados Unidos da América.

Publicado por O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Os livros da O'Reilly podem ser adquiridos para uso educacional, comercial ou promocional de vendas. Edições online também estão disponíveis para a maioria dos títulos (<http://safaribooksonline.com>). Para mais informações, entre em contato com nosso departamento de vendas corporativo/institucional: 800-998-9938 ou corporate@oreilly.com.

Editor: Brian Anderson

Indexador: Judy McConville

Editor de Produção: Kristen Brown

Designer de Intérios: David Futato

Editor de texto: Kim Cofer

Designer de capa: Karen Montgomery

Revisor: Rachel Monaghan

Ilustrador: Rebecca Demarest

Abril de 2016: Primeira edição

Histórico de revisões para a primeira edição

21-03-2016: Primeira versão

Veja <http://oreilly.com/catalog/errata.csp?isbn=9781491929124> para detalhes do lançamento.

O logotipo O'Reilly é uma marca registrada da O'Reilly Media, Inc. Site Reliability Engineering, a imagem da capa e a imagem comercial relacionada são marcas registradas da O'Reilly Media, Inc.

Embora o editor e os autores tenham feito esforços de boa fé para garantir que as informações e instruções contidas neste trabalho sejam precisas, o editor e os autores se isentam de qualquer responsabilidade por erros ou omissões, incluindo, sem limitação, responsabilidade por danos resultantes do uso ou confiança neste trabalho. O uso das informações e instruções contidas neste trabalho é por sua conta e risco. Se qualquer amostra de código ou outra tecnologia que este trabalho contém ou descreve está sujeita a licenças de código aberto ou direitos de propriedade intelectual de terceiros, é sua responsabilidade garantir que seu uso esteja em conformidade com tais licenças e/ou direitos.

978-1-491-92912-4

[LSI]

Índice

Prefácio	xiii
Prefácio	xv

Parte I. Introdução

1. Introdução.....	3
A Abordagem Sysadmin para Gerenciamento de Serviços	3
Abordagem do Google para gerenciamento de serviços: engenharia de confiabilidade do site	5
Princípios do SRE	7
O fim do começo	12
2. O Ambiente de Produção no Google, na Visão de um SRE.....	13
Hardware	13
Software de sistema que “organiza” o hardware	15
Outros softwares de sistema	18
Nossa Infraestrutura de Software	19
Nosso Ambiente de Desenvolvimento	19
Shakespeare: um serviço de amostra	20

Parte II. Princípios

3. Abraçando o Risco.....	25
Gestão de risco	25
Medindo o risco do serviço	26
Tolerância de Risco de Serviços	28

Motivação para erros de orçamento	33
4. Objetivos do nível de serviço.	37
Terminologia do nível de serviço	37
Indicadores na prática	40
Objetivos na prática	43
Acordos na prática	47
5. Eliminando o Trabalho.	49
Trabalho definido	49
Por que menos trabalho é	51
melhor O que se qualifica como engenharia?	52
A labuta é sempre ruim?	52
Conclusão	54
6. Monitoramento de Sistemas Distribuídos.	55
Definições	55
Por que Monitorar?	56
Definindo expectativas razoáveis para monitorar sintomas	57
versus causas Caixa preta versus caixa branca Os quatro	58
sinais dourados de preocupação com sua cauda (ou,	59
instrumentação e desempenho)	60
	61
Escolhendo uma resolução apropriada para medições o mais simples	62
possível, não é mais simples vincular esses princípios juntos	62
Monitoramento para a conclusão a longo prazo	63
	64
	66
7. A evolução da automação no Google.	67
O valor da automação O valor	67
para o Google SRE Os casos de	70
uso da automação Automatize-se fora	70
de um trabalho: automatize TODAS as coisas!	73
Acalmando a dor: Aplicando a automação para Turnups de cluster Borg:	75
Nascimento do computador em escala de armazém Confiabilidade é o recurso	81
fundamental Recomendações	83
	84
8. Engenharia de Liberação.	87
O papel de um engenheiro de lançamento	87
Filosofia	88

Construção e implantação contínuas	90
Gerenciamento de configurações	93
Conclusões	95
9. Simplicidade.....	97
Estabilidade do sistema versus	97
agilidade A virtude do chato Não vou	98
desistir do meu código!	98
A métrica de "linhas negativas de código"	99
APIs Mínimas Modularidade de lançamento	99
Simplicidade Uma simples conclusão	100
	100
	101
<hr/>	
Parte III. Práticas	
10. Alerta prático de dados de séries temporais.....	107
A ascensão da Borgmon	108
Instrumentação de aplicativos Coleta de	109
armazenamento de dados exportados	110
no Time-Series Arena Avaliação de regras	111
Alerta Fragmentação da topologia de	114
monitoramento Monitoramento de caixa	118
preta Mantendo a configuração dez anos	119
depois...	120
	121
	122
11. Estar de plantão.....	125
Introdução	125
Vida de um engenheiro de plantão	126
Chamada Equilibrada	127
Sentido seguro	128
Evitando Carga Operacional Inapropriada	130
Conclusões	132
12. Solução de problemas efetiva.....	133
Teoria	134
Na prática	136
Resultados negativos são mágicos	144
Estudo de caso	146

Facilitando a solução de problemas	150
Conclusão	150
13. Resposta de Emergência.....	151
O que fazer quando os sistemas falham	151
Emergência induzida por teste Emergência	152
induzida por mudança Emergência induzida	153
por processo Todos os problemas têm	155
soluções Aprenda com o passado. Não o	158
repita.	158
Conclusão	159
14. Gerenciando Incidentes.....	161
Incidentes não gerenciados	161
A anatomia de um incidente não gerenciado	162
Elementos do processo de gerenciamento de incidentes	163
Um incidente gerenciado Quando declarar um incidente	165
em resumo	166
	166
15. Cultura Postmortem: Aprendendo com o Fracasso.....	169
A filosofia postmortem do Google Colabore	169
e compartilhe conhecimento Apresentando	171
uma cultura postmortem Conclusão e	172
melhorias contínuas	175
16. Rastreamento de interrupções.....	177
Escada rolante	178
Outalador	178
17. Teste de Confiabilidade.....	183
Tipos de teste de software	185
Criando um ambiente de teste e compilação	190
Teste em escala	192
Conclusão	204
18. Engenharia de Software em SRE.....	205
Por que a engenharia de software no SRE é importante? 205	
Estudo de caso da Auxon: Histórico do projeto e planejamento de capacidade	207
baseado em intenção do espaço do problema Fomentando a engenharia de	209
software no SRE Conclusões	218
	222

19. Balanceamento de carga no frontend.	223
Poder não é a resposta	223
Balanceamento de carga usando DNS	224
Balanceamento de carga no endereço IP virtual	227
20. Balanceamento de carga no Datacenter.	231
O Caso Ideal	232
Identificando Tarefas Ruins: Controle de Fluxo e Patos Mancos	233
Limitando o pool de conexões com subconjuntos	235
Políticas de balanceamento de carga	240
21. Manuseio de Sobrecarga.	247
As armadilhas das “consultas por segundo”	248
Limites por cliente	248
Limitação do lado do cliente	249
Criticamente	251
Sinais de utilização	253
Lidando com Erros de Sobrecarga	253
Carregar das Conexões	257
Conclusões	258
22. Resolvendo Falhas em Cascata.	259
Causas de falhas em cascata e projetar para evitá-las	260
Evitando a sobrecarga do servidor	265
Inicialização lenta e armazenamento em cache frio	274
Condições de acionamento para falhas em cascata	276
Teste de falhas em cascata	278
Etapas imediatas para solucionar falhas em cascata	280
Observações finais	283
23. Gerenciando Estado Crítico: Consenso Distribuído para Confiabilidade.	285
Motivando o Uso do Consenso: Falha na Coordenação de Sistemas Distribuídos	288
Como funciona o consenso distribuído	289
Padrões de Arquitetura de Sistema para Consenso Distribuído	291
Desempenho de Consenso Distribuído	296
Implantação de Sistemas	304
Baseados em Consenso Distribuído	312
Monitoramento de Sistemas de Consenso Distribuído	313
24. Agendamento Periódico Distribuído com Cron.	315
Cron	315
Cron Jobs e Idempotência	316

Cron em grande escala	317
Construindo o Cron no Google	319
Resumo	326
25. Pipelines de Processamento de Dados	327
Origem do padrão de projeto de pipeline Efeito	327
inicial do Big Data no padrão de pipeline simples Desafios com o	328
padrão de pipeline periódico Problemas causados por distribuição	328
desigual de trabalho Desvantagens de pipelines periódicos em	328
ambientes distribuídos Introdução ao Google Workflow Estágios de execução	329
no fluxo de trabalho Garantindo a continuidade dos negócios Resumo e	333
conclusão Observações	335
	337
	338
26. Integridade dos dados: o que você lê é o que você escreveu	339
Requisitos rígidos de integridade de dados	340
Objetivos do Google SRE na manutenção da integridade e disponibilidade de dados	344
Como o Google SRE enfrenta os desafios da integridade de dados Estudos de caso	349
Princípios gerais de SRE aplicados à integridade de dados Conclusão	360
	367
	368
27. Lançamentos de produtos confiáveis em escala	369
Engenharia de Coordenação de Lançamento	370
Configurando um Processo de Lançamento	372
Desenvolvendo uma Lista de Verificação de	375
Lançamento Técnicas Selecionadas para Lançamentos	380
Confiáveis Desenvolvimento de LCE Conclusão	384
	387
<hr/>	
Parte IV. Gestão	
28. Acelerando SREs para On-Call e Além	391
Você contratou seu(s) próximo(s) SRE(s), e agora?	391
Experiências Iniciais de Aprendizagem: O Caso da Estrutura Sobre o Caos	394
Criando Engenheiros Reversos Estelares e Pensadores Improvisados	397
	400
Considerações finais	406

29. Lidando com interrupções.....	407
Gerenciando a carga operacional	408
Fatores para determinar como as interrupções são tratadas	408
Máquinas imperfeitas	409
30. Incorporando um SRE para se Recuperar da Sobrecarga Operacional.....	417
Fase 1: conheça o serviço e obtenha o contexto	418
Fase 2: Contexto de Compartilhamento	420
Fase 3: Impulsionando a Mudança	421
Conclusão	423
31. Comunicação e Colaboração em SRE.....	425
Comunicações: Colaboração de Reuniões de Produção no SRE	426
Estudo de Caso de Colaboração no SRE: Viceroy	430
Colaboração fora do SRE	432
Estudo de caso: migração do DFP para F1	437
Conclusão	440
32. O Modelo Evolutivo de Engajamento do SRE.....	441
Engajamento do SRE: o que, como e por quê	441
O Modelo PRR	442
O Modelo de Engajamento SRE	443
Revisões de prontidão de produção: modelo PRR simples	444
Evoluindo o Modelo Simples de PRR: Engajamento Antecipado	448
Desenvolvimento de Serviços em Evolução: Estruturas e Plataforma SRE	451
Conclusão	456

Parte V. Conclusões

33. Lições aprendidas de outras indústrias.....	459
Conheça nossos veteranos da indústria	460
Testes de Preparação e Desastres	462
Cultura post mortem	465
Automatizando o trabalho repetitivo e a sobrecarga operacional	467
Tomada de Decisões Estruturadas e Racionais	469
Conclusões	470
34. Conclusão.....	473

A. Tabela de Disponibilidade.....	477
B. Uma coleção de melhores práticas para serviços de produção.....	479
C. Exemplo de Documento de Estado do Incidente.....	485
D. Exemplo Postmortem.....	487
E. Lista de Verificação de Coordenação de Lançamento.....	493
F. Exemplo de Ata de Reunião de Produção.....	497
Bibliografia.....	501
Índice.....	511

Prefácio

A história do Google é uma história de expansão. É uma das grandes histórias de sucesso da indústria de computação, marcando uma mudança para negócios centrados em TI. O Google foi uma das primeiras empresas a definir o que o alinhamento entre negócios e TI significava na prática e passou a informar o conceito de DevOps para uma comunidade de TI mais ampla. Este livro foi escrito por uma ampla amostra das mesmas pessoas que tornaram essa transição uma realidade.

O Google cresceu em um momento em que o papel tradicional do administrador do sistema estava sendo transformado. Questionou a administração do sistema, como se dissesse: não podemos ter a tradição como autoridade, temos que pensar de novo e não temos tempo para esperar que todos os outros se atualizem. Na introdução de *Principles of Network and System Administration* [Bur99], afirmei que a administração de sistemas era uma forma de engenharia de computação humana. Isso foi fortemente rejeitado por alguns revisores, que disseram “ainda não estamos no estágio em que podemos chamar isso de engenharia”. Na época, senti que o campo havia se perdido, preso em sua própria cultura feiticeira, e não conseguia ver um caminho a seguir. Então, o Google traçou uma linha no silício, forçando esse destino a existir. A função revisada foi chamada de SRE, ou Site Reliability Engineer. Alguns de meus amigos estavam entre os primeiros desta nova geração de engenheiros; eles formalizaram usando software e automação. Inicialmente, eles eram ferozmente reservados, e o que acontecia dentro e fora do Google era muito diferente: a experiência do Google era única. Ao longo do tempo, informações e métodos fluíram em ambas as direções. Este livro mostra a disposição de deixar o pensamento SRE sair das sombras.

Aqui, vemos não apenas como o Google construiu sua infraestrutura lendária, mas também como estudou, aprendeu e mudou de ideia sobre as ferramentas e tecnologias ao longo do caminho. Nós também podemos enfrentar desafios assustadores com um espírito aberto. A natureza tribal da cultura de TI muitas vezes entrincheira os profissionais em posições dogmáticas que impedem a indústria. Se o Google superou essa inércia, nós também podemos.

Este livro é uma coleção de ensaios de uma empresa, com uma única visão comum. O fato de as contribuições estarem alinhadas em torno do objetivo de uma única empresa é o que a torna especial. Existem temas comuns e personagens comuns (sistemas de software)

que reaparecem em vários capítulos. Vemos as escolhas de diferentes perspectivas e sabemos que elas se correlacionam para resolver interesses conflitantes. Os artigos não são peças rigorosas, acadêmicas; são relatos pessoais, escritos com orgulho, em uma variedade de estilos pessoais e da perspectiva de conjuntos de habilidades individuais. Eles são escritos com bravura e com uma honestidade intelectual que é refrescante e incomum na literatura da indústria. Alguns afirmam “nunca faça isso, sempre faça aquilo”, outros são mais filosóficos e hesitantes, refletindo a variedade de personalidades dentro de uma cultura de TI e como isso também desempenha um papel na história. Nós, por nossa vez, os lemos com a humildade de observadores que não fizeram parte da jornada, e não temos todas as informações sobre os inúmeros desafios conflitantes. Nossas muitas perguntas são o verdadeiro legado do volume: Por que eles não fizeram X? E se eles tivessem feito Y? Como veremos isso nos próximos anos? É comparando nossas próprias ideias com o raciocínio aqui que podemos medir nossos próprios pensamentos e experiências.

A coisa mais impressionante de tudo sobre este livro é a sua própria existência. Hoje, ouvimos uma cultura descarada de “apenas me mostre o código”. Uma cultura de “não faça perguntas” cresceu em torno do código aberto, onde a comunidade, e não a experiência, é defendida. A Google é uma empresa que se atreveu a pensar os problemas desde os primeiros princípios e a empregar os melhores talentos com uma elevada proporção de doutores. As ferramentas eram apenas componentes em processos, trabalhando ao lado de cadeias de software, pessoas e dados. Nada aqui nos diz como resolver problemas universalmente, mas esse é o ponto. Histórias como essas são muito mais valiosas do que o código ou os designs em que resultaram. As implementações são efêmeras, mas o raciocínio documentado não tem preço. Raramente temos acesso a esse tipo de insight.

Esta, então, é a história de como uma empresa fez isso. O fato de haver muitas histórias sobrepostas nos mostra que o dimensionamento é muito mais do que apenas uma ampliação fotográfica de uma arquitetura de computador de livro didático. Trata-se de dimensionar um processo de negócios, e não apenas o maquinário. Esta lição por si só vale seu peso em papel eletrônico.

Não nos envolvemos muito em revisões autocriticas no mundo de TI; como tal, há muita reinvenção e repetição. Por muitos anos, houve apenas a comunidade de conferências USENIX LISA discutindo infraestrutura de TI, além de algumas conferências sobre sistemas operacionais. É muito diferente hoje, mas este livro ainda parece uma oferta rara: uma documentação detalhada da etapa do Google através de uma época divisória de águas. A história não é para copiar – embora talvez para emular – mas pode inspirar o próximo passo para todos nós.

Há uma honestidade intelectual única nestas páginas, expressando liderança e humildade. São histórias de esperanças, medos, sucessos e fracassos. Saúdo a coragem de autores e editores em permitir tal franqueza, para que nós, que não participamos das experiências práticas, também possamos nos beneficiar das lições aprendidas dentro do casulo.

— Mark Burgess,
autor de Em Busca da Certeza
Oslo, março de 2016

Prefácio

A engenharia de software tem isso em comum com ter filhos: o trabalho de parto antes do nascimento é doloroso e difícil, mas o trabalho de parto após o nascimento é onde você realmente gasta a maior parte de seu esforço. No entanto, a engenharia de software como disciplina gasta muito mais tempo falando sobre o primeiro período do que sobre o segundo, apesar das estimativas de que 40 a 90% dos custos totais de um sistema são corridos após o nascimento.¹ O modelo popular da indústria que concebe a implantação, software operacional como sendo “estabilizado” na produção e, portanto, precisando de muito menos atenção dos engenheiros de software, está errado.

Através desta lente, então, vemos que se a engenharia de software tende a se concentrar em projetar e construir sistemas de software, deve haver outra disciplina que se concentre em todo o ciclo de vida dos objetos de software, desde o início, passando pela implantação e operação, refinamento, e eventual desmantelamento pacífico. Essa disciplina usa – e precisa usar – uma ampla gama de habilidades, mas tem preocupações distintas de outros tipos de engenheiros. Hoje, nossa resposta é a disciplina que o Google chama de Engenharia de Confiabilidade do Site.

Então, o que exatamente é a Engenharia de Confiabilidade do Site (SRE)? Admitimos que não é um nome particularmente claro para o que fazemos – praticamente todos os engenheiros de confiabilidade de sites do Google são questionados sobre o que exatamente é e o que eles realmente fazem regularmente.

Descompactando um pouco o termo, em primeiro lugar, os SREs são engenheiros. Aplicamos os princípios de ciência da computação e engenharia ao projeto e desenvolvimento de sistemas de computação: geralmente, grandes sistemas distribuídos. Às vezes, nossa tarefa é escrever o software para esses sistemas junto com nossas contrapartes de desenvolvimento de produtos; às vezes, nossa tarefa é construir todas as peças adicionais que esses sistemas precisam, como backups ou balanceamento de carga, idealmente para que possam ser reutilizados nos sistemas; e, às vezes, nossa tarefa é descobrir como aplicar as soluções existentes a novos problemas.

¹ O próprio fato de haver uma variação tão grande nessas estimativas diz algo sobre engenharia de software. como uma disciplina, mas veja, por exemplo, [Gla02] para mais detalhes.

Em seguida, focamos na confiabilidade do sistema. Ben Treynor Sloss, VP de Operações 24/7 do Google, criador do termo SRE, afirma que a confiabilidade é a característica mais fundamental de qualquer produto: um sistema não é muito útil se ninguém pode usá-lo! Como a confiabilidade² é tão crítica, os SREs estão focados em encontrar maneiras de melhorar o projeto e a operação dos sistemas para torná-los mais escaláveis, confiáveis e eficientes. No entanto, despendemos esforços nessa direção apenas até certo ponto: quando os sistemas são “confiáveis o suficiente”, investimos nossos esforços na adição de recursos ou na criação de novos produtos.³ Finalmente, os SREs estão focados em serviços operacionais construídos sobre nossos sistemas de computação distribuídos, sejam esses serviços armazenamento em escala planetária, e-mail para centenas de milhões de usuários ou, onde o Google começou, pesquisa na web. O “site” em nosso nome originalmente se referia ao papel da SRE em manter o site google.com funcionando, embora agora executemos muitos outros serviços, muitos dos quais não são sites - de infraestrutura interna, como Bigtable, a produtos para desenvolvedores como o Google Cloud Platform.

Embora tenhamos representado o SRE como uma disciplina ampla, não é surpresa que ele tenha surgido no mundo veloz dos serviços da Web, e talvez em sua origem deva algo às peculiaridades de nossa infraestrutura. Também não é surpresa que de todas as características pós-implantação de software às quais poderíamos escolher dedicar atenção especial, a confiabilidade é a que consideramos primária. O software do lado do servidor é relativamente contido e, como o gerenciamento de mudanças em si está tão fortemente associado a falhas de todos os tipos, é uma plataforma natural da qual nossa abordagem pode emergir.

Apesar de ter surgido no Google e na comunidade da web em geral, achamos que essa disciplina tem lições aplicáveis a outras comunidades e outras organizações.

Este livro é uma tentativa de explicar como fazemos as coisas: tanto para que outras organizações possam fazer uso do que aprendemos, quanto para que possamos definir melhor o papel e o que o termo significa. Para isso, organizamos o livro de modo que os princípios gerais e as práticas mais específicas sejam separados sempre que possível, e onde for apropriado discutir um tópico específico com informações específicas do Google, confiamos que o leitor nos agradará isso e não terá medo de tirar conclusões úteis sobre seu próprio ambiente.

² Para nossos propósitos, confiabilidade é “A probabilidade de que [um sistema] desempenhe uma função requerida sem falhas. sob condições estabelecidas por um período de tempo determinado”, seguindo a definição em [Oco12].

³ Os sistemas de software com os quais estamos preocupados são principalmente sites e serviços semelhantes; não discutimos preocupações de confiabilidade que enfrentam software destinado a usinas nucleares, aeronaves, equipamentos médicos ou outros sistemas críticos de segurança. No entanto, comparamos nossas abordagens com aquelas usadas em outras indústrias no [Capítulo ter 33](#).

⁴ Nisso, somos distintos do termo industrial DevOps, porque embora definitivamente consideremos infraestrutura como código, temos confiabilidade como nosso foco principal. Além disso, estamos fortemente orientados a eliminar a necessidade de operações – consulte o [Capítulo 7](#) para obter mais detalhes.

Também fornecemos algum material de orientação – uma descrição do ambiente de produção do Google e um mapeamento entre alguns de nossos softwares internos e softwares disponíveis publicamente – que devem ajudar a contextualizar o que estamos dizendo e torná-lo mais diretamente utilizável.

Em última análise, é claro, software mais orientado à confiabilidade e engenharia de sistemas são inherentemente bons. No entanto, reconhecemos que as organizações menores podem estar se perguntando como podem usar melhor a experiência representada aqui: assim como a segurança, quanto mais cedo você se preocupar com a confiabilidade, melhor. Isso implica que, embora uma pequena organização tenha muitas preocupações urgentes e as escolhas de software que você faz possam diferir daquelas feitas pelo Google, ainda vale a pena implementar um suporte leve de confiabilidade desde o início, porque é menos dispendioso expandir uma estrutura mais tarde do que é introduzir um que não está presente. A Parte IV contém várias práticas recomendadas para treinamento, comunicação e reuniões que funcionam bem para nós, muitas das quais devem ser imediatamente utilizáveis por sua organização.

Mas para tamanhos entre uma startup e uma multinacional, provavelmente já existe alguém em sua organização que está fazendo um trabalho de SRE, sem necessariamente ser chamado por esse nome, ou reconhecido como tal. Outra maneira de iniciar o caminho para melhorar a confiabilidade de sua organização é reconhecer formalmente esse trabalho ou encontrar essas pessoas e promover o que elas fazem – recompensá-lo. São pessoas que estão no limite entre uma maneira de ver o mundo e outra: como Newton, que às vezes é chamado não de primeiro físico do mundo, mas de último alquimista do mundo.

E do ponto de vista histórico, quem, então, olhando para trás, pode ser o primeiro SRE?

Gostamos de pensar que Margaret Hamilton, trabalhando no programa Apollo emprestado pelo MIT, tinha todos os traços significativos do primeiro SRE.⁵ Em suas próprias palavras, “parte da cultura era aprender com todos e com tudo, inclusive com aquilo que menos se espera.”

Um caso em questão foi quando sua filha Lauren veio trabalhar com ela um dia, enquanto alguns membros da equipe estavam executando cenários de missão no computador de simulação híbrido. Como as crianças pequenas, Lauren foi explorar, e ela fez uma “missão” falhar selecionando as teclas DSKY de uma maneira inesperada, alertando a equipe sobre o que aconteceria se o programa de pré-lançamento, P01, fosse inadvertidamente selecionado por um astronauta real, durante uma missão real, durante o meio do curso real. (Lançar o P01 inadvertidamente em uma missão real seria um grande problema, porque apaga os dados de navegação e o computador não estava equipado para pilotar a nave sem dados de navegação.)

⁵ Além dessa grande história, ela também tem uma reivindicação substancial de popularizar o termo “engenharia de software”.

Com os instintos de um SRE, Margaret enviou uma solicitação de mudança de programa para adicionar um código especial de verificação de erros no software de voo a bordo, caso um astronauta, por acidente, selecione P01 durante o voo. Mas esse movimento foi considerado desnecessário pelos “superiores” da NASA: claro, isso nunca poderia acontecer! Então, em vez de adicionar o código de verificação de erros, Margaret atualizou a documentação de especificações da missão para dizer o equivalente a “Não selecione P01 durante o voo”. (Aparentemente, a atualização foi divertida para muitos no projeto, que foram informados muitas vezes que os astronautas não cometiam erros - afinal, eles foram treinados para serem perfeitos.)

Bem, a salvaguarda sugerida por Margaret só foi considerada desnecessária até a próxima missão, na Apollo 8, apenas alguns dias após a atualização das especificações. No meio do curso do quarto dia de voo com os astronautas Jim Lovell, William Anders e Frank Borman a bordo, Jim Lovell selecionou o P01 por engano – por acaso, no dia de Natal – criando muito caos para todos os envolvidos. Este foi um problema crítico, porque na ausência de uma solução alternativa, nenhum dado de navegação significava que os astronautas nunca voltariam para casa. Felizmente, a atualização da documentação expôs explicitamente essa possibilidade e foi inestimável para descobrir como fazer upload de dados utilizáveis e recuperar a missão, sem muito tempo de sobra.

Como Margaret diz, “uma compreensão completa de como operar os sistemas não foi suficiente para evitar erros humanos”, e a solicitação de mudança para adicionar software de detecção e recuperação de erros ao programa de pré-lançamento P01 foi aprovada pouco depois.

Embora o incidente da Apollo 8 tenha ocorrido décadas atrás, há muito nos parágrafos anteriores diretamente relevantes para a vida dos engenheiros hoje e muito que continuará a ser diretamente relevante no futuro. Assim, para os sistemas que você cuida, para os grupos em que trabalha ou para as organizações que está construindo, tenha em mente o SRE Way: rigor e dedicação, crença no valor da preparação e documentação e um consciênciade do que poderia dar errado, juntamente com um forte desejo de evitá-lo. Bem-vindo à nossa profissão emergente!

Como ler este livro

Este livro é uma série de ensaios escritos por membros e ex-alunos da organização Site Reliability Engineering do Google. É muito mais como anais de conferências do que como um livro padrão de um autor ou de um pequeno número de autores. Cada capítulo deve ser lido como parte de um todo coerente, mas muito pode ser obtido lendo sobre qualquer assunto que lhe interesse particularmente. (Se houver outros artigos que apoiem ou informem o texto, nós os referenciamos para que você possa acompanhar de acordo.)

Você não precisa ler em nenhuma ordem específica, embora sugerimos pelo menos começar com os Capítulos 2 e 3, que descrevem o ambiente de produção do Google e descrevem como o SRE aborda o risco, respectivamente. (O risco é, em muitos aspectos, a qualidade-chave de nossa profissão.) Ler de ponta a ponta é, claro, também útil e possível; nossos capítulos estão agrupados tematicamente, em Princípios ([Parte II](#)), Práticas ([Parte III](#)) e Gestão ([Parte IV](#)). Cada um tem uma pequena introdução que destaca sobre o que são as peças individuais e faz referência a outros artigos publicados pelos SREs do Google, abordando tópicos específicos com mais detalhes. Além disso, o site complementar deste livro, <https://g.co/SREBook>, tem vários recursos úteis.

Esperamos que isso seja pelo menos tão útil e interessante para você quanto juntá-lo foi para nós.

— Os Editores

Convenções utilizadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

Indica novos termos, URLs, endereços de e-mail, nomes de arquivo e extensões de arquivo.

Largura constante

Usado para listas de programas, bem como dentro de parágrafos para se referir a elementos de programa, como nomes de variáveis ou funções, bancos de dados, tipos de dados, variáveis de ambiente, instruções e palavras-chave.

Largura constante em negrito

Mostra comandos ou outro texto que deve ser digitado literalmente pelo usuário.

Largura constante itálico

Mostra o texto que deve ser substituído por valores fornecidos pelo usuário ou por valores determinados pelo contexto.



Este elemento significa uma dica ou sugestão.



Este elemento significa uma nota geral.



Este elemento indica um aviso ou cuidado.

Usando exemplos de código

O material suplementar está disponível em <https://g.co/SREBook>.

Este livro está aqui para ajudá-lo a fazer o seu trabalho. Em geral, se um código de exemplo for oferecido com este livro, você poderá usá-lo em seus programas e documentação. Você não precisa entrar em contato conosco para obter permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que usa vários pedaços de código deste livro não requer permissão. Vender ou distribuir um CD-ROM de exemplos dos livros da O'Reilly requer permissão. Responder a uma pergunta citando este livro e citando um código de exemplo não requer permissão. Incorporar uma quantidade significativa de código de exemplo deste livro na documentação do seu produto requer permissão.

Agradecemos, mas não exigimos, atribuição. Uma atribuição geralmente inclui o título, autor, editora e ISBN. Por exemplo: "Site Reliability Engineering, editado por Betsy Beyer, Chris Jones, Jennifer Petoff e Niall Richard Murphy (O'Reilly). Direitos autorais 2016 Google, Inc., 978-1-491-92912-4."

Se você achar que o uso de exemplos de código está fora do uso justo ou da permissão dada acima, sinta-se à vontade para nos contatar em permissions@oreilly.com.

Livros on-line do Safari®



Livros on-line do Safari é uma biblioteca digital sob demanda que oferece conteúdo especializado em formato de livro e vídeo dos principais autores do mundo em tecnologia e negócios.

Profissionais de tecnologia, desenvolvedores de software, web designers e profissionais de negócios e criativos usam o Safari Books Online como seu principal recurso para pesquisa, solução de problemas, aprendizado e treinamento de certificação.

O Safari Books Online oferece uma variedade de **planos e preços** para **empresa, governo, Educação, e indivíduos.**

Os membros têm acesso a milhares de livros, vídeos de treinamento e manuscritos de pré-publicação em um banco de dados totalmente pesquisável de editores como O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology e centenas **mais**. Para obter mais informações sobre o Safari Books Online, visite-nos **online**.

Como entrar em contato conosco

Envie comentários e perguntas sobre este livro à editora:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472 800-998-9938
(nos Estados Unidos ou Canadá) 707-829-0515
(internacional ou local) 707-829-0104 (fax)

Temos uma página web para este livro, onde listamos erratas, exemplos e qualquer informação adicional. Você pode acessar esta página em <http://bit.ly/site-reliability-engineering>.

Para comentar ou fazer perguntas técnicas sobre este livro, envie um e-mail para bookquestions@oreilly.com.

Para obter mais informações sobre nossos livros, cursos, conferências e notícias, consulte nosso site em <http://www.oreilly.com>.

Encontre-nos no Facebook: <http://facebook.com/>

oreilly Siga-nos no Twitter: <http://twitter.com/oreillymedia>

Assista-nos no YouTube: <http://www.youtube.com/oreillymedia>

Agradecimentos

Este livro não teria sido possível sem os esforços incansáveis de nossos autores e escritores técnicos. Também gostaríamos de agradecer aos seguintes revisores internos por fornecerem feedback especialmente valioso: Alex Matey, Dermot Duffy, JC van Winkel, John T.

Reese, Michael O'Reilly, Steve Carstensen e Todd Underwood. Ben Lutch e Ben Treynor Sloss foram os patrocinadores deste livro no Google; sua crença neste projeto e compartilhar o que aprendemos sobre a execução de serviços em larga escala foi essencial para fazer este livro acontecer.

Gostaríamos de enviar agradecimentos especiais a Rik Farrow, o editor de ;login:, pela parceria conosco em uma série de contribuições para pré-publicação via USENIX.

Embora os autores sejam especificamente reconhecidos em cada capítulo, gostaríamos de dedicar um tempo para reconhecer aqueles que contribuíram para cada capítulo, fornecendo contribuições, discussões e revisões ponderadas.

Capítulo 3: Abe Rahey, Ben Treynor Sloss, Brian Stoler, Dave O'Connor, David Besbris, Jill Alvidrez, Mike Curtis, Nancy Chang, Tammy Capistrant, Tom Limoncelli

Capítulo 5: Cody Smith, George Sadlier, Laurence Berland, Marc Alvidrez, Patrick Stahlberg, Peter Duff, Pim van Pelt, Ryan Anderson, Sabrina Farmer, Seth Hettich

Capítulo 6: Mike Curtis, Jamie Wilkinson, Seth Hettich

Capítulo 8: David Schnur, JT Goldstone, Marc Alvidrez, Marcus Lara-Reinhold, Noah Maxwell, Peter Dinges, Sumitran Raghunathan, Yutong Cho

Capítulo 9: Ryan Anderson

Capítulo 10: Jules Anderson, Max Luebbe, Mikel McDaniel, Raul Vera, Seth Hettich

Capítulo 11: Andrew Stribblehill, Richard Woodbury

Capítulo 12: Charles Stephen Gunn, John Hedditch, Peter Nuttall, Rob Ewaschuk, Sam Greenfield

Capítulo 13: Jelena Oertel, Kripa Krishnan, Sergio Salvi, Tim Craig

Capítulo 14: Amy Zhou, Carla Geisser, Grainne Sheerin, Hildo Biersma, Jelena Oertel, Perry Lorier, Rune Kristian Viken

Capítulo 15: Dan Wu, Heather Sherman, Jared Brick, Mike Louer, Štýpán Davidovič, Tim Craig

Capítulo 16: Andrew Stribblehill, Richard Woodbury

Capítulo 17: Isaac Clerencia, Marc Alvidrez

Capítulo 18: Ulric Longyear

Capítulo 19: Debashish Chatterjee, Perry Lorier

Capítulos 20 e 21: Adam Fletcher, Christoph Pfisterer, Lukáš Ježek, Manjot Pahwa, Micha Riser, Noah Fiedel, Pavel Herrmann, Paweł Zuzelski, Perry Lorier, Ralf Wildenhues, Tudor-Ioan Salomie, Witold Baryluk

Capítulo 22: Mike Curtis, Ryan Anderson

Capítulo 23: Ananth Shrinivas, Mike Burrows

Capítulo 24: Ben Fried, Derek Jackson, Gabe Krabbe, Laura Nolan, Seth Hettich

Capítulo 25: Abdulrahman Salem, Alex Perry, Arnar Mar Hrafnkelsson, Dieter Pearcey, Dylan Curley, Eivind Eklund, Eric Veach, Graham Poulter, Ingvar Mattsson, John Looney, Ken Grant, Michelle Duffy, Mike Hochberg, Will Robinson

Capítulo 26: Corey Vickrey, Dan Ardelean, Disney Luangsisongkham, Gordon Prioreschi, Kristina Bennett, Liang Lin, Michael Kelly, Sergey Ivanyuk

Capítulo 27: Vivek Rau

Capítulo 28: Melissa Binde, Perry Lorier, Preston Yoshioka

Capítulo 29: Ben Lutch, Carla Geisser, Dzevad Trumic, John Turek, Matt Brown

Capítulo 30: Charles Stephen Gunn, Chris Heiser, Max Luebbe, Sam Greenfield

Capítulo 31: Alex Kehlenbeck, Jeromy Carriere, Joel Becker, Sowmya Vijayaraghavan, Trevor Mattson-Hamilton

Capítulo 32: Seth Hettich

Capítulo 33: Adrian Hilton, Brad Kratochvil, Charles Ballowe, Dan Sheridan, Eddie Kennedy, Erik Gross, Gus Hartmann, Jackson Stone, Jeff Stevenson, John Li, Kevin Greer, Matt Toia, Michael Haynie, Mike Doherty, Peter Dahl, Ron Heiby

Também somos gratos aos seguintes colaboradores, que forneceram material significativo, fizeram um excelente trabalho de revisão, concordaram em ser entrevistados, forneceram conhecimentos ou recursos significativos ou tiveram algum efeito excelente neste trabalho:

Abe Hassan, Adam Rogoyski, Alex Hidalgo, Amaya Booker, Andrew Fikes, Andrew Hurst, Ariel Goh, Ashleigh Rentz, Ayman Hourieh, Barclay Osborn, Ben Appleton, Ben Love, Ben Winslow, Bernhard Beck, Bill Duane, Bill Patry, Blair Zajac, Bob Gruber, Brian Gustafson, Bruce Murphy, Buck Clay, Cedric Cellier, Chiho Saito, Chris Carlon, Christopher Hahn, Chris Kennelly, Chris Taylor, Ciara Kamahale-Sanfratello, Colin Phipps, Colm Buckley, Craig Paterson, Daniel Eisenbud, Daniel V Klein, Daniel Spoonhower, Dan Watson, Dave Phillips, David Hixson, Dina Betser, Doron Meyer, Dmitry Fedoruk, Eric Grosse, Eric Schrock, Filip Zyzniewski, Francis Tang, Gary Arneson, Georgina Wilcox, Gretta Bartels, Gustavo Franco, Harald Wagener, Healfdene Goguen, Hugo Santos, Hyrum Wright, Ian Gulliver, Jakub Turski, James Chivers, James O'Kane, James Youngman, Jan Monsch, Jason Parker-Burlingham, Jason Petsod, Jeffry McNeil, Jeff Dean, Jeff Peck, Jennifer Mace, Jerry Cen, Jess Frame, John Brady, John Gunderman, John Kochmar, John Tobin, Jordyn Buchanan, Joseph Bironas, Julio Merino, Julius Plenz, Kate Ward, Kathy Polizzi, Katrina Sostek, Kenn Hamm, Kirk Russell, Kripa Krishnan, Larry Greenfield, Lea Oliveira, Luca Cittadini,

Lucas Pereira, Magnus Ringman, Mahesh Palekar, Marco Paganini, Mario Bonilla, Mathew Mills, Mathew Monroe, Matt D. Brown, Matt Proud, Max Saltonstall, Michal Jaszczyk, Mihai Bivol, Misha Brukman, Olivier Oansaldi, Patrick Bernier, Pierre Palay estanho, Rob Shanley, Robert van Gent, Rory Ward, Rui Zhang-Shen, Salim Virji, Sanjay Ghemawat, Sarah Coty, Sean Dorward, Sean Quinlan, Sean Sechrest, Shari Trumbo McHenry, Shawn Morrissey, Shun-Tak Leung, Stan Jedrus, Stefano Lattarini, Steven Schirripa, Tanya Reilly, Terry Bolt, Tim Chaplin, Toby Weingartner, Tom Black, Udi Meiri, Victor Terron, Vlad Grama, Wes Hertlein e Zoltan Egyed.

Agradecemos muito o feedback atencioso e profundo que recebemos de revisores externos: Andrew Fong, Björn Rabenstein, Charles Border, David Blank Edelman, Frossie Economou, James Meickle, Josh Ryder, Mark Burgess e Russ Allbery.

Gostaríamos de estender nossos agradecimentos especiais a Cian Synnott, membro da equipe do livro original e co-conspirador, que deixou o Google antes da conclusão deste projeto, mas foi profundamente influente para ele, e Margaret Hamilton, que tão gentilmente nos permitiu fazer referência à sua história em nosso prefácio. Além disso, gostaríamos de agradecer especialmente a Shylaja Nukala, que generosamente doou o tempo de seus redatores técnicos e apoiou seus esforços necessários e valiosos de todo o coração.

Os editores também gostariam de agradecer pessoalmente às seguintes pessoas:

Betsy Beyer: À vovó (meu herói pessoal), por fornecer quantidades infinitas de conversas ao telefone e pipoca, e a Riba, por me fornecer as calças de moletom necessárias para abastecer várias noites. Estes, claro, além do elenco da SRE todas as estrelas que foram de fato colaboradores encantadores.

Chris Jones: Para Michelle, por me salvar de uma vida de crimes em alto mar e por sua incrível habilidade de encontrar manzanas em lugares inesperados, e para aqueles que me ensinaram sobre engenharia ao longo dos anos.

Jennifer Petoff: Ao meu marido Scott por ser incrivelmente solidário durante o processo de dois anos de escrita deste livro e por manter os editores abastecidos com bastante açúcar em nossa “Ilha da Sobremesa”.

Niall Murphy: Para Léan, Oisín e Fiachra, que foram consideravelmente mais pacientes do que eu tinha o direito de esperar com um pai e marido substancialmente mais raivoso do que o habitual, por anos. Para Dermot, pela oferta de transferência.

PARTE I

Introdução

Esta seção fornece algumas orientações de alto nível sobre o que é SRE e por que ele é diferente das práticas mais convencionais do setor de TI.

Ben Treynor Sloss, vice-presidente sênior que supervisiona as operações técnicas do Google e o criador do termo "Engenharia de confiabilidade do site" - fornece sua visão sobre o que SRE significa, como funciona e como se compara a outras maneiras de fazer as coisas na indústria - tente, no [Capítulo 1](#).

Fornecemos um guia para o ambiente de produção no Google no [Capítulo 2](#) como uma forma de ajudá-lo a se familiarizar com a riqueza de novos termos e sistemas que você conhecerá no restante do livro.

CAPÍTULO 1

Introdução

Escrito por Benjamin Treynor Sloss¹
Editado por Betsy Beyer

Esperança não é uma estratégia.

— Ditado tradicional do SRE

É uma verdade universalmente reconhecida que os sistemas não funcionam sozinhos. Como, então, um sistema – particularmente um sistema de computação complexo que opera em grande escala – deve ser executado?

A Abordagem Sysadmin para Gerenciamento de Serviços

Historicamente, as empresas empregavam administradores de sistemas para executar sistemas computacionais complexos.

Essa abordagem de administrador de sistemas, ou sysadmin, envolve montar componentes de software existentes e implantá-los para trabalharem juntos para produzir um serviço.

Os administradores de sistema são então encarregados de executar o serviço e responder a eventos e atualizações à medida que ocorrem. À medida que o sistema cresce em complexidade e volume de tráfego, gerando um aumento correspondente em eventos e atualizações, a equipe sysadmin cresce para absorver o trabalho adicional. Como a função de administrador de sistema requer um conjunto de habilidades marcadamente diferente daquele exigido dos desenvolvedores de um produto, desenvolvedores e administradores de sistema são divididos em equipes distintas: “desenvolvimento” e “operações” ou “operações”.

O modelo sysadmin de gerenciamento de serviços tem várias vantagens. Para as empresas que decidem como executar e contratar um serviço, essa abordagem é relativamente fácil de implementar: como um paradigma familiar da indústria, há muitos exemplos para aprender e

¹ Vice-presidente, Engenharia do Google, fundador do Google SRE

emular, imitar. Um pool de talentos relevante já está amplamente disponível. Uma variedade de ferramentas existentes, componentes de software (de prateleira ou não) e empresas de integração estão disponíveis para ajudar a executar esses sistemas montados, para que uma equipe de administrador de sistema novato não precise reinventar a roda e projetar um sistema do zero.

A abordagem sysadmin e a divisão de desenvolvimento/operações que a acompanha têm várias desvantagens e armadilhas. Estes caem amplamente em duas categorias: custos diretos e custos indiretos.

Os custos diretos não são sutis nem ambíguos. A execução de um serviço com uma equipe que depende de intervenção manual para gerenciamento de mudanças e tratamento de eventos torna-se cara à medida que o serviço e/ou o tráfego para o serviço cresce, porque o tamanho da equipe necessariamente escala com a carga gerada pelo sistema.

Os custos indiretos da divisão de desenvolvimento/operações podem ser sutis, mas geralmente são mais caros para a organização do que os custos diretos. Esses custos surgem do fato de que as duas equipes são bastante diferentes em termos de experiência, conjunto de habilidades e incentivos. Eles usam vocabulário diferente para descrever situações; carregam suposições diferentes sobre risco e possibilidades de soluções técnicas; eles têm suposições diferentes sobre o nível alvo de estabilidade do produto. A divisão entre os grupos pode facilmente se tornar não apenas um incentivo, mas também comunicação, objetivos e, eventualmente, confiança e respeito. Este resultado é uma patologia.

As equipes de operações tradicionais e suas contrapartes no desenvolvimento de produtos muitas vezes acabam em conflito, mais visivelmente sobre a rapidez com que o software pode ser liberado para produção. Em sua essência, as equipes de desenvolvimento querem lançar novos recursos evê-los adotados pelos usuários. Em sua essência, as equipes de operações querem garantir que o serviço não seja interrompido enquanto estão segurando o pager. Como a maioria das interrupções é causada por algum tipo de mudança – uma nova configuração, um novo lançamento de recurso ou um novo tipo de tráfego de usuários – os objetivos das duas equipes estão fundamentalmente em tensão.

Ambos os grupos entendem que é inaceitável expor seus interesses nos termos mais ríspidos possíveis (“Queremos lançar qualquer coisa, a qualquer hora, sem impedimentos” versus “Não vamos querer mudar nada no sistema uma vez que funcione”). E como seu vocabulário e suposições de risco diferem, ambos os grupos geralmente recorrem a uma forma familiar de guerra de trincheiras para promover seus interesses. A equipe de operações tenta proteger o sistema em execução contra o risco de mudança introduzindo portões de lançamento e mudança. Por exemplo, as revisões de lançamento podem conter uma verificação explícita de todos os problemas que já causaram uma interrupção no passado – que pode ser uma lista arbitrariamente longa, com nem todos os elementos fornecendo o mesmo valor. A equipe de desenvolvimento aprende rapidamente como responder.

Eles têm menos “lançamentos” e mais “trocas de bandeira”, “atualizações incrementais” ou “escolhas de cereja”. Eles adotam táticas como fragmentar o produto para que menos recursos estejam sujeitos à revisão de lançamento.

Abordagem do Google para gerenciamento de serviços: Engenharia de confiabilidade do site

O conflito não é uma parte inevitável da oferta de um serviço de software. O Google optou por executar nossos sistemas com uma abordagem diferente: nossas equipes de engenharia de confiabilidade do site se concentram na contratação de engenheiros de software para executar nossos produtos e criar sistemas para realizar o trabalho que, de outra forma, seria realizado, geralmente manualmente, por administradores de sistema.

O que exatamente é a Engenharia de Confiabilidade do Site, como foi definida no Google?

Minha explicação é simples: SRE é o que acontece quando você pede a um engenheiro de software para projetar uma equipe de operações. Quando entrei no Google em 2003 e recebi a tarefa de dirigir uma "Equipe de Produção" de sete engenheiros, minha vida inteira até aquele momento tinha sido engenharia de software. Então eu projetei e gerenciei o grupo do jeito que eu gostaria que funcionasse se eu mesmo trabalhasse como SRE. Desde então, esse grupo amadureceu para se tornar a atual equipe de SRE do Google, que permanece fiel às suas origens, conforme imaginado por um engenheiro de software ao longo da vida.

Um elemento básico da abordagem do Google para gerenciamento de serviços é a composição de cada equipe de SRE. Como um todo, o SRE pode ser dividido em duas categorias principais.

50 a 60% são engenheiros de software do Google ou, mais precisamente, pessoas que foram contratadas por meio do procedimento padrão para engenheiros de software do Google. Os outros 40 a 50% são candidatos que estavam muito próximos das qualificações de Engenharia de Software do Google (ou seja, 85 a 99% do conjunto de habilidades necessário) e que, além disso, possuíam um conjunto de habilidades técnicas úteis para o SRE, mas raras para a maioria dos engenheiros de software. De longe, a experiência interna do sistema UNIX e a experiência em rede (camada 1 a camada 3) são os dois tipos mais comuns de habilidades técnicas alternativas que buscamos.

Comum a todos os SREs é a crença e aptidão para desenvolver sistemas de software para resolver problemas complexos. No SRE, acompanhamos de perto o progresso da carreira de ambos os grupos e, até o momento, não encontramos nenhuma diferença prática no desempenho entre os engenheiros das duas faixas. Na verdade, a formação um tanto diversa da equipe SRE frequentemente resulta em sistemas inteligentes e de alta qualidade que são claramente o produto da síntese de vários conjuntos de habilidades.

O resultado de nossa abordagem de contratação para SRE é que acabamos com uma equipe de pessoas que (a) rapidamente se cansam de realizar tarefas manualmente e (b) têm o conjunto de habilidades necessário para escrever software para substituir suas tarefas manuais anteriores. Funcionar, mesmo quando a solução é complicada. Os SREs também acabam compartilhando a formação acadêmica e intelectual com o resto da organização de desenvolvimento. Portanto, o SRE está fundamentalmente fazendo um trabalho que historicamente tem sido feito por uma equipe de operações, mas usando engenheiros com experiência em software e apostando no fato de que esses engenheiros são

inerentemente ambos predispostos e têm a capacidade de projetar e implementar a automação com software para substituir o trabalho humano.

Por design, é crucial que as equipes de SRE estejam focadas na engenharia. Sem engenharia constante, a carga de operações aumenta e as equipes precisarão de mais pessoas apenas para acompanhar a carga de trabalho. Eventualmente, um grupo tradicional focado em operações é dimensionado linearmente com o tamanho do serviço: se os produtos suportados pelo serviço forem bem-sucedidos, a carga operacional aumentará com o tráfego. Isso significa contratar mais pessoas para fazer as mesmas tarefas repetidamente.

Para evitar esse destino, a equipe encarregada de gerenciar um serviço precisa codificar ou ele se afogará. Portanto, o Google impõe um limite de 50% no trabalho agregado de "operações" para todos os SREs: tíquetes, plantão, tarefas manuais etc. Esse limite garante que a equipe do SRE tenha tempo suficiente em sua programação para tornar o serviço estável e operável. Este limite é um limite superior; com o tempo, deixada por conta própria, a equipe do SRE deve acabar com muito pouca carga operacional e quase inteiramente engajada em tarefas de desenvolvimento, porque o serviço basicamente roda e se conserta: queremos sistemas que sejam automáticos, não apenas automatizados. Na prática, a escala e os novos recursos mantêm os SREs em alerta.

A regra geral do Google é que uma equipe de SRE deve gastar os 50% restantes de seu tempo realmente desenvolvendo. Então, como podemos impor esse limite? Em primeiro lugar, temos que medir como o tempo SRE é gasto. Com essa medição em mãos, garantimos que as equipes que gastam consistentemente menos de 50% de seu tempo no trabalho de desenvolvimento mudem suas práticas. Muitas vezes, isso significa transferir parte da carga de operações de volta para a equipe de desenvolvimento ou adicionar funcionários à equipe sem atribuir responsabilidades operacionais adicionais a essa equipe. Manter conscientemente esse equilíbrio entre operações e trabalho de desenvolvimento nos permite garantir que os SREs tenham largura de banda para se envolver em engenharia criativa e autônoma, mantendo a sabedoria adquirida do lado operacional da execução de um serviço.

Descobrimos que a abordagem do Google SRE para executar sistemas em grande escala tem muitas vantagens. Como os SREs estão modificando diretamente o código em sua busca de fazer com que os sistemas do Google funcionem sozinhos, as equipes de SRE são caracterizadas tanto pela inovação rápida quanto pela grande aceitação da mudança. Essas equipes são relativamente baratas – dar suporte ao mesmo serviço com uma equipe orientada a operações exigiria um número significativamente maior de pessoas. Em vez disso, o número de SREs necessários para executar, manter e melhorar um sistema é dimensionado de forma sublinear com o tamanho do sistema. Por fim, o SRE não apenas contorna a disfuncionalidade da divisão dev/ops, mas essa estrutura também melhora nossas equipes de desenvolvimento de produto: transferências fáceis entre equipes de desenvolvimento de produto e SRE treinam todo o grupo e melhoram as habilidades dos desenvolvedores que, de outra forma, pode ter dificuldade em aprender como construir um sistema distribuído de um milhão de núcleos.

Apesar desses ganhos líquidos, o modelo SRE é caracterizado por seu próprio conjunto distinto de desafios. Um desafio contínuo que o Google enfrenta é contratar SREs: não apenas o SRE

competir pelos mesmos candidatos que o pipeline de contratação de desenvolvimento de produtos, mas o fato de definirmos o padrão de contratação tão alto em termos de habilidades de codificação e engenharia de sistema significa que nosso pool de contratação é necessariamente pequeno. Como nossa disciplina é relativamente nova e única, não existem muitas informações do setor sobre como construir e gerenciar uma equipe de SRE (embora esperemos que este livro avance nessa direção!). E, uma vez que uma equipe de SRE esteja pronta, suas abordagens potencialmente não ortodoxas para o gerenciamento de serviços exigem um forte suporte de gerenciamento. Por exemplo, a decisão de interromper os lançamentos pelo restante do trimestre depois que um orçamento de erro se esgotar pode não ser adotada por uma equipe de desenvolvimento de produto, a menos que seja exigida por seu gerenciamento.

DevOps ou SRE?

O termo “DevOps” surgiu na indústria no final de 2008 e até o momento da redação deste artigo (início de 2016) ainda está em um estado de fluxo. Seus princípios básicos – envolvimento da função de TI em cada fase do projeto e desenvolvimento de um sistema, forte dependência de automação versus esforço humano, aplicação de práticas e ferramentas de engenharia para tarefas operacionais – são consistentes com muitos dos princípios e práticas do SRE. Pode-se ver o DevOps como uma generalização de vários princípios básicos de SRE para uma gama mais ampla de organizações, estruturas de gerenciamento e pessoal. Pode-se ver equivalentemente o SRE como uma implementação específica do DevOps com algumas extensões idiossincráticas.

Princípios do SRE

Embora as nuances dos fluxos de trabalho, prioridades e operações do dia a dia variem de equipe de SRE para equipe de SRE, todas compartilham um conjunto de responsabilidades básicas para o(s) serviço(s) que suportam e aderem aos mesmos princípios básicos. Em geral, uma equipe de SRE é responsável pela disponibilidade, latência, desempenho, eficiência, gerenciamento de mudanças, monitoramento, resposta a emergências e planejamento de capacidade de seu(s) serviço(s). Codificamos regras de engajamento e princípios de como as equipes de SRE interagem com seu ambiente — não apenas o ambiente de produção, mas também as equipes de desenvolvimento de produtos, as equipes de teste, os usuários e assim por diante. Essas regras e práticas de trabalho nos ajudam a manter nosso foco no trabalho de engenharia, em oposição ao trabalho de operações.

A seção a seguir discute cada um dos princípios básicos do Google SRE.

Garantindo um foco duradouro em engenharia Conforme

já discutido, o Google limita o trabalho operacional para SREs em 50% do tempo. Na prática, isso é feito monitorando a quantidade de trabalho operacional feito pelos SREs e redirecionando o excesso de trabalho operacional para as equipes de desenvolvimento de produtos: reatribuindo bugs e tickets para gerentes de desenvolvimento, [re]integrando desenvolvedores em rotações de pagers de plantão , e assim por diante. O redirecionamento termina quando a operação

carga adicional cai para 50% ou menos. Isso também fornece um mecanismo de feedback eficaz, orientando os desenvolvedores a criar sistemas que não precisam de intervenção manual.

Essa abordagem funciona bem quando toda a organização – SRE e desenvolvimento – entende por que o mecanismo de válvula de segurança existe e suporta a meta de não haver eventos de estouro porque o produto não gera carga operacional suficiente para exigir-lo.

Quando estão focados no trabalho operacional, em média, os SREs devem receber no máximo dois eventos por turno de plantão de 8 a 12 horas. Esse volume de destino dá ao engenheiro de plantão tempo suficiente para lidar com o evento com precisão e rapidez, limpar e restaurar o serviço normal e, em seguida, realizar uma autópsia. Se mais de dois eventos ocorrem regularmente por turno de plantão, os problemas não podem ser investigados minuciosamente e os engenheiros ficam suficientemente sobrecarregados para evitar que aprendam com esses eventos. Um cenário de fadiga do pager também não melhorará com a escala. Por outro lado, se os SREs de plantão recebem consistentemente menos de um evento por turno, mantê-los no ponto é uma perda de tempo.

Postmortems devem ser escritos para todos os incidentes significativos, independentemente de serem ou não paginados; postmortems que não acionaram uma página são ainda mais valiosos, pois provavelmente apontam para limpar lacunas de monitoramento. Essa investigação deve estabelecer o que aconteceu em detalhes, encontrar todas as causas-raiz do evento e atribuir ações para corrigir o problema ou melhorar a forma como ele será abordado na próxima vez. O Google opera sob uma cultura postmortem livre de culpa, com o objetivo de expor falhas e aplicar engenharia para corrigi-las, em vez de evitá-las ou minimizá-las.

Buscando a máxima velocidade de mudança sem violar o SLO de um serviço As equipes de desenvolvimento de produto e SRE podem desfrutar de uma relação de trabalho produtiva eliminando o conflito estrutural em suas respectivas metas. O conflito estrutural é entre o ritmo de inovação e a estabilidade do produto e, conforme descrito anteriormente, esse conflito muitas vezes é expresso indiretamente. No SRE, trazemos esse conflito à tona e o resolvemos com a introdução de um orçamento de erro.

O orçamento de erro decorre da observação de que 100% é a meta de confiabilidade errada para basicamente tudo (marcapassos e freios antitravamento sendo exceções notáveis). Em geral, para qualquer serviço ou sistema de software, 100% não é a meta de confiabilidade correta porque nenhum usuário pode dizer a diferença entre um sistema estar 100% disponível e 99,999% disponível. Existem muitos outros sistemas no caminho entre o usuário e o serviço (seu laptop, seu WiFi doméstico, seu ISP, a rede elétrica...) e esses sistemas coletivamente estão muito menos de 99,999% disponíveis. Assim, a diferença marginal entre 99,999% e 100% se perde no ruído de outras indisponibilidades, e o usuário não recebe nenhum benefício com o enorme esforço necessário para somar esses últimos 0,001% de disponibilidade.

Se 100% é a meta de confiabilidade errada para um sistema, qual é, então, a meta de confiabilidade correta para o sistema? Na verdade, essa não é uma questão técnica – é uma questão de produto, que deve levar em consideração as seguintes considerações:

- Com que nível de disponibilidade os usuários ficarão satisfeitos, considerando como eles usam o produto?
- Quais alternativas estão disponíveis para usuários insatisfeitos com o produto disponibilidade?
- O que acontece com o uso do produto pelos usuários em diferentes níveis de disponibilidade?

O negócio ou o produto deve estabelecer a meta de disponibilidade do sistema. Depois que essa meta é estabelecida, o erro de orçamento é um menos a meta de disponibilidade. Um serviço que está 99,99% disponível está 0,01% indisponível. Essa indisponibilidade permitida de 0,01% é o erro de orçamento do serviço. Podemos gastar o orçamento em qualquer coisa que quisermos, desde que não gastemos demais.

Então, como queremos gastar o orçamento de erro? A equipe de desenvolvimento quer lançar recursos e atrair novos usuários. Idealmente, gastaríamos todo o nosso orçamento de erros assumindo riscos com as coisas que lançamos para lançá-las rapidamente. Esta premissa básica descreve todo o modelo de orçamentos de erro. Assim que as atividades de SRE forem conceituadas nessa estrutura, liberar o orçamento de erros por meio de táticas como lançamentos em fases e experimentos de 1% pode otimizar para lançamentos mais rápidos.

O uso de um orçamento de erro resolve o conflito estrutural de incentivos entre desenvolvimento e SRE. O objetivo da SRE não é mais “zero interrupções”; em vez disso, SREs e desenvolvedores de produtos visam gastar o orçamento de erros obtendo velocidade máxima de recursos. Essa mudança faz toda a diferença. Uma interrupção não é mais uma coisa “ruim” – é uma parte esperada do processo de inovação e uma ocorrência que as equipes de desenvolvimento e SRE gerenciam em vez de temer.

Monitoramento

O monitoramento é um dos principais meios pelos quais os proprietários de serviços acompanham a integridade e a disponibilidade de um sistema. Como tal, a estratégia de monitoramento deve ser construída cuidadosamente. Uma abordagem clássica e comum de monitoramento é observar um valor ou condição específica e, em seguida, acionar um alerta de e-mail quando esse valor for excedido ou essa condição ocorrer. No entanto, esse tipo de alerta por e-mail não é uma solução eficaz: um sistema que exige que um humano leia um e-mail e decida se algum tipo de ação precisa ser tomada em resposta é fundamentalmente falho. O monitoramento nunca deve exigir que um humano interprete qualquer parte do domínio de alerta. Em vez disso, o software deve fazer a interpretação e os humanos devem ser notificados apenas quando precisam agir.

Existem três tipos de saída de monitoramento válida:

Alertas

Significa que um ser humano precisa agir imediatamente em resposta a algo que está acontecendo ou prestes a acontecer, a fim de melhorar a situação.

Ingressos

Significa que um ser humano precisa agir, mas não imediatamente. O sistema não pode lidar automaticamente com a situação, mas se um humano agir em poucos dias, nenhum dano resultará.

Registro

Ninguém precisa ver essas informações, mas elas são registradas para fins de diagnóstico ou forense. A expectativa é que ninguém leia os logs, a menos que algo mais os incite a fazê-lo.

A confiabilidade da **resposta**

de emergência é uma função do tempo médio até a falha (MTTF) e do tempo médio para reparo (MTTR) [Sch15]. A métrica mais relevante na avaliação da eficácia da resposta de emergência é a rapidez com que a equipe de resposta pode trazer o sistema de volta à saúde – ou seja, o MTTR.

Humanos adicionam latência. Mesmo que um determinado sistema sofra mais falhas reais, um sistema que possa evitar emergências que exijam intervenção humana terá maior disponibilidade do que um sistema que exija intervenção prática. Quando os humanos são necessários, descobrimos que pensar e registrar as melhores práticas antecipadamente em um “manual” produz aproximadamente uma melhoria de 3x no MTTR em comparação com a estratégia de “dar um jeito”. O engenheiro de plantão herói faz-tudo funciona, mas o engenheiro de plantão experiente armado com uma cartilha funciona muito melhor.

Embora nenhum manual, por mais abrangente que seja, seja um substituto para engenheiros inteligentes capazes de pensar em tempo real, etapas e dicas claras e completas de solução de problemas são valiosas ao responder a uma página de alto risco ou sensível ao tempo. Assim, o Google SRE conta com cartilhas de plantão, além de exercícios como a “Roda do Infortúnio”² para preparar os engenheiros para reagir a eventos de plantão.

Gerenciamento de Mudanças

O SRE descobriu que aproximadamente 70% das interrupções são devido a mudanças em um sistema ativo. As práticas recomendadas neste domínio usam a automação para realizar o seguinte:

² Consulte “Desastre Role Playing” na página 401.

- Implementação de lançamentos progressivos
 - Detecção de problemas com rapidez e precisão •
- Revertendo alterações com segurança quando surgem problemas

Esse trio de práticas minimiza efetivamente o número agregado de usuários e operações expostas a mudanças ruins. Ao remover os humanos do circuito, essas práticas evitam os problemas normais de fadiga, familiaridade/desprezo e desatenção a tarefas altamente repetitivas. Como resultado, a velocidade de liberação e a segurança aumentam.

Previsão de Demanda e Planejamento de Capacidade

A previsão de demanda e planejamento de capacidade podem ser vistos como garantia de que há capacidade e redundância suficientes para atender a demanda futura projetada com a disponibilidade necessária. Não há nada de especial nesses conceitos, exceto que um número surpreendente de serviços e equipes não torna as medidas necessárias para garantir que a capacidade necessária esteja disponível no momento necessário. O planejamento de capacidade deve levar em consideração tanto o crescimento orgânico (que decorre da adoção e uso natural de produtos pelos clientes) quanto o crescimento inorgânico (que resulta de eventos como lançamentos de recursos, campanhas de marketing ou outras mudanças direcionadas aos negócios).

Várias etapas são obrigatórias no planejamento de capacidade:

- Uma previsão de demanda orgânica precisa, que se estende além do lead time necessário para adquirir capacidade
- Uma incorporação precisa de fontes de demanda inorgânica na demanda previsão
- Testes regulares de carga do sistema para correlacionar a capacidade bruta (servidores, discos etc.) ligado) para a capacidade de serviço

Como a capacidade é fundamental para a disponibilidade, segue-se naturalmente que a equipe de SRE deve ser responsável pelo planejamento da capacidade, o que significa que ela também deve ser responsável pelo provisionamento.

Provisionamento

O provisionamento combina gerenciamento de mudanças e planejamento de capacidade. Em nossa experiência, o provisionamento deve ser realizado rapidamente e somente quando necessário, pois a capacidade é cara. Este exercício também deve ser feito corretamente ou a capacidade não funciona quando necessário. A adição de nova capacidade geralmente envolve a ativação de uma nova instância ou local, fazendo modificações significativas nos sistemas existentes (arquivos de configuração,平衡adores de carga, rede) e validando se a nova capacidade executa e fornece resultados corretos. Assim, é uma operação mais arriscada do que o deslocamento de carga, que muitas vezes é

feito várias vezes por hora, e deve ser tratado com um grau correspondente de cuidado extra.

E ciência e desempenho O uso eficiente

de recursos é importante sempre que um serviço se preocupa com dinheiro. Como o SRE, em última análise, controla o provisionamento, ele também deve estar envolvido em qualquer trabalho de utilização, pois a utilização é uma função de como um determinado serviço funciona e como ele é provisionado. Segue-se que prestar atenção à estratégia de provisionamento de um serviço e, portanto, sua utilização, fornece uma alavanca muito, muito grande nos custos totais do serviço.

O uso de recursos é uma função da demanda (carga), capacidade e eficiência do software. Os SREs preveem a demanda, provisionam capacidade e podem modificar o software. Esses três fatores são uma grande parte (embora não a totalidade) da eficiência de um serviço.

Os sistemas de software ficam mais lentos à medida que a carga é adicionada a eles. Uma desaceleração em um serviço equivale a uma perda de capacidade. Em algum momento, um sistema de lentidão deixa de servir, o que corresponde a uma lentidão infinita. SREs fornecem para atender a uma meta de capacidade em uma velocidade de resposta específica e, portanto, estão muito interessados no desempenho de um serviço. SREs e desenvolvedores de produtos irão (e devem) monitorar e modificar um serviço para melhorar seu desempenho, aumentando assim a capacidade e melhorando a eficiência.³

O fim do começo

A Engenharia de Confiabilidade do Site representa uma ruptura significativa com as práticas recomendadas existentes do setor para gerenciar serviços grandes e complicados. Motivado originalmente pela familiaridade – “como engenheiro de software, é assim que eu gostaria de investir meu tempo para realizar um conjunto de tarefas repetitivas” – tornou-se muito mais: um conjunto de princípios, um conjunto de práticas, um conjunto de incentivos e um campo de atuação dentro da disciplina maior de engenharia de software. O resto do livro explora o SRE Way em detalhes.

³ Para uma discussão mais detalhada de como essa colaboração pode funcionar na prática, consulte “[Comunicação: Reuniões de Produção](#)” na página 426.

CAPÍTULO 2

O Ambiente de Produção no Google, sob o ponto de vista de um SRE

**Escrito por JC van Winkel
Editado por Betsy Beyer**

Os datacenters do Google são muito diferentes da maioria dos datacenters convencionais e dos farms de servidores de pequena escala. Essas diferenças apresentam problemas e oportunidades extras.

Este capítulo discute os desafios e oportunidades que caracterizam os datacenters do Google e apresenta a terminologia usada em todo o livro.

Hardware

A maioria dos recursos de computação do Google está em datacenters projetados pelo Google com distribuição de energia, resfriamento, rede e hardware de computação proprietários (consulte [Bar13]).

Ao contrário dos datacenters de colocation “padrão”, o hardware de computação em um datacenter projetado pelo Google é o mesmo em todos os aspectos.¹ Para eliminar a confusão entre hardware de servidor e software de servidor, usamos a seguinte terminologia ao longo do livro:

Máquina

Uma peça de hardware (ou talvez uma VM)

Servidor

Um pedaço de software que implementa um serviço

¹ Bem, mais ou menos o mesmo. Majoritariamente. Exceto pelas coisas que são diferentes. Alguns datacenters acabam com várias gerações de hardware de computação e, às vezes, aumentamos os datacenters depois que eles são criados. Mas, na maioria das vezes, nosso hardware de datacenter é homogêneo.

As máquinas podem executar qualquer servidor, portanto, não dedicamos máquinas específicas a programas de servidor específicos. Não há uma máquina específica que execute nosso servidor de e-mail, por exemplo. Em vez disso, a alocação de recursos é tratada pelo nosso sistema operacional de cluster, Borg.

Percebemos que esse uso da palavra servidor é incomum. O uso comum da palavra confunde “binário que aceita conexão de rede” com máquina, mas diferenciar entre os dois é importante quando se fala de computação no Google. Depois que você se acostumar com o uso do servidor, fica mais claro por que faz sentido usar essa terminologia especializada, não apenas no Google, mas também no restante deste livro.

A [Figura 2-1](#) ilustra a topologia de um datacenter do Google:

- Dezenas de máquinas são colocadas em um rack.
- Os racks ficam em fila.
- Uma ou mais linhas formam um cluster.
- Normalmente, um edifício de datacenter abriga vários clusters. • Vários edifícios de datacenter localizados próximos formam um campus.

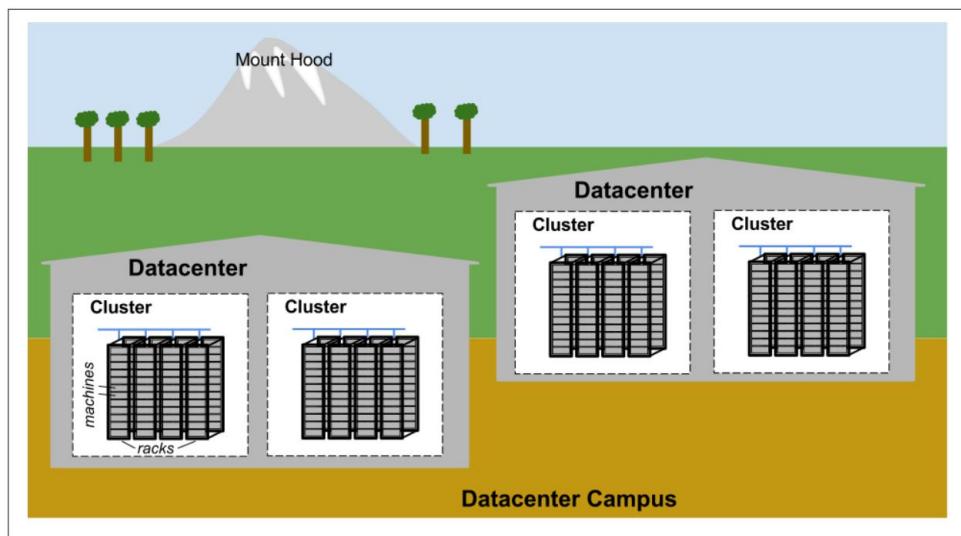


Figura 2-1. Exemplo de topologia de campus de datacenter do Google

As máquinas dentro de um determinado datacenter precisam ser capazes de se comunicar umas com as outras, por isso criamos um switch virtual muito rápido com dezenas de milhares de portas. Conseguimos isso conectando centenas de switches criados pelo Google em uma malha de rede Clos [\[Clos53\]](#) chamada Jupiter [\[Sin15\]](#). Em sua maior configuração, o Jupiter suporta largura de banda de bisseção de 1,3 Pbps entre os servidores.

Os datacenters são conectados uns aos outros com nossa rede de backbone global B4 [Jai13]. B4 é uma arquitetura de rede definida por software (e usa o protocolo de comunicação de padrão aberto OpenFlow). Ele fornece largura de banda maciça para um número modesto de sites e usa alocação de largura de banda elástica para maximizar a largura de banda média [Kum15].

Software de sistema que “organiza” o hardware

Nosso hardware deve ser controlado e administrado por software que possa lidar com uma escala massiva. As falhas de hardware são um problema notável que gerenciamos com software. Dado o grande número de componentes de hardware em um cluster, as falhas de hardware ocorrem com bastante frequência. Em um único cluster em um ano típico, milhares de máquinas falham e milhares de discos rígidos quebram; quando multiplicados pelo número de clusters que operamos globalmente, esses números se tornam um tanto impressionantes. Portanto, queremos abstrair esses problemas dos usuários, e as equipes que executam nossos serviços da mesma forma não querem ser incomodadas por falhas de hardware. Cada campus de datacenter tem equipes dedicadas à manutenção do hardware e da infraestrutura do datacenter.

Gerenciando Máquinas O

Borg, ilustrado na [Figura 2-2](#), é um sistema operacional de cluster distribuído [Ver15], semelhante ao Apache Mesos.² O Borg gerencia seus trabalhos no nível do cluster.

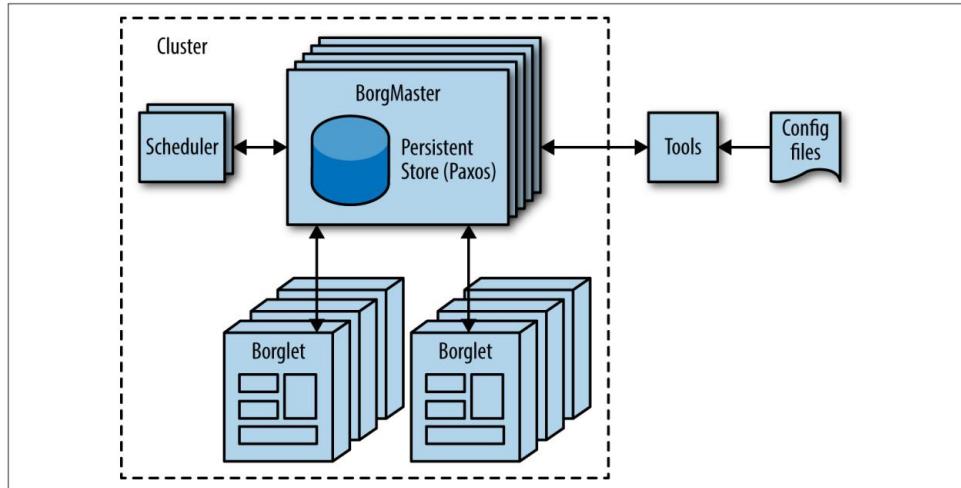


Figura 2-2. Arquitetura de cluster Borg de alto nível

² Alguns leitores podem estar mais familiarizados com o descendente de Borg, o Kubernetes — uma estrutura de orquestração de cluster de contêiner de código aberto iniciada pelo Google em 2014; veja <http://kubernetes.io> e [Bur16]. Para mais detalhes sobre as semelhanças entre Borg e Apache Mesos, veja [Ver15].

Borg é responsável por executar os trabalhos dos usuários, que podem ser servidores executando indefinidamente ou processos em lote como um MapReduce [Dea04]. Os trabalhos podem consistir em mais de uma (e às vezes milhares) de tarefas idênticas, tanto por motivos de confiabilidade quanto porque um único processo geralmente não pode lidar com todo o tráfego do cluster. Quando o Borg inicia um trabalho, ele encontra máquinas para as tarefas e diz às máquinas para iniciarem o programa servidor.

Borg então monitora continuamente essas tarefas. Se uma tarefa não funcionar corretamente, ela será encerrada e reiniciada, possivelmente em uma máquina diferente.

Como as tarefas são alocadas de maneira fluida nas máquinas, não podemos simplesmente confiar em endereços IP e números de porta para fazer referência às tarefas. Resolvemos esse problema com um nível extra de indireção: ao iniciar um trabalho, o Borg aloca um nome e um número de índice para cada tarefa usando o Borg Naming Service (BNS). Em vez de usar o endereço IP e o número da porta, outros processos se conectam às tarefas do Borg por meio do nome BNS, que é traduzido para um endereço IP e número de porta pelo BNS. Por exemplo, o caminho BNS pode ser uma string como `/bns/<cluster>/<user>/<job name>/<task number>`, que resolveria para `</IP address>:<port>`.

Borg também é responsável pela alocação de recursos para trabalhos. Cada trabalho precisa especificar seus recursos necessários (por exemplo, 3 núcleos de CPU, 2 GiB de RAM). Usando a lista de requisitos para todos os trabalhos, o Borg pode empacotar as tarefas nas máquinas de uma maneira ideal que também leva em consideração os domínios de falha (por exemplo: o Borg não executará todas as tarefas de um trabalho no mesmo rack, pois isso significa que a parte superior do comutador do rack é um ponto único de falha para esse trabalho).

Se uma tarefa tenta usar mais recursos do que o solicitado, o Borg mata a tarefa e a reinicia (já que uma tarefa de loop lento geralmente é preferível a uma tarefa que não foi reiniciada).

As Tarefas

de **Armazenamento** podem usar o disco local nas máquinas como um bloco de rascunho, mas temos várias opções de armazenamento em cluster para armazenamento permanente (e até mesmo o espaço de rascunho acabará sendo movido para o modelo de armazenamento em cluster). Eles são comparáveis ao Lustre e ao Hadoop Distributed File System (HDFS), que são ambos sistemas de arquivos de cluster de código aberto.

A camada de armazenamento é responsável por oferecer aos usuários acesso fácil e confiável ao armazenamento disponível para um cluster. Conforme mostrado na [Figura 2-3](#), o armazenamento tem muitas camadas:

1. A camada mais baixa é chamada D (para disco, embora D use discos giratórios e armazenamento flash). D é um servidor de arquivos rodando em quase todas as máquinas em um cluster. No entanto, os usuários que desejam acessar seus dados não querem ter que lembrar qual máquina está armazenando seus dados, que é onde a próxima camada entra em ação.
2. Uma camada em cima de D chamada Colossus cria um sistema de arquivos em todo o cluster que oferece a semântica usual do sistema de arquivos, bem como replicação e criptografia. Colossus é o sucessor do GFS, o Google File System [Ghe03].

3. Existem vários serviços semelhantes a banco de dados construídos sobre o Colossus:

uma. Bigtable [Cha06] é um sistema de banco de dados NoSQL que pode lidar com bancos de dados com tamanho de petabytes. Um Bigtable é um mapa classificado multidimensional esparsa, distribuído e persistente que é indexado por chave de linha, chave de coluna e carimbo de data/hora; cada valor no mapa é uma matriz de bytes não interpretada. O Bigtable oferece suporte a replicação consistente entre datacenters. b. O Spanner [Cor12] oferece uma interface do tipo SQL para usuários que exigem consistência real em todo o mundo.

c. Vários outros sistemas de banco de dados, como o Blobstore, estão disponíveis. Cada um desses opções vem com seu próprio conjunto de compensações (consulte o Capítulo 26).

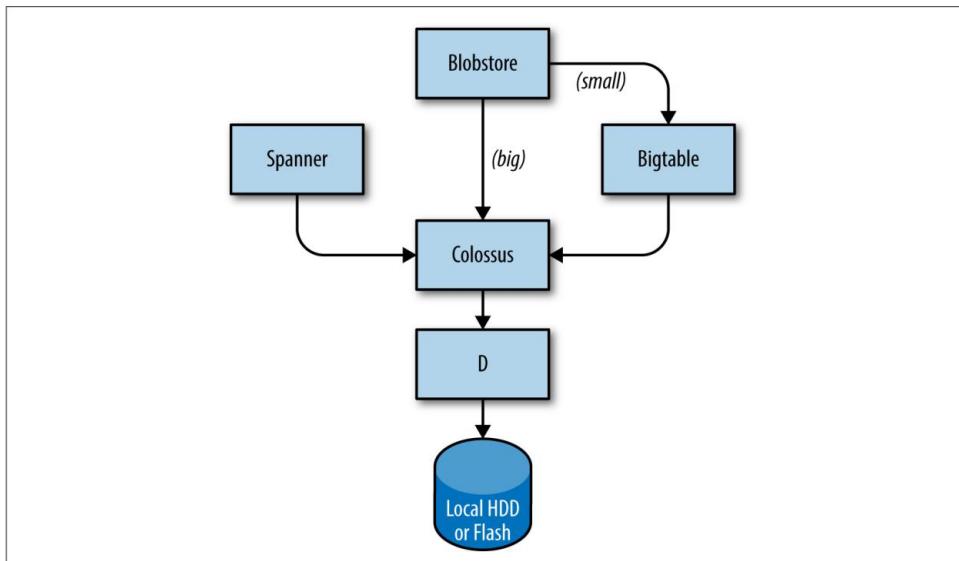


Figura 2-3. Partes da pilha de armazenamento do Google

Rede O hardware

de rede do Google é controlado de várias maneiras. Conforme discutido anteriormente, usamos uma rede definida por software baseada em OpenFlow. Em vez de usar hardware de roteamento “inteligente”, contamos com componentes de comutação “burros” mais baratos em combinação com um controlador central (duplicado) que pré-computa os melhores caminhos na rede.

Portanto, podemos afastar as decisões de roteamento de alto custo computacional dos roteadores e usar hardware de comutação simples.

A largura de banda da rede precisa ser alocada com sabedoria. Assim como o Borg limita os recursos de computação que uma tarefa pode usar, o Bandwidth Enforcer (BwE) gerencia a largura de banda disponível para maximizar a largura de banda média disponível. Otimizar a largura de banda não é

apenas sobre o custo: a engenharia de tráfego centralizada demonstrou resolver vários problemas que são tradicionalmente extremamente difíceis de resolver por meio de uma combinação de roteamento distribuído e engenharia de tráfego [Kum15].

Alguns serviços têm trabalhos executados em vários clusters, que são distribuídos em todo o mundo. Para minimizar a latência de serviços distribuídos globalmente, queremos direcionar os usuários para o datacenter mais próximo com capacidade disponível. Nosso Global So ware Load Balancer (GSLB) executa o balanceamento de carga em três níveis:

- Balanceamento de carga geográfica para solicitações de DNS (por exemplo, para www.google.com), descrito no [Capítulo 19](#)
- Balanceamento de carga em um nível de serviço do usuário (por exemplo, YouTube ou Google Maps)
- Balanceamento de carga na chamada de procedimento remoto (RPC), descrito no [Capítulo 20](#)

Os proprietários de serviço especificam um nome simbólico para um serviço, uma lista de endereços BNS de servidores e a capacidade disponível em cada um dos locais (normalmente medido em consultas por segundo). O GSLB então direciona o tráfego para os endereços BNS.

Outros softwares de sistema

Vários outros componentes em um datacenter também são importantes.

Serviço de bloqueio

O serviço de bloqueio Chubby [Bur06] fornece uma API semelhante a um sistema de arquivos para manter bloqueios. Chubby lida com esses bloqueios em locais de datacenter. Utiliza o protocolo Paxos para Consenso assíncrono (ver [Capítulo 23](#)).

Chubby também desempenha um papel importante na eleição do mestre. Quando um serviço tem cinco réplicas de um trabalho em execução para fins de confiabilidade, mas apenas uma réplica pode executar o trabalho real, Chubby é usado para selecionar qual réplica pode prosseguir.

Os dados que devem ser consistentes são adequados para armazenamento no Chubby. Por esse motivo, o BNS usa o Chubby para armazenar o mapeamento entre os caminhos do BNS e os pares de endereço IP:porta .

Monitoramento e alerta

Queremos ter certeza de que todos os serviços estão sendo executados conforme necessário. Portanto, executamos muitas instâncias de nosso programa de monitoramento Borgmon (consulte o [Capítulo 10](#)). Borgmon regularmente “raspa” métricas de servidores monitorados. Essas métricas podem ser usadas instantaneamente para alertas e também armazenadas para uso em visões gerais históricas (por exemplo, gráficos). Podemos usar o monitoramento de várias maneiras:

- Configure alertas para problemas agudos. •
- Compare o comportamento: uma atualização de software tornou o servidor mais rápido? • Examinar como o comportamento de consumo de recursos evolui ao longo do tempo, o que é essencial para o planejamento de capacidade.

Nossa Infraestrutura de Software

Nossa arquitetura de software é projetada para fazer o uso mais eficiente de nossa infraestrutura de hardware. Nosso código é altamente multithread, então uma tarefa pode facilmente usar muitos núcleos. Para facilitar painéis, monitoramento e depuração, cada servidor possui um servidor HTTP que fornece diagnósticos e estatísticas para uma determinada tarefa.

Todos os serviços do Google se comunicam usando uma infraestrutura de chamada de procedimento remoto (RPC) chamada Stubby; uma versão de código aberto, gRPC, está disponível.³ Freqüentemente, uma chamada RPC é feita mesmo quando uma chamada para uma sub-rotina no programa local precisa ser realizada. Isso torna mais fácil refatorar a chamada em um servidor diferente se mais modularidade for necessária, ou quando a base de código de um servidor crescer. O GSLB pode balancear a carga de RPCs da mesma forma que equilibra a carga de serviços visíveis externamente.

Um servidor recebe solicitações de RPC de seu front-end e envia RPCs para seu back-end. Em termos tradicionais, o frontend é chamado de cliente e o backend é chamado de servidor.

Os dados são transferidos de e para um RPC usando buffers de protocolo, 4 geralmente abreviados para “protobufs”, que são semelhantes ao Thrift do Apache. Os buffers de protocolo têm muitas vantagens sobre o XML para serializar dados estruturados: são mais simples de usar, 3 a 10 vezes menores, 20 a 100 vezes mais rápidos e menos ambíguos.

Nosso Ambiente de Desenvolvimento

A velocidade de desenvolvimento é muito importante para o Google, por isso criamos um ambiente de desenvolvimento completo para fazer uso de nossa infraestrutura [Mor12b].

Além de alguns grupos que possuem seus próprios repositórios de código aberto (por exemplo, Android e Chrome), os engenheiros de software do Google trabalham a partir de um único repositório compartilhado [Pot16]. Isso tem algumas implicações práticas importantes para nossos fluxos de trabalho:

3 Veja <http://grpc.io>.

4 Os buffers de protocolo são um mecanismo extensível neutro de linguagem e plataforma neutra para serializar dados. Para obter mais detalhes, consulte <https://developers.google.com/protocol-buffers/>.

- Se os engenheiros encontrarem um problema em um componente fora do projeto, eles poderão corrigir o problema, enviar as alterações propostas ("changelist" ou CL) ao proprietário para revisão e enviar o CL à linha principal.
- As alterações no código-fonte em um projeto do próprio engenheiro exigem uma revisão. Todo software é revisado antes de ser enviado.

Quando o software é compilado, a solicitação de compilação é enviada aos servidores de compilação em um datacenter. Mesmo grandes compilações são executadas rapidamente, pois muitos servidores de compilação podem compilar em paralelo. Essa infraestrutura também é usada para testes contínuos. Cada vez que um CL é enviado, os testes são executados em todos os softwares que podem depender desse CL, direta ou indiretamente. Se a estrutura determinar que a alteração provavelmente quebrou outras partes do sistema, ela notificará o proprietário da alteração enviada. Alguns projetos usam um sistema push-on-green, onde uma nova versão é automaticamente enviada para produção após passar nos testes.

Shakespeare: um serviço de amostra

Para fornecer um modelo de como um serviço seria implantado hipoteticamente no ambiente de produção do Google, vejamos um exemplo de serviço que interage com várias tecnologias do Google. Suponha que queremos oferecer um serviço que permita determinar onde uma determinada palavra é usada em todas as obras de Shakespeare.

Podemos dividir este sistema em duas partes:

- Um componente de lote que lê todos os textos de Shakespeare, cria um índice e grava o índice em um Bigtable. Este trabalho precisa ser executado apenas uma vez, ou talvez com pouca frequência (já que você nunca sabe se um novo texto pode ser descoberto!).
- Um front-end de aplicativo que trata das solicitações do usuário final. Este trabalho está sempre em alta, pois usuários em todos os fusos horários vão querer pesquisar nos livros de Shakespeare.

O componente de lote é um MapReduce composto por três fases.

A fase de mapeamento lê os textos de Shakespeare e os divide em palavras individuais.

Isso é mais rápido se executado em paralelo por vários trabalhadores.

A fase de embaralhamento classifica as tuplas por palavra.

Na fase de redução, uma tupla de (palavra, lista de locais) é criada.

Cada tupla é gravada em uma linha em um Bigtable, usando a palavra como chave.

Vida útil de uma solicitação

solicitação A Figura 2-4 mostra como a solicitação de um usuário é atendida: primeiro, o usuário aponta seu navegador para `shakespeare.google.com`. Para obter o endereço IP correspondente, o dispositivo do usuário resolve o endereço com seu servidor DNS (1). Essa solicitação acaba no servidor DNS do Google, que se comunica com o GSLB. Como o GSLB acompanha a carga de tráfego entre os servidores front-end em todas as regiões, ele escolhe qual endereço IP do servidor enviar a esse usuário.

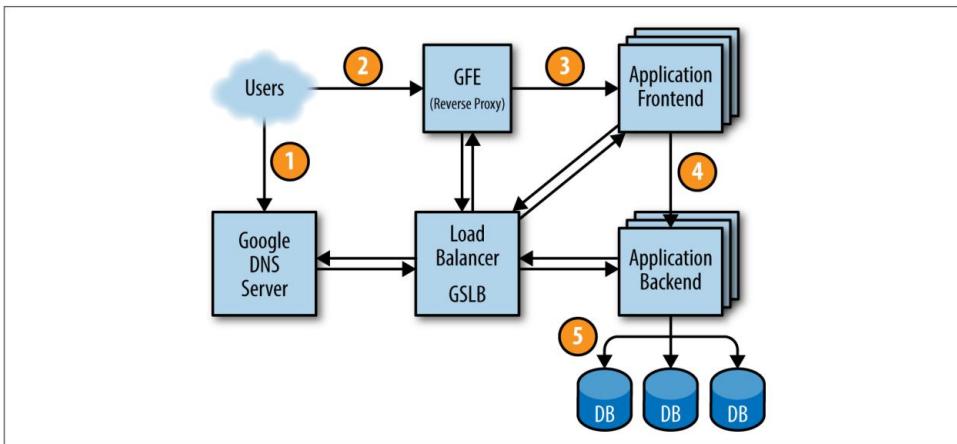


Figura 2-4. A vida de um pedido

O navegador se conecta ao servidor HTTP neste IP. Esse servidor (chamado Google Frontend ou GFE) é um proxy reverso que encerra a conexão TCP (2). O GFE procura qual serviço é necessário (pesquisa na web, mapas ou – neste caso – Shakespeare). Novamente usando GSLB, o servidor encontra um servidor front-end Shakespeare disponível e envia a esse servidor um RPC contendo a solicitação HTML (3).

O servidor Shakespeare analisa a solicitação HTML e constrói um protobuf contendo a palavra a ser pesquisada. O servidor front-end Shakespeare agora precisa entrar em contato com o servidor backend Shakespeare: o servidor front-end contata o GSLB para obter o endereço BNS de um servidor backend adequado e descarregado (4). Esse servidor backend Shakespeare agora contata um servidor Bigtable para obter os dados solicitados (5).

A resposta é gravada no protobuf de resposta e retornada ao servidor back-end de Shakespeare. O backend entrega um protobuf contendo os resultados ao servidor frontend Shakespeare, que monta o HTML e retorna a resposta ao usuário.

Toda essa cadeia de eventos é executada em um piscar de olhos – apenas algumas centenas de milissegundos! Como muitas partes móveis estão envolvidas, há muitos pontos de falha em potencial; em particular, um GSLB com falha causaria estragos. No entanto, as políticas do Google de testes rigorosos e distribuição cuidadosa, além de nossa recuperação proativa de erros

métodos como degradação graciosa, nos permitem entregar o serviço confiável que nossos usuários esperam. Afinal, as pessoas usam regularmente www.google.com para verificar se a conexão com a Internet está configurada corretamente.

Organização de Trabalho e Dados

O teste de carga determinou que nosso servidor de back-end pode lidar com cerca de 100 consultas por segundo (QPS). Testes realizados com um conjunto limitado de usuários nos levam a esperar um pico de carga de cerca de 3.470 QPS, portanto, precisamos de pelo menos 35 tarefas. No entanto, as seguintes considerações significam que precisamos de pelo menos 37 tarefas no trabalho, ou $N + 2$:

- Durante as atualizações, uma tarefa por vez ficará indisponível, deixando 36 tarefas. • Uma falha de máquina pode ocorrer durante uma atualização de tarefa, deixando apenas 35 tarefas, apenas o suficiente para atender a carga de pico.⁵

Uma análise mais detalhada do tráfego de usuários mostra que nosso pico de uso é distribuído globalmente: 1.430 QPS da América do Norte, 290 da América do Sul, 1.400 da Europa e África e 350 da Ásia e Austrália. Em vez de localizar todos os back-ends em um site, nós os distribuímos nos EUA, América do Sul, Europa e Ásia. Permitir redundância $N + 2$ por região significa que acabamos com 17 tarefas nos EUA, 16 na Europa e 6 na Ásia. No entanto, decidimos usar 4 tarefas (em vez de 5) na América do Sul, para reduzir a sobrecarga de $N + 2$ para $N + 1$. Nesse caso, estamos dispostos a tolerar um pequeno risco de maior latência em troca para custos de hardware mais baixos: se o GSLB redirecionar o tráfego de um continente para outro quando nosso datacenter sul-americano estiver

acima da capacidade, podemos economizar 20% dos recursos que gastaríamos em hardware. Nas regiões maiores, distribuiremos as tarefas em dois ou três clusters para maior resiliência.

Como os back-ends precisam entrar em contato com o Bigtable que contém os dados, também precisamos projetar esse elemento de armazenamento estrategicamente. Um back-end na Ásia contatando um Bigtable nos EUA adiciona uma quantidade significativa de latência, então replicamos o Bigtable em cada região. A replicação do Bigtable nos ajuda de duas maneiras: fornece resiliência caso um servidor do Bigtable falhe e reduz a latência de acesso a dados. Embora o Bigtable ofereça apenas consistência eventual, não é um grande problema porque não precisamos atualizar o conteúdo com frequência.

Introduzimos muita terminologia aqui; embora você não precise se lembrar de tudo, é útil para enquadrar muitos dos outros sistemas aos quais nos referiremos mais tarde.

5 Assumimos que a probabilidade de duas falhas de tarefas simultâneas em nosso ambiente é baixa o suficiente para ser desprezível. Pontos únicos de falha, como interruptores no topo do rack ou distribuição de energia, podem tornar essa suposição inválida em outros ambientes.

PARTE II

Princípios

Esta seção examina os princípios subjacentes ao modo como as equipes de SRE normalmente trabalham - os padrões, comportamentos e áreas de preocupação que influenciam o domínio geral das operações de SRE.

O primeiro capítulo desta seção, e a parte mais importante a ser lida se você quiser obter a visão mais ampla do que exatamente o SRE faz, e como raciocinamos sobre isso, é o [Capítulo 3, Abraçando o Risco](#). Ele analisa o SRE através da lente do risco – sua avaliação, gerenciamento e o uso de orçamentos de erro para fornecer abordagens neutras úteis para o gerenciamento de serviços.

Os objetivos de nível de serviço são outra unidade conceitual fundamental para o SRE. A indústria comumente agrupa conceitos dispareos sob a bandeira geral de acordos de nível de serviço, uma tendência que torna mais difícil pensar sobre esses conceitos com clareza.

O [Capítulo 4, Objetivos de nível de serviço](#), tenta separar indicadores de objetivos de acordos, examina como o SRE usa cada um desses termos e fornece algumas recomendações sobre como encontrar métricas úteis para seus próprios aplicativos.

Eliminar a labuta é uma das tarefas mais importantes do SRE e é o assunto do [Capítulo 5, Eliminando a labuta](#). Definimos labuta como um trabalho operacional mundano e repetitivo que não fornece valor duradouro, que escala linearmente com o crescimento do serviço.

Seja no Google ou em outro lugar, o monitoramento é um componente absolutamente essencial para fazer a coisa certa na produção. Se você não pode monitorar um serviço, você não sabe o que está acontecendo, e se você está cego para o que está acontecendo, você não pode ser confiável.

Leia o [Capítulo 6, Monitorando Sistemas Distribuídos](#), para obter algumas recomendações sobre o que e como monitorar e algumas práticas recomendadas independentes de implementação.

No [Capítulo 7, A Evolução da Automação no Google](#), examinamos a abordagem do SRE à automação e percorremos alguns estudos de caso de como o SRE implementou a automação, tanto com sucesso quanto sem sucesso.

A maioria das empresas trata a engenharia de lançamento como uma reflexão tardia. No entanto, como você aprenderá no [Capítulo 8, Engenharia de Liberação](#), a engenharia de liberação não é apenas crítica para a estabilidade geral do sistema – já que a maioria das interrupções resulta de uma mudança de algum tipo. É também a melhor maneira de garantir que as versões sejam consistentes.

Um princípio-chave de qualquer engenharia de software eficaz, não apenas engenharia orientada à confiabilidade, a simplicidade é uma qualidade que, uma vez perdida, pode ser extraordinariamente difícil de recuperar. No entanto, como diz o velho ditado, um sistema complexo que funciona necessariamente evolui de um sistema simples que funciona. O [Capítulo 9, Simplicidade](#), aborda esse tópico em detalhes.

Leitura adicional do Google SRE

Aumentar a velocidade do produto com segurança é um princípio fundamental para qualquer organização. Em “Making Push On Green a Reality” [Kle14], publicado em outubro de 2014, mostramos que tirar humanos do processo de liberação pode paradoxalmente reduzir o trabalho dos SREs enquanto aumenta a confiabilidade do sistema.

CAPÍTULO 3

Abraçando o Risco

**Escrito por Marc Alvidrez
Editado por Kavita Giuliani**

Você pode esperar que o Google tente construir serviços 100% confiáveis – aqueles que nunca falham. Acontece que depois de um certo ponto, no entanto, aumentar a confiabilidade é pior para um serviço (e seus usuários) em vez de melhor! A confiabilidade extrema tem um custo: maximizar a estabilidade limita a rapidez com que novos recursos podem ser desenvolvidos e a rapidez com que os produtos podem ser entregues aos usuários e aumenta drasticamente seu custo, o que, por sua vez, reduz o número de recursos que uma equipe pode oferecer. Além disso, os usuários normalmente não percebem a diferença entre alta confiabilidade e extrema confiabilidade em um serviço, porque a experiência do usuário é dominada por componentes menos confiáveis, como a rede celular ou o dispositivo com o qual estão trabalhando. Simplificando, um usuário em um smartphone 99% confiável não pode dizer a diferença entre 99,99% e 99,999% de confiabilidade do serviço! Com isso em mente, em vez de simplesmente maximizar o tempo de atividade, a Engenharia de Confiabilidade do Site procura equilibrar o risco de indisponibilidade com os objetivos de inovação rápida e operações de serviço eficientes, para que a felicidade geral dos usuários – com recursos, serviço e desempenho — é otimizado.

Gestão de risco

Sistemas não confiáveis podem minar rapidamente a confiança dos usuários, por isso queremos reduzir a chance de falha do sistema. No entanto, a experiência mostra que, à medida que construímos sistemas, o custo não aumenta linearmente à medida que a confiabilidade aumenta – uma melhoria incremental na confiabilidade pode custar 100 vezes mais do que o incremento anterior. O custo tem duas dimensões:

O custo de recursos de máquina/computação redundantes

O custo associado a equipamentos redundantes que, por exemplo, nos permite colocar sistemas offline para manutenção de rotina ou imprevista, ou oferece espaço para armazenarmos blocos de código de paridade que fornecem uma garantia mínima de durabilidade dos dados.

O custo de oportunidade

O custo suportado por uma organização quando ela aloca recursos de engenharia para construir sistemas ou recursos que diminuem o risco em vez de recursos que são diretamente visíveis ou utilizáveis pelos usuários finais. Esses engenheiros não trabalham mais em novos recursos e produtos para usuários finais.

No SRE, gerenciamos a confiabilidade do serviço em grande parte gerenciando o risco. Conceituamos o risco como um continuum. Damos igual importância a descobrir como criar maior confiabilidade nos sistemas do Google e identificar o nível adequado de tolerância para os serviços que executamos. Isso nos permite realizar uma análise de custo/benefício para determinar, por exemplo, onde no continuum de risco (não linear) devemos colocar Pesquisa, Anúncios, Gmail ou Fotos. Nossa objetivo é alinhar explicitamente o risco assumido por um determinado serviço com o risco que o negócio está disposto a assumir. Nós nos esforçamos para tornar um serviço confiável o suficiente, mas não mais confiável do que precisa ser. Ou seja, quando estabelecemos uma meta de disponibilidade de 99,99%, queremos superá-la, mas não muito: isso desperdiçaria oportunidades de adicionar recursos ao sistema, sanar dívida técnica ou reduzir seus custos operacionais. De certa forma, vemos a meta de disponibilidade como mínimo e máximo.

A principal vantagem desse enquadramento é que ele desbloqueia riscos explícitos e ponderados.

Medindo o risco do serviço

Como prática padrão no Google, geralmente somos mais bem atendidos identificando uma métrica objetiva para representar a propriedade de um sistema que queremos otimizar. Ao definir uma meta, podemos avaliar nosso desempenho atual e rastrear melhorias ou degradações ao longo do tempo. Para o risco de serviço, não está imediatamente claro como reduzir todos os fatores potenciais em uma única métrica. As falhas de serviço podem ter muitos efeitos potenciais, incluindo insatisfação do usuário, danos ou perda de confiança; perda de receita direta ou indireta; impacto de marca ou reputação; e indesejável cobertura da imprensa. Claramente, alguns desses fatores são muito difíceis de medir. Para tornar esse problema tratável e consistente em muitos tipos de sistemas que executamos, focamos no tempo de inatividade não planejado.

Para a maioria dos serviços, a maneira mais direta de representar a tolerância ao risco é em termos do nível aceitável de tempo de inatividade não planejado. O tempo de inatividade não planejado é capturado pelo nível desejado de disponibilidade do serviço, geralmente expresso em termos do número de “nove” que gostaríamos de fornecer: 99,9%, 99,99% ou 99,999% de disponibilidade.

Cada nove adicionais corresponde a uma melhoria de ordem de magnitude em direção a 100% de disponibilidade. Para sistemas de atendimento, essa métrica é tradicionalmente calculada com base na proporção do tempo de atividade do sistema (consulte a [Equação 3-1](#)).

Equação 3-1. Disponibilidade baseada no tempo

$$\text{disponibilidade} = \frac{\text{uptime}}{(\text{uptime} + \text{downtime})}$$

Usando esta fórmula ao longo do período de um ano, podemos calcular o número aceitável de minutos de inatividade para atingir um determinado número de nove de disponibilidade. Por exemplo, um sistema com uma meta de disponibilidade de 99,99% pode ficar inativo por até 52,56 minutos em um ano e permanecer dentro da meta de disponibilidade; consulte o [Apêndice A](#) para obter uma tabela.

No Google, no entanto, uma métrica baseada em tempo para disponibilidade geralmente não é significativa porque estamos analisando serviços distribuídos globalmente. Nossa abordagem para o isolamento de falhas torna muito provável que estejamos servindo pelo menos um subconjunto de tráfego para um determinado serviço em algum lugar do mundo a qualquer momento (ou seja, estamos pelo menos parcialmente “ativos” o tempo todo). Portanto, em vez de usar métricas em torno do tempo de atividade, definimos a disponibilidade em termos da taxa de sucesso da solicitação. A [Equação 3-2](#) mostra como essa métrica baseada em rendimento é calculada em uma janela contínua (ou seja, proporção de solicitações bem-sucedidas em uma janela de um dia).

Equação 3-2. Disponibilidade agregada

$$\text{disponibilidade de} \\ \text{solicitações bem-sucedidas} = \frac{\text{total de solicitações}}{\text{}}$$

Por exemplo, um sistema que atende a 2,5 milhões de solicitações em um dia com uma meta de disponibilidade diária de 99,99% pode atender até 250 erros e ainda atingir sua meta para aquele determinado dia.

Em um aplicativo típico, nem todas as solicitações são iguais: falhar em uma solicitação de inscrição de novo usuário é diferente de falhar em uma pesquisa de solicitação de novo email em segundo plano. Em muitos casos, entretanto, a disponibilidade calculada como a taxa de sucesso da solicitação sobre todas as solicitações é uma aproximação razoável do tempo de inatividade não planejado, visto da perspectiva do usuário final.

Quantificar o tempo de inatividade não planejado como uma taxa de sucesso de solicitação também torna essa métrica de disponibilidade mais acessível para uso em sistemas que normalmente não atendem diretamente aos usuários finais. A maioria dos sistemas que não atendem (por exemplo, lote, pipeline, armazenamento e sistemas transacionais) tem uma noção bem definida de unidades de trabalho bem-sucedidas e malsucedidas. De fato, embora os sistemas discutidos neste capítulo sejam principalmente sistemas de atendimento ao consumidor e à infraestrutura, muitos dos mesmos princípios também se aplicam a sistemas que não atendem com modificações mínimas.

Por exemplo, um processo em lote que extrai, transforma e insere o conteúdo de um de nossos bancos de dados de clientes em um data warehouse para permitir análises adicionais pode ser configurado para ser executado periodicamente. Usando uma taxa de sucesso de solicitação definida em termos de registros processados com sucesso e sem sucesso, podemos calcular uma métrica de disponibilidade útil, apesar do sistema em lote não funcionar constantemente.

Na maioria das vezes, definimos metas de disponibilidade trimestrais para um serviço e acompanhamos nosso desempenho em relação a essas metas semanalmente ou até diariamente. Essa estratégia nos permite gerenciar o serviço para um objetivo de disponibilidade de alto nível, procurando, rastreando e corrigindo desvios significativos à medida que surgem inevitavelmente. Consulte o [Capítulo 4](#) para obter mais detalhes.

Tolerância de Risco de Serviços

O que significa identificar a tolerância ao risco de um serviço? Em um ambiente formal ou no caso de sistemas críticos de segurança, a tolerância ao risco dos serviços normalmente é construída diretamente na definição básica do produto ou serviço. No Google, a tolerância ao risco dos serviços tende a ser menos claramente definida.

Para identificar a tolerância ao risco de um serviço, os SREs devem trabalhar com os proprietários do produto para transformar um conjunto de metas de negócios em objetivos explícitos para os quais podemos projetar. Nesse caso, os objetivos de negócios que nos preocupam têm impacto direto no desempenho e confiabilidade do serviço oferecido. Na prática, esta tradução é mais fácil dizer do que fazer. Embora os serviços ao consumidor geralmente tenham proprietários de produtos claros, é incomum que serviços de infraestrutura (por exemplo, sistemas de armazenamento ou uma camada de cache HTTP de uso geral) tenham uma estrutura semelhante de propriedade do produto. Discutiremos os casos do consumidor e da infraestrutura por sua vez.

Identificando a tolerância ao risco dos serviços ao consumidor Nossos

serviços ao consumidor geralmente têm uma equipe de produto que atua como proprietária do negócio de um aplicativo. Por exemplo, a Pesquisa, o Google Maps e o Google Docs têm seus próprios gerentes de produto. Esses gerentes de produto são encarregados de entender os usuários e os negócios e moldar o produto para o sucesso no mercado.

Quando existe uma equipe de produto, essa equipe geralmente é o melhor recurso para discutir os requisitos de confiabilidade de um serviço. Na ausência de uma equipe de produto dedicada, os engenheiros que constroem o sistema muitas vezes desempenham esse papel, consciente ou inconscientemente.

Há muitos fatores a serem considerados ao avaliar a tolerância ao risco dos serviços, como os seguintes:

- Que nível de disponibilidade é necessário? •

Diferentes tipos de falhas têm efeitos diferentes no serviço? • Como podemos usar o custo do serviço para ajudar a localizar um serviço no continuum de risco? • Que outras métricas de serviço são importantes levar em consideração?

Nível alvo de

disponibilidade O nível alvo de disponibilidade para um determinado serviço do Google geralmente depende da função que ele oferece e de como o serviço está posicionado no mercado. A lista a seguir inclui questões a serem consideradas:

- Que nível de serviço os usuários esperam? • Este serviço está diretamente relacionado à receita (nossa receita ou de nossos clientes? receita)?
- Este é um serviço pago ou gratuito? •

Se houver concorrentes no mercado, que nível de serviço esses comü os peticionários fornecem? • Este serviço é direcionado a consumidores ou empresas?

Considere os requisitos do Google Apps for Work. A maioria de seus usuários são usuários corporativos, alguns grandes e outros pequenos. Essas empresas dependem dos serviços do Google Apps for Work (por exemplo, Gmail, Agenda, Drive, Documentos) para fornecer ferramentas que permitem que seus funcionários realizem seu trabalho diário. Dito de outra forma, uma interrupção de um serviço do Google Apps for Work é uma interrupção não apenas para o Google, mas também para todas as empresas que dependem criticamente de nós. Para um serviço típico do Google Apps for Work, podemos definir uma meta de disponibilidade trimestral externa de 99,9% e respaldar essa meta com uma meta de disponibilidade interna mais forte e um contrato que estipule penalidades se não cumprirmos a meta externa.

O YouTube oferece um conjunto contrastante de considerações. Quando o Google adquiriu o YouTube, tivemos que decidir sobre a meta de disponibilidade apropriada para o site. Em 2006, o YouTube estava focado nos consumidores e estava em uma fase de seu ciclo de vida muito diferente do que o Google estava na época. Embora o YouTube já tivesse um ótimo produto, ele ainda estava mudando e crescendo rapidamente. Definimos uma meta de disponibilidade menor para o YouTube do que para nossos produtos corporativos porque o desenvolvimento rápido de recursos era correspondentemente mais importante.

Tipos de falhas

A forma esperada de falhas para um determinado serviço é outra consideração importante. Quão resiliente é o nosso negócio para atender o tempo de inatividade? O que é pior para o serviço: um baixa taxa constante de falhas ou uma interrupção ocasional de todo o site? Ambos os tipos de falha podem resultar no mesmo número absoluto de erros, mas podem ter impactos muito diferentes nos negócios.

Um exemplo ilustrativo da diferença entre interrupções completas e parciais surge naturalmente em sistemas que atendem a informações privadas. Considere um aplicativo de gerenciamento de contatos e a diferença entre falhas intermitentes que causam falhas na renderização de imagens de perfil versus um caso de falha que resulta em contatos privados de um usuário

sendo mostrado a outro usuário. O primeiro caso é claramente uma experiência de usuário ruim, e os SREs trabalhariam para remediar o problema rapidamente. No segundo caso, no entanto, o risco de expor dados privados pode facilmente minar a confiança básica do usuário de maneira significativa. Como resultado, a desativação total do serviço seria apropriada durante a fase de depuração e de limpeza potencial para o segundo caso.

Na outra ponta dos serviços oferecidos pelo Google, às vezes é aceitável ter interrupções regulares durante as janelas de manutenção. Há alguns anos, o Ads Frontend costumava ser um desses serviços. Ele é usado por anunciantes e editores de sites para configurar, configurar, executar e monitorar suas campanhas publicitárias. Como a maior parte desse trabalho ocorre durante o horário comercial normal, determinamos que interrupções ocasionais, regulares e programadas na forma de janelas de manutenção seriam aceitáveis e contamos essas interrupções programadas como tempo de inatividade planejado, não tempo de inatividade não planejado.

Custo

O custo é muitas vezes o fator chave na determinação da meta de disponibilidade apropriada para um serviço. A Ads está em uma posição particularmente boa para fazer essa troca porque os sucessos e as falhas de solicitação podem ser diretamente traduzidos em receita obtida ou perdida. Ao determinar a meta de disponibilidade para cada serviço, fazemos perguntas como:

- Se fôssemos construir e operar esses sistemas com mais nove de disponibilidade, qual seria nosso aumento incremental na receita?
- Essa receita adicional compensa o custo de atingir esse nível de confiabilidade?

Para tornar essa equação de compensação mais concreta, considere o seguinte custo/benefício para um serviço de exemplo em que cada solicitação tem o mesmo valor:

Melhoria proposta na meta de disponibilidade: 99,9% ÿ 99,99% Aumento
proposto na disponibilidade: 0,09% Receita de serviço: US\$ 1 milhão Valor
da disponibilidade aprimorada: US\$ 1 milhão * 0,0009 = US\$ 900

Nesse caso, se o custo de melhorar a disponibilidade em um nove for inferior a US\$ 900, o investimento valerá a pena. Se o custo for superior a \$900, os custos excederão o aumento projetado na receita.

Pode ser mais difícil definir essas metas quando não temos uma função de tradução simples entre confiabilidade e receita. Uma estratégia útil pode ser considerar a taxa de erro de fundo dos ISPs na Internet. Se as falhas estiverem sendo medidas da perspectiva do usuário final e for possível direcionar a taxa de erro para o serviço abaixo da taxa de erro em segundo plano, esses erros estarão dentro do ruído para a conexão de Internet de um determinado usuário. Embora existam diferenças significativas entre ISPs e protocolos (por exemplo,

TCP versus UDP, IPv4 versus IPv6), medimos a taxa de erro em segundo plano típica para ISPs entre 0,01% e 1%.

Outras métricas de serviço

Examinar a tolerância ao risco dos serviços em relação às métricas além da disponibilidade costuma ser proveitosa. Entender quais métricas são importantes e quais métricas não são importantes nos dá graus de liberdade ao tentar assumir riscos ponderados.

A latência do serviço para nossos sistemas de anúncios fornece um exemplo ilustrativo. Quando o Google lançou o Web Search pela primeira vez, um dos principais diferenciais do serviço era a velocidade.

Quando introduzimos o AdWords, que exibe anúncios ao lado dos resultados de pesquisa, um requisito fundamental do sistema era que os anúncios não retardassem a experiência de pesquisa. Esse requisito impulsionou as metas de engenharia em cada geração de sistemas do Google AdWords e é tratado como invariável.

O AdSense, o sistema de anúncios do Google que veicula anúncios contextuais em resposta a solicitações do código JavaScript que os editores inserem em seus sites, tem uma meta de latência muito diferente. A meta de latência do AdSense é evitar a lentidão na renderização da página de terceiros ao inserir anúncios contextuais. A meta de latência específica, portanto, depende da velocidade com que a página de um determinado editor é processada. Isso significa que os anúncios do AdSense geralmente podem ser veiculados centenas de milissegundos mais lentos do que os anúncios do AdWords.

Esse requisito de latência de serviço mais flexível nos permitiu fazer muitas compensações inteligentes no provisionamento (ou seja, determinar a quantidade e os locais dos recursos de serviço que usamos), o que nos economiza custos substanciais em relação ao provisionamento ingênuo. Em outras palavras, dada a relativa insensibilidade do serviço AdSense para moderar mudanças no desempenho da latência, podemos consolidar o atendimento em menos localizações geográficas, reduzindo nossa sobrecarga operacional.

Identificando a tolerância ao risco dos serviços de infraestrutura

Os requisitos para criar e executar componentes de infraestrutura diferem dos requisitos para produtos de consumo de várias maneiras. Uma diferença fundamental é que, por definição, os componentes de infraestrutura têm vários clientes, muitas vezes com necessidades variadas.

Nível alvo de disponibilidade

Considere o Bigtable [Cha06], um sistema de armazenamento distribuído em grande escala para dados estruturados. Alguns serviços ao consumidor fornecem dados diretamente do Bigtable no caminho de uma solicitação do usuário. Esses serviços precisam de baixa latência e alta confiabilidade. Outras equipes usam o Bigtable como um repositório de dados que usam para realizar análises offline (por exemplo, MapReduce) em

a base regular. Essas equipes tendem a se preocupar mais com o rendimento do que com a confiabilidade. A tolerância ao risco para esses dois casos de uso é bastante distinta.

Uma abordagem para atender às necessidades de ambos os casos de uso é projetar todos os serviços de infraestrutura para serem ultraconfiáveis. Dado o fato de que esses serviços de infraestrutura também tendem a agregar enormes quantidades de recursos, essa abordagem geralmente é muito cara na prática. Para entender as diferentes necessidades dos diferentes tipos de usuários, você pode observar o estado desejado da fila de solicitações para cada tipo de usuário do Bigtable.

Tipos de falhas O

usuário de baixa latência deseja que as filas de solicitações do Bigtable estejam (quase sempre) vazias para que o sistema possa processar cada solicitação pendente imediatamente após a chegada.

(Na verdade, o enfileiramento ineficiente geralmente é a causa da alta latência de cauda.) O usuário preocupado com a análise offline está mais interessado no rendimento do sistema, de modo que o usuário deseja que as filas de solicitações nunca fiquem vazias. Para otimizar a taxa de transferência, o sistema Bigtable nunca precisa ficar ocioso enquanto aguarda sua próxima solicitação.

Como você pode ver, sucesso e fracasso são antitéticos para esses conjuntos de usuários. O sucesso para o usuário de baixa latência é o fracasso para o usuário preocupado com a análise offline.

Custo

Uma maneira de satisfazer essas restrições concorrentes de maneira econômica é partitionar a infraestrutura e oferecê-la em vários níveis independentes de serviço. No exemplo do Bigtable, podemos criar dois tipos de clusters: clusters de baixa latência e clusters de taxa de transferência. Os clusters de baixa latência são projetados para serem operados e usados por serviços que precisam de baixa latência e alta confiabilidade. Para garantir comprimentos de fila curtos e satisfazer requisitos de isolamento de cliente mais rigorosos, o sistema Bigtable pode ser provisionado com uma quantidade substancial de capacidade de folga para contenção reduzida e redundância aumentada. Os clusters de taxa de transferência, por outro lado, podem ser provisionados para funcionar muito quente e com menos redundância, otimizando a taxa de transferência sobre a latência.

Na prática, podemos satisfazer essas necessidades a um custo muito menor, talvez de 10 a 50% do custo de um cluster de baixa latência. Dada a enorme escala do Bigtable, essa economia de custos se torna significativa muito rapidamente.

A principal estratégia em relação à infraestrutura é fornecer serviços com níveis de serviço explicitamente delineados, permitindo assim que os clientes façam as compensações corretas de risco e custo ao construir seus sistemas. Com níveis de serviço explicitamente delineados, os provedores de infraestrutura podem efetivamente externalizar a diferença no custo necessário para fornecer serviço em um determinado nível aos clientes. Exportar o custo dessa forma motiva os clientes a escolher o nível de serviço com o menor custo que ainda atende às suas necessidades. Por

Por exemplo, o Google+ pode decidir colocar dados críticos para impor a privacidade do usuário em um armazenamento de dados globalmente consistente e de alta disponibilidade (por exemplo, um sistema semelhante a SQL replicado globalmente como Spanner [Cor12]), enquanto coloca dados opcionais (dados que não são críticos, mas isso

aprimora a experiência do usuário) em um armazenamento de dados mais barato, menos confiável, menos atualizado e eventualmente consistente (por exemplo, um armazenamento NoSQL com replicação de melhor esforço como o Bigtable).

Observe que podemos executar várias classes de serviços usando hardware e software idênticos. Podemos fornecer garantias de serviço muito diferentes ajustando uma variedade de características de serviço, como as quantidades de recursos, o grau de redundância, as restrições geográficas de provisionamento e, criticamente, a configuração do software da infraestrutura.

Exemplo: infraestrutura de front-

end Para demonstrar que esses princípios de avaliação de tolerância a riscos não se aplicam apenas à infraestrutura de armazenamento, vejamos outra grande classe de serviço: a infraestrutura de front-end do Google. A infraestrutura de front-end consiste em proxy reverso e sistemas de平衡amento de carga executados próximos à borda de nossa rede. Esses são os sistemas que, entre outras coisas, servem como um ponto final das conexões dos usuários finais (por exemplo, terminam o TCP do navegador do usuário). Dado seu papel crítico, projetamos esses sistemas para fornecer um nível extremamente alto de confiabilidade. Embora os serviços ao consumidor muitas vezes possam limitar a visibilidade da falta de confiabilidade nos back-ends, esses sistemas de infraestrutura não têm tanta sorte. Se uma solicitação nunca chegar ao servidor front-end do serviço de aplicativo, ela será perdida.

Exploramos as formas de identificar a tolerância ao risco dos serviços ao consumidor e de infraestrutura. Agora, discutiremos o uso desse nível de tolerância para gerenciar a não confiabilidade por meio de orçamentos de erro.

Motivação para Orçamentos de Erro¹

Escrito por Mark Roth

Editado por Carmela Quinio

Outros capítulos deste livro discutem como as tensões podem surgir entre as equipes de desenvolvimento de produtos e as equipes de SRE, uma vez que geralmente são avaliadas com base em métricas diferentes. O desempenho do desenvolvimento de produtos é amplamente avaliado pela velocidade do produto, o que cria um incentivo para enviar um novo código o mais rápido possível. Enquanto isso, o desempenho do SRE é (sem surpresa) avaliado com base na confiabilidade de um serviço, o que implica um incentivo para resistir a uma alta taxa de mudança. A assimetria de informação entre as duas equipes amplifica ainda mais essa tensão inerente. Os desenvolvedores de produtos têm mais visibilidade do tempo e esforço envolvidos na escrita e liberação de seu código, enquanto os SREs têm mais visibilidade na confiabilidade do serviço (e no estado da produção em geral).

¹ Uma versão inicial desta seção apareceu como um artigo em ;login: (agosto de 2015, vol. 40, no. 4).

Essas tensões geralmente se refletem em diferentes opiniões sobre o nível de esforço que deve ser colocado nas práticas de engenharia. A lista a seguir apresenta algumas tensões típicas:

Tolerância a falhas de

software Quão endurecidos tornamos o software para eventos inesperados? Muito pouco, e temos um produto frágil e inutilizável. Demais, e temos um produto que ninguém quer usar (mas que funciona de forma muito estável).

Testes

Novamente, não há testes suficientes e você terá interrupções embaralhadas, vazamentos de dados de privacidade ou vários outros eventos dignos de imprensa. Muitos testes e você pode perder seu mercado.

Frequência de

push Cada push é arriscado. Quanto devemos trabalhar para reduzir esse risco, versus fazer outro trabalho?

Duração e tamanho do canário

É uma prática recomendada testar uma nova versão em algum pequeno subconjunto de uma carga de trabalho típica, uma prática geralmente chamada de canário. Quanto tempo esperamos e quão grande é o canário?

Normalmente, as equipes preexistentes elaboraram algum tipo de equilíbrio informal entre elas sobre onde está o limite de risco/esforço. Infelizmente, raramente se pode provar que esse equilíbrio é ótimo, e não apenas uma função das habilidades de negociação dos engenheiros envolvidos. Tampouco tais decisões devem ser motivadas por política, medo ou esperança.

(Na verdade, o lema não oficial do Google SRE é “Esperança não é uma estratégia”.) Em vez disso, nosso objetivo é definir uma métrica objetiva, acordada por ambos os lados, que possa ser usada para orientar as negociações de forma reproduzível. Quanto mais baseada em dados a decisão puder ser, melhor.

Formando seu orçamento de erros Para

basear essas decisões em dados objetivos, as duas equipes definem conjuntamente um orçamento de erros trimestral com base no objetivo de nível de serviço do serviço, ou SLO (consulte o [Capítulo 4](#)). O orçamento de erro fornece uma métrica clara e objetiva que determina o quanto o serviço não é confiável em um único trimestre. Essa métrica remove a política das negociações entre os SREs e os desenvolvedores de produtos ao decidir quanto risco permitir.

Nossa prática é então a seguinte:

- O Gerenciamento de Produto define um SLO, que define uma expectativa de quanto tempo de atividade que o serviço deve ter por trimestre.

- O tempo de atividade real é medido por um terceiro neutro: nosso sistema de monitoramento. • A diferença entre esses dois números é o “orçamento” de quanta “falta de confiabilidade” resta para o trimestre. • Enquanto o tempo de atividade medido estiver acima do SLO – em outras palavras, enquanto houver erro de orçamento restante – novos lançamentos podem ser enviados.

Por exemplo, imagine que o SLO de um serviço deve atender com sucesso 99,999% de todas as consultas por trimestre. Isso significa que o orçamento de erro do serviço é uma taxa de falha de 0,001% para um determinado trimestre. Se um problema nos fizer falhar em 0,0002% das consultas esperadas para o trimestre, o problema gastará 20% do orçamento de erro trimestral do serviço.

Benefícios

O principal benefício de um orçamento de erro é que ele fornece um incentivo comum que permite que o desenvolvimento do produto e o SRE se concentrem em encontrar o equilíbrio certo entre inovação e confiabilidade.

Muitos produtos usam esse loop de controle para gerenciar a velocidade de lançamento: desde que os SLOs do sistema sejam atendidos, os lançamentos podem continuar. Se as violações de SLO ocorrerem com frequência suficiente para gastar o orçamento de erros, os lançamentos serão temporariamente interrompidos enquanto recursos adicionais são investidos no teste e desenvolvimento do sistema para torná-lo mais resiliente, melhorar seu desempenho e assim por diante. Abordagens mais sutis e eficazes estão disponíveis do que essa técnica simples de ligar/desligar:² por exemplo, retardar lançamentos ou revertê-los quando o orçamento de erro de violação de SLO estiver próximo de ser usado.

Por exemplo, se o desenvolvimento de produtos quiser economizar nos testes ou aumentar a velocidade de envio e o SRE for resistente, o orçamento de erros orienta a decisão. Quando o orçamento é grande, os desenvolvedores de produtos podem correr mais riscos. Quando o orçamento estiver quase esgotado, os próprios desenvolvedores de produtos pressionarão por mais testes ou velocidade de envio mais lenta, pois não querem arriscar usar o orçamento e atrasar o lançamento. Com efeito, a equipe de desenvolvimento de produto torna-se autopolicida. Eles conhecem o orçamento e podem gerenciar seu próprio risco. (É claro que esse resultado depende de uma equipe SRE ter autoridade para realmente interromper os lançamentos se o SLO for interrompido.)

O que acontece se uma interrupção de rede ou falha do datacenter reduzir o SLO medido? Esses eventos também consomem o orçamento de erros. Como resultado, o número de novos pushes pode ser reduzido para o restante do trimestre. Toda a equipe apoia essa redução porque todos compartilham a responsabilidade pelo tempo de atividade.

O orçamento também ajuda a destacar alguns dos custos de metas de confiabilidade excessivamente altas, em termos de inflexibilidade e inovação lenta. Se a equipe está tendo problemas

² Conhecido como controle “bang/bang”—consulte https://en.wikipedia.org/wiki/Bang–bang_control.

lançando novos recursos, eles podem optar por afrouxar o SLO (aumentando assim o orçamento de erros) para aumentar a inovação.

Principais insights

- Gerenciar a confiabilidade do serviço é em grande parte gerenciar riscos e gerenciar riscos pode ser caro.
- 100% provavelmente nunca é a meta de confiabilidade certa: além de ser impossível de alcançar, normalmente é mais confiabilidade do que os usuários de um serviço desejam ou percebem. Combine o perfil do serviço com o risco que a empresa está disposta a assumir.
- Um orçamento de erro alinha incentivos e enfatiza a propriedade conjunta entre SRE e desenvolvimento de produtos. Os orçamentos de erro facilitam a decisão da taxa de lançamentos e desativam efetivamente as discussões sobre interrupções com as partes interessadas e permitem que várias equipes cheguem à mesma conclusão sobre o risco de produção sem rancor.

CAPÍTULO 4

Objetivos de nível de serviço

**Escrito por Chris Jones, John Wilkes e Niall Murphy com
Cody Smith
Editado por Betsy Beyer**

É impossível gerenciar um serviço corretamente, muito menos bem, sem entender quais comportamentos realmente importam para esse serviço e como medir e avaliar esses comportamentos. Para isso, gostaríamos de definir e entregar um determinado nível de serviço aos nossos usuários, sejam eles uma API interna ou um produto público.

Usamos intuição, experiência e uma compreensão do que os usuários desejam para definir indicadores de nível de serviço (SLIs), objetivos (SLOs) e acordos (SLAs). Essas medições descrevem as propriedades básicas das métricas que importam, quais valores queremos que essas métricas tenham e como reagiremos se não pudermos fornecer o serviço esperado. Em última análise, a escolha de métricas apropriadas ajuda a conduzir a ação certa se algo der errado e também dá a uma equipe de SRE a confiança de que um serviço é saudável.

Este capítulo descreve a estrutura que usamos para lidar com os problemas de modelagem métrica, seleção métrica e análise métrica. Grande parte dessa explicação seria bastante abstrata sem um exemplo, então usaremos o serviço de Shakespeare descrito em “[Shakespeare: um exemplo de serviço](#)” na [página 20](#) para ilustrar nossos pontos principais.

Terminologia do nível de serviço

Muitos leitores provavelmente estão familiarizados com o conceito de SLA, mas os termos SLI e SLO também merecem uma definição cuidadosa, porque no uso comum, o termo SLA é sobrecarregado e assumiu vários significados dependendo do contexto. Preferimos separar esses significados para maior clareza.

Indicadores

Um SLI é um indicador de nível de serviço – uma medida quantitativa cuidadosamente definida de algum aspecto do nível de serviço fornecido.

A maioria dos serviços considera a latência de solicitação — quanto tempo leva para retornar uma resposta a uma solicitação — como um SLI de chave. Outros SLIs comuns incluem a taxa de erro, geralmente expressa como uma fração de todas as solicitações recebidas, e a taxa de transferência do sistema, normalmente medida em solicitações por segundo. As medições geralmente são agregadas: ou seja, os dados brutos são coletados em uma janela de medição e depois transformados em uma taxa, média ou percentil.

Idealmente, o SLI mede diretamente um nível de serviço de interesse, mas às vezes apenas um proxy está disponível porque a medida desejada pode ser difícil de obter ou interpretar. Por exemplo, a latência do lado do cliente geralmente é a métrica mais relevante para o usuário, mas pode ser possível apenas medir a latência no servidor.

Outro tipo de SLI importante para os SREs é a disponibilidade, ou a fração do tempo em que um serviço pode ser usado. Muitas vezes, é definido em termos da fração de solicitações bem formadas que são bem-sucedidas, às vezes chamadas de rendimento. (Durabilidade - a probabilidade de que os dados sejam retidos por um longo período de tempo - é igualmente importante para sistemas de armazenamento de dados.) Embora 100% de disponibilidade seja impossível, quase 100% de disponibilidade geralmente é facilmente alcançável, e a indústria geralmente expressa valores de alta disponibilidade em termos do número de "nove" na porcentagem de disponibilidade. Por exemplo, disponibilidades de 99% e 99,999% podem ser chamadas de disponibilidade "2 noves" e "5 noves", respectivamente, e a meta publicada atual para a disponibilidade do Google Compute Engine é "três noves e meio" — disponibilidade de 99,95%. .

Objetivos Um

SLO é um objetivo de nível de serviço: um valor alvo ou intervalo de valores para um nível de serviço que é medido por um SLI. Uma estrutura natural para SLOs é, portanto, SLI \leq alvo ou limite inferior \leq SLI \leq limite superior. Por exemplo, podemos decidir que retornaremos os resultados da pesquisa do Shakespeare “rapidamente”, adotando um SLO de que nossa latência média de solicitação de pesquisa deve ser inferior a 100 milissegundos.

Escolher um SLO apropriado é complexo. Para começar, nem sempre você pode escolher seu valor! Para solicitações HTTP recebidas do mundo externo para seu serviço, a métrica de consultas por segundo (QPS) é essencialmente determinada pelos desejos de seus usuários, e você não pode realmente definir um SLO para isso.

Por outro lado, você pode dizer que deseja que a latência média por solicitação seja inferior a 100 milissegundos, e definir essa meta pode motivá-lo a escrever seu frontend com comportamentos de baixa latência de vários tipos ou comprar certos tipos de equipamentos de baixa latência. (100 milissegundos é obviamente um valor arbitrário, mas em geral números de latência mais baixos são bons. Existem excelentes razões para acreditar que rápido é

melhor do que lento, e essa latência experimentada pelo usuário acima de certos valores na verdade afasta as pessoas – veja “Speed Matters” [Bru09] para mais detalhes.)

Novamente, isso é mais sutil do que pode parecer à primeira vista, pois esses dois SLIs - QPS e latência - podem estar conectados nos bastidores: QPS mais alto geralmente leva a latências maiores e é comum que os serviços tenham um penhasco de desempenho além de alguns limites de carga.

Escolher e publicar SLOs para usuários define as expectativas sobre o desempenho de um serviço. Essa estratégia pode reduzir reclamações infundadas aos proprietários do serviço sobre, por exemplo, a lentidão do serviço. Sem um SLO explícito, os usuários geralmente desenvolvem suas próprias crenças sobre o desempenho desejado, que podem não estar relacionadas às crenças das pessoas que projetam e operam o serviço. Essa dinâmica pode levar tanto à dependência excessiva do serviço, quando os usuários acreditam incorretamente que um serviço estará mais disponível do que realmente é (como aconteceu com Chubby: consulte [“A interrupção planejada global da Chubby”](#)), quanto à subconfiança, quando os usuários em potencial acreditam que um sistema é mais esquisito e menos confiável do que realmente é.

A interrupção planejada global Chubby

Escrito por Marc Alvidrez

Chubby [\[Bur06\]](#) é o serviço de bloqueio do Google para sistemas distribuídos fracamente acoplados. No caso global, distribuímos instâncias Chubby de forma que cada réplica esteja em uma região geográfica diferente. Com o tempo, descobrimos que as falhas da instância global do Chubby geravam interrupções de serviço consistentemente, muitas das quais eram visíveis para os usuários finais. Como se vê, as verdadeiras interrupções globais do Chubby são tão raras que os proprietários de serviços começaram a adicionar dependências ao Chubby assumindo que ele nunca cairia. Sua alta confiabilidade forneceu uma falsa sensação de segurança porque os serviços não podiam funcionar adequadamente quando Chubby estava indisponível, embora isso raramente ocorresse.

A solução para este cenário do Chubby é interessante: o SRE garante que o Chubby global atenda, mas não excede significativamente, seu objetivo de nível de serviço. Em qualquer trimestre, se uma falha real não tiver reduzido a disponibilidade abaixo da meta, uma interrupção controlada será sintetizada pela desativação intencional do sistema. Dessa forma, podemos eliminar dependências não razoáveis no Chubby logo após serem adicionadas. Isso força os proprietários de serviços a considerar a realidade dos sistemas distribuídos mais cedo ou mais tarde.

Contratos Por

fim, SLAs são contratos de nível de serviço: um contrato explícito ou implícito com seus usuários que inclui as consequências de cumprir (ou não) os SLOs que eles contêm. As consequências são mais facilmente reconhecidas quando são financeiras – um desconto ou uma pena.

mas podem assumir outras formas. Uma maneira fácil de dizer a diferença entre um SLO e um SLA é perguntar “o que acontece se os SLOs não forem atendidos?”: se não houver uma consequência explícita, então você quase certamente está olhando para um SLO¹. I normalmente se envolve na construção de SLAs, porque os SLAs estão intimamente ligados às decisões de negócios e produtos. O SRE, no entanto, se envolve em ajudar a evitar o desencadeamento das consequências de SLOs perdidos. Eles também podem ajudar a definir os SLIs: obviamente precisa haver uma maneira objetiva de medir os SLOs no acordo, ou surgirão desacordos.

A Pesquisa Google é um exemplo de serviço importante que não possui SLA para o público: queremos que todos usem a Pesquisa da forma mais fluida e eficiente possível, mas não assinamos um contrato com o mundo inteiro. Mesmo assim, ainda há consequências se a Pesquisa não estiver disponível. A indisponibilidade resulta em um impacto em nossa reputação, bem como em uma queda na receita de publicidade. Muitos outros serviços do Google, como o Google for Work, têm SLAs explícitos com seus usuários. Independentemente de um serviço específico ter ou não um SLA, é importante definir SLIs e SLOs e usá-los para gerenciar o serviço.

Tanto para a teoria, agora para a experiência.

Indicadores na prática

Dado que defendemos a importância da escolha de métricas apropriadas para medir seu serviço, como você identifica quais métricas são significativas para seu serviço ou sistema?

Com o que você e seus usuários se importam?

Você não deve usar todas as métricas que puder acompanhar em seu sistema de monitoramento como um SLI; uma compreensão do que seus usuários desejam do sistema informará a seleção criteriosa de alguns indicadores. Escolher muitos indicadores torna difícil prestar o nível certo de atenção aos indicadores que importam, enquanto escolher muito poucos pode deixar comportamentos significativos do seu sistema não examinados. Normalmente descobrimos que um punhado de indicadores representativos é suficiente para avaliar e raciocinar sobre a saúde de um sistema.

¹ A maioria das pessoas realmente quer dizer SLO quando dizem “SLA”. Um brinde: se alguém fala sobre uma “violação de SLA”, quase sempre está falando sobre um SLO perdido. Uma violação real do SLA pode desencadear um processo judicial por quebra de contrato.

Os serviços tendem a se enquadrar em algumas categorias amplas em termos dos SLIs que consideram relevantes:

- Os sistemas de atendimento voltados para o usuário, como os frontends de pesquisa do Shakespeare, geralmente se preocupam com disponibilidade, latência e taxa de transferência. Em outras palavras: poderíamos responder ao pedido? Quanto tempo demorou para responder? Quantas solicitações poderiam ser tratadas?
- Os sistemas de armazenamento geralmente enfatizam a latência, a disponibilidade e a durabilidade. Em outras palavras: quanto tempo leva para ler ou gravar dados? Podemos acessar os dados sob demanda? Os dados ainda estão lá quando precisamos deles? Consulte o [Capítulo 26](#) para uma discussão mais ampla dessas questões.
- Sistemas de big data, como pipelines de processamento de dados, tendem a se preocupar com a taxa de transferência e a latência de ponta a ponta. Em outras palavras: quantos dados estão sendo processados? Quanto tempo leva para os dados progredirem da ingestão até a conclusão? (Alguns pipelines também podem ter metas de latência em estágios de processamento individuais.) • Todos os sistemas devem se preocupar com a exatidão: a resposta certa foi retornada, os dados certos recuperados, a análise certa foi feita? A correção é importante para rastrear como um indicador da integridade do sistema, embora muitas vezes seja uma propriedade dos dados no sistema e não da infraestrutura em si e, portanto, geralmente não é uma responsabilidade do SRE a ser atendida.

Coletando indicadores Muitas

métricas de indicadores são mais naturalmente reunidas no lado do servidor, usando um sistema de monitoramento como Borgmon (consulte o [Capítulo 10](#)) ou Prometheus, ou com análise periódica de logs – por exemplo, respostas HTTP 500 como uma fração de todas as solicitações. No entanto, alguns sistemas devem ser instrumentados com coleta do lado do cliente, porque não medir o comportamento no cliente pode perder uma série de problemas que afetam os usuários, mas não afetam as métricas do lado do servidor. Por exemplo, concentrar-se na latência de resposta do backend de pesquisa do Shakespeare pode perder uma latência ruim do usuário devido a problemas com o JavaScript da página: nesse caso, medir quanto tempo leva para uma página se tornar utilizável no navegador é um proxy melhor para o que o usuário realmente experimenta.

Agregação Para

simplicidade e usabilidade, geralmente agregamos medições brutas. Isso precisa ser feito com cuidado.

Algumas métricas são aparentemente diretas, como o número de solicitações por segundo atendidas, mas mesmo essa medição aparentemente direta agrupa dados implicitamente na janela de medição. A medição é obtida uma vez por segundo ou pela média de solicitações de um minuto? O último pode ocultar taxas de solicitação instantânea muito mais altas em rajadas que duram apenas alguns segundos. Considere um sistema que serve

200 solicitações/s em segundos pares e 0 nos demais. Tem a mesma carga média de uma que atende a 100 solicitações/s constantes, mas tem uma carga instantânea duas vezes maior que a média. Da mesma forma, a média das latências de solicitação pode parecer atraente, mas obscurece um detalhe importante: é perfeitamente possível que a maioria das solicitações seja rápida, mas que uma longa cauda de solicitações seja muito, muito mais lenta.

A maioria das métricas é melhor pensada como distribuições em vez de médias. Por exemplo, para um SLI de latência, algumas solicitações serão atendidas rapidamente, enquanto outras invariavelmente levarão mais tempo, às vezes muito mais. Uma média simples pode obscurecer essas latências de cauda, bem como mudanças nelas. A Figura 4-1 fornece um exemplo: embora uma solicitação típica seja atendida em cerca de 50 ms, 5% das solicitações são 20 vezes mais lentas! Monitoramento e alerta com base apenas na latência média não mostrariam nenhuma mudança no comportamento ao longo do dia, quando de fato há mudanças significativas na latência da cauda (a linha superior).

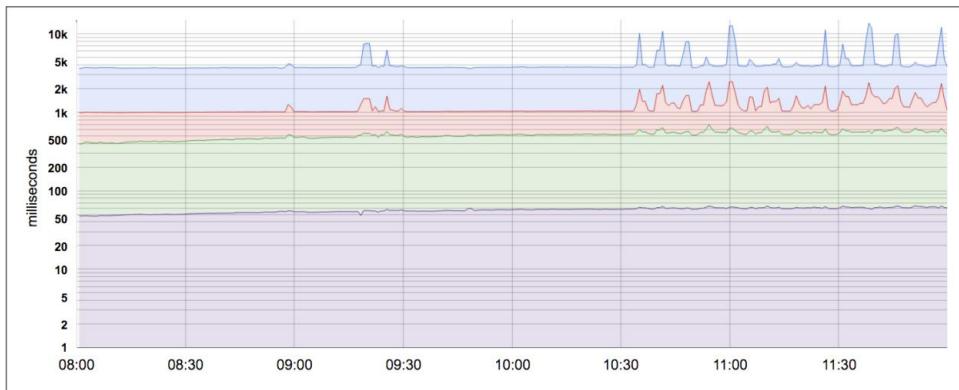


Figura 4-1. Latências de 50º, 85º, 95º e 99º percentil para um sistema. Observe que o eixo Y tem uma escala logarítmica.

O uso de percentis para indicadores permite que você considere a forma da distribuição e seus diferentes atributos: um percentil de alta ordem, como o 99º ou 99,9º, mostra um valor plausível de pior caso, enquanto usa o percentil 50 (também conhecido como a mediana) enfatiza o caso típico. Quanto maior a variação nos tempos de resposta, mais a experiência típica do usuário é afetada pelo comportamento de cauda longa, um efeito exacerbado em alta carga por efeitos de fila. Estudos com usuários mostraram que as pessoas normalmente preferem um sistema um pouco mais lento a um com alta variação no tempo de resposta, então algumas equipes de SRE se concentram apenas em valores percentuais altos, alegando que, se o comportamento do percentil 99,9 for bom, então a experiência típica certamente será.

Uma nota sobre falácia estatísticas

Geralmente preferimos trabalhar com percentis em vez da média (média aritmética) de um conjunto de valores. Isso torna possível considerar a cauda longa dos pontos de dados, que geralmente têm características significativamente diferentes (e mais interessantes) do que a média. Devido à natureza artificial dos sistemas de computação, os pontos de dados geralmente são distorcidos - por exemplo, nenhuma solicitação pode ter uma resposta em menos de 0 ms, e um tempo limite de 1.000 ms significa que não pode haver respostas bem-sucedidas com valores maiores que o tempo limite. Como resultado, não podemos supor que a média e a mediana sejam iguais – ou mesmo próximas uma da outra!

Tentamos não assumir que nossos dados são normalmente distribuídos sem antes verificá-los, caso algumas intuições e aproximações padrão não sejam válidas. Por exemplo, se a distribuição não for a esperada, um processo que aja quando vê discrepâncias (por exemplo, reiniciar um servidor com altas latências de solicitação) pode fazer isso com muita ou pouca frequência.

Padronizar indicadores

Recomendamos que você padronize as definições comuns para SLIs para que você não precise raciocinar sobre eles a partir de princípios básicos a cada vez. Qualquer recurso que esteja em conformidade com os modelos de definição padrão pode ser omitido da especificação de um SLI individual, por exemplo:

- Intervalos de agregação: “Média de mais de 1 minuto” •
- Regiões de agregação: “Todas as tarefas em um cluster” •
- Com que frequência as medições são feitas: “A cada 10 segundos” • Quais solicitações estão incluídas: “HTTP GETs de trabalhos de monitoramento de caixa preta” •
- Como os dados são adquiridos: “Através do nosso monitoramento, medido no servidor” •
- Latência de acesso aos dados: “Tempo até o último byte”

Para economizar esforços, crie um conjunto de modelos SLI reutilizáveis para cada métrica comum; isso também torna mais simples para todos entender o que um SLI específico significa.

Objetivos na prática

Comece pensando (ou descobrindo!) com o que seus usuários se importam, não com o que você pode medir. Muitas vezes, o que importa para seus usuários é difícil ou impossível de medir, então você acabará se aproximando das necessidades dos usuários de alguma forma. No entanto, se você simplesmente começar com o que é fácil de medir, acabará com SLOs menos úteis. Como resultado, temos

às vezes descobri que trabalhar a partir de objetivos desejados para trás em indicadores específicos funciona melhor do que escolher indicadores e depois chegar a metas.

Definição de objetivos Para

maior clareza, os SLOs devem especificar como são medidos e as condições sob as quais são válidos. Por exemplo, podemos dizer o seguinte (a segunda linha é igual à primeira, mas depende dos padrões SLI da seção anterior para remover a redundância):

- 99% (média de mais de 1 minuto) das chamadas Get RPC serão concluídas em menos de 100 ms (medido em todos os servidores de back-end).
- 99% das chamadas Get RPC serão concluídas em menos de 100 ms.

Se a forma das curvas de desempenho for importante, você poderá especificar vários Metas de SLO:

- 90% das chamadas Get RPC serão concluídas em menos de 1 ms. •
- 99% das chamadas Get RPC serão concluídas em menos de 10 ms. •
- 99,9% das chamadas Get RPC serão concluídas em menos de 100 ms.

Se você tiver usuários com cargas de trabalho heterogêneas, como um pipeline de processamento em massa que se preocupa com a taxa de transferência e um cliente interativo que se preocupa com a latência, pode ser apropriado definir objetivos separados para cada classe de carga de trabalho:

- 95% das chamadas Set RPC dos clientes de throughput serão concluídas em < 1 s.
- 99% das chamadas RPC definidas dos clientes de latência com cargas úteis < 1 kB serão concluídas em < 10 EM.

É irrealista e indesejável insistir que os SLOs serão atendidos 100% do tempo: isso pode reduzir a taxa de inovação e implantação, exigir soluções caras e excessivamente conservadoras, ou ambos. Em vez disso, é melhor permitir um erro de orçamento - uma taxa na qual os SLOs podem ser perdidos - e rastreá-lo diariamente ou semanalmente. A alta administração provavelmente também desejará uma avaliação mensal ou trimestral. (Um orçamento de erro é apenas um SLO para atender a outros SLOs!)

A taxa em que os SLOs são perdidos é um indicador útil para a integridade do serviço percebida pelo usuário. É útil rastrear SLOs (e violações de SLO) diariamente ou semanalmente para ver tendências e receber avisos antecipados de possíveis problemas antes que eles aconteçam.

A alta administração provavelmente também desejará uma avaliação mensal ou trimestral.

A taxa de violação de SLO pode ser comparada com o orçamento de erro (consulte “[Motivação para orçamentos de erro](#)” na página 33), com a lacuna usada como entrada para o processo que decide quando lançar novas versões.

Escolhendo alvos A

escolha de alvos (SLOs) não é uma atividade puramente técnica devido às implicações do produto e do negócio, que devem ser refletidas tanto nos SLIs quanto nos SLOs (e talvez SLAs) selecionados. Da mesma forma, pode ser necessário trocar certos atributos do produto por outros dentro das restrições impostas pela equipe, tempo de colocação no mercado, disponibilidade de hardware e financiamento. Embora o SRE deva fazer parte desta conversa e aconselhar sobre os riscos e a viabilidade de diferentes opções, aprendemos algumas lições que podem ajudar a tornar essa discussão mais produtiva:

Não escolha uma meta com base no desempenho atual

Embora a compreensão dos méritos e limites de um sistema seja essencial, a adoção de valores sem reflexão pode prendê-lo a apoiar um sistema que exige esforços heróicos para atingir suas metas e que não pode ser melhorado sem reformulação significativa.

Mantenha a

simplicidade Agregações complicadas em SLIs podem obscurecer as alterações no desempenho do sistema e também são mais difíceis de raciocinar.

Evite absolutos

Embora seja tentador pedir um sistema que possa dimensionar sua carga “infinitamente” sem nenhum aumento de latência e que esteja “sempre” disponível, esse requisito não é realista.

Mesmo um sistema que se aproxime de tais ideais provavelmente levará muito tempo para ser projetado e construído, e será caro para operar – e provavelmente se tornará desnecessariamente melhor do que os usuários ficariam felizes (ou mesmo encantados) em ter.

Tenha o mínimo de SLOs possível

Escolha apenas SLOs suficientes para fornecer uma boa cobertura dos atributos do seu sistema.

Defenda os SLOs que você escolher: se você nunca conseguir ganhar uma conversa sobre prioridades citando um SLO específico, provavelmente não vale a pena ter esse SLO.² No entanto, nem todos os atributos do produto são receptivos aos SLOs: é difícil especificar “com um SLO”.

A perfeição pode

esperar Você sempre pode refinar as definições e metas de SLO ao longo do tempo à medida que aprende sobre o comportamento de um sistema. É melhor começar com um alvo solto que você aperta do que

² Se você nunca consegue ganhar uma conversa sobre SLOs, provavelmente não vale a pena ter uma equipe de SRE para o produto.

escolha um alvo excessivamente estrito que deve ser relaxado quando você descobrir que é inatingível.

Os SLOs podem – e devem – ser um fator importante na priorização do trabalho para SREs e desenvolvedores de produtos, porque refletem o que os usuários se preocupam. Um bom SLO é uma função de força útil e legítima para uma equipe de desenvolvimento. Mas um SLO mal pensado pode resultar em trabalho desperdiçado se uma equipe usar esforços heróicos para atender a um SLO excessivamente agressivo, ou um produto ruim se o SLO for muito fraco. Os SLOs são uma alavanca enorme: use-os com sabedoria.

Medidas de controle

SLIs e SLOs são elementos cruciais nas malhas de controle usadas para gerenciar sistemas:

1. Monitore e meça os SLIs do sistema.
2. Compare os SLIs com os SLOs e decida se uma ação é necessária ou não.
3. Se for necessária ação, descubra o que precisa acontecer para atingir a meta.
4. Tome essa ação.

Por exemplo, se a etapa 2 mostrar que a latência da solicitação está aumentando e perderá o SLO em algumas horas, a menos que algo seja feito, a etapa 3 pode incluir testar a hipótese de que os servidores estão vinculados à CPU e decidir adicionar mais deles ao espalhar a carga. Sem o SLO, você não saberia se (ou quando) agir.

SLOs definem expectativas A

publicação de SLOs define expectativas para o comportamento do sistema. Os usuários (e usuários em potencial) geralmente querem saber o que podem esperar de um serviço para entender se ele é apropriado para seu caso de uso. Por exemplo, uma equipe que deseja criar um site de compartilhamento de fotos pode querer evitar o uso de um serviço que promete durabilidade muito forte e baixo custo em troca de disponibilidade um pouco menor, embora o mesmo serviço possa ser perfeito para um sistema de gerenciamento de registros de arquivo .

Para definir expectativas realistas para seus usuários, considere usar uma ou ambas as táticas a seguir:

Mantenha uma margem

de segurança O uso de um SLO interno mais rígido do que o SLO anunciado aos usuários oferece espaço para responder a problemas crônicos antes que eles se tornem visíveis externamente. Um buffer SLO também permite acomodar reimplementações que trocam desempenho por outros atributos, como custo ou facilidade de manutenção, sem ter que decepcionar os usuários.

Não exagere Os

usuários baseiam-se na realidade do que você oferece, em vez do que você diz que fornecerá, principalmente para serviços de infraestrutura. Se o desempenho real do seu serviço for muito melhor do que o SLO declarado, os usuários passarão a confiar em seu desempenho atual.

Você pode evitar a dependência excessiva colocando o sistema off-line deliberadamente ocasionalmente (o serviço Chubby do Google introduziu interrupções planejadas em resposta à disponibilidade excessiva),³ limitando algumas solicitações ou projetando o sistema para que não seja mais rápido sob cargas leves.

Compreender o quão bem um sistema está atendendo às suas expectativas ajuda a decidir se deve investir para tornar o sistema mais rápido, mais disponível e mais resiliente. Alternativamente, se o serviço estiver indo bem, talvez o tempo da equipe deva ser gasto em outras prioridades, como pagar dívidas técnicas, adicionar novos recursos ou introduzir outros produtos.

Acordos na prática

A elaboração de um SLA exige que as equipes jurídica e de negócios escolham as consequências e penalidades apropriadas para uma violação. O papel do SRE é ajudá-los a entender a probabilidade e a dificuldade de atender aos SLOs contidos no SLA. Muitos dos conselhos sobre a construção de SLOs também se aplicam a SLAs. É sensato ser conservador no que você anuncia aos usuários, pois quanto mais amplo o público, mais difícil é alterar ou excluir SLAs que provam ser imprudentes ou difíceis de trabalhar.

³ A injeção de falhas [Ben12] serve a um propósito diferente, mas também pode ajudar a definir expectativas.

CAPÍTULO 5

Eliminando o Trabalho

**Escrito por Vivek Rau
Editado por Betsy Beyer**

Se um operador humano precisar tocar em seu sistema durante as operações normais, você terá um bug.

A definição de normal muda à medida que seus sistemas crescem.

—Carla Geisser, Google SRE

No SRE, queremos dedicar tempo ao trabalho de projeto de engenharia de longo prazo em vez do trabalho operacional. Como o termo trabalho operacional pode ser mal interpretado, usamos uma palavra específica: labuta.

Trabalho definido

A labuta não é apenas “trabalho que não gosto de fazer”. Também não é simplesmente equivalente a tarefas administrativas ou trabalho sujo. As preferências sobre quais tipos de trabalho são satisfatórios e agradáveis variam de pessoa para pessoa, e algumas pessoas até gostam de trabalho manual e repetitivo. Há também tarefas administrativas que precisam ser feitas, mas não devem ser categorizadas como labuta: isso é sobrecarga. A sobrecarga geralmente é um trabalho não diretamente vinculado à execução de um serviço de produção e inclui tarefas como reuniões de equipe, definição e classificação de metas,¹ fragmentos² e papelada de RH. O trabalho sujo às vezes pode ter valor a longo prazo e, nesse caso, também não é labuta. Limpar toda a configuração de alerta para o seu serviço e remover a desordem pode ser sujo, mas não é trabalhoso.

Então, o que é labuta? A labuta é o tipo de trabalho vinculado à execução de um serviço de produção que tende a ser manual, repetitivo, automatizável, tático, desprovido de valor duradouro e

¹ Usamos o sistema de Objetivos e Resultados-Chave, iniciado por Andy Grove na Intel; veja [Kla12].

² Googlers gravam resumos curtos em formato livre, ou “trechos”, do que trabalhamos a cada semana.

que escala linearmente à medida que um serviço cresce. Nem toda tarefa considerada labuta tem todos esses atributos, mas quanto mais o trabalho corresponder a uma ou mais das seguintes descrições, maior a probabilidade de ser labuta:

Manual

Isso inclui trabalho como executar manualmente um script que automatiza alguma tarefa.

A execução de um script pode ser mais rápida do que a execução manual de cada etapa do script, mas o tempo prático que um ser humano gasta executando esse script (não o tempo decorrido) ainda é tempo de trabalho.

Repetitivo

Se você está realizando uma tarefa pela primeira vez, ou mesmo pela segunda vez, esse trabalho não é labuta. A labuta é um trabalho que você faz repetidamente. Se você está resolvendo um novo problema ou inventando uma nova solução, este trabalho não é labuta.

Automatizável

Se uma máquina pudesse realizar a tarefa tão bem quanto um ser humano, ou se a necessidade da tarefa pudesse ser eliminada, essa tarefa é cansativa. Se o julgamento humano é essencial para a tarefa, há uma boa chance de não ser labuta.³

Tático

A labuta é orientada por interrupções e reativa, em vez de orientada por estratégia e proativa. Lidar com alertas de pager é trabalhoso. Talvez nunca consigamos eliminar completamente esse tipo de trabalho, mas temos que trabalhar continuamente para minimizá-lo.

Sem valor duradouro

Se o seu serviço permanecer no mesmo estado após a conclusão de uma tarefa, provavelmente a tarefa foi trabalhosa. Se a tarefa produziu uma melhoria permanente em seu serviço, provavelmente não foi labuta, mesmo que alguma quantidade de trabalho pesado – como cavar código legado e configurações e corrigi-los – estivesse envolvido.

O(n) com crescimento do

serviço Se o trabalho envolvido em uma tarefa aumenta linearmente com o tamanho do serviço, volume de tráfego ou contagem de usuários, essa tarefa provavelmente é cansativa. Um serviço idealmente gerenciado e projetado pode crescer em pelo menos uma ordem de magnitude com zero trabalho adicional, além de alguns esforços pontuais para adicionar recursos.

3 Devemos ter cuidado ao dizer que uma tarefa “não é labuta porque precisa de julgamento humano”. Precisamos pensar cuidadosamente sobre se a natureza da tarefa requer intrinsecamente julgamento humano e não pode ser abordada por um melhor design. Por exemplo, pode-se construir (e alguns construíram) um serviço que alerta seus SREs várias vezes ao dia, onde cada alerta requer uma resposta complexa envolvendo muito julgamento humano. Tal serviço é mal projetado, com complexidade desnecessária. O sistema precisa ser simplificado e reconstruído para eliminar as condições de falha subjacentes ou lidar com essas condições automaticamente. Até que o redesenho e a reimplementação sejam concluídos e o serviço aprimorado seja implementado, o trabalho de aplicar o julgamento humano para responder a cada alerta é definitivamente árduo.

Por que menos trabalho é melhor

Nossa organização SRE tem uma meta anunciada de manter o trabalho operacional (ou seja, labuta) abaixo de 50% do tempo de cada SRE. Pelo menos 50% do tempo de cada SRE deve ser gasto em trabalho de projeto de engenharia que reduzirá o trabalho futuro ou adicionará recursos de serviço.

O desenvolvimento de recursos geralmente se concentra em melhorar a confiabilidade, o desempenho ou a utilização, o que geralmente reduz a labuta como um efeito de segunda ordem.

Compartilhamos essa meta de 50% porque a labuta tende a se expandir se não for controlada e pode ocupar rapidamente 100% do tempo de todos. O trabalho de reduzir a labuta e ampliar os serviços é a "Engenharia" na Engenharia de Confiabilidade do Site. O trabalho de engenharia é o que permite que a organização SRE escale de forma sublinear com o tamanho do serviço e gerencie serviços com mais eficiência do que uma equipe pura de desenvolvimento ou uma equipe pura de operações.

Além disso, quando contratamos novos SREs, prometemos a eles que o SRE não é uma organização típica de Ops, citando a regra dos 50% mencionada. Precisamos manter essa promessa, não permitindo que a organização SRE ou qualquer subequipe dentro dela se transforme em um Ops equipe.

Calculando o Trabalho

Se procurarmos limitar o tempo que um SRE gasta em labuta para 50%, como esse tempo é gasto?

Há um limite para a quantidade de trabalho que qualquer SRE tem que lidar se estiver de plantão. Um SRE típico tem uma semana de plantão primário e uma semana de plantão secundário em cada ciclo (para discussão de turnos de plantão primário versus secundário, consulte o [Capítulo 11](#)). Segue-se que, em uma rotação de 6 pessoas, pelo menos 2 de cada 6 semanas são dedicadas a turnos de plantão e tratamento de interrupção, o que significa que o limite inferior da labuta potencial é $2/6 = 33\%$ do tempo de um SRE. Em uma rotação de 8 pessoas, o limite inferior é $2/8 = 25\%$.

Consistente com esses dados, os SREs relatam que sua principal fonte de trabalho são as interrupções (ou seja, mensagens e e-mails não relacionados a serviços urgentes). A próxima fonte principal é a resposta de plantão (urgente), seguida por liberações e pushes. Embora nossos processos de liberação e push sejam geralmente tratados com uma quantidade razoável de automação, ainda há muito espaço para melhorias nessa área.

Pesquisas trimestrais dos SREs do Google mostram que o tempo médio gasto trabalhando é de cerca de 33%, então temos um desempenho muito melhor do que nossa meta geral de 50%. No entanto, a média não captura outliers: alguns SREs reivindicam 0% de labuta (projetos de desenvolvimento puro sem trabalho de plantão) e outros reivindicam 80% de labuta. Quando SREs individuais relatam trabalho excessivo, isso geralmente indica a necessidade de os gerentes distribuirem a carga de trabalho de maneira mais uniforme pela equipe e incentivar esses SREs a encontrar projetos de engenharia satisfatórios.

O que Quali es como Engenharia?

O trabalho de engenharia é novo e intrinsecamente requer julgamento humano. Produz uma melhoria permanente no seu serviço e é orientado por uma estratégia. É frequentemente criativo e inovador, adotando uma abordagem orientada ao design para resolver um problema – quanto mais generalizado, melhor. O trabalho de engenharia ajuda sua equipe ou a organização SRE a lidar com um serviço maior, ou mais serviços, com o mesmo nível de pessoal.

As atividades típicas de SRE se enquadram nas seguintes categorias aproximadas:

Engenharia de software

Envolve escrever ou modificar código, além de qualquer projeto associado e trabalho de documentação. Os exemplos incluem escrever scripts de automação, criar ferramentas ou estruturas, adicionar recursos de serviço para escalabilidade e confiabilidade ou modificar o código de infraestrutura para torná-lo mais robusto.

Engenharia de sistemas

Envolve configurar sistemas de produção, modificar configurações ou documentar sistemas de uma forma que produza melhorias duradouras a partir de um esforço único. Os exemplos incluem configuração e atualizações de monitoramento, configuração de balanceamento de carga, configuração de servidor, ajuste de parâmetros de SO e configuração de平衡ador de carga. A engenharia de sistemas também inclui consultoria em arquitetura, design e produção para equipes de desenvolvedores.

Labuta

Trabalho diretamente ligado à execução de um serviço repetitivo, manual etc.

Overhead

Trabalho administrativo não vinculado diretamente à execução de um serviço. Os exemplos incluem contratação, papelada de RH, reuniões de equipe/empresa, higiene da fila de bugs, trechos, revisões por pares e autoavaliações e cursos de treinamento.

Cada SRE precisa gastar pelo menos 50% de seu tempo em trabalho de engenharia, quando em média alguns trimestres ou um ano. A labuta tende a ser pontiaguda, portanto, 50% do tempo gasto em engenharia pode não ser realista para algumas equipes de SRE, e elas podem ficar abaixo dessa meta em alguns trimestres. Mas se a fração de tempo gasto em projetos estiver significativamente abaixo de 50% no longo prazo, a equipe afetada precisa dar um passo atrás e descobrir o que está errado.

A labuta é sempre ruim?

A labuta não deixa todos infelizes o tempo todo, especialmente em pequenas quantidades. Tarefas previsíveis e repetitivas podem ser bastante calmantes. Produzem uma sensação de realização e vitórias rápidas. Eles podem ser atividades de baixo risco e baixo estresse. Algumas pessoas gravitam em torno de tarefas que envolvem labuta e podem até gostar desse tipo de trabalho.

A labuta nem sempre é invariavelmente é ruim, e todos precisam estar absolutamente claros de que uma certa quantidade de labuta é inevitável na função de SRE e, de fato, em quase todas as funções de engenharia. É bom em pequenas doses, e se você está feliz com essas pequenas doses, o trabalho não é um problema. A labuta se torna tóxica quando experimentada em grandes quantidades. Se você está sobrecarregado com muito trabalho, deve ficar muito preocupado e reclamar em voz alta.

Entre as muitas razões pelas quais trabalhar demais é ruim, considere o seguinte:

Estagnação na carreira

Seu progresso na carreira vai desacelerar ou parar se você gastar muito pouco tempo em projetos. O Google recompensa o trabalho sujo quando é inevitável e tem um grande impacto positivo, mas você não pode fazer do grunge uma carreira.

Baixa moral

As pessoas têm limites diferentes para quanta labuta podem tolerar, mas todos têm um limite. Muito trabalho leva ao esgotamento, ao tédio e ao descontentamento.

Além disso, gastar muito tempo trabalhando em detrimento do tempo gasto em engenharia prejudica uma organização de SRE das seguintes maneiras:

Cria confusão

Trabalhamos arduamente para garantir que todos que trabalham na organização SRE ou com ela entendam que somos uma organização de engenharia. Indivíduos ou equipes dentro do SRE que se envolvem em muito trabalho minam a clareza dessa comunicação e confundem as pessoas sobre nosso papel.

Retarda o

progresso O trabalho excessivo torna a equipe menos produtiva. A velocidade dos recursos de um produto diminuirá se a equipe de SRE estiver muito ocupada com trabalho manual e combate a incêndios para lançar novos recursos imediatamente.

Estabelece

precedente Se você estiver muito disposto a trabalhar duro, seus colegas Dev terão incentivos para sobrecarregar você com ainda mais trabalho, às vezes transferindo tarefas operacionais que deveriam ser executadas por Devs para SRE. Outras equipes também podem começar a esperar que os SREs assumam esse trabalho, o que é ruim por razões óbvias.

Promove o desgaste

Mesmo que você não esteja pessoalmente insatisfeito com a labuta, seus atuais ou futuros companheiros de equipe podem gostar muito menos. Se você incluir muito trabalho nos procedimentos de sua equipe, você motiva os melhores engenheiros da equipe a começarem a procurar em outro lugar um trabalho mais recompensador.

Causa quebra de fé Novas

contratações ou transferências que ingressaram na SRE com a promessa de trabalhar no projeto se sentirão enganadas, o que é ruim para o moral.

Conclusão

Se todos nos comprometermos a eliminar um pouco de labuta a cada semana com uma boa engenharia, limparemos constantemente nossos serviços e poderemos mudar nossos esforços coletivos para engenharia de escala, arquitetando a próxima geração de serviços e construindo SRE cruzado cadeias de ferramentas. Vamos inventar mais e trabalhar menos.

CAPÍTULO 6

Monitoramento de Sistemas Distribuídos

**Escrito por Rob Ewaschuk
Editado por Betsy Beyer**

As equipes de SRE do Google têm alguns princípios básicos e práticas recomendadas para criar sistemas bem-sucedidos de monitoramento e alerta. Este capítulo oferece diretrizes sobre quais problemas devem interromper um humano por meio de uma página e como lidar com problemas que não são sérios o suficiente para acionar uma página.

Definições

Não há vocabulário uniformemente compartilhado para discutir todos os tópicos relacionados ao monitoramento. Mesmo dentro do Google, o uso dos termos a seguir varia, mas as interpretações mais comuns estão listadas aqui.

Monitoramento

Coleta, processamento, agregação e exibição de dados quantitativos em tempo real sobre um sistema, como contagens e tipos de consultas, contagens e tipos de erros, tempos de processamento e tempos de vida do servidor.

Monitoramento de caixa branca

Monitoramento baseado em métricas expostas pelos componentes internos do sistema, incluindo logs, interfaces como a Java Virtual Machine Profiling Interface ou um manipulador HTTP que emite estatísticas internas.

Monitoramento de caixa preta

Testando o comportamento visível externamente como um usuário o veria.

Painel

Um aplicativo (geralmente baseado na Web) que fornece uma visão resumida das principais métricas de um serviço. Um painel pode ter filtros, seletores e assim por diante, mas é pré-construído para

expor as métricas mais importantes para seus usuários. O painel também pode exibir informações da equipe, como comprimento da fila de tickets, uma lista de bugs de alta prioridade, o engenheiro de plantão atual para uma determinada área de responsabilidade ou pushes recentes.

Alerta

Uma notificação destinada a ser lida por um humano e que é enviada a um sistema, como uma fila de bugs ou tíquetes, um alias de e-mail ou um pager. Respectivamente, esses alertas são classificados como tickets, alertas por e-mail, 1 e páginas.¹

Causa raiz

Um defeito em um software ou sistema humano que, se reparado, infunde a confiança de que esse evento não acontecerá novamente da mesma maneira. Um determinado incidente pode ter várias causas-raiz: por exemplo, talvez tenha sido causado por uma combinação de automação de processo insuficiente, software que travou na entrada falsa e teste insuficiente do script usado para gerar a configuração. Cada um desses fatores pode estar sozinho como uma causa raiz e cada um deve ser reparado.

Nó e máquina Usados

alternadamente para indicar uma única instância de um kernel em execução em um servidor físico, máquina virtual ou contêiner. Pode haver vários serviços que valem a pena monitorar em uma única máquina. Os serviços podem ser:

- Relacionados entre si: por exemplo, um servidor de cache e um servidor web
- Hardware de compartilhamento de serviços não relacionados: por exemplo, um repositório de código e um master para um sistema de configuração como [Puppet](#) ou [Cozinheiro](#)

Empurre

Qualquer alteração no software em execução de um serviço ou em sua configuração.

Por que Monitorar?

Há muitas razões para monitorar um sistema, incluindo:

Análise de tendências de longo prazo

Qual é o tamanho do meu banco de dados e quanto rápido ele está crescendo? Com que rapidez minha contagem diária de usuários ativos está crescendo?

Comparação ao longo do tempo ou grupos de experiência

As consultas são mais rápidas com Acme Bucket of Bytes 2.72 versus Ajax DB 3.14? Quão melhor é minha taxa de acertos do memcache com um nó extra? Meu site está mais lento do que na semana passada?

¹ Às vezes conhecido como "spam de alerta", pois raramente são lidos ou tratados.

Alerta

Algo está quebrado e alguém precisa consertar agora mesmo! Ou, algo pode quebrar em breve, então alguém deve olhar em breve.

Construindo painéis

Os painéis devem responder a perguntas básicas sobre seu serviço e normalmente incluem alguma forma dos quatro sinais dourados (discutidos em “[Os Quatro Sinais Dourados](#)” na [página 60](#)).

Realização de análise retrospectiva ad hoc (ou seja, depuração)

Nossa latência disparou; o que mais aconteceu na mesma época?

O monitoramento do sistema também é útil para fornecer dados brutos para análises de negócios e para facilitar a análise de violações de segurança. Como este livro se concentra nos domínios de engenharia nos quais a SRE possui experiência específica, não discutiremos essas aplicações de monitoramento aqui.

Monitoramento e alerta permitem que um sistema nos diga quando está quebrado, ou talvez nos diga o que está prestes a quebrar. Quando o sistema não é capaz de se corrigir automaticamente, queremos que um humano investigue o alerta, determine se há um problema real em mãos, mitigue o problema e determine a causa raiz do problema. A menos que você esteja realizando uma auditoria de segurança em componentes de escopo muito restrito de um sistema, você nunca deve acionar um alerta simplesmente porque “algo parece um pouco estranho”.

Paginar um humano é um uso bastante caro do tempo de um funcionário. Se um funcionário estiver no trabalho, uma página interrompe seu fluxo de trabalho. Se o funcionário estiver em casa, uma página interrompe seu tempo pessoal e talvez até seu sono. Quando as páginas ocorrem com muita frequência, os funcionários adivinham, ignoram ou até ignoram os alertas recebidos, às vezes até mesmo ignorando uma página “real” que é mascarada pelo ruído. As interrupções podem ser prolongadas porque outros ruídos interferem no diagnóstico e na correção rápidos. Sistemas de alerta eficazes têm bom sinal e ruído muito baixo.

Definindo expectativas razoáveis para monitoramento

O monitoramento de um aplicativo complexo é um esforço de engenharia significativo por si só. Mesmo com uma infraestrutura substancial existente para instrumentação, coleta, exibição e alerta, uma equipe de SRE do Google com 10 a 12 membros normalmente tem um ou, às vezes, dois membros cuja principal atribuição é criar e manter sistemas de monitoramento para seu serviço. Esse número diminuiu ao longo do tempo à medida que generalizamos e centralizamos a infraestrutura de monitoramento comum, mas cada equipe de SRE normalmente tem pelo menos uma “pessoa de monitoramento”. (Dito isto, embora possa ser divertido ter acesso a painéis de gráficos de tráfego e similares, as equipes de SRE evitam cuidadosamente qualquer situação que exija que alguém “olhe para uma tela para observar problemas”.)

Em geral, o Google tende a sistemas de monitoramento mais simples e rápidos, com melhores ferramentas para análise post hoc. Evitamos sistemas “mágicos” que tentam aprender limites ou detectar automaticamente a causalidade. As regras que detectam alterações inesperadas nas taxas de solicitação do usuário final são um contra-exemplo; embora essas regras ainda sejam mantidas o mais simples possível, elas fornecem uma detecção muito rápida de uma anomalia muito simples, específica e grave. Outros usos de dados de monitoramento, como planejamento de capacidade e previsão de tráfego, podem tolerar mais fragilidade e, portanto, mais complexidade. Experimentos observacionais conduzidos em um horizonte de tempo muito longo (meses ou anos) com uma baixa taxa de amostragem (horas ou dias) também podem tolerar mais fragilidade porque amostras perdidas ocasionais não escondem uma tendência de longa duração.

O Google SRE obteve sucesso limitado com hierarquias de dependência complexas. Raramente usamos regras como: “Se eu souber que o banco de dados está lento, alerte para um banco de dados lento; caso contrário, alerta para o site geralmente lento.” Regras dependentes de dependência geralmente pertencem a partes muito estáveis de nosso sistema, como nosso sistema para drenar o tráfego de usuários de um datacenter. Por exemplo, “Se um datacenter estiver esgotado, não me alerte sobre sua latência” é uma regra comum de alerta de datacenter. Poucas equipes no Google mantêm hierarquias de dependência complexas porque nossa infraestrutura tem uma taxa constante de refatoração contínua.

Algumas das ideias descritas neste capítulo ainda são aspiracionais: sempre há espaço para passar mais rapidamente do sintoma para a(s) causa(s) raiz(es), especialmente em sistemas em constante mudança. Portanto, embora este capítulo estabeleça algumas metas para sistemas de monitoramento e algumas maneiras de atingir essas metas, é importante que os sistemas de monitoramento - especialmente o caminho crítico desde o início de um problema de produção, passando de uma página para um humano, por meio de triagem básica e depuração—ser simples e compreensível por todos da equipe.

Da mesma forma, para manter o ruído baixo e o sinal alto, os elementos do seu sistema de monitoramento que direcionam para um pager precisam ser muito simples e robustos. As regras que geram alertas para humanos devem ser simples de entender e representar uma falha clara.

Sintomas versus Causas

Seu sistema de monitoramento deve abordar duas questões: o que está quebrado e por quê?

O “que está quebrado” indica o sintoma; o “porquê” indica uma causa (possivelmente intermediária).

A Tabela 6-1 lista alguns sintomas hipotéticos e causas.

Tabela 6-1. Exemplos de sintomas e causas

Sintoma	Causa
Estou servindo HTTP 500s ou 404s	Os servidores de banco de dados estão recusando conexões
Minhas respostas são lentas	As CPUs estão sobrecarregadas por um bogosort ou um cabo Ethernet é cravado sob um rack, visível como perda parcial de pacotes
Usuários na Antártida não estão recebendo GIFs animados de gatos	Sua rede de distribuição de conteúdo odeia cientistas e felinos e, portanto, colocou na lista negra alguns IPs de clientes
O conteúdo privado é legível em todo o mundo	Um novo push de software fez com que as ACLs fossem esquecidas e permitisse todas as solicitações

“O que” versus “por que” é uma das distinções mais importantes ao escrever um bom monitoramento com sinal máximo e ruído mínimo.

Caixa preta versus caixa branca

Combinamos o uso intenso de monitoramento de caixa branca com usos modestos, mas críticos, de monitoramento de caixa preta. A maneira mais simples de pensar sobre monitoramento de caixa preta versus monitoramento de caixa branca é que o monitoramento de caixa preta é orientado a sintomas e representa problemas ativos – não previstos: “O sistema não está funcionando corretamente agora”. O monitoramento de caixa branca depende da capacidade de inspecionar as entranhas do sistema, como logs ou endpoints HTTP, com instrumentação. O monitoramento de caixa branca, portanto, permite a detecção de problemas iminentes, falhas mascaradas por tentativas e assim por diante.

Observe que em um sistema multicamadas, o sintoma de uma pessoa é a causa de outra. Por exemplo, suponha que o desempenho de um banco de dados seja lento. As leituras lentas do banco de dados são um sintoma para o SRE do banco de dados que as detecta. No entanto, para o SRE front-end que observa um site lento, as mesmas leituras lentas do banco de dados são uma causa. Portanto, o monitoramento de caixa branca às vezes é orientado a sintomas e, às vezes, à causa, dependendo de quanto informativa é sua caixa branca.

Ao coletar telemetria para depuração, o monitoramento de caixa branca é essencial. Se os servidores da Web parecem lentos em solicitações de banco de dados pesados, você precisa saber o quanto rápido o servidor da Web percebe que o banco de dados é e o quanto rápido o banco de dados acredita ser. Caso contrário, você não pode distinguir um servidor de banco de dados realmente lento de um problema de rede entre seu servidor web e seu banco de dados.

Para a paginação, o monitoramento de caixa preta tem o principal benefício de forçar a disciplina a incomodar um humano apenas quando um problema já está em andamento e contribui para sintomas reais. Por outro lado, para problemas que ainda não ocorrem, mas iminentes, o monitoramento caixa-preta é bastante inútil.

Os quatro sinais dourados

Os quatro sinais de ouro do monitoramento são latência, tráfego, erros e saturação. Se você puder medir apenas quatro métricas do seu sistema voltado para o usuário, concentre-se nessas quatro.

Latência

O tempo que leva para atender uma solicitação. É importante distinguir entre a latência de solicitações bem-sucedidas e a latência de solicitações com falha. Por exemplo, um erro HTTP 500 acionado devido à perda de conexão com um banco de dados ou outro back-end crítico pode ser atendido muito rapidamente; no entanto, como um erro HTTP 500 indica uma solicitação com falha, fatorar 500s em sua latência geral pode resultar em cálculos enganosos. Por outro lado, um erro lento é ainda pior do que um erro rápido!

Portanto, é importante rastrear a latência do erro, em vez de apenas filtrar erros.

Tráfego

Uma medida de quanta demanda está sendo colocada em seu sistema, medida em uma métrica específica do sistema de alto nível. Para um serviço da Web, essa medida geralmente é de solicitações HTTP por segundo, talvez divididas pela natureza das solicitações (por exemplo, conteúdo estático versus conteúdo dinâmico). Para um sistema de streaming de áudio, essa medição pode se concentrar na taxa de E/S da rede ou sessões simultâneas. Para um sistema de armazenamento de valor-chave, essa medida pode ser de transações e recuperações por segundo.

Erros

A taxa de solicitações que falham, seja explicitamente (por exemplo, HTTP 500s), implicitamente (por exemplo, uma resposta de sucesso HTTP 200, mas associada ao conteúdo errado) ou por política (por exemplo, "Se você se comprometeu com um segundos tempos de resposta, qualquer solicitação acima de um segundo é um erro"). Onde os códigos de resposta do protocolo são insuficientes para expressar todas as condições de falha, protocolos secundários (internos) podem ser necessários para rastrear os modos de falha parcial. O monitoramento desses casos pode ser drasticamente diferente: capturar HTTP 500s em seu平衡ador de carga pode fazer um trabalho decente ao capturar todas as solicitações com falha completa, enquanto apenas testes de sistema de ponta a ponta podem detectar que você está servindo o conteúdo errado.

Saturação

Quão "cheio" é o seu serviço. Uma medida da fração do seu sistema, enfatizando os recursos que são mais restritos (por exemplo, em um sistema com restrição de memória, mostre a memória; em um sistema com restrição de E/S, mostre E/S). Observe que muitos sistemas degradam em desempenho antes de atingir 100% de utilização, portanto, ter uma meta de utilização é essencial.

Em sistemas complexos, a saturação pode ser complementada com uma medição de carga de nível superior: seu serviço pode lidar adequadamente com o dobro do tráfego, lidar com apenas 10% mais tráfego ou lidar com ainda menos tráfego do que recebe atualmente? Por muito

serviços simples que não possuem parâmetros que alteram a complexidade da solicitação (por exemplo, "Me dê um nonce" ou "Preciso de um inteiro monotônico globalmente exclusivo") que raramente alteram a configuração, um valor estático de um teste de carga pode ser adequado. Conforme discutido no parágrafo anterior, no entanto, a maioria dos serviços precisa usar sinais indiretos, como utilização da CPU ou largura de banda da rede, que têm um limite superior conhecido. Os aumentos de latência costumam ser um indicador importante de saturação. Medir o tempo de resposta do percentil 99 em uma pequena janela (por exemplo, um minuto) pode fornecer um sinal muito precoce de saturação.

Por fim, a saturação também se preocupa com as previsões de saturação iminente, como "Parece que seu banco de dados encherá seu disco rígido em 4 horas".

Se você medir todos os quatro sinais dourados e chamar um humano quando um sinal for problemático (ou, no caso de saturação, quase problemático), seu serviço será pelo menos decentemente coberto pelo monitoramento.

Preocupando-se com sua cauda (ou, Instrumentação e Atuação)

Ao construir um sistema de monitoramento do zero, é tentador projetar um sistema com base na média de alguma quantidade: a latência média, o uso médio de CPU de seus nós ou a plenitude média de seus bancos de dados. O perigo apresentado pelos dois últimos casos é óbvio: CPUs e bancos de dados podem ser facilmente utilizados de forma muito desequilibrada. O mesmo vale para a latência. Se você executar um serviço da web com uma latência média de 100 ms a 1.000 solicitações por segundo, 1% das solicitações pode levar facilmente 5 segundos . um backend pode facilmente se tornar a resposta mediana do seu frontend.

A maneira mais simples de diferenciar entre uma média lenta e uma "cauda" muito lenta de solicitações é coletar contagens de solicitações agrupadas por latências (adequadas para renderizar um histograma), em vez de latências reais: quantas solicitações serviram que levaram entre 0 ms e 10 ms, entre 10 ms e 30 ms, entre 30 ms e 100 ms, entre 100 ms e 300 ms, e assim por diante?

Distribuir os limites do histograma de forma aproximadamente exponencial (neste caso por fatores de aproximadamente 3) geralmente é uma maneira fácil de visualizar a distribuição de suas solicitações.

2 Se 1% de suas solicitações for 10x a média, isso significa que o restante de suas solicitações é cerca de duas vezes mais rápido que a média. Mas se você não está medindo sua distribuição, a ideia de que a maioria de seus pedidos está perto da média é apenas um pensamento esperançoso.

Escolhendo uma resolução apropriada para medições

Diferentes aspectos de um sistema devem ser medidos com diferentes níveis de granularidade.

Por exemplo:

- Observar a carga da CPU ao longo de um minuto não revelará nem mesmo picos de longa duração que geram latências de cauda altas. • Por outro lado, para um serviço da Web que visa não mais de 9 horas de inatividade agregada por ano (99,9% de tempo de atividade anual), a sondagem de um status 200 (sucesso) mais de uma ou duas vezes por minuto provavelmente é desnecessariamente frequente.
- Da mesma forma, verificar a plenitude do disco rígido para um serviço que visa 99,9% de disponibilidade mais de uma vez a cada 1–2 minutos provavelmente é desnecessário.

Tome cuidado em como você estrutura a granularidade de suas medições. A coleta de medições por segundo da carga da CPU pode gerar dados interessantes, mas essas medições frequentes podem ser muito caras para coletar, armazenar e analisar. Se sua meta de monitoramento exigir alta resolução, mas não exigir latência extremamente baixa, você poderá reduzir esses custos realizando amostragem interna no servidor e, em seguida, configurando um sistema externo para coletar e agragar essa distribuição ao longo do tempo ou entre servidores.

Você pode:

1. Registre a utilização atual da CPU a cada segundo.
2. Usando buckets de granularidade de 5%, incremente a utilização apropriada da CPU balde a cada segundo.
3. Agregue esses valores a cada minuto.

Essa estratégia permite observar pontos de acesso de CPU breves sem incorrer em custos muito altos devido à coleta e retenção.

Tão simples quanto possível, não mais simples

Empilhar todos esses requisitos uns sobre os outros pode resultar em um sistema de monitoramento muito complexo – seu sistema pode acabar com os seguintes níveis de complexidade:

- Alertas sobre diferentes limites de latência, em diferentes percentis, em todos os tipos de métricas diferentes
- Código extra para detectar e expor possíveis causas • Painéis associados para cada uma dessas possíveis causas

As fontes de complexidade potencial são intermináveis. Como todos os sistemas de software, o monitoramento pode se tornar tão complexo que se torna frágil, complicado de mudar e um fardo de manutenção.

Portanto, projete seu sistema de monitoramento com foco na simplicidade. Ao escolher o que monitorar, tenha em mente as seguintes diretrizes:

- As regras que detectam incidentes reais com mais frequência devem ser tão simples, previsíveis e confiáveis quanto possível.
- A coleta de dados, agregação e configuração de alertas que raramente são exercidas (por exemplo, menos de uma vez por trimestre para algumas equipes de SRE) devem ser removidas.
- Os sinais que são coletados, mas não expostos em nenhum painel pré-configurado nem usados por qualquer alerta, são candidatos à remoção.

Na experiência do Google, a coleta básica e a agregação de métricas, combinadas com alertas e painéis, funcionaram bem como um sistema relativamente autônomo. (Na verdade, o sistema de monitoramento do Google é dividido em vários binários, mas normalmente as pessoas aprendem sobre todos os aspectos desses binários). depuração, rastreamento de detalhes sobre exceções ou falhas, teste de carga, coleta e análise de logs ou inspeção de tráfego. Embora a maioria desses assuntos compartilhe pontos em comum com o monitoramento básico, misturar muitos resultados em sistemas excessivamente complexos e frágeis. Como em muitos outros aspectos da engenharia de software, manter sistemas distintos com pontos de integração claros, simples e fracamente acoplados é uma estratégia melhor (por exemplo, usar APIs da Web para extrair dados de resumo em um formato que possa permanecer constante por um longo período). de tempo).

Unindo esses princípios

Os princípios discutidos neste capítulo podem ser vinculados a uma filosofia de monitoramento e alerta amplamente endossada e seguida pelas equipes de SRE do Google. Embora essa filosofia de monitoramento seja um pouco ambiciosa, é um bom ponto de partida para escrever ou revisar um novo alerta e pode ajudar sua organização a fazer as perguntas certas, independentemente do tamanho de sua organização ou da complexidade de seu serviço ou sistema.

Ao criar regras para monitoramento e alerta, fazer as seguintes perguntas pode ajudar a evitar falsos positivos e esgotamento do pager:³

- Esta regra detecta uma condição não detectada que é urgente, açãoável e ativa ou iminentemente visível ao usuário?⁴

³ Consulte Aplicação de técnicas de gerenciamento de alarme cardíaco ao seu [plantão](#) [Hol14] para obter um exemplo de fadiga de alerta em outro contexto.

⁴ Situações de redundância zero ($N + 0$) contam como iminentes, assim como partes “quase cheias” do seu serviço! Para obter mais detalhes sobre o conceito de redundância, consulte https://en.wikipedia.org/wiki/N%2B1_redundancy.

- Serei capaz de ignorar esse alerta, sabendo que é benigno? Quando e por que poderei ignorar esse alerta e como evitar esse cenário? • Este alerta indica definitivamente que os usuários estão sendo afetados negativamente? Existem casos detectáveis em que os usuários não estão sendo impactados negativamente, como tráfego drenado ou implantações de teste, que devem ser filtrados?
- Posso agir em resposta a este alerta? Essa ação é urgente ou pode esperar até de manhã? A ação poderia ser automatizada com segurança? Essa ação será uma correção de longo prazo ou apenas uma solução alternativa de curto prazo?
- Outras pessoas estão sendo chamadas para este problema, tornando desnecessária pelo menos uma das páginas?

Essas perguntas refletem uma filosofia fundamental em páginas e pagers:

- Cada vez que o pager toca, devo ser capaz de reagir com um senso de urgência. Só consigo reagir com um senso de urgência algumas vezes por dia antes de ficar fatigado.
- Cada página deve ser acionável. • Cada resposta de página deve exigir inteligência. Se uma página apenas merece um robótico resposta, não deve ser uma página.
- As páginas devem ser sobre um problema novo ou um evento que não tenha sido visto antes.

Tal perspectiva dissipava certas distinções: se uma página satisfaz os quatro marcadores anteriores, é irrelevante se a página é acionada por monitoramento de caixa branca ou caixa preta. Essa perspectiva também amplia certas distinções: é melhor se esforçar muito mais para detectar os sintomas do que as causas; quando se trata de causas, preocupe-se apenas com causas muito definidas, muito iminentes.

Monitoramento de Longo Prazo

Em sistemas de produção modernos, os sistemas de monitoramento rastreiam um sistema em constante evolução com mudanças na arquitetura de software, características de carga e metas de desempenho. Um alerta que atualmente é excepcionalmente raro e difícil de automatizar pode se tornar frequente, talvez até merecendo um script hackeado para resolvê-lo. Neste ponto, alguém deve encontrar e eliminar as causas-raiz do problema; se tal resolução não for possível, a resposta de alerta merece ser totalmente automatizada.

É importante que as decisões sobre o monitoramento sejam tomadas com objetivos de longo prazo em mente. Cada página que acontece hoje distrai um ser humano de melhorar o sistema para amanhã, então muitas vezes há um caso para um impacto de curto prazo na disponibilidade ou desempenho para melhorar as perspectivas de longo prazo para o sistema. Vamos dar uma olhada em dois estudos de caso que ilustram essa troca.

Bigtable SRE: uma história de alertas excessivos

A infraestrutura interna do Google geralmente é oferecida e avaliada em relação a um objetivo de nível de serviço (SLO; consulte o [Capítulo 4](#)). Muitos anos atrás, o SLO do serviço Bigtable era baseado no desempenho médio de um cliente sintético bem comportado. Devido a problemas no Bigtable e nas camadas inferiores da pilha de armazenamento, o desempenho médio foi impulsionado por uma cauda “grande”: os 5% piores das solicitações geralmente eram significativamente mais lentos do que o restante.

Alertas de e-mail eram acionados quando o SLO se aproximava e alertas de paginação eram acionados quando o SLO era excedido. Ambos os tipos de alertas estavam disparando volumosamente, consumindo quantidades inaceitáveis de tempo de engenharia: a equipe gastou uma quantidade significativa de tempo triando os alertas para encontrar os poucos que eram realmente acionáveis, e muitas vezes perdemos os problemas que realmente afetavam os usuários, porque tão poucos deles fez. Muitas das páginas não eram urgentes, devido a problemas bem compreendidos na infraestrutura, e tinham respostas mecânicas ou não recebiam resposta.

Para remediar a situação, a equipe usou uma abordagem em três frentes: ao fazer grandes esforços para melhorar o desempenho do Bigtable, também reduzimos temporariamente nossa meta de SLO, usando a latência de solicitação do 75º percentil. Também desabilitamos os alertas de e-mail, pois eram tantos que perder tempo diagnosticando-os era inviável.

Essa estratégia nos deu espaço suficiente para realmente corrigir os problemas de longo prazo no Bigtable e nas camadas inferiores da pilha de armazenamento, em vez de corrigir constantemente os problemas táticos. Engenheiros de plantão podiam realmente realizar o trabalho quando não estavam sendo mantidos por páginas o tempo todo. Por fim, recuar temporariamente em nossos alertas nos permitiu progredir mais rapidamente em direção a um serviço melhor.

Gmail: respostas previsíveis e programáveis de humanos

Nos primórdios do Gmail, o serviço foi construído em um sistema de gerenciamento de processo distribuído adaptado chamado Workqueue, que foi originalmente criado para processamento em lote de partes do índice de pesquisa. O Workqueue foi “adaptado” a processos de longa duração e posteriormente aplicado ao Gmail, mas certos bugs na base de código relativamente opaca do agendador provaram ser difíceis de superar.

Naquela época, o monitoramento do Gmail era estruturado de tal forma que alertas disparavam quando tarefas individuais eram “desagendadas” pelo Workqueue. Essa configuração não era ideal porque, mesmo naquela época, o Gmail tinha muitos, muitos milhares de tarefas, cada tarefa representando uma fração de um por cento de nossos usuários. Nós nos preocupamos muito em fornecer uma boa experiência de usuário para os usuários do Gmail, mas essa configuração de alerta era insustentável.

Para resolver esse problema, o Gmail SRE criou uma ferramenta que ajudou a “cutucar” o agendador da maneira certa para minimizar o impacto para os usuários. A equipe teve várias discussões sobre se deveríamos ou não automatizar todo o loop, desde a detecção do problema até o reagendamento, até que uma solução melhor a longo prazo fosse alcançada, mas alguns temiam que esse tipo de solução atrasasse uma correção real.

Esse tipo de tensão é comum dentro de uma equipe e geralmente reflete uma desconfiança subjacente na autodisciplina da equipe: enquanto alguns membros da equipe desejam implementar um “hack” para dar tempo para uma correção adequada, outros se preocupam que um hack ser esquecido ou que a correção adequada será despriorizada indefinidamente. Essa preocupação é crível, pois é fácil construir camadas de dívida técnica insustentável corrigindo problemas em vez de fazer correções reais. Gerentes e líderes técnicos desempenham um papel fundamental na implementação de correções verdadeiras e de longo prazo, apoiando e priorizando correções de longo prazo potencialmente demoradas, mesmo quando a “dor” inicial da paginação diminui.

Páginas com respostas automáticas e algorítmicas devem ser uma bandeira vermelha. A falta de vontade por parte de sua equipe para automatizar essas páginas implica que a equipe não tem confiança de que pode limpar sua dívida técnica. Este é um grande problema que vale a pena escalar.

O longo prazo Um

tema comum conecta os exemplos anteriores do Bitable e do Gmail: uma tensão entre disponibilidade de curto e longo prazo. Muitas vezes, a pura força de esforço pode ajudar um sistema frágil a alcançar alta disponibilidade, mas esse caminho geralmente é de curta duração e repleto de esgotamento e dependência de um pequeno número de membros heróicos da equipe.

Fazer uma diminuição controlada e de curto prazo na disponibilidade é muitas vezes uma troca dolorosa, mas estratégica, para a estabilidade de longo prazo do sistema. É importante não pensar em cada página como um evento isolado, mas considerar se o nível geral de paginação leva a um sistema saudável e adequadamente disponível, com uma equipe saudável e viável e uma perspectiva de longo prazo. Revisamos estatísticas sobre frequência de páginas (geralmente expressas como incidentes por turno, em que um incidente pode ser composto de algumas páginas relacionadas) em relatórios trimestrais com a gerência, garantindo que os tomadores de decisão sejam mantidos atualizados sobre a carga do pager e a integridade geral de suas equipes.

Conclusão

Um pipeline de alerta e monitoramento saudável é simples e fácil de entender. Ele se concentra principalmente em sintomas para paginação, reservando heurísticas orientadas a causa para servir como auxílio para depurar problemas. Monitorar os sintomas é mais fácil quanto mais você estiver monitorando na pilha, embora o monitoramento da saturação e do desempenho de subsistemas, como bancos de dados, muitas vezes deva ser executado diretamente no próprio subsistema.

Os alertas de e-mail são de valor muito limitado e tendem a ser facilmente invadidos por ruídos; em vez disso, você deve preferir um painel que monitore todos os problemas subcríticos em andamento para o tipo de informação que normalmente termina em alertas de e-mail. Um painel também pode ser emparelhado com um log, para analisar correlações históricas.

A longo prazo, alcançar uma rotação e um produto de plantão bem-sucedidos inclui optar por alertar sobre sintomas ou problemas reais iminentes, adaptar suas metas a metas realmente alcançáveis e garantir que seu monitoramento ofereça suporte a diagnósticos rápidos.

CAPÍTULO 7

A evolução da automação no Google

Escrito por Niall Murphy com John Looney e Michael Kacirek

Editado por Betsy Beyer

Além da arte negra, há apenas automação e mecanização.

—Federico García Lorca (1898–1936), poeta e dramaturgo espanhol

Para o SRE, a automação é um multiplicador de força, não uma panacéia. É claro que apenas multiplicar a força não altera naturalmente a precisão de onde essa força é aplicada: fazer automação sem pensar pode criar tantos problemas quanto resolver. Portanto, embora acreditemos que a automação baseada em software seja superior à operação manual na maioria das circunstâncias, melhor do que qualquer uma das opções é um projeto de sistema de nível superior que não exija nenhuma delas – um sistema autônomo. Ou, dito de outra forma, o valor da automação vem tanto do que ela faz quanto de sua aplicação criteriosa. Discutiremos o valor da automação e como nossa atitude evoluiu ao longo do tempo.

O valor da automação

Qual é exatamente o valor da automação?¹

Consistência

Embora a escala seja uma motivação óbvia para a automação, existem muitas outras razões para usá-la. Tomemos o exemplo dos sistemas de computação universitários, onde muitos engenheiros de sistemas iniciaram suas carreiras. Os administradores de sistemas com essa formação geralmente eram encarregados de executar uma coleção de máquinas ou alguns

¹ Para os leitores que já sentem que entendem precisamente o valor da automação, pule para "O valor para o Google SRE" na página 70. No entanto, observe que nossa descrição contém algumas nuances que podem ser úteis para se ter em mente ao ler o restante o capítulo.

software, e estavam acostumados a executar manualmente várias ações no cumprimento dessa função. Um exemplo comum é a criação de contas de usuário; outros incluem tarefas puramente operacionais, como garantir que os backups aconteçam, gerenciar o failover do servidor e pequenas manipulações de dados, como alterar o resolv.conf dos servidores DNS upstream, dados da zona do servidor DNS e atividades semelhantes. Em última análise, no entanto, essa prevalência de tarefas manuais é insatisfatória tanto para as organizações quanto para as pessoas que mantêm sistemas dessa maneira. Para começar, qualquer ação realizada por um humano ou humanos centenas de vezes não será executada da mesma maneira todas as vezes: mesmo com a melhor vontade do mundo, muito poucos de nós serão tão consistentes quanto uma máquina. Essa inevitável falta de consistência leva a erros, descuidos, problemas com a qualidade dos dados e, sim, problemas de confiabilidade. Nesse domínio – a execução de procedimentos conhecidos e bem definidos – o valor da consistência é, em muitos aspectos, o principal valor da automação.

Uma plataforma

A automação não fornece apenas consistência. Projetados e executados corretamente, os sistemas automáticos também fornecem uma plataforma que pode ser estendida, aplicada a mais sistemas ou talvez até desmembrada com fins lucrativos.² (A alternativa, sem automação, não é econômica nem extensível: é um imposto cobrado sobre o funcionamento de um sistema.)

Uma plataforma também centraliza os erros. Em outras palavras, um bug corrigido no código será corrigido lá uma vez e para sempre, ao contrário de um conjunto suficientemente grande de humanos realizando o mesmo procedimento, conforme discutido anteriormente. Uma plataforma pode ser estendida para executar tarefas adicionais com mais facilidade do que os humanos podem ser instruídos a realizá-las (ou às vezes até perceber que elas precisam ser feitas). Dependendo da natureza da tarefa, ela pode ser executada continuamente ou com muito mais frequência do que os humanos poderiam realizar adequadamente a tarefa, ou em momentos inconvenientes para os humanos. Além disso, uma plataforma pode exportar métricas sobre seu desempenho ou permitir que você descubra detalhes sobre seu processo que você não conhecia anteriormente, porque esses detalhes são mais facilmente mensuráveis no contexto de uma plataforma.

Reparos mais

rápidos Há um benefício adicional para sistemas em que a automação é usada para resolver falhas comuns em um sistema (uma situação frequente para automação criada por SRE). Se a automação for executada regularmente e com sucesso suficiente, o resultado é um tempo médio de reparo (MTTR) reduzido para essas falhas comuns. Você pode então gastar seu tempo em outras tarefas, alcançando assim uma maior velocidade do desenvolvedor porque você não precisa gastar tempo prevenindo um problema ou (mais comumente) limpando depois dele.

² A experiência adquirida na construção dessa automação também é valiosa por si só; os engenheiros entendem profundamente os processos existentes que automatizaram e podem automatizar novos processos mais rapidamente.

Como é bem entendido na indústria, quanto mais tarde no ciclo de vida do produto um problema for descoberto, mais caro será sua correção; consulte o [Capítulo 17](#). Geralmente, os problemas que ocorrem na produção real são mais caros para corrigir, tanto em termos de tempo quanto de dinheiro, o que significa que um sistema automatizado procurando problemas assim que eles surgem tem uma boa chance de reduzir o custo total de o sistema, dado que o sistema é suficientemente grande.

Ação mais rápida

Nas situações de infraestrutura em que a automação SRE tende a ser implantada, os humanos geralmente não reagem tão rápido quanto as máquinas. Nos casos mais comuns, onde, por exemplo, failover ou comutação de tráfego podem ser bem definidos para um aplicativo específico, não faz sentido exigir efetivamente que um humano pressione intermitentemente um botão chamado “Permitir que o sistema continue em execução”. (Sim, é verdade que, às vezes, os procedimentos automáticos podem acabar piorando uma situação ruim, mas é por isso que esses procedimentos devem ter escopo em domínios bem definidos.) O Google tem uma grande quantidade de automação; em muitos casos, os serviços aos quais oferecemos suporte não sobreviveriam por muito tempo sem essa automação, pois ultrapassaram o limite da operação manual gerenciável há muito tempo.

Economia de

tempo Finalmente, a economia de tempo é uma razão muito citada para a automação. Embora as pessoas citem esse raciocínio para a automação mais do que os outros, em muitos aspectos o benefício geralmente é menos calculável imediatamente. Os engenheiros muitas vezes hesitam sobre se vale a pena escrever uma parte específica de automação ou código, em termos de esforço economizado em não exigir que uma tarefa seja executada manualmente versus o esforço necessário para escrevê-la.³ É fácil ignorar o fato de que, uma vez encapsulado, alguma tarefa em automação, qualquer um pode executar a tarefa. Portanto, a economia de tempo se aplica a qualquer pessoa que usaria plausivelmente a automação. O desacoplamento do operador da operação é muito poderoso.



Joseph Bironas, um SRE que liderou os esforços de ativação do datacenter do Google por um tempo, argumentou vigorosamente: “Se estivermos projetando processos e soluções que não são automatizáveis, continuaremos tendo que contratar humanos para manter o sistema. Se tivermos que contratar humanos para fazer o trabalho, estaremos alimentando as máquinas com sangue, suor e lágrimas de seres humanos. Pense em Matrix com menos efeitos especiais e administradores de sistema mais irritados.”

³ Veja o seguinte desenho do XKCD: <http://xkcd.com/1205/>.

O valor para o Google SRE

Todos esses benefícios e compensações se aplicam a nós tanto quanto a qualquer outra pessoa, e o Google tem uma forte tendência à automação. Parte de nossa preferência pela automação vem de nossos desafios comerciais específicos: os produtos e serviços que cuidamos são de escala planetária e normalmente não temos tempo para nos envolver no mesmo tipo de máquina ou serviço de mão comum em outras organizações.⁴ Para serviços realmente grandes, os fatores de consistência, rapidez e confiabilidade dominam a maioria das conversas sobre as vantagens e desvantagens da execução da automação.

Outro argumento a favor da automação, particularmente no caso do Google, é nosso ambiente de produção complicado, mas surpreendentemente uniforme, descrito no [Capítulo 2](#). Enquanto outras organizações podem ter um equipamento importante sem uma API prontamente acessível, software para o qual não código-fonte está disponível ou outro impedimento para o controle total sobre as operações de produção, o Google geralmente evita esses cenários. Construímos APIs para sistemas quando nenhuma API estava disponível do fornecedor. Embora a compra de software para uma tarefa específica fosse muito mais barata no curto prazo, optamos por escrever nossas próprias soluções, porque isso produzia APIs com potencial para benefícios muito maiores no longo prazo. Passamos muito tempo superando obstáculos ao gerenciamento automático do sistema e, em seguida, desenvolvemos resolutamente o próprio gerenciamento automático do sistema. Dado como o Google gerencia seu código-fonte [\[Pot16\]](#), a disponibilidade desse código para mais ou menos qualquer sistema que o SRE toque também significa que nossa missão de “possuir o produto em produção” é muito mais fácil porque controlamos a totalidade da pilha.

É claro que, embora o Google esteja ideologicamente inclinado a usar máquinas para gerenciar máquinas sempre que possível, a realidade exige algumas modificações em nossa abordagem. Não é apropriado automatizar todos os componentes de todos os sistemas e nem todos têm a capacidade ou inclinação para desenvolver automação em um determinado momento. Alguns sistemas essenciais começaram como protótipos rápidos, não projetados para durar ou interagir com a automação. Os parágrafos anteriores apresentam uma visão maximalista de nossa posição, mas que temos sido amplamente bem-sucedidas em colocar em ação no contexto do Google. Em geral, optamos por criar plataformas onde pudéssemos, ou nos posicionar para que pudéssemos criar plataformas ao longo do tempo. Vemos essa abordagem baseada em plataforma como necessária para gerenciamento e escalabilidade.

Os casos de uso para automação

Na indústria, automação é o termo geralmente usado para escrever código para resolver uma ampla variedade de problemas, embora as motivações para escrever esse código e as soluções

⁴ Veja, por exemplo, <http://blog.engineyard.com/2014/pets-vs-cattle>.

em si, muitas vezes são bem diferentes. De maneira mais ampla, nessa visão, a automação é um “meta software” – software para agir sobre software.

Como sugerimos anteriormente, há vários casos de uso para automação. Aqui está uma lista não exaustiva de exemplos:

- Criação de conta de usuário
- Ativação e desativação de cluster para serviços •
- Preparação e desativação de instalação de software ou hardware • Lançamentos de novas versões de software
- Mudanças de configuração de tempo de execução
- Um caso especial de mudanças de configuração de tempo de execução: mudanças em suas dependências

Esta lista poderia continuar essencialmente ad infinitum.

Casos de uso do Google SRE para automação No

Google, temos todos os casos de uso listados e muito mais.

No entanto, no Google SRE, nossa principal afinidade geralmente tem sido a execução de infraestrutura, em vez de gerenciar a qualidade dos dados que passam por essa infraestrutura. Essa linha não é totalmente clara - por exemplo, nos importamos muito se metade de um conjunto de dados desaparece após um push e, portanto, alertamos sobre diferenças grosseiras como essa, mas é raro escrevermos o equivalente a alterar as propriedades de algum subconjunto arbitrário de contas em um sistema.

Portanto, o contexto de nossa automação geralmente é a automação para gerenciar o ciclo de vida dos sistemas, não seus dados: por exemplo, implantações de um serviço em um novo cluster.

Nessa medida, os esforços de automação do SRE não estão longe do que muitas outras pessoas e organizações fazem, exceto que usamos ferramentas diferentes para gerenciá-lo e temos um foco diferente (como discutiremos).

Ferramentas amplamente disponíveis como Puppet, Chef, cfengine e até mesmo Perl, que fornecem maneiras de automatizar tarefas específicas, diferem principalmente em termos do nível de abstração dos componentes fornecidos para ajudar no ato de automatizar. Uma linguagem completa como Perl fornece recursos de nível POSIX, que em teoria fornecem um escopo essencialmente ilimitado de automação nas APIs acessíveis ao sistema,⁵ enquanto Chef e Puppet fornecem abstrações prontas para uso com quais serviços ou outras entidades de nível superior podem ser manipuladas. A compensação aqui é clássica: abstrações de nível superior são mais fáceis de gerenciar e raciocinar, mas quando você encontra uma “abstração com vazamento”,

⁵ É claro que nem todo sistema que precisa ser gerenciado realmente fornece APIs que podem ser chamadas para gerenciamento – forçando o uso de algumas ferramentas, por exemplo, invocações de CLI ou cliques automatizados em sites.

você falha sistematicamente, repetidamente e potencialmente de forma inconsistente. Por exemplo, geralmente assumimos que enviar um novo binário para um cluster é atômico; o cluster terminará com a versão antiga ou com a nova versão. No entanto, o comportamento do mundo real é mais complicado: a rede desse cluster pode falhar no meio do caminho; máquinas podem falhar; a comunicação com a camada de gerenciamento do cluster pode falhar, deixando o sistema em um estado inconsistente; dependendo da situação, novos binários podem ser preparados, mas não enviados, ou enviados, mas não reiniciados, ou reiniciados, mas não verificáveis. Muito poucas abstrações modelam esses tipos de resultados com sucesso e, geralmente, acabam se interrompendo e pedindo intervenção. Sistemas de automação realmente ruins nem fazem isso.

O SRE tem várias filosofias e produtos no domínio da automação, alguns dos quais se parecem mais com ferramentas de distribuição genéricas sem modelagem particularmente detalhada de entidades de nível superior e alguns dos quais se parecem mais com linguagens para descrever a implantação de serviços (e assim por diante) em um nível muito abstrato. O trabalho feito no último tende a ser mais reutilizável e uma plataforma mais comum do que o anterior, mas a complexidade do nosso ambiente de produção às vezes significa que a primeira abordagem é a opção mais imediatamente tratável.

Uma hierarquia de classes de automação

Embora todas essas etapas de automação sejam valiosas e, de fato, uma plataforma de automação seja valiosa por si só, em um mundo ideal, não precisaríamos de automação externalizada. Na verdade, em vez de ter um sistema que tenha que ter lógica de cola externa, seria ainda melhor ter um sistema que não precise de lógica de cola, não apenas porque a internalização é mais eficiente (embora essa eficiência seja útil), mas porque ele foi projetado para não precisar de lógica de cola em primeiro lugar. Conseguir isso envolve levar os casos de uso para a lógica de cola – geralmente manipulações de “primeira ordem” de um sistema, como adicionar contas ou executar a ativação do sistema – e encontrar uma maneira de lidar com esses casos de uso diretamente no aplicativo.

Como um exemplo mais detalhado, a maior parte da automação de turnup no Google é problemática porque acaba sendo mantida separadamente do sistema central e, portanto, sofre de “bit rot”, ou seja, não muda quando os sistemas subjacentes mudam. Apesar da melhor das intenções, tentar acoplar mais firmemente os dois (automação de ativação e o sistema central) geralmente falha devido a prioridades desalinhadas, pois os desenvolvedores de produtos resistirão, não sem razão, a um requisito de implantação de teste para cada mudança. Em segundo lugar, a automação que é crucial, mas executada apenas em intervalos infrequentes e, portanto, difícil de testar, muitas vezes é particularmente frágil devido ao ciclo de feedback estendido. O failover de cluster é um exemplo clássico de automação executada com pouca frequência: os failovers podem ocorrer apenas a cada poucos meses ou com pouca frequência para que sejam introduzidas inconsistências entre instâncias. A evolução da automação segue um caminho:

- 1) Nenhum mestre de banco de dados de automação é submetido a failover manualmente entre os locais.
- 2) Automação específica do sistema mantida externamente Um SRE tem um script de failover em seu diretório inicial.
- 3) Automação genérica mantida externamente O SRE adiciona suporte de banco de dados a um script de “failover genérico” que todos usam.
- 4) Automação específica do sistema mantida internamente O banco de dados é fornecido com seu próprio script de failover.
- 5) Sistemas que não precisam de automação O banco de dados detecta problemas e faz failover automaticamente sem intervenção humana.

O SRE odeia operações manuais, então obviamente tentamos criar sistemas que não as requeiram. No entanto, às vezes as operações manuais são inevitáveis.

Além disso, há uma subvariedade de automação que aplica alterações não no domínio da configuração específica relacionada ao sistema, mas no domínio da produção como um todo. Em um ambiente de produção proprietário altamente centralizado como o do Google, há um grande número de alterações que não têm escopo específico de serviço, por exemplo, alterar servidores Chubby upstream, uma alteração de sinalizador na biblioteca cliente Bigtable para tornar o acesso mais confiável e assim on - que, no entanto, precisam ser gerenciados com segurança e revertidos, se necessário. Além de um certo volume de alterações, é inviável que as alterações em toda a produção sejam realizadas manualmente e, em algum momento antes desse ponto, é um desperdício ter supervisão manual para um processo em que grande parte das alterações são triviais ou realizadas com sucesso por meio de estratégias básicas de relançamento e verificação.

Vamos usar estudos de caso internos para ilustrar alguns dos pontos anteriores em detalhes. O primeiro estudo de caso é sobre como, devido a um trabalho diligente e perspicaz, conseguimos alcançar o nirvana autoprovocado do SRE: nos automatizarmos fora de um trabalho.

Automatize-se fora de um trabalho: automatize TODAS as coisas!

Por muito tempo, os produtos Ads do Google armazenaram seus dados em um banco de dados MySQL. Como os dados de anúncios obviamente têm altos requisitos de confiabilidade, uma equipe de SRE foi encarregada de cuidar dessa infraestrutura. De 2005 a 2008, o banco de dados de anúncios funcionou principalmente no que consideramos um estado maduro e gerenciado. Por exemplo, automatizamos o pior, mas não todo, o trabalho de rotina para substituições de réplicas padrão. Acreditávamos que o banco de dados de anúncios era bem gerenciado e que havíamos colhido a maior parte dos frutos mais fáceis em termos de otimização e escala. No entanto, à medida que as operações diárias se tornaram confortáveis, os membros da equipe começaram a olhar para o próximo nível

de desenvolvimento de sistemas: migrando o MySQL para o sistema de agendamento de clusters do Google, Borg.

Esperávamos que essa migração fornecesse dois benefícios principais:

- Eliminar completamente a manutenção da máquina/réplica: o Borg automaticamente lidar com a configuração/reinicialização de tarefas novas e quebradas.
- Habilitar bin-packing de várias instâncias MySQL na mesma máquina física: o Borg permitiria o uso mais eficiente dos recursos da máquina por meio de Containers.

No final de 2008, implantamos com sucesso uma instância MySQL de prova de conceito no Borg.

Infelizmente, isso foi acompanhado por uma nova dificuldade significativa. Uma característica operacional central do Borg é que suas tarefas se movem automaticamente. As tarefas geralmente se movem dentro do Borg com a mesma frequência de uma ou duas vezes por semana. Essa frequência era tolerável para nossas réplicas de banco de dados, mas inaceitável para nossos mestres.

Naquela época, o processo de failover mestre levava de 30 a 90 minutos por instância. Simplesmente porque rodamos em máquinas compartilhadas e estávamos sujeitos a reinicializações para atualizações de kernel, além da taxa normal de falha de máquina, tínhamos que esperar uma série de failovers não relacionados a cada semana. Esse fator, em combinação com o número de shards em que nosso sistema estava hospedado, significava que:

- Failovers manuais consumiriam uma quantidade substancial de horas humanas e nos dariam disponibilidade no melhor caso de 99% de tempo de atividade, o que ficou aquém dos requisitos comerciais reais do produto. • Para atender aos nossos orçamentos de erros, cada failover teria que levar menos de 30 segundos de tempo de inatividade. Não havia como otimizar um procedimento humano-dependente para reduzir o tempo de inatividade em menos de 30 segundos.

Portanto, nossa única opção era automatizar o failover. Na verdade, precisávamos automatizar mais do que apenas o failover.

Em 2009, os Ads SRE concluiu nosso daemon de failover automatizado, que apelidamos de “Decider”. O Decider pode concluir failovers do MySQL para failovers planejados e não planejados em menos de 30 segundos em 95% das vezes. Com a criação do Decider, o MySQL on Borg (MoB) finalmente se tornou uma realidade. Passamos da otimização de nossa infraestrutura por falta de failover para abraçar a ideia de que a falha é inevitável e, portanto, otimizar para recuperar rapidamente por meio da automação.

Embora a automação nos permitisse alcançar o MySQL altamente disponível em um mundo que forçava até duas reinicializações por semana, ela veio com seu próprio conjunto de custos. Todos os nossos aplicativos tiveram que ser alterados para incluir significativamente mais lógica de tratamento de falhas do que antes. Dado que a norma no mundo de desenvolvimento MySQL é assumir que a instância MySQL será o componente mais estável na pilha, essa mudança significou customizar software como JDBC para ser mais tolerante ao nosso ambiente propenso a falhas. Quão-

sempre, os benefícios de migrar para o MoB com o Decider valeram a pena esses custos. Uma vez no MoB, o tempo que nossa equipe gastou em tarefas operacionais mundanas caiu 95%.

Nossos failovers foram automatizados, portanto, uma interrupção de uma única tarefa de banco de dados não paginava mais um humano.

O principal resultado dessa nova automação foi que tivemos muito mais tempo livre para gastar na melhoria de outras partes da infraestrutura. Essas melhorias tiveram um efeito cascata: quanto mais tempo economizávamos, mais tempo podíamos gastar na otimização e automatização de outros trabalhos tediosos. Eventualmente, conseguimos automatizar as mudanças de esquema, fazendo com que o custo de manutenção operacional total do banco de dados de anúncios caísse quase 95%. Alguns podem dizer que nos automatizamos com sucesso para fora deste trabalho. O lado de hardware do nosso domínio também viu melhorias.

A migração para o MoB liberou recursos consideráveis porque pudemos agendar várias instâncias do MySQL nas mesmas máquinas, o que melhorou a utilização de nosso hardware. No total, conseguimos liberar cerca de 60% do nosso hardware. Nossa equipe agora estava cheia de recursos de hardware e engenharia.

Este exemplo demonstra a sabedoria de ir além para entregar uma plataforma em vez de substituir os procedimentos manuais existentes. O próximo exemplo vem do grupo de infraestrutura de cluster e ilustra algumas das compensações mais difíceis que você pode encontrar no caminho para automatizar todas as coisas.

Acalmando a dor: aplicando a automação ao cluster Turnups

Dez anos atrás, a equipe de SRE de infraestrutura de cluster parecia conseguir uma nova contratação a cada poucos meses. Como se viu, essa foi aproximadamente a mesma frequência em que encontramos um novo cluster. Como a ativação de um serviço em um novo cluster dá aos novos contratados exposição aos componentes internos de um serviço, essa tarefa parecia uma ferramenta de treinamento natural e útil.

As etapas executadas para deixar um cluster pronto para uso foram algo como o seguinte:

1. Instale um edifício de datacenter para energia e refrigeração.
2. Instale e configure os switches principais e as conexões com o backbone.
3. Instale alguns racks iniciais de servidores.
4. Configure serviços básicos como DNS e instaladores, então configure um servidor de bloqueio vice, armazenamento e computação.
5. Implante os racks de máquinas restantes.
6. Atribua recursos de serviços voltados para o usuário, para que suas equipes possam configurar os serviços.

As etapas 4 e 6 foram extremamente complexas. Embora serviços básicos como DNS sejam relativamente simples, os subsistemas de armazenamento e computação na época ainda estavam em desenvolvimento intenso, então novos sinalizadores, componentes e otimizações foram adicionados semanalmente.

Alguns serviços tinham mais de cem subsistemas de componentes diferentes, cada um com uma complexa teia de dependências. Deixar de configurar um subsistema, ou configurar um sistema ou componente de forma diferente de outras implantações, é uma interrupção que afeta o cliente esperando para acontecer.

Em um caso, um cluster Bigtable de vários petabytes foi configurado para não usar o primeiro disco (registro) em sistemas de 12 discos, por motivos de latência. Um ano depois, alguma automação supôs que, se o primeiro disco de uma máquina não estivesse sendo usado, essa máquina não tinha nenhum armazenamento configurado; portanto, era seguro limpar a máquina e configura-la do zero. Todos os dados do Bigtable foram apagados instantaneamente. Felizmente, tivemos várias réplicas em tempo real do conjunto de dados, mas essas surpresas não são bem-vindas. A automação precisa ter cuidado ao confiar em sinais implícitos de “segurança”.

Automação inicial focada em acelerar a entrega de cluster. Essa abordagem tendia a depender do uso criativo do SSH para problemas tediosos de distribuição de pacotes e inicialização de serviços. Essa estratégia foi uma vitória inicial, mas esses roteiros de forma livre tornaram-se um colesterol de dívida técnica.

Detectando inconsistências com o Prodtest

À medida que o número de clusters crescia, alguns clusters exigiam sinalizadores e configurações ajustados manualmente. Como resultado, as equipes perdiam cada vez mais tempo perseguindo configurações incorretas difíceis de detectar. Se um sinalizador que tornasse o GFS mais responsável ao processamento de log vazasse nos modelos padrão, as células com muitos arquivos poderiam ficar sem memória sob carga.

Erros de configuração irritantes e demorados surgiram com quase todas as grandes alterações de configuração.

Os scripts de shell criativos – embora frágeis – que usamos para configurar clusters não eram dimensionados para o número de pessoas que queriam fazer alterações nem para o grande número de permutações de cluster que precisavam ser construídas. Esses scripts de shell também não resolveram preocupações mais significativas antes de declarar que um serviço era bom para receber tráfego voltado para o cliente, como:

- Todas as dependências do serviço estavam disponíveis e configuradas corretamente? • Todas as configurações e pacotes foram consistentes com outras implantações? • A equipe poderia confirmar se todas as exceções de configuração eram desejadas?

Prodtest (Production Test) foi uma solução engenhosa para essas surpresas indesejadas.

Estendemos a estrutura de teste de unidade do Python para permitir testes de unidade de serviços do mundo real. Esses testes de unidade têm dependências, permitindo uma cadeia de testes, e uma falha em um teste seria abortada rapidamente. Faça o teste mostrado na [Figura 7-1](#) como exemplo.

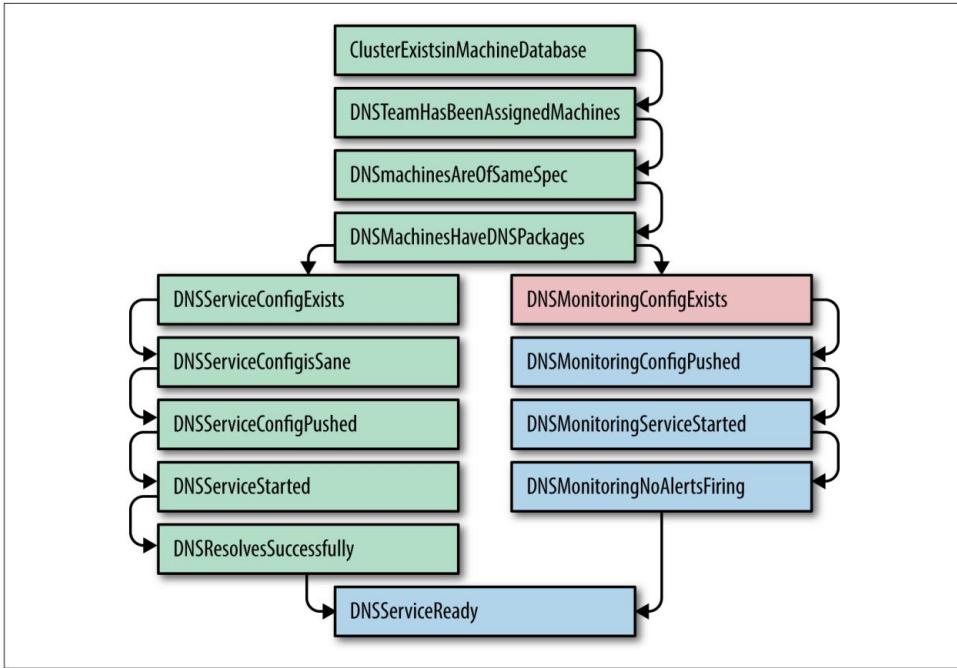


Figura 7-1. ProdTest for DNS Service, mostrando como um teste com falha aborta a cadeia subsequente de testes

O Prodtest de uma determinada equipe recebeu o nome do cluster e poderia validar os serviços dessa equipe nesse cluster. Adições posteriores nos permitiram gerar um gráfico dos testes de unidade e seus estados. Essa funcionalidade permitiu que um engenheiro visse rapidamente se seu serviço estava configurado corretamente em todos os clusters e, se não, por quê. O gráfico destacou a etapa com falha e o teste de unidade do Python com falha gera uma mensagem de erro mais detalhada.

Sempre que uma equipe encontrasse um atraso devido a uma configuração incorreta de outra equipe, um bug poderia ser arquivado para estender seu Prodtest. Isso garantiu que um problema semelhante fosse descoberto mais cedo no futuro. Os SREs estavam orgulhosos de poder garantir a seus clientes que todos os serviços – tanto serviços recém-ativados quanto serviços existentes com nova configuração – atenderiam de forma confiável o tráfego de produção.

Pela primeira vez, nossos gerentes de projeto puderam prever quando um cluster poderia “ir ao vivo” e tinham uma compreensão completa de por que cada cluster levava seis ou mais semanas para passar de “pronto para rede” para “servir tráfego ao vivo”. Do nada, a SRE recebeu uma missão da alta administração: em três meses, cinco novos clusters estarão prontos para a rede no mesmo dia. Por favor, transforme-os em uma semana.

Resolvendo Inconsistências Idempotentemente A

“Uma Semana de Ativação” era uma missão aterrorizante. Tínhamos dezenas de milhares de linhas de script de shell pertencentes a dezenas de equipes. Podíamos dizer rapidamente o quão despreparado um determinado cluster estava, mas corrigi-lo significava que as dezenas de equipes teriam que registrar centenas de bugs, e então tínhamos que esperar que esses bugs fossem prontamente corrigidos.

Percebemos que evoluir de “testes de unidade Python encontrando configurações incorretas” para “código Python corrigindo configurações incorretas” poderia nos permitir corrigir esses problemas mais rapidamente.

O teste de unidade já sabia qual cluster estávamos examinando e o teste específico que estava falhando, então combinamos cada teste com uma correção. Se cada correção foi escrita para ser idempotente e pudesse assumir que todas as dependências foram atendidas, resolver o problema deveria ser fácil e seguro de resolver. Exigir correções idempotentes significava que as equipes poderiam executar seu “script de correção” a cada 15 minutos sem temer danos à configuração do cluster. Se o teste da equipe DNS fosse bloqueado na configuração da equipe do Machine Database de um novo cluster, assim que o cluster aparecesse no banco de dados, os testes e correções da equipe DNS começariam a funcionar.

Faça o teste mostrado na [Figura 7-2](#) como exemplo. Se `TestDnsMonitoringConfigExists` falhar, como mostrado, podemos chamar `FixDnsMonitoringCreateConfig`, que extrai a configuração de um banco de dados e verifica um arquivo de configuração de esqueleto em nosso sistema de controle de revisão. Em seguida, `TestDnsMonitoringConfigExists` passa na nova tentativa e o teste `TestDnsMonitoringConfigPushed` pode ser tentado. Se o teste falhar, a etapa `FixDnsMonitoringPushConfig` será executada. Se uma correção falhar várias vezes, a automação assume que a correção falhou e para, notificando o usuário.

Armado com esses scripts, um pequeno grupo de engenheiros poderia garantir que pudéssemos passar de “A rede funciona e as máquinas estão listadas no banco de dados” para “Servir 1% do tráfego de pesquisa na web e anúncios” em questão de uma ou duas semanas. Na época, isso parecia ser o ápice da tecnologia de automação.

Olhando para trás, essa abordagem era profundamente falha; a latência entre o teste, a correção e, em seguida, um segundo teste introduziu testes irregulares que às vezes funcionavam e às vezes falhavam. Nem todas as correções eram naturalmente idempotentes, portanto, um teste irregular seguido por uma correção pode deixar o sistema em um estado inconsistente.

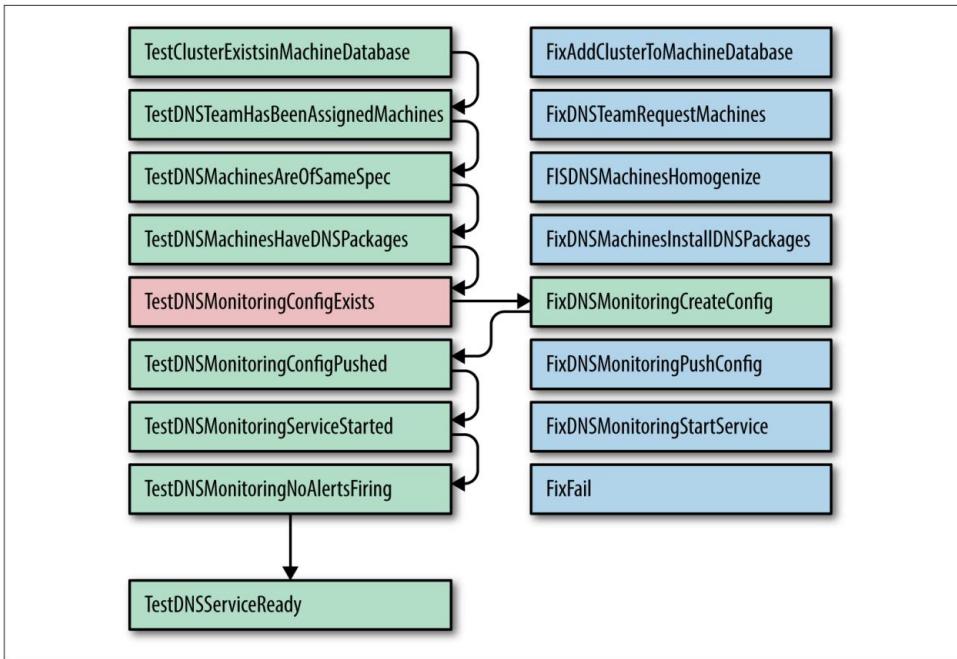


Figura 7-2. ProdTest for DNS Service, mostrando que um teste com falha resultou na execução de apenas uma correção

A inclinação para se especializar

Os processos de automação podem variar em três aspectos:

- Competência, ou seja, sua precisão •

Latência, a rapidez com que todas as etapas são executadas quando

iniciadas • Relevância ou proporção do processo do mundo real coberto pela automação

Começamos com um processo altamente competente (mantido e executado pelos proprietários do serviço), de alta latência (os proprietários do serviço executavam o processo em seu tempo livre ou o atribuíam a novos engenheiros) e muito relevante (os proprietários do serviço sabia quando o mundo real mudou e poderia consertar a automação).

Para reduzir a latência de ativação, muitas equipes proprietárias de serviços instruíram uma única “equipe de ativação” sobre qual automação executar. A equipe de ativação usou tickets para iniciar cada etapa da ativação para que pudéssemos rastrear as tarefas restantes e a quem essas tarefas foram atribuídas. Se as interações humanas em relação aos módulos de automação ocorressem entre pessoas na mesma sala, as mudanças de cluster poderiam acontecer em um tempo muito menor. Finalmente, tivemos nosso processo de automação competente, preciso e oportuno!

Mas esse estado não durou muito. O mundo real é caótico: software, configuração, dados, etc. alterados, resultando em mais de mil alterações separadas por dia nos sistemas afetados.

As pessoas mais afetadas por bugs de automação não eram mais especialistas em domínio, então a automação tornou-se menos relevante (o que significa que novas etapas foram perdidas) e menos competentes (novos sinalizadores podem ter causado falhas na automação). No entanto, demorou um pouco para essa queda na qualidade afetar a velocidade.

O código de automação, como o código de teste de unidade, morre quando a equipe de manutenção não é obsessiva em manter o código em sincronia com a base de código que cobre. O mundo muda em torno do código: a equipe de DNS adiciona novas opções de configuração, a equipe de armazenamento altera os nomes dos pacotes e a equipe de rede precisa oferecer suporte a novos dispositivos.

Ao liberar as equipes que executavam serviços da responsabilidade de manter e executar seu código de automação, criamos incentivos organizacionais feios:

- Uma equipe cuja principal tarefa é acelerar o turnup atual não tem incentivo para reduzir a dívida técnica da equipe proprietária do serviço que executa o serviço em produção posteriormente.
- Uma equipe que não executa a automação não tem incentivo para construir sistemas fáceis de automatizar.
- Um gerente de produto cujo cronograma não é afetado pela automação de baixa qualidade sempre priorizará novos recursos sobre simplicidade e automação.

As ferramentas mais funcionais geralmente são escritas por quem as usa. Um argumento semelhante se aplica ao motivo pelo qual as equipes de desenvolvimento de produtos se beneficiam de manter pelo menos alguma consciência operacional de seus sistemas em produção.

Os turnups foram novamente de alta latência, imprecisos e incompetentes - o pior de todos os mundos. No entanto, um mandato de segurança não relacionado nos permitiu sair dessa armadilha. Grande parte da automação distribuída dependia naquela época do SSH. Isso é desajeitado do ponto de vista da segurança, porque as pessoas precisam ter root em muitas máquinas para executar a maioria dos comandos.

Uma crescente conscientização sobre ameaças de segurança avançadas e persistentes nos levou a reduzir os privilégios de que os SREs desfrutavam ao mínimo absoluto de que precisavam para realizar seus trabalhos. Tivemos que substituir nosso uso de sshd por um daemon de administração local autenticado, orientado por ACL e baseado em RPC, também conhecido como servidores administrativos, que tinha permissões para realizar essas alterações locais. Como resultado, ninguém poderia instalar ou modificar um servidor sem uma trilha de auditoria. Mudanças no Local Admin Daemon e no Package Repo foram bloqueadas em revisões de código, tornando muito difícil para alguém exceder sua autoridade; dar a alguém o acesso para instalar pacotes não permitiria a visualização de logs colocados. O Admin Server registrou o solicitante de RPC, quaisquer parâmetros e os resultados de todos os RPCs para aprimorar a depuração e as auditorias de segurança.

Ativação de cluster orientada a serviço Na

próxima iteração, os servidores administrativos se tornaram parte dos fluxos de trabalho das equipes de serviço, tanto relacionados aos servidores administrativos específicos da máquina (para instalar pacotes e reiniciar) quanto aos servidores administrativos em nível de cluster (para ações como drenagem ou ativar um serviço).

Os SREs deixaram de escrever scripts de shell em seus diretórios pessoais para construir servidores RPC revisados por pares com ACLs refinadas.

Mais tarde, após a percepção de que os processos de ativação tinham que ser de propriedade das equipes que possuíam os serviços, vimos isso como uma maneira de abordar a ativação de cluster como um problema de Arquitetura Orientada a Serviços (SOA): os proprietários de serviços seriam responsáveis por criar um Admin Server para lidar com RPCs de ativação/desativação de clusters, enviados pelo sistema que sabia quando os clusters estavam prontos. Por sua vez, cada equipe forneceria o contrato (API) que a automação de ativação precisava, enquanto ainda estaria livre para alterar a implementação subjacente. À medida que um cluster estava “pronto para a rede”, a automação enviava um RPC para cada Admin Server que desempenhava um papel na ativação do cluster.

Agora temos um processo de baixa latência, competente e preciso; mais importante, esse processo permaneceu forte, pois a taxa de mudança, o número de equipes e o número de serviços parecem dobrar a cada ano.

Como mencionado anteriormente, nossa evolução da automação de turnup seguiu um caminho:

1. Ação manual acionada pelo operador (sem automação)
2. Automação específica do sistema escrita pelo operador
3. Automação genérica mantida externamente
4. Automação específica do sistema mantida internamente
5. Sistemas autônomos que não precisam de intervenção humana

Embora essa evolução tenha sido, em termos gerais, um sucesso, o estudo de caso Borg ilustra outra maneira pela qual passamos a pensar o problema da automação.

Borg: Nascimento do Computador em Escala de Armazém

Outra maneira de entender o desenvolvimento de nossa atitude em relação à automação é quando e onde essa automação é melhor implantada é considerar a história do desenvolvimento de nossos sistemas de gerenciamento de cluster.⁶ Como o MySQL no Borg, que demonstrou o sucesso da conversão de operações manuais aos automáticas, e o processo de ativação do cluster, que demonstrou a desvantagem de não pensar com cuidado suficiente sobre onde e como a automação foi implementada, desenvolvendo o cluster manuamente.

⁶ Comprimimos e simplificamos esse histórico para facilitar a compreensão.

A gestão também acabou demonstrando mais uma lição sobre como a automação deve ser feita. Como nossos dois exemplos anteriores, algo bastante sofisticado foi criado como resultado final de uma evolução contínua de origens mais simples.

Os clusters do Google foram implantados inicialmente como as pequenas redes de todos os outros da época: racks de máquinas com propósitos específicos e configurações heterogêneas.

Engenheiros fariam login em alguma máquina “mestre” bem conhecida para realizar tarefas administrativas; binários e configurações “dourados” viviam nesses mestres. Como tínhamos apenas um provedor colo, a maior parte da lógica de nomenclatura assumiu implicitamente esse local. À medida que a produção crescia e começamos a usar vários clusters, diferentes domínios (nomes de cluster) entraram em cena. Tornou-se necessário ter um arquivo descrevendo o que cada máquina fazia, que agrupava as máquinas sob alguma estratégia de nomenclatura solta. Este arquivo descritor, em combinação com o equivalente a um SSH paralelo, nos permitiu reiniciar (por exemplo) todas as máquinas de busca de uma só vez. Nessa época, era comum obter tickets como “a pesquisa é feita com a máquina x1, o rastreamento pode ter a máquina agora”.

O desenvolvimento da automação começou. Inicialmente, a automação consistia em scripts Python simples para operações como as seguintes:

- Gerenciamento de serviços: mantendo os serviços em execução (por exemplo, reinicia após falhas de segmentação)
- Rastreando quais serviços deveriam ser executados em quais máquinas
- Análise de mensagens de log: SSHing em cada máquina e procurando regexps

A automação acabou se transformando em um banco de dados adequado que rastreava o estado da máquina e também incorporava ferramentas de monitoramento mais sofisticadas. Com o conjunto de união da automação disponível, agora podíamos gerenciar automaticamente grande parte do ciclo de vida das máquinas: perceber quando as máquinas quebravam, removendo os serviços, enviando-as para reparo e restaurando a configuração quando voltavam do reparo.

Mas para dar um passo atrás, essa automação foi útil, mas profundamente limitada, devido ao fato de que as abstrações do sistema estavam implacavelmente ligadas a máquinas físicas. Precisávamos de uma nova abordagem, daí o Borg [Ver15] nasceu: um sistema que se afastou das atribuições relativamente estáticas de host/porta/trabalho do mundo anterior, para tratar uma coleção de máquinas como um gerenciado de recursos. Central para seu sucesso — e sua concepção — foi a noção de transformar o gerenciamento de cluster em uma entidade para a qual as chamadas de API pudessem ser emitidas, para algum coordenador central. Isso liberou dimensões extras de eficiência, flexibilidade e confiabilidade: diferentemente do modelo anterior de “propriedade” da máquina, Borg poderia permitir que as máquinas agendassem, por exemplo, tarefas em lote e voltadas para o usuário na mesma máquina.

Essa funcionalidade resultou em atualizações contínuas e automáticas do sistema operacional com uma quantidade muito pequena de esforço constante⁷ — esforço que não se adapta ao tamanho total das implantações de produção. Pequenos desvios no estado da máquina agora são corrigidos automaticamente; a quebra e o gerenciamento do ciclo de vida são essencialmente inoperantes para o SRE neste momento. Milhares de máquinas nascem, morrem e passam por reparos diariamente sem nenhum esforço de SRE. Para repetir as palavras de Ben Treynor Sloss: ao adotar a abordagem de que isso era um problema de software, a automação inicial nos deu tempo suficiente para transformar o gerenciamento de cluster em algo autônomo, em vez de automatizado. Alcançamos esse objetivo trazendo ideias relacionadas à distribuição de dados, APIs, arquiteturas hub-and-spoke e desenvolvimento de software de sistema distribuído clássico para apoiar o domínio do gerenciamento de infraestrutura.

Uma analogia interessante é possível aqui: podemos fazer um mapeamento direto entre o caso de uma única máquina e o desenvolvimento de abstrações de gerenciamento de cluster. Nessa visão, o reagendamento em outra máquina se parece muito com um processo movendo-se de uma CPU para outra: é claro, esses recursos de computação estão na outra extremidade de um link de rede, mas até que ponto isso realmente importa? Pensando nesses termos, o reagendamento parece uma característica intrínseca do sistema, em vez de algo que se “automatiza” — os humanos não conseguem reagir rápido o suficiente de qualquer maneira. Da mesma forma, no caso da ativação do cluster: nessa metáfora, a ativação do cluster é simplesmente capacidade programável adicional, um pouco como adicionar disco ou RAM a um único computador. No entanto, não se espera que um computador de nó único, em geral, continue operando quando um grande número de componentes falha. O computador global deve ser auto-reparável para operar uma vez que ultrapassa um certo tamanho, devido ao grande número de falhas essencialmente garantido estatisticamente que ocorre a cada segundo. Isso implica que, à medida que movemos os sistemas para cima na hierarquia, de acionados manualmente para acionados automaticamente, para autônomos, alguma capacidade de auto-introspecção é necessária para sobreviver.

Confiabilidade é a característica fundamental

É claro que, para uma solução de problemas eficaz, os detalhes da operação interna em que a introspecção depende também devem ser expostos aos humanos que gerenciam o sistema geral. Discussões análogas sobre o impacto da automação no domínio não computacional — por exemplo, em voos de avião⁸ ou aplicações industriais — muitas vezes apontam o lado negativo da automação altamente eficaz:⁹ os operadores humanos ficam progressivamente mais livres do contato direto útil com o sistema à medida que a automação cobre cada vez mais atividades diárias ao longo do tempo. Inevitavelmente, então, surge uma situação em que a automação falha e os humanos são agora incapazes de operar o sistema com sucesso.

⁷ Como em um número pequeno e imutável.

⁸ Veja, por exemplo, https://en.wikipedia.org/wiki/Air_France_Flight_447.

⁹ Veja, por exemplo, [Bai83] e [Sar97].

A fluidez de suas reações foi perdida devido à falta de prática, e seus modelos mentais do que o sistema deveria estar fazendo não refletem mais a realidade do que está fazendo.¹⁰ Essa situação surge mais quando o sistema não é autônomo - ou seja, onde a automação substitui as ações manuais, e presume-se que as ações manuais sejam sempre executáveis e disponíveis exatamente como antes. Infelizmente, com o tempo, isso acaba se tornando falso: essas ações manuais nem sempre são executáveis porque a funcionalidade para permiti-las não existe mais.

Nós também passamos por situações em que a automação foi ativamente prejudicial em várias ocasiões – consulte “[Automação: habilitando falhas em escala](#)” na página 85 – mas, na experiência do Google, há mais sistemas para os quais a automação ou o comportamento autônomo não são mais extras opcionais. À medida que você escala, é claro que esse é o caso, mas ainda há argumentos fortes para um comportamento mais autônomo dos sistemas, independentemente do tamanho. A confiabilidade é a característica fundamental, e o comportamento autônomo e resiliente é uma maneira útil de conseguir isso.

Recomendações

Você pode ler os exemplos neste capítulo e decidir que precisa ser escalado pelo Google antes de ter qualquer coisa a ver com automação. Isso não é verdade, por dois motivos: a automação oferece mais do que apenas economia de tempo, então vale a pena implementá-la em mais casos do que um simples cálculo de tempo gasto versus economia de tempo pode sugerir. Mas a abordagem com a maior alavancagem realmente ocorre na fase de projeto: enviar e iterar rapidamente pode permitir que você implemente a funcionalidade mais rapidamente, mas raramente contribui para um sistema resiliente. A operação autônoma é difícil de adaptar de forma convincente a sistemas suficientemente grandes, mas as boas práticas padrão em engenharia de software ajudarão consideravelmente: desacoplar subsistemas, introduzir APIs, minimizar efeitos colaterais e assim por diante.

10 Esta é mais uma boa razão para treinos regulares; consulte “[Desastre Role Playing](#)” na página 401.

Automação: permitindo falhas em escala O

Google executa mais de uma dúzia de seus próprios grandes datacenters, mas também dependemos de máquinas em muitas instalações de colocation de terceiros (ou “colos”). Nossas máquinas nesses colos são usadas para encerrar a maioria das conexões de entrada ou como cache para nossa própria Rede de Entrega de Conteúdo, a fim de diminuir a latência do usuário final. A qualquer momento, vários desses racks estão sendo instalados ou desativados; ambos os processos são amplamente automatizados. Uma etapa durante o descomissionamento envolve a substituição de todo o conteúdo do disco de todas as máquinas no rack, após o qual um sistema independente verifica o apagamento bem-sucedido. Chamamos esse processo de “Diskerase”.

Era uma vez, a automação responsável pelo descomissionamento de um determinado rack falhou, mas somente após a etapa do Diskerase ter sido concluída com sucesso. Mais tarde, o processo de descomissionamento foi reiniciado desde o início, para depurar a falha. Nessa iteração, ao tentar enviar o conjunto de máquinas do rack para o Diskerase, a automação determinou que o conjunto de máquinas que ainda precisava ser Diskerase estava (corretamente) vazio. Infelizmente, o conjunto vazio foi usado como um valor especial, interpretado como “tudo”. Isso significa que a automação enviou quase todas as máquinas que temos em todos os colos para Diskerase.

Em minutos, o Diskerase altamente eficiente limpou os discos em todas as máquinas em nossa CDN, e as máquinas não foram mais capazes de encerrar conexões de usuários (ou fazer qualquer outra coisa útil). Ainda conseguimos atender todos os usuários de nossos próprios datacenters e, após alguns minutos, o único efeito visível externamente foi um leve aumento na latência. Até onde pudemos dizer, pouquíssimos usuários notaram o problema, graças ao bom planejamento de capacidade (pelo menos acertamos!). Enquanto isso, passamos quase dois dias reinstalando as máquinas nos racks de cores afetados; em seguida, passamos as semanas seguintes auditando e adicionando mais verificações de integridade — incluindo limitação de taxa — em nossa automação e tornando nosso fluxo de trabalho de desativação idempotente.

CAPÍTULO 8

Engenharia de Liberação

**Escrito por Dinah McNutt
Editado por Betsy Beyer e Tim Harvey**

A engenharia de lançamento é uma disciplina de engenharia de software relativamente nova e de rápido crescimento que pode ser concisamente descrita como construção e entrega de software [McN14a]. Os engenheiros de versão têm uma compreensão sólida (se não especializada) de gerenciamento de código-fonte, compiladores, linguagens de configuração de compilação, ferramentas de compilação automatizadas, gerenciadores de pacotes e instaladores. Seu conjunto de habilidades inclui profundo conhecimento de vários domínios: desenvolvimento, gerenciamento de configuração, integração de teste, administração de sistema e suporte ao cliente.

A execução de serviços confiáveis requer processos de liberação confiáveis. Os Engenheiros de Confiabilidade do Site (SREs) precisam saber que os binários e as configurações que eles usam são construídos de maneira reproduzível e automatizada para que os lançamentos sejam repetíveis e não sejam “flocos de neve exclusivos”. As alterações em qualquer aspecto do processo de liberação devem ser intencionais, e não acidentais. Os SREs se preocupam com esse processo desde o código-fonte até a implantação.

A engenharia de lançamento é uma função de trabalho específica no Google. Os engenheiros de versão trabalham com engenheiros de software (SWEs) no desenvolvimento de produtos e SREs para definir todas as etapas necessárias para liberar o software - desde como o software é armazenado no repositório de código-fonte, para criar regras para compilação, como testar, empacotar e implantar são conduzidas.

O papel de um engenheiro de lançamento

O Google é uma empresa orientada a dados e a engenharia de lançamento segue o exemplo. Temos ferramentas que relatam uma série de métricas, como quanto tempo leva para uma mudança de código ser implantada em produção (em outras palavras, velocidade de lançamento) e estatísticas sobre o que

recursos estão sendo usados em arquivos de configuração de compilação [Ada15]. A maioria dessas ferramentas foi imaginada e desenvolvida por engenheiros de lançamento.

Os engenheiros de lançamento definem as melhores práticas para usar nossas ferramentas para garantir que os projetos sejam lançados usando metodologias consistentes e repetíveis. Nossas melhores práticas cobrem todos os elementos do processo de liberação. Os exemplos incluem sinalizadores do compilador, formatos para tags de identificação de compilação e etapas necessárias durante uma compilação. Certificar-se de que nossas ferramentas se comportem corretamente por padrão e sejam documentadas adequadamente facilita para as equipes manterem o foco em recursos e usuários, em vez de gastar tempo reinventando a roda (mal) quando se trata de liberar software.

O Google tem um grande número de SREs encarregados de implantar produtos com segurança e manter os serviços do Google em funcionamento. Para garantir que nossos processos de lançamento atendam aos requisitos de negócios, os engenheiros de lançamento e os SREs trabalham juntos para desenvolver estratégias para alterar as mudanças, lançar novos lançamentos sem interromper os serviços e reverter recursos que demonstrem problemas.

Filosofia

A engenharia de lançamento é guiada por uma filosofia de engenharia e serviço expressa por meio de quatro princípios principais, detalhados nas seções a seguir.

Modelo de autoatendimento

Para trabalhar em escala, as equipes devem ser autossuficientes. A engenharia de lançamento desenvolveu as melhores práticas e ferramentas que permitem que nossas equipes de desenvolvimento de produtos controlem e executem seus próprios processos de lançamento. Embora tenhamos milhares de engenheiros e produtos, podemos atingir uma alta velocidade de lançamento porque as equipes individuais podem decidir com que frequência e quando lançar novas versões de seus produtos. Os processos de lançamento podem ser automatizados a ponto de exigir o mínimo de envolvimento dos engenheiros, e muitos projetos são criados e lançados automaticamente usando uma combinação de nosso sistema de compilação automatizado e nossas ferramentas de implantação. Os lançamentos são realmente automáticos e exigem apenas o envolvimento do engenheiro se e quando surgirem problemas.

O software voltado

para o usuário de **alta velocidade** (como muitos componentes da Pesquisa Google) é reconstruído com frequência, pois nosso objetivo é lançar recursos voltados para o cliente o mais rápido possível. Adotamos a filosofia de que lançamentos frequentes resultam em menos alterações entre as versões. Essa abordagem facilita o teste e a solução de problemas. Algumas equipes executam compilações de hora em hora e, em seguida, selecionam a versão para realmente implantar na produção do pool de compilações resultante. A seleção é baseada nos resultados do teste e nos recursos contidos em uma determinada compilação. Outras equipes adotaram um modelo de lançamento “Push on Green” e implantam todas as compilações que passam em todos os testes [Kle14].

Construções Herméticas

As ferramentas de construção devem nos permitir garantir consistência e repetibilidade. Se duas pessoas tentarem construir o mesmo produto com o mesmo número de revisão no repositório de código fonte em máquinas diferentes, esperamos resultados idênticos.¹ Nossas compilações são herméticas, o que significa que são insensíveis às bibliotecas e outros softwares instalados na máquina de construção. Em vez disso, as compilações dependem de versões conhecidas de ferramentas de compilação, como compiladores e dependências, como bibliotecas. O processo de compilação é autocontido e não deve depender de serviços externos ao ambiente de compilação.

Reconstruir versões mais antigas quando precisamos corrigir um bug no software que está sendo executado em produção pode ser um desafio. Realizamos essa tarefa reconstruindo na mesma revisão da compilação original e incluindo alterações específicas que foram enviadas após esse ponto no tempo. Chamamos essa tática de colheita de cereja. Nossas ferramentas de compilação são versionadas com base na revisão no repositório de código-fonte para o projeto que está sendo compilado. Portanto, um projeto construído no mês passado não usará a versão deste mês do compilador se for necessário escolher uma cereja, porque essa versão pode conter recursos incompatíveis ou indesejados.

Aplicação de Políticas e Procedimentos

Várias camadas de segurança e controle de acesso determinam quem pode realizar operações específicas ao liberar um projeto. As operações fechadas incluem:

- Aprovação de alterações no código-fonte - esta operação é gerenciada por meio de configuração espalhados por toda a base de código
- Especificar as ações a serem executadas durante o processo de liberação
- Criar uma nova versão
- Aprovar a proposta de integração inicial (que é uma solicitação para executar uma compilação em um número de revisão específico no repositório de código-fonte) e escolhas especiais
- Implantando uma nova versão
- Fazendo alterações na configuração de compilação de um projeto

Quase todas as alterações na base de código exigem uma revisão de código, que é uma ação simplificada integrada ao nosso fluxo de trabalho normal do desenvolvedor. Nossa sistema de lançamento automatizado produz um relatório de todas as alterações contidas em um lançamento, que é arquivado com outros artefatos de compilação. Ao permitir que os SREs entendam quais mudanças estão incluídas em um novo

¹ O Google usa um repositório de código-fonte unificado monolítico; veja [Pote16].

lançamento de um projeto, este relatório pode agilizar a solução de problemas quando houver problemas com um lançamento.

Construção e implantação contínuas

O Google desenvolveu um sistema de lançamento automatizado chamado Rapid. O Rapid é um sistema que aproveita várias tecnologias do Google para fornecer uma estrutura que oferece versões escalonáveis, herméticas e confiáveis. As seções a seguir descrevem o ciclo de vida do software no Google e como ele é gerenciado usando o Rapid e outras ferramentas associadas.

Building

Blaze2 é a ferramenta de construção preferida do Google. Ele suporta a construção de binários de uma variedade de linguagens, incluindo nossas linguagens padrão de C++, Java, Python, Go e Java-Script. Os engenheiros usam o Blaze para definir destinos de compilação (por exemplo, a saída de uma compilação, como um arquivo JAR) e para especificar as dependências para cada destino. Ao executar uma compilação, o Blaze compila automaticamente os destinos de dependência.

Os destinos de compilação para binários e testes de unidade são definidos nos arquivos de configuração do projeto do Rapid. Os sinalizadores específicos do projeto, como um identificador de compilação exclusivo, são passados pelo Rapid para o Blaze. Todos os binários suportam um sinalizador que exibe a data de compilação, o número de revisão e o identificador de compilação, o que nos permite associar facilmente um binário a um registro de como ele foi construído.

Ramificação

Todo o código é verificado na ramificação principal da árvore de código-fonte (linha principal). No entanto, a maioria dos grandes projetos não são lançados diretamente da linha principal. Em vez disso, ramificamos da linha principal em uma revisão específica e nunca mesclamos as alterações da ramificação de volta à linha principal. As correções de bugs são enviadas para a linha principal e, em seguida, selecionadas para a ramificação para inclusão na versão. Essa prática evita pegar inadvertidamente alterações não relacionadas enviadas à linha principal desde que a compilação original ocorreu. Usando esse método de ramificação e seleção de cereja, sabemos o conteúdo exato de cada versão.

Teste Um

sistema de teste contínuo executa testes de unidade em relação ao código na linha principal sempre que uma alteração é enviada, permitindo detectar falhas de compilação e teste rapidamente. A engenharia de lançamento recomenda que os alvos de teste de compilação contínua correspondam aos mesmos alvos de teste que controlam o lançamento do projeto. Também recomendamos a criação de versões em

² Blaze foi open source como Bazel. Consulte "Perguntas frequentes do Bazel" no site do Bazel, <http://bazel.io/faq.html>.

o número de revisão (versão) da última compilação de teste contínuo que concluiu com êxito todos os testes. Essas medidas diminuem a chance de que as alterações subsequentes feitas na linha principal causem falhas durante a compilação executada no momento da liberação.

Durante o processo de lançamento, reexecutamos os testes de unidade usando o branch de lançamento e criamos uma trilha de auditoria mostrando que todos os testes foram aprovados. Essa etapa é importante porque, se um lançamento envolve escolhas especiais, a ramificação de lançamento pode conter uma versão do código que não existe em nenhum lugar na linha principal. Queremos garantir que os testes passem no contexto do que realmente está sendo lançado.

Para complementar o sistema de teste contínuo, usamos um ambiente de teste independente que executa testes em nível de sistema em artefatos de compilação empacotados. Esses testes podem ser iniciados manualmente ou a partir do Rapid.

O Software de

Embalagem é distribuído para nossas máquinas de produção através do Midas Package Manager (MPM) [McN14c]. O MPM monta pacotes com base nas regras do Blaze que listam os artefatos de compilação a serem incluídos, juntamente com seus proprietários e permissões. Os pacotes são nomeados (por exemplo, search/shakespeare/frontend), versionados com um hash exclusivo e assinados para garantir a autenticidade. O MPM suporta a aplicação de rótulos a uma versão específica de um pacote. O Rapid aplica um rótulo contendo o ID da compilação, o que garante que um pacote possa ser referenciado exclusivamente usando o nome do pacote e esse rótulo.

Os rótulos podem ser aplicados a um pacote MPM para indicar a localização de um pacote no processo de lançamento (por exemplo, dev, canary ou production). Se aplicar uma etiqueta existente a uma nova embalagem, a etiqueta é automaticamente movida da embalagem antiga para a nova embalagem. Por exemplo: se um pacote for rotulado como canary, alguém que instalar posteriormente a versão canary desse pacote receberá automaticamente a versão mais recente do pacote com o rótulo canary.

Rapid

A [Figura 8-1](#) mostra os principais componentes do sistema Rapid. O Rapid é configurado com arquivos chamados blueprints. Os blueprints são escritos em uma linguagem de configuração interna e são usados para definir destinos de compilação e teste, regras para implantação e informações administrativas (como proprietários de projetos). As listas de controle de acesso baseadas em função determinam quem pode executar ações específicas em um projeto Rapid.

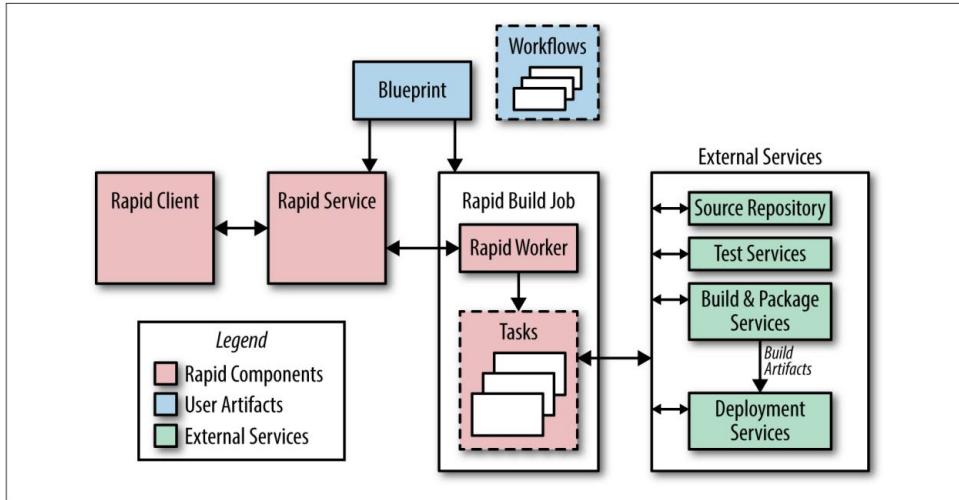


Figura 8-1. Visão simplificada da arquitetura Rapid mostrando os principais componentes do sistema

Cada projeto do Rapid possui fluxos de trabalho que definem as ações a serem executadas durante o processo de liberação. As ações de fluxo de trabalho podem ser executadas em série ou em paralelo, e um fluxo de trabalho pode iniciar outros fluxos de trabalho. O Rapid despacha solicitações de trabalho para tarefas executadas como um trabalho Borg em nossos servidores de produção. Como o Rapid usa nossa infraestrutura de produção, ele pode lidar com milhares de solicitações de lançamento simultaneamente.

Um processo de liberação típico ocorre da seguinte forma:

1. O Rapid usa o número de revisão de integração solicitado (geralmente obtido automaticamente do nosso sistema de teste contínuo) para criar uma ramificação de lançamento.
2. O Rapid usa o Blaze para compilar todos os binários e executar os testes de unidade, muitas vezes realizando essas duas etapas em paralelo. A compilação e o teste ocorrem em ambientes dedicados a essas tarefas específicas, em vez de ocorrer no trabalho Borg, onde o fluxo de trabalho do Rapid está sendo executado. Essa separação nos permite parallelizar o trabalho facilmente.
3. Os artefatos de construção ficam então disponíveis para teste do sistema e implementações canárias. Uma implantação canária típica envolve iniciar alguns trabalhos em nosso ambiente de produção após a conclusão dos testes do sistema.
4. Os resultados de cada etapa do processo são registrados. Um relatório de todas as alterações desde a última versão é criado.

O Rapid nos permite gerenciar nossas ramificações de lançamento e escolhas de cereja; solicitações individuais de coleta seletiva podem ser aprovadas ou rejeitadas para inclusão em uma versão.

O Deployment

Rapid é frequentemente usado para conduzir implantações simples diretamente. Ele atualiza os trabalhos do Borg para usar pacotes MPM recém-criados com base nas definições de implantação nos arquivos de blueprint e executores de tarefas especializados.

Para implantações mais complicadas, usamos o Sisyphus, que é uma estrutura de automação de implementação geral desenvolvida pela SRE. Uma distribuição é uma unidade lógica de trabalho composta por uma ou mais tarefas individuais. O Sisyphus fornece um conjunto de classes Python que podem ser estendidas para dar suporte a qualquer processo de implantação. Ele tem um painel que permite um controle mais preciso sobre como o lançamento é realizado e fornece uma maneira de monitorar o progresso do lançamento.

Em uma integração típica, o Rapid cria uma distribuição em um trabalho Sisyphus de longa duração. O Rapid conhece o rótulo de compilação associado ao pacote MPM que criou e pode especificar esse rótulo de compilação ao criar o lançamento no Sisyphus. O Sisyphus usa o rótulo de compilação para especificar qual versão dos pacotes MPM deve ser implantada.

Com o Sisyphus, o processo de implantação pode ser tão simples ou complicado quanto necessário. Por exemplo, ele pode atualizar todos os trabalhos associados imediatamente ou pode implementar um novo binário em clusters sucessivos por um período de várias horas.

Nosso objetivo é adequar o processo de implantação ao perfil de risco de um determinado serviço. Em ambientes de desenvolvimento ou pré-produção, podemos construir lançamentos de hora em hora e enviar automaticamente quando todos os testes forem aprovados. Para grandes serviços voltados para o usuário, podemos iniciar em um cluster e expandir exponencialmente até que todos os clusters sejam atualizados. Para partes sensíveis de infraestrutura, podemos estender o lançamento por vários dias, deixando-os entre instâncias em diferentes regiões geográficas.

Gerenciamento de configuração

O gerenciamento de configuração é uma área de colaboração particularmente próxima entre engenheiros de lançamento e SREs. Embora o gerenciamento de configuração possa parecer inicialmente um problema aparentemente simples, as mudanças de configuração são uma fonte potencial de instabilidade. Como resultado, nossa abordagem para liberar e gerenciar configurações de sistema e serviço evoluiu substancialmente ao longo do tempo. Hoje utilizamos vários modelos para distribuição de arquivos de configuração, conforme descrito nos parágrafos a seguir. Todos os esquemas envolvem o armazenamento da configuração em nosso repositório de código-fonte primário e a aplicação de um requisito rigoroso de revisão de código.

Use a linha principal para configuração. Este foi o primeiro método usado para configurar serviços no Borg (e nos sistemas anteriores ao Borg). Usando esse esquema, desenvolvedores e SREs modificam os arquivos de configuração no cabeçalho da ramificação principal. As alterações são revisadas e aplicadas ao sistema em execução. Como resultado, as versões binárias e as alterações de configuração são desacopladas. Embora conceitualmente e processualmente simples,

essa técnica geralmente leva a um desvio entre a versão com check-in dos arquivos de configuração e a versão em execução do arquivo de configuração porque os trabalhos devem ser atualizados para receber as alterações.

Inclua arquivos de configuração e binários no mesmo pacote MPM. Para projetos com poucos arquivos de configuração ou projetos em que os arquivos (ou um subconjunto de arquivos) mudam a cada ciclo de lançamento, os arquivos de configuração podem ser incluídos no pacote MPM com os binários. Embora essa estratégia limite a flexibilidade vinculando firmemente os arquivos binários e de configuração, ela simplifica a implantação, pois requer apenas a instalação de um pacote.

Empacote os arquivos de configuração em “pacotes de configuração” do MPM. Podemos aplicar o princípio hermético ao gerenciamento de configuração. As configurações binárias tendem a ser fortemente vinculadas a versões específicas de binários, então aproveitamos os sistemas de compilação e empacotamento para capturar e liberar arquivos de configuração junto com seus binários. Semelhante ao nosso tratamento de binários, podemos usar o build ID para reconstruir a configuração em um ponto específico no tempo.

Por exemplo, uma mudança que implementa um novo recurso pode ser lançada com uma configuração de sinalizador que configura esse recurso. Ao gerar dois pacotes MPM, um para o binário e outro para a configuração, mantemos a capacidade de alterar cada pacote independentemente. Ou seja, se o recurso foi lançado com uma configuração de sinalizador de `first_folio`, mas percebermos que deveria ser `bad_quarto`, podemos escolher essa alteração na ramificação de lançamento, reconstruir o pacote de configuração e implantá-lo. Essa abordagem tem a vantagem de não exigir uma nova compilação binária.

Podemos aproveitar o recurso de rotulagem do MPM para indicar quais versões dos pacotes MPM devem ser instaladas juntas. Um rótulo de `much_ado` pode ser aplicado aos pacotes MPM descritos no parágrafo anterior, o que nos permite buscar ambos os pacotes usando este rótulo. Quando uma nova versão do projeto é construída, o rótulo `much_ado` será aplicado aos novos pacotes. Como essas tags são exclusivas no namespace de um pacote MPM, somente o pacote mais recente com essa tag será usado.

Leia os arquivos de configuração de um armazenamento externo. Alguns projetos têm arquivos de configuração que precisam ser alterados com frequência ou dinamicamente (ou seja, enquanto o binário está em execução). Esses arquivos podem ser armazenados em Chubby, Bigtable ou nosso sistema de arquivos baseado em fonte [[Kem11](#)].

Em resumo, os proprietários do projeto consideram as diferentes opções para distribuir e gerenciar arquivos de configuração e decidem qual funciona melhor caso a caso.

Conclusões

Embora este capítulo tenha discutido especificamente a abordagem do Google para a engenharia de lançamento e as maneiras pelas quais os engenheiros de lançamento trabalham e colaboram com os SREs, essas práticas também podem ser aplicadas de forma mais ampla.

Não é apenas para Googlers Quando

equipados com as ferramentas certas, automação adequada e políticas bem definidas, os desenvolvedores e SREs não precisam se preocupar com o lançamento de software. Os lançamentos podem ser tão indolores quanto simplesmente pressionar um botão.

A maioria das empresas lida com o mesmo conjunto de problemas de engenharia de lançamento, independentemente de seu tamanho ou das ferramentas que usam: Como você deve lidar com o controle de versão de seus pacotes? Você deve usar um modelo contínuo de compilação e implantação ou executar compilações periódicas? Com que frequência você deve liberar? Quais políticas de gerenciamento de configuração você deve usar? Quais métricas de lançamento são de interesse?

Os engenheiros de versão do Google desenvolveram nossas próprias ferramentas por necessidade, pois ferramentas de código aberto ou fornecidas por fornecedores não funcionam na escala que exigimos. Ferramentas personalizadas nos permitem incluir funcionalidades para dar suporte (e até mesmo impor) políticas de processo de lançamento. No entanto, essas políticas devem primeiro ser definidas para adicionar recursos apropriados às nossas ferramentas, e todas as empresas devem se esforçar para definir seus processos de liberação, sejam eles automatizados e/ou aplicados ou não.

Comece a engenharia de lançamentos no início A engenharia de

lançamentos muitas vezes é uma reflexão tardia, e essa maneira de pensar deve mudar à medida que plataformas e serviços continuam a crescer em tamanho e complexidade.

As equipes devem fazer um orçamento para recursos de engenharia de lançamento no início do ciclo de desenvolvimento do produto. É mais barato implementar boas práticas e processos antecipadamente, em vez de ter que adaptar seu sistema posteriormente.

É essencial que os desenvolvedores, SREs e engenheiros de lançamento trabalhem juntos. O engenheiro de lançamento precisa entender a intenção de como o código deve ser construído e implantado. Os desenvolvedores não devem construir e “jogar os resultados por cima do muro” para serem tratados pelos engenheiros de lançamento.

As equipes de projeto individuais decidem quando a engenharia de lançamento se envolve em um projeto. Como a engenharia de lançamento ainda é uma disciplina relativamente jovem, os gerentes nem sempre planejam e orçam a engenharia de lançamento nos estágios iniciais de um projeto.

Portanto, ao considerar como incorporar práticas de engenharia de lançamento, certifique-se de considerar seu papel como aplicado a todo o ciclo de vida de seu produto ou serviço, particularmente nos estágios iniciais.

Mais Informações

Para obter mais informações sobre engenharia de lançamento, consulte as apresentações a seguir, cada uma com vídeo disponível online:

- [Como a adoção do lançamento contínuo reduziu a complexidade da mudança](#), USENIX Release Engineering Summit West 2014, [Dic14]
- [Mantendo a consistência em um ambiente massivamente paralelo](#), Configuração USENIX Cúpula de Gestão de 2013, [McN13]
- [Os 10 Mandamentos da Engenharia de Liberação](#), 2º Workshop Internacional de Engenharia de Lançamento 2014, [McN14b]
- [Distribuindo So ware em um Ambiente Massivamente Paralelo](#), LISA 2014, [McN14c]

CAPÍTULO 9

Simplicidade

Escrito por Max Luebbecke
Editado por Tim Harvey

O preço da confiabilidade é a busca da máxima simplicidade.

—CAR Hoare, palestra do Prêmio Turing

Os sistemas de software são inherentemente dinâmicos e instáveis.¹ Um sistema de software só pode ser perfeitamente estável se existir no vácuo. Se pararmos de alterar a base de código, paramos de introduzir bugs. Se o hardware ou as bibliotecas subjacentes nunca mudarem, nenhum desses componentes apresentará bugs. Se congelarmos a base de usuários atual, nunca teremos que dimensionar o sistema. Na verdade, um bom resumo da abordagem SRE para gerenciar sistemas é: “No final das contas, nosso trabalho é manter a agilidade e a estabilidade em equilíbrio no sistema.”²

Estabilidade do sistema versus agilidade

Às vezes faz sentido sacrificar a estabilidade em nome da agilidade. Muitas vezes abordei um domínio de problema desconhecido conduzindo o que chamo de codificação exploratória - definindo um prazo de validade explícito para qualquer código que escrevo com o entendimento de que preciso tentar e falhar uma vez para realmente entender a tarefa que preciso concluir. O código que vem com uma data de expiração pode ser muito mais liberal com cobertura de teste e gerenciamento de lançamento porque nunca será enviado para produção ou visto pelos usuários.

¹ Isso geralmente é verdade para sistemas complexos em geral; veja [Per99] e [Coo00].

² Criado por meu ex-empresário, Johan Anderson, na época em que me tornei um SRE.

Para a maioria dos sistemas de software de produção, queremos uma combinação equilibrada de estabilidade e agilidade. Os SREs trabalham para criar procedimentos, práticas e ferramentas que tornam o software mais confiável. Ao mesmo tempo, os SREs garantem que esse trabalho tenha o menor impacto possível na agilidade do desenvolvedor. Na verdade, a experiência do SRE descobriu que processos confiáveis tendem a aumentar a agilidade do desenvolvedor: implementações de produção rápidas e confiáveis tornam as mudanças na produção mais fáceis de ver. Como resultado, uma vez que um bug aparece, leva menos tempo para encontrar e corrigir esse bug. Construir confiabilidade no desenvolvimento permite que os desenvolvedores concentrem sua atenção no que realmente importa – a funcionalidade e o desempenho de seus softwares e sistemas.

A virtude do tédio

Ao contrário de quase tudo na vida, “chatô” é na verdade um atributo positivo quando se trata de software! Não queremos que nossos programas sejam espontâneos e interessantes; queremos que eles sigam o roteiro e cumpram previsivelmente seus objetivos de negócios.

Nas palavras do engenheiro do Google Robert Muth, “ao contrário de uma história de detetive, a falta de emoção, suspense e quebra-cabeças é, na verdade, uma propriedade desejável do código-fonte”. Surpresas na produção são os inimigos do SRE.

Como Fred Brooks sugere em seu ensaio “No Silver Bullet” [Bro95], é muito importante considerar a diferença entre complexidade essencial e complexidade acidental.

A complexidade essencial é a complexidade inerente a uma determinada situação que não pode ser removida de uma definição de problema, enquanto a complexidade acidental é mais fluida e pode ser resolvida com esforço de engenharia. Por exemplo, escrever um servidor web envolve lidar com a complexidade essencial de servir páginas web rapidamente. No entanto, se escrevermos um servidor web em Java, podemos introduzir complexidade acidental ao tentar minimizar o impacto no desempenho da coleta de lixo.

Com o objetivo de minimizar a complexidade acidental, as equipes de SRE devem:

- Recuar quando a complexidade acidental é introduzida nos sistemas pelos quais eles são responsáveis • Esforçar-se constantemente para eliminar a complexidade nos sistemas que integram e pelos quais assumem responsabilidade operacional

Eu não vou desistir do meu código!

Como os engenheiros são seres humanos que muitas vezes formam uma ligação emocional com suas criações, confrontos sobre expurgos em grande escala da árvore de origem não são incomuns. Alguns podem protestar: “E se precisarmos desse código mais tarde?” “Por que não comentamos o código para que possamos adicioná-lo facilmente mais tarde?” ou “Por que não bloqueamos o código com um sinalizador em vez de excluí-lo?” Estas são todas sugestões terríveis. Fonte

sistemas de controle facilitam a reversão de alterações, enquanto centenas de linhas de código comentado criam distrações e confusão (especialmente porque os arquivos de origem continuam a evoluir), e código que nunca é executado, fechado por um sinalizador que está sempre desabilitado, é uma bomba-relógio metafórica esperando para explodir, dolorosamente vivenciada pela Knight Capital, por exemplo (ver “Order In the Matter of Knight Capital Americas LLC” [Sec13]).

Correndo o risco de parecer extremo, quando você considera um serviço da Web que deve estar disponível 24 horas por dia, 7 dias por semana, até certo ponto, cada nova linha de código escrita é um passivo. O SRE promove práticas que tornam mais provável que todo código tenha um propósito essencial, como examinar o código para garantir que ele realmente conduza os objetivos de negócios, remover rotineiramente código morto e incorporar detecção de inchaço em todos os níveis de teste.

A métrica "Linhas de código negativas"

O termo “software bloat” foi cunhado para descrever a tendência do software de se tornar mais lento e maior ao longo do tempo como resultado de um fluxo constante de recursos adicionais.

Embora o software inchado pareça intuitivamente indesejável, seus aspectos negativos se tornam ainda mais claros quando considerados da perspectiva do SRE: cada linha de código alterada ou adicionada a um projeto cria o potencial para a introdução de novos defeitos e bugs. Um projeto menor é mais fácil de entender, mais fácil de testar e frequentemente tem menos defeitos. Tendo essa perspectiva em mente, talvez devêssemos ter reservas quando temos o desejo de adicionar novos recursos a um projeto.

Algumas das codificações mais satisfatórias que já fiz foram deletar milhares de linhas de código em um momento em que não era mais útil.

APIs mínimas

O poeta francês Antoine de Saint Exupery escreveu: “A perfeição é finalmente alcançada não quando não há mais a acrescentar, mas quando não há mais nada a tirar”.

[Sai39]. Este princípio também é aplicável ao projeto e construção de software.

As APIs são uma expressão particularmente clara de por que essa regra deve ser seguida.

Escrever APIs claras e mínimas é um aspecto essencial do gerenciamento da simplicidade em um sistema de software. Quanto menos métodos e argumentos fornecermos aos consumidores da API, mais fácil será a compreensão da API e mais esforço poderemos dedicar para tornar esses métodos tão bons quanto possível. Mais uma vez, um tema recorrente aparece: a decisão consciente de não assumir certos problemas nos permite focar em nosso problema central e tornar as soluções que explicitamente propusemos criar substancialmente melhores. Em software, menos é mais! Uma API pequena e simples geralmente também é uma marca registrada de um problema bem compreendido.

Modularidade

Expandindo a partir de APIs e binários únicos, muitas das regras básicas que se aplicam à programação orientada a objetos também se aplicam ao projeto de sistemas distribuídos.

A capacidade de fazer alterações em partes do sistema isoladamente é essencial para criar um sistema suportável. Especificamente, o baixo acoplamento entre binários, ou entre binários e configuração, é um padrão de simplicidade que simultaneamente promove a agilidade do desenvolvedor e a estabilidade do sistema. Se um bug for descoberto em um programa que é um componente de um sistema maior, esse bug pode ser corrigido e enviado para produção independente do resto do sistema.

Embora a modularidade que as APIs oferecem possa parecer simples, não é tão aparente que a noção de modularidade também se estenda à forma como as mudanças nas APIs são introduzidas.

Apenas uma única alteração em uma API pode forçar os desenvolvedores a reconstruir todo o seu sistema e correr o risco de introduzir novos bugs. As APIs de controle de versão permitem que os desenvolvedores continuem a usar a versão da qual seu sistema depende enquanto atualizam para uma versão mais recente de maneira segura e ponderada. A cadência de lançamento pode variar em todo o sistema, em vez de exigir um impulso de produção completo de todo o sistema toda vez que um recurso é adicionado ou aprimorado.

À medida que um sistema se torna mais complexo, a separação de responsabilidades entre APIs e entre binários se torna cada vez mais importante. Esta é uma analogia direta com o design de classes orientadas a objetos: assim como é entendido que é uma má prática escrever uma classe “grab bag” que contém funções não relacionadas, também é uma má prática criar e colocar em produção uma classe “util” ou binário “misc”. Um sistema distribuído bem projetado consiste em colaboradores, cada um com um propósito claro e bem definido.

O conceito de modularidade também se aplica aos formatos de dados. Um dos pontos fortes e objetivos de design dos buffers de protocolo do Google³ era criar um formato de fio que fosse compatível com versões anteriores e posteriores.

Liberar Simplicidade

Lançamentos simples são geralmente melhores do que lançamentos complicados. É muito mais fácil medir e entender o impacto de uma única mudança do que de um lote de mudanças lançadas simultaneamente. Se lançarmos 100 alterações não relacionadas a um sistema ao mesmo tempo e o desempenho piorar, entender quais alterações afetaram o desempenho e como elas fizeram isso exigirá um esforço considerável ou instrumentação adicional. Se a liberação for realizada em lotes menores, podemos avançar mais rápido com mais

³ Os buffers de protocolo, também chamados de “protobufs”, são um mecanismo extensível de plataforma neutra e neutra para serialização de dados estruturados. Para obter mais detalhes, consulte <https://developers.google.com/protocol-buffers/docs/overview#a-bit-of-history>.

confiança porque cada mudança de código pode ser entendida isoladamente no sistema maior. Essa abordagem de lançamentos pode ser comparada ao gradiente descendente no aprendizado de máquina, no qual encontramos uma solução ótima dando pequenos passos de cada vez e considerando se cada mudança resulta em uma melhoria ou degradação.

Uma simples conclusão

Este capítulo repetiu um tema várias vezes: a simplicidade do software é um pré-requisito para a confiabilidade. Não estamos sendo preguiçosos quando consideramos como podemos simplificar cada etapa de uma determinada tarefa. Em vez disso, estamos esclarecendo o que realmente queremos realizar e como podemos fazê-lo mais facilmente. Toda vez que dizemos “não” a um recurso, não estamos restringindo a inovação; estamos mantendo o ambiente livre de distrações para que o foco permaneça diretamente na inovação e a engenharia real possa prosseguir.

PARTE III

Práticas

Simplificando, os SREs executam serviços – um conjunto de sistemas relacionados, operados para usuários, que podem ser internos ou externos – e são, em última análise, responsáveis pela integridade desses serviços. A operação bem-sucedida de um serviço envolve uma ampla gama de atividades: desenvolvimento de sistemas de monitoramento, capacidade de planejamento, resposta a incidentes, garantia de que as causas-raiz das interrupções sejam abordadas e assim por diante. Esta seção aborda a teoria e a prática da atividade diária de um SRE: construir e operar grandes sistemas de computação distribuída.

Podemos caracterizar a saúde de um serviço – da mesma forma que Abraham Maslow categorizou as necessidades humanas [Mas43] – desde os requisitos mais básicos necessários para que um sistema funcione como um serviço até os níveis mais altos de função – permitindo a autogestão, atualização e assumir o controle ativo da direção do serviço, em vez de combater incêndios de forma reativa. Esse entendimento é tão fundamental para a forma como avaliamos os serviços no Google que não foi desenvolvido explicitamente até que vários SREs do Google, incluindo nosso ex-colega Mikey Dickerson,¹ se juntaram temporariamente à cultura radicalmente diferente do governo dos Estados Unidos para ajudar com o lançamento do Healthcare.gov no final de 2013 e início de 2014: eles precisavam de uma maneira de explicar como aumentar a confiabilidade dos sistemas.

Usaremos essa hierarquia, ilustrada na [Figura III-1](#), para examinar os elementos que tornam um serviço confiável, do mais básico ao mais avançado.

¹ Mikey deixou o Google no verão de 2014 para se tornar o primeiro administrador do US Digital Service (<https://www.whitehouse.gov/digital/united-states-digital-service>), uma agência destinada (em parte) a trazer princípios e práticas de SRE para os sistemas de TI do governo dos EUA.

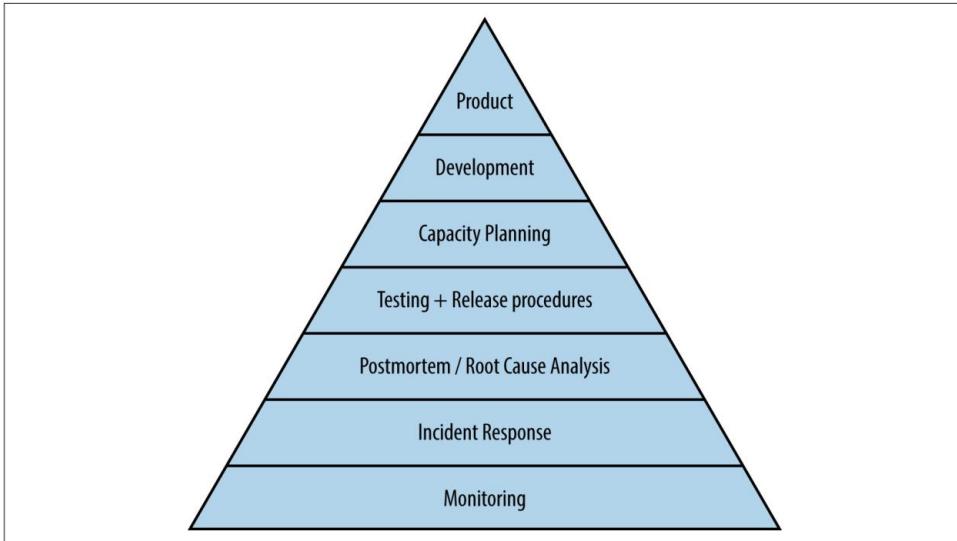


Figura III-1. Hierarquia de Confiabilidade do Serviço

Monitoramento

Sem monitoramento, você não tem como saber se o serviço está funcionando; sem uma infraestrutura de monitoramento cuidadosamente projetada, você está voando às cegas. Talvez todos que tentam usar o site recebam um erro, talvez não, mas você quer estar ciente dos problemas antes que seus usuários os percebam. Discutimos ferramentas e filosofia no [Capítulo 10, Alertas práticos de dados de séries temporais](#).

Resposta a incidentes

Os SREs não ficam de plantão apenas por causa disso: em vez disso, o suporte de plantão é uma ferramenta que usamos para alcançar nossa missão maior e permanecer em contato com a forma como os sistemas de computação distribuídos realmente funcionam (e falham!). Se pudéssemos encontrar uma maneira de nos aliviar de carregar um pager, fariímos. No [Capítulo 11, Ficando de plantão](#), explicamos como equilibrarmos os deveres de plantão com nossas outras responsabilidades.

Uma vez que você está ciente de que há um problema, como você faz com que ele desapareça? Isso não significa necessariamente corrigi-lo de uma vez por todas - talvez você possa parar o sangramento reduzindo a precisão do sistema ou desligando alguns recursos temporariamente, permitindo que ele se degrade normalmente, ou talvez você possa direcionar o tráfego para outra instância do servidor. vice que está funcionando corretamente. Os detalhes da solução que você escolhe implementar são necessariamente específicos para seu serviço e sua organização. Responder efetivamente a incidentes, no entanto, é algo aplicável a todas as equipes.

Descobrir o que está errado é o primeiro passo; oferecemos uma abordagem estruturada no [Capítulo 12, Solução de Problemas Eficaz](#).

Durante um incidente, muitas vezes é tentador ceder à adrenalina e começar a responder ad hoc. Aconselhamos contra essa tentação no [Capítulo 13, Resposta a Emergências](#), e aconselhamos no [Capítulo 14, Gerenciando Incidentes](#), que o gerenciamento eficaz de incidentes deve reduzir seu impacto e limitar a ansiedade induzida por interrupções.

Análise Postmortem e de Causa Raiz

Pretendemos ser alertados e resolver manualmente apenas problemas novos e interessantes apresentados pelo nosso serviço; é terrivelmente chato “consertar” o mesmo problema repetidamente. Na verdade, essa mentalidade é um dos principais diferenciais entre a filosofia SRE e alguns ambientes mais tradicionais focados em operações. Este tema é explorado em dois capítulos.

Construir uma cultura postmortem sem culpa é o primeiro passo para entender o que deu errado (e o que deu certo!), conforme descrito no [Capítulo 15, Cultura Postmortem: Aprendendo com o Fracasso](#).

Relacionado a essa discussão, no [Capítulo 16, Rastreando interrupções](#), descrevemos brevemente uma ferramenta interna, o rastreador de interrupções, que permite que as equipes de SRE acompanhem incidentes de produção recentes, suas causas e ações tomadas em resposta a eles.

Teste

Uma vez que entendemos o que tende a dar errado, nosso próximo passo é tentar evitá-lo, porque um grama de prevenção vale um quilo de cura. As suítes de teste oferecem alguma garantia de que nosso software não está cometendo certas classes de erros antes de ser lançado para produção; falamos sobre a melhor forma de usá-los no [Capítulo 17, Testando a Confiabilidade](#).

Planejamento de capacidade

No [Capítulo 18, Engenharia de Software em SRE](#), oferecemos um estudo de caso de engenharia de software em SRE com Auxon, uma ferramenta para automatizar o planejamento de capacidade.

Seguindo naturalmente o planejamento de capacidade, o balanceamento de carga garante que estamos usando adequadamente a capacidade que construímos. Discutimos como as solicitações aos nossos serviços são enviadas aos datacenters no [Capítulo 19, Balanceamento de carga no frontend](#). Em seguida, continuamos a discussão no [Capítulo 20, Balanceamento de carga no datacenter](#) e no [Capítulo 21, Lidando com sobrecarga](#), ambos essenciais para garantir a confiabilidade do serviço.

Finalmente, no [Capítulo 22, Lidando com Falhas em Cascata](#), oferecemos conselhos para lidar com falhas em cascata, tanto no projeto do sistema quanto no caso de seu serviço ser pego em uma falha em cascata.

Desenvolvimento

Um dos principais aspectos da abordagem do Google à Engenharia de confiabilidade do site é que fazemos um trabalho significativo de projeto de sistemas e engenharia de software em grande escala dentro da organização.

No [Capítulo 23, Gerenciando estado crítico: consenso distribuído para confiabilidade](#), explicamos o consenso distribuído, que (na forma de Paxos) está no centro de muitos sistemas distribuídos do Google, incluindo nosso sistema Cron distribuído globalmente. No [Capítulo 24, Agendamento periódico distribuído com Cron](#), descrevemos um sistema que pode ser dimensionado para datacenters inteiros e além, o que não é tarefa fácil.

O [Capítulo 25, Pipelines de Processamento de Dados](#), discute as várias formas que os pipelines de processamento de dados podem assumir: desde tarefas MapReduce únicas executadas periodicamente até sistemas que operam quase em tempo real. Diferentes arquiteturas podem levar a desafios surpreendentes e contra-intuitivos.

Certificar-se de que os dados armazenados ainda estejam lá quando você quiser lê-los é o coração da integridade dos dados; no [Capítulo 26, Integridade dos dados: o que você lê é o que você escreveu](#), explicamos como manter os dados seguros.

produtos

Finalmente, tendo subido na pirâmide de confiabilidade, chegamos ao ponto de ter um produto viável.

No [Capítulo 27, Lançamentos confiáveis de produtos em escala](#), escrevemos sobre como o Google faz lançamentos confiáveis de produtos em escala para tentar oferecer aos usuários a melhor experiência possível a partir do dia zero.

Leitura adicional do Google SRE

Conforme discutido anteriormente, o teste é útil e sua execução inadequada pode ter grandes efeitos na estabilidade geral. Em um artigo do ACM [\[Kri12\]](#), explicamos como o Google realiza testes de resiliência em toda a empresa para garantir que somos capazes de resistir ao inesperado caso ocorra um apocalipse zumbi ou outro desastre.

Embora muitas vezes seja considerado uma arte obscura, cheia de planilhas misticantes que adivinham o futuro, o planejamento de capacidade é vital e, como mostra [\[Hix15a\]](#), você não precisa de uma bola de cristal para fazer isso direito.

Por fim, uma abordagem interessante e nova para a segurança de redes corporativas é detalhada em [\[War14\]](#), uma iniciativa para substituir intranets privilegiadas por credenciais de dispositivos e usuários. Impulsionado por SREs no nível de infraestrutura, essa é definitivamente uma abordagem a ser lembrada ao criar sua próxima rede.

CAPÍTULO 10

Alerta prático de dados de séries temporais

**Escrito por Jamie Wilkinson
Editado por Kavita Giuliani**

Que as consultas fluam e o pager fique em silêncio.

—Bênção tradicional SRE

O monitoramento, a camada inferior da Hierarquia das Necessidades de Produção, é fundamental para a execução de um serviço estável. O monitoramento permite que os proprietários de serviços tomem decisões racionais sobre o impacto das mudanças no serviço, apliquem o método científico à resposta a incidentes e, claro, garantam a razão de sua existência: medir o alinhamento do serviço com as metas de negócios (consulte o [Capítulo 6](#)).).

Independentemente de um serviço ter ou não suporte SRE, ele deve ser executado em uma relação simbiótica com seu monitoramento. Mas tendo sido incumbidos da responsabilidade final pelo Google Production, os SREs desenvolvem um conhecimento particularmente íntimo da infraestrutura de monitoramento que suporta seu serviço.

Monitorar um sistema muito grande é desafiador por alguns motivos:

- O grande número de componentes sendo analisados • A necessidade de manter uma carga de manutenção razoavelmente baixa para os engenheiros responsáveis pelo sistema

Os sistemas de monitoramento do Google não medem apenas métricas simples, como o tempo médio de resposta de um servidor web europeu sem carga; também precisamos entender a distribuição desses tempos de resposta em todos os servidores da Web naquela região. Esse conhecimento nos permite identificar os fatores que contribuem para a cauda de latência.

Na escala em que nossos sistemas operam, ser alertado sobre falhas em uma única máquina é inaceitável porque esses dados são muito barulhentos para serem acionáveis. Em vez disso, tentamos construir sistemas robustos contra falhas nos sistemas dos quais eles dependem. Em vez de exigir o gerenciamento de muitos componentes individuais, um grande sistema deve ser projetado para agregar sinais e eliminar discrepâncias. Precisamos de sistemas de monitoramento que nos permitam alertar para objetivos de serviço de alto nível, mas que mantenham a granularidade para inspecionar componentes individuais conforme necessário.

Os sistemas de monitoramento do Google evoluíram ao longo de 10 anos do modelo tradicional de scripts personalizados que verificam respostas e alertas, totalmente separados da exibição visual de tendências, para um novo paradigma. Esse novo modelo tornou a coleção de séries temporais uma função de primeira classe do sistema de monitoramento e substituiu esses scripts de verificação por uma linguagem rica para manipular séries temporais em gráficos e alertas.

A Ascensão de Borgmon

Logo após a criação da infraestrutura de agendamento de tarefas Borg [Ver15] em 2003, um novo sistema de monitoramento – Borgmon – foi construído para complementá-la.

Monitoramento de séries temporais fora do Google

Este capítulo descreve a arquitetura e a interface de programação de uma ferramenta de monitoramento interno que foi fundamental para o crescimento e a confiabilidade do Google por quase 10 anos... mas como isso ajuda você, nosso caro leitor?

Nos últimos anos, o monitoramento passou por uma explosão cambriana: Riemann, Heka, Bosun e Prometheus surgiram como ferramentas de código aberto que são muito semelhantes aos alertas baseados em séries temporais de Borgmon. Em particular, Prometheus¹ compartilha muitas semelhanças com Borgmon, especialmente quando você compara as duas linguagens de regras. Os princípios de coleta de variáveis e avaliação de regras permanecem os mesmos em todas essas ferramentas e fornecem um ambiente com o qual você pode experimentar e, esperançosamente, lançar em produção as ideias inspiradas por este capítulo.

Em vez de executar scripts personalizados para detectar falhas do sistema, Borgmon conta com um formato comum de exposição de dados; isso permite a coleta de dados em massa com baixa sobrecarga e evita os custos de execução de subprocessos e configuração de conexão de rede.

Chamamos isso de monitoramento de caixa branca (consulte o [Capítulo 6](#) para uma comparação entre monitoramento de caixa branca e caixa preta).

¹ Prometheus é um sistema de monitoramento de código aberto e banco de dados de séries temporais disponível em <http://prometheus.io>.

Os dados são usados para renderizar gráficos e criar alertas, que são realizados usando aritmética simples. Como a coleta não é mais um processo de curta duração, o histórico dos dados coletados também pode ser usado para esse cálculo de alerta.

Esses recursos ajudam a atingir a meta de simplicidade descrita no [Capítulo 6](#). Eles permitem que a sobrecarga do sistema seja mantida baixa para que as pessoas que executam os serviços possam permanecer ágeis e responder às mudanças contínuas no sistema à medida que ele cresce.

Para facilitar a coleta em massa, o formato das métricas teve que ser padronizado. Um método mais antigo de exportação do estado interno (conhecido como varz) 2 foi ~~fornecido para permitir que os serviços anotem o destino em uma busca HTTP~~ destinado a permitir que os serviços anotem o destino em uma busca HTTP. Por exemplo, para visualizar uma página de métricas manualmente, você pode usar o seguinte comando:

```
% curl http://webserver:80/varz
http_requests 37 errors_total 12
```

Um Borgmon pode coletar de outro Borgmon,³ para que possamos construir hierarquias que sigam a topologia do serviço, agregando e resumindo informações e descartando algumas estrategicamente em cada nível. Normalmente, uma equipe executa um único Borgmon por cluster e um par no nível global. Alguns serviços muito grandes se fragmentam abaixo do nível do cluster em muitos Borgmons scraper, que por sua vez alimentam o Borgmon no nível do cluster.

Instrumentação de aplicativos

O manipulador HTTP /varz simplesmente lista todas as variáveis exportadas em texto simples, como chaves e valores separados por espaço, um por linha. Uma extensão posterior adicionou uma variável mapeada, que permite ao exportador definir vários rótulos em um nome de variável e, em seguida, exportar uma tabela de valores ou um histograma. Um exemplo de variável com valor de mapa se parece com o seguinte, mostrando 25 respostas HTTP 200 e 12 HTTP 500s:

```
http_responses map:code 200:25 404:0 500:12
```

Adicionar uma métrica a um programa requer apenas uma única declaração no código onde a métrica é necessária.

Em retrospectiva, é evidente que essa interface textual sem esquema torna a barreira para adicionar nova instrumentação muito baixa, o que é positivo tanto para a engenharia de software quanto para as equipes de SRE. No entanto, isso tem uma desvantagem em relação à manutenção contínua; o desacoplamento da definição de variável de seu uso nas regras de Borgmon requer cuidado.

² O Google nasceu nos EUA, então pronunciamos “var-zee”.

³ O plural de Borgmon é Borgmon, como carneiro.

gerenciamento de mudanças completo. Na prática, essa troca tem sido satisfatória porque as ferramentas para validar e gerar regras também foram escritas.⁴

Exportando variáveis As

raízes da web do Google são profundas: cada uma das principais linguagens usadas no Google tem uma implementação da interface de variável exportada que se registra automaticamente com o servidor HTTP embutido em cada binário do Google por padrão.⁵ As instâncias da variável a ser exportada permitir que o autor do servidor execute operações óbvias, como adicionar uma quantia ao valor atual, definir uma chave para um valor específico e assim por diante. A biblioteca Go expvar⁶ e seu formulário de saída JSON têm uma variante dessa API.

Coleta de dados exportados

Para encontrar seus destinos, uma instância do Borgmon é configurada com uma lista de destinos usando um dos muitos métodos de resolução de nomes.⁷ A lista de destinos geralmente é dinâmica, portanto, usar a descoberta de serviço reduz o custo de manutenção e permite que o monitoramento seja dimensionado.

Em intervalos predefinidos, o Borgmon busca o URI /varz em cada destino, decodifica os resultados e armazena os valores na memória. O Borgmon também distribui a coleção de cada instância na lista de destino por todo o intervalo, de modo que a coleta de cada destino não esteja em sintonia com seus pares.

Borgmon também registra variáveis “sintéticas” para cada alvo para identificar:

- Se o nome foi resolvido para um host e porta
- Se o destino respondeu a uma coleta
- Se o destino respondeu a uma verificação de integridade
- A que horas a coleta terminou

Essas variáveis sintéticas facilitam a gravação de regras para detectar se as tarefas monitoradas estão indisponíveis.

⁴ Muitas equipes não SRE usam um gerador para eliminar o clichê inicial e as atualizações contínuas e encontrar o gerador muito mais fácil de usar (embora menos poderoso) do que editar diretamente as regras.

⁵ Muitos outros aplicativos também usam seu protocolo de serviço para exportar seu estado interno. O OpenLDAP o exporta através da subárvore cn=Monitor ; O MySQL pode relatar o estado com uma consulta SHOW VARIABLES ; O Apache tem seu manipulador mod_status .

⁶ <https://golang.org/pkg/expvar/>

⁷ O Borg Name System (BNS) é descrito no [Capítulo 2](#).

É interessante que varz seja bastante diferente do SNMP (Simple Networking Monitoring Protocol), que “é projetado [...] para ter requisitos mínimos de transporte e continuar funcionando quando a maioria dos outros aplicativos de rede falham” [Mic03]. A raspagem de alvos sobre HTTP parece estar em desacordo com este princípio de design; no entanto, a experiência mostra que isso raramente é um problema.⁸ O próprio sistema já foi projetado para ser robusto contra falhas de rede e de máquina, e a Borgmon permite que os engenheiros escrevam regras de alerta mais inteligentes usando a própria falha de coleta como um sinal.

Armazenamento na Arena de Séries Temporais

Um serviço é normalmente composto de muitos binários executando tantas tarefas, em muitas máquinas, em muitos clusters. A Borgmon precisa manter todos esses dados organizados, ao mesmo tempo em que permite consultas e fatias flexíveis desses dados.

Borgmon armazena todos os dados em um banco de dados na memória, regularmente verificado no disco. Os pontos de dados têm a forma (timestamp, value) e são armazenados em listas cronológicas chamadas séries temporais, e cada série temporal é nomeada por um conjunto único de rótulos, do formulário nome=valor.

Conforme apresentado na Figura 10-1, uma série temporal é conceitualmente uma matriz de números unidimensional, progredindo ao longo do tempo. À medida que você adiciona permutações de rótulos a essa série temporal, a matriz se torna multidimensional.

"http_requests"	:	:	:	:	:	:	:	:	:	:
	0	0	0	0	0	0	0	0	0	0
:	0	0	0	0	0	0	0	0	0	0
now - 2Δt	0	0	0	0	0	0	0	0	0	0
now - Δt	0	0	0	0	0	0	0	0	0	0
now	0	0	0	0	0	0	0	0	0	0
	host1	host2	host3	host4	host5	...				

Figura 10-1. Uma série temporal para erros rotulados pelo host original foi coletada de

Na prática, a estrutura é um bloco de memória de tamanho fixo, conhecido como arena de séries temporais, com um coletor de lixo que expira as entradas mais antigas assim que a arena estiver cheia. O intervalo de tempo entre as entradas mais recentes e mais antigas na arena é o horizonte, que indica quantos dados que podem ser consultados são mantidos na RAM. Normalmente, o datacenter

⁸ Relembre no [Capítulo 6](#) a distinção entre alertar sobre os sintomas e sobre as causas.

e Borgmon global são dimensionados para armazenar cerca de 12 horas de dados⁹ para renderização de consoles, e muito menos tempo se forem os fragmentos de coletor de nível mais baixo. O requisito de memória para um único ponto de dados é de cerca de 24 bytes, então podemos ajustar 1 milhão de séries temporais únicas por 12 horas em intervalos de 1 minuto em menos de 17 GB de RAM.

Periodicamente, o estado na memória é arquivado em um sistema externo conhecido como Time-Series Database (TSDB). O Borgmon pode consultar o TSDB para dados mais antigos e, embora mais lento, o TSDB é mais barato e maior que a RAM do Borgmon.

Etiquetas e Vetores

Conforme mostrado no exemplo de série temporal na Figura 10-2, as séries temporais são armazenadas como sequências de números e carimbos de data/hora, que são referidos como vetores. Como vetores em álgebra linear, esses vetores são fatias e seções transversais da matriz multidimensional de pontos de dados na arena. Conceitualmente, os timestamps podem ser ignorados, porque os valores são inseridos no vetor em intervalos regulares no tempo – por exemplo, 1 ou 10 segundos ou 1 minuto de intervalo.

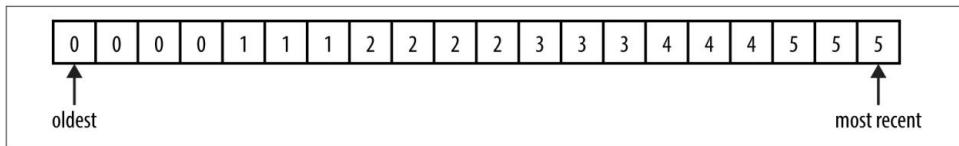


Figura 10-2. Um exemplo de série temporal

O nome de uma série temporal é um conjunto de rótulos, porque é implementado como um conjunto de rótulos expressos como pares chave=valor . Um desses rótulos é o próprio nome da variável, a chave que aparece na página varz.

Alguns nomes de rótulos são declarados como importantes. Para que a série temporal no banco de dados de série temporal seja identificável, ela deve ter, no mínimo, os seguintes rótulos:

var

O nome da variável

trabalho

O nome dado ao tipo de servidor que está sendo monitorado

serviço

Uma coleção vagamente definida de trabalhos que fornecem um serviço aos usuários, internos ou externos

⁹ Esse horizonte de 12 horas é um número mágico que visa ter informações suficientes para depurar um incidente na RAM para consultas rápidas sem custar muita RAM.

zona

Uma convenção do Google que se refere ao local (normalmente o datacenter) do Borgmon que realizou a coleta dessa variável

Juntas, essas variáveis aparecem como o seguinte, chamado de expressão de variável:

```
{var=http_requests,job=webserver,instance=host0:80,service=web,zone=us-west}
```

Uma consulta por uma série temporal não requer a especificação de todos esses rótulos e uma pesquisa por um conjunto de rótulos retorna todas as séries temporais correspondentes em um vetor. Assim, poderíamos retornar um vetor de resultados removendo o rótulo da instância na consulta anterior, caso houvesse mais de uma instância no cluster. Por exemplo:

```
{var=http_requests,job=webserver,service=web,zone=us-west}
```

pode ter um resultado de cinco linhas em um vetor, com o valor mais recente na série temporal assim:

```
{var=http_requests,job=webserver,instance=host0:80,service=web,zone=us-west} 10
{var=http_requests,job=webserver,instance=host1:80,service=web,zone=us-west} 9
{var=http_requests,job=webserver,instance=host2:80,service=web,zone=us-west} 11
{var=http_requests,job=webserver,instance=host3:80,service=web,zone=us -west} 0
{var=http_requests,job=webserver,instance=host4:80,service=web,zone=us-west} 10
```

Rótulos podem ser adicionados a uma série temporal de:

- O nome do destino, por exemplo, o trabalho e a instância
- O próprio destino, por exemplo, por meio de variáveis com valor de mapa
- A configuração de Borgmon, por exemplo, anotações sobre localização ou reetiquetagem
- As regras de Borgmon sendo avaliadas

Também podemos consultar séries temporais no tempo, especificando uma duração para a variável expressão:

```
{var=http_requests,job=webserver,service=web,zone=us-west}[10m]
```

Isso retorna os últimos 10 minutos do histórico da série temporal que corresponde à expressão. Se estivéssemos coletando pontos de dados uma vez por minuto, esperaríamos retornar 10 pontos de dados em uma janela de 10 minutos, assim:¹⁰

```
{var=http_requests,job=webserver,instance=host0:80, ...} 0 1 2 3 4 5 6 7 8 9 10
{var=http_requests,job=webserver,instance=host1:80, ...} 0 1 2 3 4 4 5 6 7 8 9
{var=http_requests,job=webserver,instance=host2:80, ...} 0 1 2 3 5 6 7 8 9 9 11
{var=http_requests,job=webserver,instance =host3:80, ...} 0 0 0 0 0 0 0 0 0 0 0
{var=http_requests,job=webserver,instance=host4:80, ...} 0 1 2 3 4 5 6 7 8 9 10
```

¹⁰ Os rótulos de serviço e zona são eliminados aqui por espaço, mas estão presentes na expressão retornada.

Avaliação de regras

Borgmon é realmente apenas uma calculadora programável, com algum açúcar sintático que permite gerar alertas. Os componentes de coleta e armazenamento de dados já descritos são apenas maiores necessários para tornar essa calculadora programável finalmente adequada ao propósito aqui como um sistema de monitoramento. :)



Centralizar a avaliação da regra em um sistema de monitoramento, em vez de delegá-la a subprocessos bifurcados, significa que os cálculos podem ser executados em paralelo em muitos destinos semelhantes. Essa prática mantém a configuração relativamente pequena em tamanho (por exemplo, removendo a duplicação de código) ainda mais poderosa por meio de sua expressividade.

O código do programa Borgmon, também conhecido como regras de Borgmon, consiste em expressões algébricas simples que computam séries temporais de outras séries temporais. Essas regras podem ser bastante poderosas porque podem consultar o histórico de uma única série temporal (ou seja, o eixo do tempo), consultar diferentes subconjuntos de rótulos de muitas séries temporais de uma só vez (ou seja, o eixo do espaço) e aplicar muitas operações.

As regras são executadas em um conjunto de encadeamentos paralelo sempre que possível, mas dependem da ordenação ao usar regras definidas anteriormente como entrada. O tamanho dos vetores retornados por suas expressões de consulta também determina o tempo de execução geral de uma regra. Assim, normalmente é possível adicionar recursos de CPU a uma tarefa Borgmon em resposta à sua lentidão. Para auxiliar na análise mais detalhada, são exportadas métricas internas sobre o tempo de execução das regras para depuração de desempenho e monitoramento do monitoramento.

A agregação é a base da avaliação de regras em um ambiente distribuído.

A agregação envolve tomar a soma de um conjunto de séries temporais das tarefas em um trabalho para tratar o trabalho como um todo. A partir dessas somas, as taxas globais podem ser calculadas. Por exemplo, a taxa total de consultas por segundo de um trabalho em um datacenter é a soma de todas as taxas de alteração¹¹ de todos os contadores de consulta.¹²

¹¹ Calcular a soma das taxas em vez da taxa de somas defende o resultado contra reinicializações do contador ou falta de dados, talvez devido a uma reinicialização de tarefa ou falha na coleta de dados.

¹² Apesar de não digitadas, a maioria das varz são contadores simples. A função de taxa de Borgmon lida com todas as maiores casos de reinicializações do contador.



Um contador é qualquer variável não monotonicamente decrescente – o que significa que os contadores só aumentam de valor. Os medidores, por outro lado, podem assumir qualquer valor que desejarem. Os contadores medem valores crescentes, como o número total de quilômetros percorridos, enquanto os medidores mostram o estado atual, como a quantidade de combustível restante ou a velocidade atual. Ao coletar dados no estilo Borgmon, é melhor usar contadores, porque eles não perdem o significado quando os eventos ocorrem entre os intervalos de amostragem. Caso ocorra alguma atividade ou mudança entre os intervalos de amostragem, é provável que uma coleta de medidores perca essa atividade.

Para um exemplo de servidor web, podemos querer alertar quando nosso cluster de servidor web começar a servir mais erros como porcentagem de solicitações do que achamos normal – ou mais tecnicamente, quando a soma das taxas de não HTTP-200 códigos de retorno em todas as tarefas no cluster, dividido pela soma das taxas de solicitações para todas as tarefas nesse cluster, é maior que algum valor.

Isso é realizado por:

1. Agregando as taxas de códigos de resposta em todas as tarefas, gerando um vetor de taxas naquele momento, uma para cada código.
2. Calcular a taxa de erro total como a soma desse vetor, gerando um único valor para o cluster naquele momento. Essa taxa de erro total exclui o código 200 da soma, porque não é um erro.
3. Calcular a proporção de erros para solicitações em todo o cluster, dividindo a taxa de erro total pela taxa de solicitações que chegaram e, novamente, gerando um único valor para o cluster naquele momento.

Cada uma dessas saídas em um ponto no tempo é anexada à sua expressão de variável nomeada, que cria a nova série temporal. Como resultado, poderemos inspecionar o histórico de taxas de erro e taxas de erro em outra ocasião.

As regras de taxa de solicitações seriam escritas na linguagem de regras de Borgmon da seguinte forma:

```
regras <<<
# Calcula a taxa de solicitações para cada tarefa a partir da contagem de solicitações
{var=task:http_requests:rate10m,job=webserver} = rate({var=http_requests,job=webserver}{10m});

# Soma as taxas para obter a taxa agregada de consultas para o cluster; # 'sem instância'
instrui o Borgmon a remover o rótulo da instância # do lado direito.
{var=dc:http_requests:rate10m,job=webserver} =

    soma sem instância({var=task:http_requests:rate10m,job=webserver})
>>>
```

A função `rate()` pega a expressão incluída e retorna o delta total dividido pelo tempo total entre os valores mais antigos e mais recentes.

Com os dados de série temporal de exemplo da consulta anterior, os resultados para a regra `task:http_requests:rate10m seriam:13`

```
{var=task:http_requests:rate10m,job=webserver,instance=host0:80, ...} 1
{var=task:http_requests:rate10m,job=webserver,instance=host2:80, ...} 0,9 {var
=task:http_requests:rate10m,job=webserver,instance=host3:80, ...} 1,1
{var=task:http_requests:rate10m,job=webserver,instance=host4:80, ...} 0
{var=task :http_requests:rate10m,job=webserver,instance=host5:80, ...} 1
```

e os resultados para a regra `dc:http_requests:rate10m seriam:`

```
{var=dc:http_requests:rate10m,job=webserver,service=web,zone=us-west} 4
```

porque a segunda regra usa a primeira como entrada.



O rótulo da instância está ausente na saída agora, descartado pela regra de agregação. Se tivesse permanecido na regra, Borgmon não seria capaz de somar as cinco linhas.

Nesses exemplos, usamos uma janela de tempo porque estamos lidando com pontos discretos na série temporal, em oposição a funções contínuas. Fazer isso torna o cálculo da taxa mais fácil do que realizar o cálculo, mas significa que para calcular uma taxa, precisamos selecionar um número suficiente de pontos de dados. Também temos que lidar com a possibilidade de que algumas coleções recentes tenham falhado. Lembre-se de que a notação de expressão de variável histórica usa o intervalo [10m] para evitar pontos de dados perdidos causados por erros de coleta.

O exemplo também usa uma convenção do Google que ajuda na legibilidade. Cada nome de variável calculado contém um trio separado por dois pontos indicando o nível de agregação, o nome da variável e a operação que criou esse nome. Neste exemplo, as variáveis do lado esquerdo são “taxa de 10 minutos de solicitações HTTP de tarefa” e “taxa de 10 minutos de solicitações HTTP de datacenter”.

Agora que sabemos como criar uma taxa de consultas, podemos construir sobre isso para também calcular uma taxa de erros e, então, podemos calcular a proporção de respostas a solicitações para entender quanto trabalho útil o serviço está fazendo. Podemos comparar a taxa de taxa de erros com nosso objetivo de nível de serviço (consulte o [Capítulo 4](#)) e alertar se esse objetivo for perdido ou estiver em risco de ser perdido:

13 Os rótulos de serviço e zona são omitidos por espaço.

```

regras <<<
# Calcula uma taxa por tarefa e por rótulo de
'código' {var=task:http_responses:rate10m,job=webserver} =
    taxa por código({var=http_responses,job=webserver}[10m]);

# Calcula uma taxa de resposta de nível de cluster por rótulo de
'código' {var=dc:http_responses:rate10m,job=webserver} =
    soma sem instância({var=task:http_responses:rate10m,job=webserver});

# Calcula uma nova taxa de nível de cluster somando todos os códigos não 200
{var=dc:http_errors:rate10m,job=webserver} = soma sem
    código( {var=dc:http_responses:rate10m,job=webserver,code!=/200/};

# Calcula a proporção da taxa de erros para a taxa de solicitações
{var=dc:http_errors:ratio_rate10m,job=webserver} = {var=dc:http_errors:rate10m,job=webserver}

/
{var=dc:http_requests:rate10m,job=webserver};
>>>

```

Novamente, esse cálculo demonstra a convenção de sufixar o novo nome da variável de série temporal com a operação que a criou. Esse resultado é lido como “erros HTTP do datacenter em 10 minutos de taxa”.

A saída dessas regras pode se parecer com:¹⁴

```

{var=task:http_responses:rate10m,job=webserver}

{var=task:http_responses:rate10m,job=webserver,code=200,instance=host0:80, ...} 1
{var=task:http_responses:rate10m,job=webserver,code=500,instance=host0:80, ...} 0
{var=task:http_responses:rate10m,job=webserver,code=200,instance=host1:80, ...} 0,5
{var=task:http_responses:rate10m,job=webserver,code=500,instance=host1:80, ...} 0,4
{var=task:http_responses:rate10m,job=webserver,code=200,instance=host2:80, ...} 1
{var=task:http_responses:rate10m,job=webserver,code=500,instance=host2:80, ...} 0,1
{var=task:http_responses:rate10m,job=webserver,code=200,instance=host3:80, ...} 0
{var=task:http_responses:rate10m,job=webserver,code=500,instance=host3:80, ...} 0
{var=task:http_responses:rate10m,job=webserver,code=200,instance=host4:80, ...} 0,9
{var=task:http_responses:rate10m,job=webserver,code=500,instance=host4:80, ...} 0,1

{var=dc:http_responses:rate10m,job=webserver}

{var=dc:http_responses:rate10m,job=webserver,code=200, ...} 3,4
{var=dc:http_responses:rate10m,job=webserver,code=500, ...} 0,6

{var=dc:http_responses:rate10m,job=webserver,code=/!200/}

{var=dc:http_responses:rate10m,job=webserver,code=500, ...} 0,6

{var=dc:http_errors:rate10m,job=webserver}

{var=dc:http_errors:rate10m,job=webserver, ...} 0,6

```

¹⁴ Os rótulos de serviço e zona são omitidos por espaço.

```
{var=dc:http_errors:ratio_rate10m,job=webserver}
```

```
{var=dc:http_errors:ratio_rate10m,job=webserver} 0,15
```



A saída anterior mostra a consulta intermediária na regra dc:http_errors:rate10m que filtra os códigos de erro não 200. Embora o valor das expressões seja o mesmo, observe que o rótulo do código é retido em uma, mas removido da outra.

Como mencionado anteriormente, as regras de Borgmon criam novas séries temporais, de modo que os resultados dos cálculos são mantidos na arena das séries temporais e podem ser inspecionados da mesma forma que as séries temporais de origem. A capacidade de fazer isso permite consultas ad hoc, avaliação e exploração como tabelas ou gráficos. Esse é um recurso útil para depuração enquanto estiver de plantão e, se essas consultas ad hoc forem úteis, elas podem se tornar visualizações permanentes em um console de serviço.

Alerta

Quando uma regra de alerta é avaliada por um Borgmon, o resultado é verdadeiro, caso em que o alerta é acionado, ou falso. A experiência mostra que os alertas podem “flap” (alternar seu estado rapidamente); portanto, as regras permitem uma duração mínima para a qual a regra de alerta deve ser verdadeira antes que o alerta seja enviado. Normalmente, essa duração é definida para pelo menos dois ciclos de avaliação de regras para garantir que nenhuma coleta perdida cause um alerta falso.

O exemplo a seguir cria um alerta quando a taxa de erros acima de 10 minutos excede 1% e o número total de erros excede 1:

```
regras <<<
{var=dc:http_errors:ratio_rate10m,job=webserver} > 0.01 e por trabalho, erro

{var=dc:http_errors:rate10m,job=webserver} > 1
por 2m
=> ErrorRatioTooHigh
detalha "taxa de erro do servidor web em [[trigger_value]]" rótulos
{severity=page};

>>>
```

Nosso exemplo mantém a taxa de proporção em 0,15, que está bem acima do limite de 0,01 na regra de alerta. No entanto, o número de erros não é maior que 1 neste momento, então o alerta não estará ativo. Quando o número de erros exceder 1, o alerta ficará pendente por dois minutos para garantir que não seja um estado transitório e só então será acionado.

A regra de alerta contém um pequeno modelo para preencher uma mensagem contendo informações contextuais: para qual trabalho o alerta se destina, o nome do alerta, o valor numérico da regra de disparo e assim por diante. As informações contextuais são preenchidas pelo Borgmon quando o alerta é acionado e são enviadas no Alert RPC.

O Borgmon está conectado a um serviço executado centralmente, conhecido como Alertmanager, que recebe RPCs de alerta quando a regra é acionada pela primeira vez e, novamente, quando o alerta é considerado “disparado”. O Alertmanager é responsável por encaminhar a notificação de alerta para o destino correto. O Alertmanager pode ser configurado para fazer o seguinte:

- Inibe certos alertas quando outros estão ativos
- Deduplicar alertas de vários Borgmon que possuem os mesmos conjuntos de rótulos
- Alertas de fan-in ou fan-out com base em seus conjuntos de rótulos quando vários alertas com fogo de etiquetas

Conforme descrito no [Capítulo 6](#), as equipes enviam seus alertas dignos de página para sua rotação de plantão e seus alertas importantes, mas subcríticos, para suas filas de tickets. Todos os outros alertas devem ser mantidos como dados informativos para painéis de status.

Um guia mais abrangente para o design de alertas pode ser encontrado no [Capítulo 4](#).

Dividindo a topologia de monitoramento

Um Borgmon também pode importar dados de séries temporais de outros Borgmon. Embora se possa tentar coletar de todas as tarefas em um serviço globalmente, fazer isso rapidamente se torna um gargalo de escala e introduz um único ponto de falha no design. Em vez disso, um protocolo de streaming é usado para transmitir dados de séries temporais entre Borgmon, economizando tempo de CPU e bytes de rede em comparação com o formato varz baseado em texto. Uma implantação típica desse tipo usa dois ou mais Borgmon globais para agregação de nível superior e um Borgmon em cada datacenter para monitorar todos os trabalhos executados nesse local. (O Google divide a rede de produção em zonas para alterações de produção, portanto, ter duas ou mais réplicas globais oferece diversidade em caso de manutenção e interrupções para esse ponto único de falha.)

Conforme mostrado na [Figura 10-3](#), implantações mais complicadas fragmentam ainda mais o datacenter Borgmon em uma camada puramente de raspagem (geralmente devido a restrições de RAM e CPU em um único Borgmon para serviços muito grandes) e uma camada de agregação DC que executa principalmente avaliação de regras para agregação. Às vezes, a camada global é dividida entre avaliação de regras e painéis. O Borgmon de nível superior pode filtrar os dados que deseja transmitir do Borgmon de nível inferior, para que o Borgmon global não preencha sua arena com todas as séries temporais por tarefa dos níveis inferiores. Assim, a hierarquia de agregação constrói caches locais de séries temporais relevantes que podem ser detalhadas quando necessário.

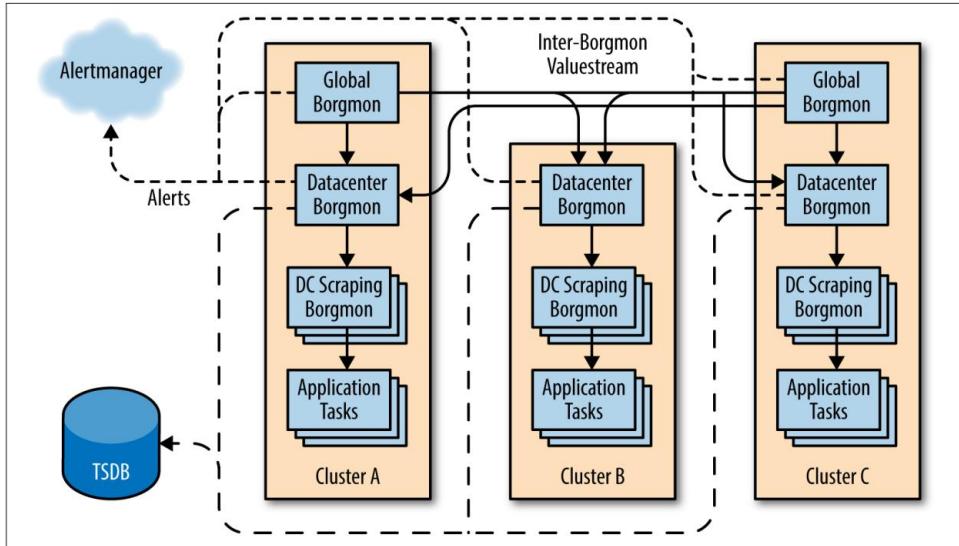


Figura 10-3. Um modelo de fluxo de dados de uma hierarquia de Borgmon em três clusters

Monitoramento de caixa preta

Borgmon é um sistema de monitoramento de caixa branca – ele inspeciona o estado interno do serviço de destino e as regras são escritas tendo em mente o conhecimento interno. A natureza transparente desse modelo fornece grande poder para identificar rapidamente quais componentes estão falhando, quais filas estão cheias e onde ocorrem gargalos, tanto ao responder a um incidente quanto ao testar a implantação de um novo recurso.

No entanto, o monitoramento de caixa branca não fornece uma visão completa do sistema que está sendo monitorado; confiar apenas no monitoramento de caixa branca significa que você não está ciente do que os usuários veem. Você vê apenas as consultas que chegam ao destino; as consultas que nunca são feitas devido a um erro de DNS são invisíveis, enquanto as consultas perdidas devido a uma falha do servidor nunca emitem som. Você só pode alertar sobre as falhas que esperava.

As equipes do Google resolvem esse problema de cobertura com o Prober, que executa uma verificação de protocolo em relação a um destino e relata sucesso ou falha. O prober pode enviar alertas diretamente para o Alertmanager, ou seu próprio varz pode ser coletado por um Borgmon. O Prober pode validar a carga útil de resposta do protocolo (por exemplo, o conteúdo HTML de uma resposta HTTP) e validar se o conteúdo é esperado e até extrair e exportar valores como séries temporais. As equipes costumam usar o Prober para exportar histogramas de tempos de resposta por tipo de operação e tamanho da carga útil, para que possam dividir o desempenho visível ao usuário.

O Prober é um híbrido do modelo check-and-test com algumas extrações de variáveis mais ricas para criar séries temporais.

O Prober pode ser apontado para o domínio front-end ou por trás do balanceador de carga. Ao usar os dois destinos, podemos detectar falhas localizadas e suprimir alertas. Por exemplo, podemos monitorar o balanceamento de carga www.google.com e os servidores da Web em cada datacenter atrás do balanceador de carga. Essa configuração nos permite saber se o tráfego ainda é servido quando um datacenter falha ou isolar rapidamente uma borda no gráfico de fluxo de tráfego onde ocorreu uma falha.

Mantendo a configuração

A configuração do Borgmon separa a definição das regras dos alvos que estão sendo monitorados. Isso significa que os mesmos conjuntos de regras podem ser aplicados a muitos destinos de uma só vez, em vez de escrever configurações quase idênticas repetidamente. Essa separação de interesses pode parecer incidental, mas reduz bastante o custo de manutenção do monitoramento, evitando muitas repetições na descrição dos sistemas de destino.

Borgmon também suporta modelos de linguagem. Esse sistema semelhante a macro permite que os engenheiros construam bibliotecas de regras que podem ser reutilizadas. Esta funcionalidade reduz novamente a repetição, reduzindo assim a probabilidade de erros na configuração.

É claro que qualquer ambiente de programação de alto nível cria a oportunidade de complexidade, então Borgmon fornece uma maneira de construir extensos testes de unidade e regressão sintetizando dados de séries temporais, a fim de garantir que as regras se comportem como o autor pensa. A equipe de Monitoramento de Produção executa um serviço de integração contínua que executa um conjunto desses testes, empacota a configuração e envia a configuração para todos os Borgmon em produção, que validam a configuração antes de aceitá-la.

Na vasta biblioteca de modelos comuns que foram criados, surgiram duas classes de configuração de monitoramento. A primeira classe simplesmente codifica o esquema emergente de variáveis exportadas de uma determinada biblioteca de código, de forma que qualquer usuário da biblioteca possa reutilizar o modelo de sua vez. Esses modelos existem para a biblioteca do servidor HTTP, alocação de memória, biblioteca do cliente de armazenamento e serviços RPC genéricos, entre outros. (Enquanto a interface varz não declara nenhum esquema, a biblioteca de regras associada à biblioteca de código acaba declarando um esquema.)

A segunda classe de biblioteca surgiu à medida que criamos modelos para gerenciar a agregação de dados de uma tarefa de servidor único para o espaço de serviço global. Essas bibliotecas contêm regras de agregação genéricas para variáveis exportadas que os engenheiros podem usar para modelar a topologia de seu serviço.

Por exemplo, um serviço pode fornecer uma única API global, mas estar hospedado em muitos datacenters. Dentro de cada datacenter, o serviço é composto por vários shards e cada shard é composto por vários trabalhos com vários números de tarefas. Um engenheiro pode modelar essa divisão com regras Borgmon para que, durante a depuração, os subcomponentes possam ser isolados do resto do sistema. Esses agrupamentos geralmente seguem o destino compartilhado dos componentes; Por exemplo, tarefas individuais compartilham destino devido a arquivos de configuração, trabalhos em um shard compartilham destino porque estão hospedados no mesmo datacenter e sites físicos compartilham destino devido à rede.

As convenções de rotulagem tornam essa divisão possível: um Borgmon adiciona rótulos indicando o nome da instância do destino e o shard e o datacenter que ocupa, que podem ser usados para agrupar e agregar essas séries temporais.

Assim, temos vários usos para rótulos em uma série temporal, embora todos sejam intercambiáveis:

- Rótulos que definem detalhamentos dos próprios dados (por exemplo, nosso código de resposta HTTP em a variável `http_responses`)
- Rótulos que definem a origem dos dados (por exemplo, a instância ou nome do trabalho) •

Rótulos que indicam a localidade ou agregação dos dados dentro do serviço como um todo (por exemplo, o rótulo de zona descrevendo um local físico, um rótulo de fragmento descrevendo um agrupamento lógico de tarefas)

A natureza modelada dessas bibliotecas permite flexibilidade em seu uso. O mesmo modelo pode ser usado para agrregar de cada camada.

Dez anos depois...

Borgmon transpõe o modelo de verificação e alerta por alvo para a coleta de variáveis em massa e uma avaliação de regras centralizada em toda a série temporal para alertas e diagnósticos.

Esse desacoplamento permite que o tamanho do sistema monitorado seja dimensionado independentemente do tamanho das regras de alerta. Essas regras custam menos para manter porque são abstratas em um formato comum de série temporal. Novos aplicativos vêm prontos com exportações de métricas em todos os componentes e bibliotecas aos quais eles se vinculam, agregação e modelos de console bem viajados, o que reduz ainda mais a carga de implementação.

Garantir que o custo de manutenção seja dimensionado de forma sublinear com o tamanho do serviço é a chave para tornar o monitoramento (e todas as operações de sustentação) sustentável. Esse tema é recorrente em todos os trabalhos de SRE, pois os SREs trabalham para escalar todos os aspectos de seu trabalho em escala global.

Dez anos é muito tempo, no entanto, e é claro que hoje a forma do cenário de monitoramento dentro do Google evoluiu com experimentos e mudanças, buscando melhorias contínuas à medida que a empresa cresce.

Embora Borgmon permaneça interno ao Google, a ideia de tratar dados de séries temporais como uma fonte de dados para gerar alertas agora está acessível a todos por meio de ferramentas de código aberto como Prometheus, Riemann, Heka e Bosun, e provavelmente outras quando você Leia isso.

CAPÍTULO 11**Estar de plantão**

Escrito por Andrea Spadaccini¹
Editado por Kavita Giuliani

Estar de plantão é um dever crítico que muitas equipes de operações e engenharia devem realizar para manter seus serviços confiáveis e disponíveis. No entanto, existem várias armadilhas na organização de rodízios e responsabilidades de plantão que podem trazer sérias consequências para os serviços e para as equipes se não forem evitadas. Este capítulo descreve os princípios básicos da abordagem de plantão que os engenheiros de confiabilidade do site (SREs) do Google desenvolveram ao longo dos anos e explica como essa abordagem levou a serviços confiáveis e carga de trabalho sustentável ao longo do tempo.

Introdução

Diversas profissões exigem que os funcionários exerçam algum tipo de plantão, o que implica em estar disponível para atendimentos tanto no horário de trabalho quanto no de folga. No contexto de TI, as atividades de plantão têm sido historicamente realizadas por equipes de operações dedicadas com a responsabilidade principal de manter o(s) serviço(s) pelos quais são responsáveis em boas condições.

Muitos serviços importantes no Google, por exemplo, Pesquisa, Anúncios e Gmail, têm equipes dedicadas de SREs responsáveis pelo desempenho e confiabilidade desses serviços. Assim, os SREs estão de plantão para os serviços que suportam. As equipes de SRE são bem diferentes das equipes puramente operacionais, pois colocam grande ênfase no uso da engenharia para abordar problemas. Esses problemas, que normalmente se enquadram no domínio operacional, existem em uma escala que seria intratável sem soluções de engenharia de software.

¹ Uma versão anterior deste capítulo apareceu como artigo em *:login:* (outubro 2015, vol. 40, nº 5).

Para impor esse tipo de solução de problemas, o Google contrata pessoas com formação diversificada em sistemas e engenharia de software para equipes de SRE. Limitamos a quantidade de tempo que os SREs gastam em trabalho puramente operacional em 50%; no mínimo, 50% do tempo de um SRE deve ser destinado a projetos de engenharia que dimensionem ainda mais o impacto da equipe por meio da automação, além de melhorar o atendimento.

Vida de um engenheiro de plantão

Esta seção descreve as atividades típicas de um engenheiro de plantão e fornece algumas informações básicas para o restante do capítulo.

Como guardiões dos sistemas de produção, os engenheiros de plantão cuidam de suas operações designadas gerenciando interrupções que afetam a equipe e realizando e/ou verificando alterações de produção.

Quando de plantão, um engenheiro está disponível para realizar operações nos sistemas de produção em minutos, de acordo com os tempos de resposta de paging acordados pela equipe e pelos proprietários do sistema de negócios. Os valores típicos são 5 minutos para serviços voltados para o usuário ou de outra forma altamente críticos e 30 minutos para sistemas menos sensíveis ao tempo. A empresa fornece o dispositivo de recepção de page, que normalmente é um telefone. O Google tem sistemas flexíveis de entrega de alertas que podem enviar páginas por meio de vários mecanismos (e-mail, SMS, chamada de robô, aplicativo) em vários dispositivos.

Os tempos de resposta estão relacionados à disponibilidade desejada do serviço, como demonstrado pelo exemplo simplista a seguir: se um sistema voltado para o usuário deve obter 4 noves de disponibilidade em um determinado trimestre (99,99%), o tempo de inatividade trimestral permitido é de cerca de 13 minutos ([Apêndice UMA](#)). Esta restrição implica que o tempo de reação dos engenheiros de plantão deve ser da ordem de minutos (estritamente falando, 13 minutos). Para sistemas com SLOs mais relaxados, o tempo de reação pode ser da ordem de dezenas de minutos.

Assim que uma mensagem é recebida e reconhecida, espera-se que o engenheiro de plantão faça a triagem do problema e trabalhe para sua resolução, possivelmente envolvendo outros membros da equipe e escalando conforme necessário.

Eventos de produção sem paginação, como alertas de prioridade mais baixa ou lançamentos de software, também podem ser tratados e/ou verificados pelo engenheiro de plantão durante o horário comercial. Essas atividades são menos urgentes do que os eventos de paginação, que têm prioridade sobre quase todas as outras tarefas, incluindo o trabalho do projeto. Para obter mais informações sobre interrupções e outros eventos de não paginação que contribuem para a carga operacional, consulte o [Capítulo 29](#).

Muitas equipes têm uma rotação de plantão primária e secundária. A distribuição de tarefas entre o primário e o secundário varia de equipe para equipe. Uma equipe pode雇用 o secundário como um substituto para as páginas que o plantão principal perde. Outra equipe pode especificar que o plantão primário trata apenas de páginas, enquanto o secundário trata de todas as outras atividades de produção não urgentes.

Em equipes para as quais uma rotação secundária não é estritamente necessária para a distribuição de tarefas, é comum que duas equipes relacionadas sirvam como secundárias de plantão uma para a outra, com tarefas de tratamento de emergência. Esta configuração elimina a necessidade de um rodízio de plantão secundário exclusivo.

Há muitas maneiras de organizar rotações de plantão; para análise detalhada, consulte o capítulo “Oncall” de [Lim14].

Chamada Equilibrada

As equipes de SRE têm restrições específicas quanto à quantidade e qualidade dos turnos de plantão. A quantidade de plantão pode ser calculada pelo percentual de tempo gasto pelos engenheiros em plantão. A qualidade do plantão pode ser calculada pelo número de incidentes que ocorrem durante um turno de plantão.

Os gerentes de SRE têm a responsabilidade de manter a carga de trabalho de plantão equilibrada e sustentável nesses dois eixos.

Equilíbrio na quantidade

Acreditamos firmemente que o “E” em “SRE” é uma característica definidora de nossa organização, por isso nos esforçamos para investir pelo menos 50% do tempo de SRE em engenharia: do restante, não mais que 25 % pode ser gasto em plantão, deixando até outros 25% em outros tipos de trabalho operacional, fora do projeto.

Usando a regra de plantão de 25%, podemos derivar o número mínimo de SREs necessários para sustentar uma rotação de plantão 24 horas por dia, 7 dias por semana. Assumindo que há sempre duas pessoas de plantão (primário e secundário, com funções diferentes), o número mínimo de engenheiros necessários para plantão de uma equipe de um único local é oito: assumindo turnos de uma semana, cada engenheiro está de plantão -call (primário ou secundário) por uma semana a cada mês. Para equipes de dois locais, um tamanho mínimo razoável de cada equipe é seis, tanto para honrar a regra de 25% quanto para garantir uma massa substancial e crítica de engenheiros para a equipe.

Se um serviço implicar trabalho suficiente para justificar o crescimento de uma equipe de um único local, preferimos criar uma equipe de vários locais. Uma equipe multi-site é vantajosa por dois motivos:

- Os turnos noturnos têm efeitos prejudiciais na saúde das pessoas [Dur05], e uma rotação “sigo o sol” em vários locais permite que as equipes evitem totalmente os turnos noturnos.
- Limitar o número de engenheiros na rotação de plantão garante que os engenheiros não percam o contato com os sistemas de produção (consulte “Um inimigo traiçoeiro: sobrecarga operacional” na página 132).

No entanto, as equipes de vários locais incorrem em sobrecarga de comunicação e coordenação. Portanto, a decisão de ir multi-site ou single-site deve ser baseada nos trade-offs

cada opção implica, a importância do sistema e a carga de trabalho que cada sistema gera.

Equilíbrio na qualidade

Para cada turno de plantão, um engenheiro deve ter tempo suficiente para lidar com quaisquer incidentes e atividades de acompanhamento, como escrever post -mortems [Loo10]. Vamos definir um incidente como uma sequência de eventos e alertas relacionados à mesma causa raiz e que seriam discutidos como parte da mesma autópsia. Descobrimos que, em média, lidar com as tarefas envolvidas em um incidente de plantão – análise de causa raiz, correção e atividades de acompanhamento, como escrever uma autópsia e corrigir bugs – leva 6 horas. Segue-se que o número máximo de incidentes por dia é de 2 por turno de plantão de 12 horas. Para ficar dentro desse limite superior, a distribuição de eventos de paginação deve ser muito plana ao longo do tempo, com um valor mediano provável de 0: se um determinado componente ou problema causa páginas todos os dias (mediana de incidentes/dia > 1), é provável que alguma outra coisa quebre em algum ponto, causando mais incidentes do que deveria ser permitido.

Se esse limite for excedido temporariamente, por exemplo, por um quarto, medidas corretivas devem ser implementadas para garantir que a carga operacional retorne a um estado sustentável (consulte “Sobrecarga operacional” na página 130 e Capítulo 30).

Remuneração Uma

compensação adequada deve ser considerada para suporte fora do horário de expediente. Diferentes organizações lidam com a compensação de plantão de maneiras diferentes; O Google oferece folga em substituição ou compensação direta em dinheiro, limitada a uma proporção do salário total. O teto de remuneração representa, na prática, um limite na quantidade de trabalho de plantão que será assumido por qualquer indivíduo. Essa estrutura de remuneração garante o incentivo ao envolvimento em plantões conforme exigido pela equipe, mas também promove uma distribuição equilibrada do trabalho de plantão e limita os possíveis inconvenientes do excesso de plantão, como esgotamento ou tempo inadequado para o projeto. trabalhar.

Sentindo seguro

Conforme mencionado anteriormente, as equipes de SRE oferecem suporte aos sistemas mais críticos do Google. Ser um SRE de plantão normalmente significa assumir a responsabilidade por sistemas críticos de receita voltados para o usuário ou pela infraestrutura necessária para manter esses sistemas em funcionamento. A metodologia SRE para pensar e enfrentar os problemas é vital para a operação adequada dos serviços.

A pesquisa moderna identifica duas formas distintas de pensar que um indivíduo pode, consciente ou inconscientemente, escolher quando confrontado com desafios [Kah11]:

- Ação intuitiva, automática e rápida • Funções

cognitivas racionais, focadas e deliberadas

Quando se está lidando com as interrupções relacionadas a sistemas complexos, a segunda dessas opções tem maior probabilidade de produzir melhores resultados e levar a um tratamento de incidentes bem planejado.

Para garantir que os engenheiros estejam no estado de espírito apropriado para alavancar a última mentalidade, é importante reduzir o estresse relacionado a estar de plantão. A importância e o impacto dos serviços e as consequências de possíveis interrupções podem criar uma pressão significativa sobre os engenheiros de plantão, prejudicando o bem-estar de cada membro da equipe e possivelmente levando os SREs a fazer escolhas incorretas que podem colocar em risco a disponibilidade do serviço. Hormônios do estresse como cortisol e hormônio liberador de corticotropina (CRH) são conhecidos por causar consequências comportamentais – incluindo medo – que podem prejudicar as funções cognitivas e causar tomada de decisão abaixo do ideal [Chr09].

Sob a influência desses hormônios do estresse, a abordagem cognitiva mais deliberada é tipicamente subsumida por ação irrefletida e não considerada (mas imediata), levando a um potencial abuso de heurística. Heurísticas são comportamentos muito tentadores quando se está de plantão. Por exemplo, quando as mesmas páginas de alerta pela quarta vez na semana e as três páginas anteriores foram iniciadas por um sistema de infraestrutura externo, é extremamente tentador exercer viés de confirmação associando automaticamente essa quarta ocorrência do problema à causa anterior .

Embora a intuição e as reações rápidas possam parecer características desejáveis no meio do gerenciamento de incidentes, elas têm desvantagens. A intuição pode estar errada e muitas vezes é menos suportável por dados óbvios. Assim, seguir a intuição pode levar um engenheiro a perder tempo perseguindo uma linha de raciocínio que está incorreta desde o início. As reações rápidas estão profundamente enraizadas no hábito e as respostas habituais não são consideradas, o que significa que podem ser desastrosas. A metodologia ideal no gerenciamento de incidentes atinge o equilíbrio perfeito entre tomar as medidas no ritmo desejado quando há dados suficientes disponíveis para tomar uma decisão razoável e, ao mesmo tempo, examinar criticamente suas suposições.

É importante que os SREs de plantão entendam que podem contar com vários recursos que tornam a experiência de plantão menos assustadora do que parece. Os recursos de plantão mais importantes são:

- Caminhos de escalonamento claros • Procedimentos de gerenciamento de incidentes bem definidos • Uma cultura post mortem sem culpa ([Loo10], [All12])

As equipes de desenvolvedores de sistemas suportados pelo SRE geralmente participam de um rodízio de plantão 24 horas por dia, 7 dias por semana, e sempre é possível escalar para essas equipes parceiras quando necessário.

A escalada apropriada de interrupções geralmente é uma maneira de reagir a interrupções graves com dimensões desconhecidas significativas.

Quando se trata de incidentes, se o problema for complexo o suficiente para envolver várias equipes ou se, após alguma investigação, ainda não for possível estimar um limite superior para o período de tempo do incidente, pode ser útil adotar um gerenciamento formal de incidentes. protocolo. O Google SRE usa o protocolo descrito no [Capítulo 14](#), que oferece um conjunto de etapas fáceis de seguir e bem definidas que ajudam um engenheiro de plantão a buscar racionalmente uma resolução de incidente satisfatória com toda a ajuda necessária. Esse protocolo é suportado internamente por uma ferramenta baseada na web que automatiza a maioria das ações de gerenciamento de incidentes, como transferência de funções e registro e comunicação de atualizações de status. Essa ferramenta permite que os gerentes de incidentes se concentrem em lidar com o incidente, em vez de gastar tempo e esforço cognitivo em ações mundanas, como formatar e-mails ou atualizar vários canais de comunicação ao mesmo tempo.

Finalmente, quando ocorre um incidente, é importante avaliar o que deu errado, reconhecer o que deu certo e tomar medidas para evitar que os mesmos erros se repitam no futuro. As equipes de SRE devem escrever post-mortems após incidentes significativos e detalhar uma linha do tempo completa dos eventos ocorridos. Concentrando-se nos eventos em vez das pessoas, essas autópsias fornecem um valor significativo. Em vez de culpar indivíduos, eles extraem valor da análise sistemática de incidentes de produção. Erros acontecem, e o software deve garantir que cometemos o mínimo de erros possível.

Reconhecer oportunidades de automação é uma das melhores maneiras de prevenir erros humanos [\[Loo10\]](#).

Evitando Carga Operacional Inapropriada

Conforme mencionado em “[Chamada balanceada](#)” na [página 127](#), os SREs gastam no máximo 50% de seu tempo em trabalho operacional. O que acontece se as atividades operacionais ultrapassarem esse limite?

Sobrecarga Operacional A

equipe e a liderança do SRE são responsáveis por incluir objetivos concretos no planejamento trimestral do trabalho para garantir que a carga de trabalho retorne a níveis sustentáveis. O empréstimo temporário de um SRE experiente para uma equipe sobrecarregada, discutido no [Capítulo 30](#), pode fornecer espaço suficiente para que a equipe possa avançar no tratamento dos problemas.

Idealmente, os sintomas de sobrecarga operacional devem ser mensuráveis, para que as metas possam ser quantificadas (por exemplo, número de tickets diários < 5, eventos de paging por turno < 2).

O monitoramento mal configurado é uma causa comum de sobrecarga operacional. Os alertas de paginação devem estar alinhados com os sintomas que ameaçam os SLOs de um serviço. Todos os alertas de paginação também devem ser acionáveis. Alertas de baixa prioridade que incomodam o engenheiro de plantão a cada hora (ou com mais frequência) interrompem a produtividade, e a fadiga que esses alertas induzem pode

também fazem com que alertas sérios sejam tratados com menos atenção do que o necessário. Veja o [Capítulo 29](#) para discussão adicional.

Também é importante controlar o número de alertas que os engenheiros de plantão recebem para um único incidente. Às vezes, uma única condição anormal pode gerar vários alertas, portanto, é importante regular a distribuição de alertas garantindo que os alertas relacionados sejam agrupados pelo sistema de monitoramento ou alerta. Se, por qualquer motivo, alertas duplicados ou não informativos forem gerados durante um incidente, silenciar esses alertas pode fornecer o silêncio necessário para que o engenheiro de plantão se concentre no incidente em si.

Alertas ruidosos que geram sistematicamente mais de um alerta por incidente devem ser ajustados para se aproximarem de uma proporção de alerta/incidente de 1:1. Isso permite que o engenheiro de plantão se concentre no incidente em vez de fazer a triagem de alertas duplicados.

Às vezes, as mudanças que causam sobrecarga operacional não estão sob o controle das equipes do SRE. Por exemplo, os desenvolvedores de aplicativos podem introduzir alterações que tornem o sistema mais barulhento, menos confiável ou ambos. Nesse caso, é apropriado trabalhar em conjunto com os desenvolvedores de aplicativos para definir metas comuns para melhorar o sistema.

Em casos extremos, as equipes do SRE podem ter a opção de “devolver o pager” – o SRE pode solicitar que a equipe do desenvolvedor fique exclusivamente de plantão para o sistema até que ele atenda aos padrões da equipe do SRE em questão. A devolução do pager não acontece com muita frequência, pois quase sempre é possível trabalhar com a equipe de desenvolvedores para reduzir a carga operacional e tornar um determinado sistema mais confiável. Em alguns casos, porém, podem ser necessárias mudanças complexas ou arquitetônicas que abrangem vários trimestres para tornar um sistema sustentável do ponto de vista operacional. Nesses casos, a equipe SRE não deve estar sujeita a uma carga operacional excessiva. Em vez disso, é apropriado negociar a reorganização das responsabilidades de plantão com a equipe de desenvolvimento, possivelmente roteando alguns ou todos os alertas de paging para o desenvolvedor de plantão.

Essa solução geralmente é uma medida temporária, durante a qual as equipes de SRE e desenvolvedor trabalham juntas para colocar o serviço em forma para ser integrado pela equipe de SRE novamente.

A possibilidade de renegociar responsabilidades de plantão entre SRE e equipes de desenvolvimento de produtos atesta o equilíbrio de poderes entre as equipes.² Essa relação de trabalho também exemplifica como a tensão saudável entre essas duas equipes e os valores que elas representam - confiabilidade versus velocidade de recursos - normalmente é resolvida com grande benefício para o serviço e, por extensão, para a empresa como um todo.

² Para mais discussão sobre a tensão natural entre SRE e equipes de desenvolvimento de produto, consulte o [Capítulo 1](#).

Um inimigo traiçoeiro: subcarga operacional Estar de plantão

para um sistema silencioso é uma bênção, mas o que acontece se o sistema estiver muito quieto ou quando os SREs não estiverem de plantão com frequência suficiente? Uma subcarga operacional é indesejável para uma equipe SRE. Estar fora de contato com a produção por longos períodos de tempo pode levar a problemas de confiança, tanto em termos de excesso de confiança quanto de falta de confiança, enquanto as lacunas de conhecimento são descobertas apenas quando ocorre um incidente.

Para neutralizar essa eventualidade, as equipes de SRE devem ser dimensionadas para permitir que cada engenheiro esteja de plantão pelo menos uma ou duas vezes por trimestre, garantindo assim que cada membro da equipe esteja suficientemente exposto à produção. Os exercícios da “Roda do Infortúnio” (discutidos no [Capítulo 28](#)) também são atividades de equipe úteis que podem ajudar a aprimorar e melhorar as habilidades de resolução de problemas e o conhecimento do serviço. O Google também tem um evento anual de recuperação de desastres em toda a empresa chamado DiRT (Treinamento de Recuperação de Desastres) que combina exercícios teóricos e práticos para realizar testes de vários dias de sistemas de infraestrutura e serviços individuais; veja [\[Kri12\]](#).

Conclusões

A abordagem de plantão descrita neste capítulo serve como diretriz para todas as equipes de SRE no Google e é fundamental para promover um ambiente de trabalho sustentável e gerenciável. A abordagem do Google ao plantão nos permitiu usar o trabalho de engenharia como o principal meio de dimensionar as responsabilidades de produção e manter alta confiabilidade e disponibilidade, apesar da crescente complexidade e número de sistemas e serviços pelos quais os SREs são responsáveis.

Embora essa abordagem possa não ser imediatamente aplicável a todos os contextos em que os engenheiros precisam estar de plantão para serviços de TI, acreditamos que ela representa um modelo sólido que as organizações podem adotar no dimensionamento para atender a um volume crescente de trabalho de plantão.

CAPÍTULO 12

Solução de problemas eficaz

Escrito por Chris Jones

Esteja avisado que ser um especialista é mais do que entender como um sistema deve funcionar. A experiência é adquirida investigando por que um sistema não funciona.

—Brian Redman

As maneiras pelas quais as coisas dão certo são casos especiais das maneiras pelas quais as coisas dão errado.

—John Allspaw

A resolução de problemas é uma habilidade crítica para qualquer pessoa que opere sistemas de computação distribuídos – especialmente SREs – mas muitas vezes é vista como uma habilidade inata que algumas pessoas têm e outras não. Uma razão para essa suposição é que, para aqueles que solucionam problemas com frequência, é um processo arraigado; explicar como solucionar problemas é difícil, assim como explicar como andar de bicicleta. No entanto, acreditamos que a resolução de problemas pode ser aprendida e ensinada.

Os novatos muitas vezes tropeçam na solução de problemas porque o exercício idealmente depende de dois fatores: uma compreensão de como solucionar problemas de forma genérica (ou seja, sem nenhum conhecimento específico do sistema) e um conhecimento sólido do sistema.

Embora você possa investigar um problema usando apenas o processo genérico e a derivação dos primeiros princípios¹, geralmente achamos essa abordagem menos eficiente e menos eficaz do que entender como as coisas devem funcionar. O conhecimento do sistema normalmente limita a eficácia de um SRE novo para um sistema; há pouco substituto para aprender como o sistema é projetado e construído.

¹ De fato, usar apenas os primeiros princípios e habilidades de solução de problemas geralmente é uma maneira eficaz de aprender como um sistema funciona; veja o [Capítulo 28](#).

Vejamos um modelo geral do processo de solução de problemas. Leitores com experiência em solução de problemas podem discordar de nossas definições e processos; se o seu método é eficaz para você, não há razão para não ficar com ele.

Teoria

Formalmente, podemos pensar no processo de solução de problemas como uma aplicação do método hipotético-dedutivo:² dado um conjunto de observações sobre um sistema e uma base teórica para entender o comportamento do sistema, nós iterativamente hipotetizamos as causas potenciais para a falha e tentamos testar essas hipóteses.

Em um modelo idealizado como o da [Figura 12-1](#), começariamos com um relatório de problema nos dizendo que algo está errado com o sistema. Em seguida, podemos examinar a telemetria³ e os logs do sistema para entender seu estado atual. Essas informações, combinadas com nosso conhecimento de como o sistema é construído, como ele deve operar e seus modos de falha, nos permite identificar algumas possíveis causas.

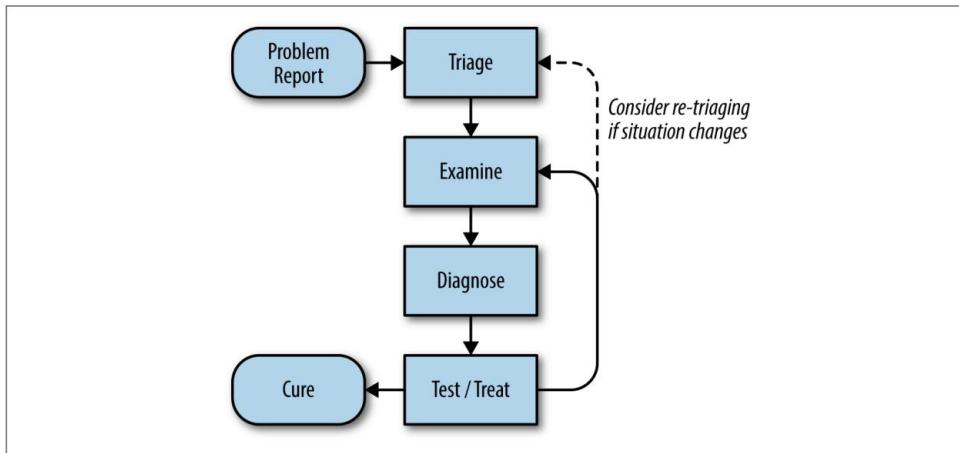


Figura 12-1. Um processo para solução de problemas

2 Veja https://en.wikipedia.org/wiki/Hypothetico-deducing_model.

3 Por exemplo, variáveis exportadas conforme descrito no [Capítulo 10](#).

Podemos então testar nossas hipóteses de duas maneiras. Podemos comparar o estado observado do sistema com nossas teorias para encontrar evidências confirmatórias ou não. Ou, em alguns casos, podemos “tratar” ativamente o sistema – ou seja, mudar o sistema de forma controlada – e observar os resultados. Essa segunda abordagem refina nossa compreensão do estado do sistema e da(s) possível(is) causa(s) dos problemas relatados. Usando qualquer uma dessas estratégias, testamos repetidamente até que uma causa raiz seja identificada, quando então podemos tomar ações corretivas para evitar uma recorrência e escrever uma autópsia. É claro que corrigir a(s) causa(s) próxima(s) nem sempre precisa esperar pela escrita da causa raiz ou post-mortem.

Armadilhas comuns

Sessões de solução de problemas ineficazes são atormentadas por problemas nas etapas de Triage, Examinar e Diagnosticar, geralmente devido à falta de compreensão profunda do sistema. As seguintes são armadilhas comuns a serem evitadas:

- Observar sintomas que não são relevantes ou entender mal o significado de métricas do sistema. Perseguições de ganso selvagem geralmente resultam.
- Incompreensão de como alterar o sistema, suas entradas ou seu ambiente, para testar hipóteses com segurança e eficácia. • Inventar teorias altamente improváveis sobre o que está errado, ou agarrar-se às causas de problemas passados, raciocinando que, como aconteceu uma vez, deve estar acontecendo novamente.
- Caçar correlações espúrias que são realmente coincidências ou são correlações com causas compartilhadas.

Corrigir a primeira e a segunda armadilhas comuns é uma questão de aprender o sistema em questão e adquirir experiência com os padrões comuns usados em sistemas distribuídos. A terceira armadilha é um conjunto de falácias lógicas que podem ser evitadas lembrando que nem todas as falhas são igualmente prováveis – como os médicos são ensinados, “quando você ouvir batidas de casco, pense em cavalos, não em zebras”. sendo iguais, devemos preferir explicações mais simples.⁵

⁴ Atribuído a Theodore Woodward, da Faculdade de Medicina da Universidade de Maryland, na década de 1940. Veja [https://en.wikipedia.org/wiki/Zebra_\(medicina\)](https://en.wikipedia.org/wiki/Zebra_(medicina)). Isso funciona em alguns domínios, mas em alguns sistemas, classes inteiras de falhas podem ser eliminadas: por exemplo, usar um sistema de arquivos de cluster bem projetado significa que é improvável que um problema de latência seja devido a um único disco morto.

⁵ Navalha de Occam; consulte https://en.wikipedia.org/wiki/Occam%27s_razor. Mas lembre-se de que ainda pode haver vários problemas; em particular, pode ser mais provável que um sistema tenha vários problemas comuns de baixo grau que, juntos, explicam todos os sintomas, em vez de um único problema raro que causa todos eles. Cf https://en.wikipedia.org/wiki/Hickam%27s_dictum.

Por fim, devemos lembrar que a correlação não é causa:⁶ alguns eventos correlacionados, como perda de pacotes dentro de um cluster e falha de discos rígidos no cluster, compartilham causas comuns - neste caso, uma queda de energia, embora a falha de rede claramente não cause as falhas do disco rígido nem vice-versa. Pior ainda, à medida que os sistemas crescem em tamanho e complexidade e à medida que mais métricas são monitoradas, é inevitável que haja eventos que se correlacionam bem com outros eventos, puramente por coincidência.⁷

Compreender as falhas em nosso processo de raciocínio é o primeiro passo para evitá-las e nos tornarmos mais eficazes na resolução de problemas. Uma abordagem metódica para saber o que sabemos, o que não sabemos e o que precisamos saber torna mais simples e direto descobrir o que deu errado e como corrigi-lo.

Na prática

Na prática, é claro, a solução de problemas nunca é tão limpa quanto nosso modelo idealizado sugere que deveria ser. Existem algumas etapas que podem tornar o processo menos doloroso e mais produtivo tanto para quem está enfrentando problemas no sistema quanto para quem está respondendo a eles.

Relatório de problemas

Todo problema começa com um relatório de problemas, que pode ser um alerta automatizado ou um de seus colegas dizendo: "O sistema está lento". Um relatório eficaz deve informar o comportamento esperado, o comportamento real e, se possível, como reproduzir o comportamento.⁸ Idealmente, os relatórios devem ter um formato consistente e ser armazenados em um local pesquisável, como um rastreamento de bugs sistema. Aqui, nossas equipes geralmente têm formulários personalizados ou pequenos aplicativos da Web que solicitam informações relevantes para diagnosticar os sistemas específicos aos quais eles dão suporte, que geram e roteiam automaticamente um bug.

Este também pode ser um bom ponto para fornecer ferramentas para que os relatores de problemas tentem autodiagnosticar ou auto-reparar problemas comuns por conta própria.

É prática comum no Google abrir um bug para cada problema, mesmo aqueles recebidos por e-mail ou mensagens instantâneas. Isso cria um log de atividades de investigação e correção que podem ser referenciadas no futuro. Muitas equipes desencorajam relatar problemas diretamente a uma pessoa por vários motivos: esta prática introduz uma etapa adicional de transcrição do relatório em um bug, produz relatórios de qualidade inferior que não são

⁶ Claro, veja <https://xkcd.com/552>.

⁷ Pelo menos, não temos uma teoria plausível para explicar por que o número de doutorados em Ciência da Computação nos EUA deve ser extremamente bem correlacionado ($r^2 = 0,9416$) com o consumo per capita de queijo, entre 2000 e 2009: http://tylervigen.com/view_correlation?id=1099.

⁸ Pode ser útil encaminhar possíveis relatores de bugs para [Tat99] para ajudá-los a fornecer problemas de alta qualidade relatórios.

visível para outros membros da equipe e tende a concentrar a carga de solução de problemas em um punhado de membros da equipe que os repórteres conhecem, em vez da pessoa que está de serviço (veja também o [Capítulo 29](#)).

Shakespeare tem um problema

Você está de plantão para o serviço de pesquisa Shakespeare e recebe um alerta, Shakespeare BlackboxProbe_SearchFailure: seu monitoramento de caixa preta não conseguiu encontrar resultados de pesquisa para “as formas das coisas desconhecidas” nos últimos cinco minutos. O sistema de alertas registrou um bug – com links para os resultados recentes da sonda de caixa preta e para a entrada do manual para este alerta – e o atribuiu a você. Hora de entrar em ação!

Triagem

Depois de receber um relatório de problema, o próximo passo é descobrir o que fazer a respeito. Os problemas podem variar em gravidade: um problema pode afetar apenas um usuário em circunstâncias muito específicas (e pode ter uma solução alternativa) ou pode acarretar uma interrupção global completa de um serviço. Sua resposta deve ser apropriada para o impacto do problema: é apropriado declarar uma emergência total para o último (veja o [Capítulo 14](#)), mas fazê-lo para o primeiro é um exagero. Avaliar a gravidade de um problema requer um exercício de bom julgamento de engenharia e, muitas vezes, um grau de calma sob pressão.

Sua primeira resposta em uma grande interrupção pode ser iniciar a solução de problemas e tentar encontrar uma causa raiz o mais rápido possível. Ignore esse instinto!

Em vez disso, seu curso de ação deve ser fazer o sistema funcionar tão bem quanto possível sob as circunstâncias. Isso pode envolver opções de emergência, como desviar o tráfego de um cluster quebrado para outros que ainda estão funcionando, descartar o tráfego por atacado para evitar uma falha em cascata ou desabilitar subsistemas para aliviar a carga. Parar o sangramento deve ser sua primeira prioridade; você não está ajudando seus usuários se o sistema morrer enquanto você está causando root. É claro que a ênfase na triagem rápida não impede a adoção de medidas para preservar as evidências do que está errado, como logs, para ajudar na análise subsequente da causa raiz.

Pilotos novatos são ensinados que sua primeira responsabilidade em uma emergência é pilotar o avião [\[Gaw09\]](#); a solução de problemas é secundária para colocar o avião e todos nele com segurança no solo. Essa abordagem também é aplicável a sistemas de computador: por exemplo, se um bug estiver levando a uma corrupção de dados possivelmente irrecuperável, congelar o sistema para evitar mais falhas pode ser melhor do que deixar esse comportamento continuar.

Essa percepção é muitas vezes bastante inquietante e contra-intuitiva para novos SREs, particularmente aqueles cuja experiência anterior foi em organizações de desenvolvimento de produtos.

Examinar

Precisamos ser capazes de examinar o que cada componente do sistema está fazendo para entender se está ou não se comportando corretamente.

Idealmente, um sistema de monitoramento está registrando as métricas do seu sistema, conforme discutido no [Capítulo 10](#). Essas métricas são um bom ponto de partida para descobrir o que está errado. A representação gráfica de séries temporais e operações em séries temporais pode ser uma maneira eficaz de entender o comportamento de partes específicas de um sistema e encontrar correlações que possam sugerir onde os problemas começaram.⁹ O registro em log é outra ferramenta inestimável. A exportação de informações sobre cada operação e sobre o estado do sistema permite entender exatamente o que um processo estava fazendo em um determinado momento. Pode ser necessário analisar os logs do sistema em um ou vários processos. O rastreamento de solicitações em toda a pilha usando ferramentas como Dapper [\[Sig10\]](#) fornece uma maneira muito poderosa de entender como um sistema distribuído está funcionando, embora casos de uso variados impliquem projetos de rastreamento significativamente diferentes [\[Sam14\]](#).

Logging

Os logs de texto são muito úteis para a depuração reativa em tempo real, enquanto o armazenamento de logs em um formato binário estruturado pode possibilitar a construção de ferramentas para realizar análises retrospectivas com muito mais informações.

É realmente útil ter vários níveis de verbosidade disponíveis, juntamente com uma maneira de aumentar esses níveis rapidamente. Essa funcionalidade permite que você examine qualquer uma ou todas as operações com detalhes incríveis sem precisar reiniciar seu processo, enquanto ainda permite que você diminua os níveis de verbosidade quando seu serviço estiver operando normalmente. Dependendo do volume de tráfego que seu serviço recebe, pode ser melhor usar amostragem estatística; por exemplo, você pode mostrar uma em cada 1.000 operações.

A próxima etapa é incluir um idioma de seleção para que você possa dizer "mostre-me as operações que correspondem a X", para uma ampla faixa de X - por exemplo, Definir RPCs com um tamanho de carga útil inferior a 1.024 bytes ou operações que demoraram mais de 10 ms para retornar, ou que chamou `doSomeThingInteresting()` em `rpc_handler.py`. Você pode até querer projetar sua infraestrutura de log para poder ativá-la conforme necessário, de forma rápida e seletiva.

Exportar o estado atual é o terceiro truque em nossa caixa de ferramentas. Por exemplo, os servidores do Google têm endpoints que mostram uma amostra de RPCs enviados ou recebidos recentemente. Assim, é possível entender como um servidor está se comunicando com outros sem fazer referência

⁹ Mas cuidado com as falsas correlações que podem levá-lo a caminhos errados!

um diagrama de arquitetura. Esses endpoints também mostram histogramas de taxas de erro e latência para cada tipo de RPC, para que seja possível identificar rapidamente o que não está íntegro. Alguns sistemas possuem endpoints que mostram sua configuração atual ou permitem o exame de seus dados; por exemplo, os servidores Borgmon do Google ([Capítulo 10](#)) podem mostrar as regras de monitoramento que estão usando e até mesmo permitir o rastreamento de uma determinada computação passo a passo para as métricas de origem das quais um valor é derivado.

Finalmente, você pode até precisar instrumentar um cliente para experimentar, a fim de descobrir o que um componente está retornando em resposta a solicitações.

Depurando Shakespeare

Usando o link para os resultados do monitoramento de caixa preta no bug, você descobre que o prober envia uma solicitação HTTP GET para o endpoint /api/search :

```
{
  'search_text': 'as formas de coisas desconhecidas'
}
```

Ele espera receber uma resposta com um código de resposta HTTP 200 e uma carga JSON exatamente correspondente:

```
[
  {
    "trabalho": "Sonho de uma noite de
    verão", "ato": 5, "cena": 1, "linha": 2526,
    "orador": "Teseu"

  }
]
```

O sistema está configurado para enviar uma sonda uma vez por minuto; nos últimos 10 minutos, cerca de metade das sondas foram bem-sucedidas, embora sem um padrão discernível. Infelizmente, o prober não mostra o que foi retornado quando falhou; você faz uma nota para corrigir isso para o futuro.

Usando curl, você envia solicitações manualmente para o endpoint de pesquisa e obtém uma resposta com falha com o código de resposta HTTP 502 (Bad Gateway) e sem carga útil. Ele tem um cabeçalho HTTP, X-Request-Trace, que lista os endereços dos servidores backend responsáveis por responder a essa solicitação. Com essas informações, agora você pode examinar esses backends para testar se estão respondendo adequadamente.

Diagnóstico

Uma compreensão completa do projeto do sistema é decididamente útil para criar hipóteses plausíveis sobre o que deu errado, mas também existem algumas práticas genéricas que ajudarão mesmo sem conhecimento de domínio.

Simplifique e reduza

Idealmente, os componentes de um sistema têm interfaces bem definidas e realizam transformações conhecidas de sua entrada para sua saída (em nosso exemplo, dado um texto de pesquisa de entrada, um componente pode retornar uma saída contendo possíveis correspondências). É então possível observar as conexões entre os componentes – ou, de forma equivalente, os dados que fluem entre eles – para determinar se um determinado componente está funcionando corretamente. A injeção de dados de teste conhecidos para verificar se a saída resultante é esperada (uma forma de teste de caixa preta) em cada etapa pode ser especialmente eficaz, assim como a injeção de dados destinados a investigar possíveis causas de erros. Ter um caso de teste reproduzível sólido torna a depuração muito mais rápida, e pode ser possível usar o caso em um ambiente de não produção onde técnicas mais invasivas ou arriscadas estão disponíveis do que seria possível em produção.

Dividir e conquistar é uma técnica de solução de propósito geral muito útil. Em um sistema multicamadas em que o trabalho acontece em uma pilha de componentes, geralmente é melhor começar sistematicamente de uma extremidade da pilha e trabalhar em direção à outra, examinando cada componente por vez. Essa estratégia também é adequada para uso com pipelines de processamento de dados. Em sistemas excepcionalmente grandes, proceder linearmente pode ser muito lento; uma alternativa, a bissecção, divide o sistema ao meio e examina os caminhos de comunicação entre os componentes de um lado e do outro. Depois de determinar se uma metade parece estar funcionando corretamente, repita o processo até ficar com um componente possivelmente defeituoso.

Pergunte “o quê”, “onde” e “por que”

Um sistema com defeito muitas vezes ainda está tentando fazer alguma coisa, mas não o que você quer que ele faça. Descobrir o que está fazendo, então perguntar por que está fazendo isso e onde seus recursos estão sendo usados ou para onde está indo sua saída pode ajudá-lo a entender como as coisas deram errado.¹⁰

¹⁰ Em muitos aspectos, isso é semelhante à técnica “Five Whys” [Ohn88] introduzida por Taiichi Ohno para entender as causas-raiz dos erros de fabricação.

Descompactando as causas de um sintoma Sintoma:

Um cluster do Spanner tem alta latência e os RPCs para seus servidores estão expirando.

Por quê? As tarefas do servidor Spanner estão usando todo o tempo de CPU e não podem progredir em todas as solicitações que os clientes enviam.

Onde no servidor o tempo de CPU está sendo usado? A criação de perfil do servidor mostra suas entradas de classificação nos logs com checkpoint no disco.

Onde no código de classificação de log ele está sendo usado? Ao avaliar uma expressão regular em relação a caminhos para arquivos de log.

Soluções: Reescreva a expressão regular para evitar retrocesso. Procure na base de código para padrões semelhantes. Considere usar RE2, que não retrocede e garante o crescimento linear do tempo de execução com o tamanho da entrada.¹¹

O que tocou por último

Os sistemas têm inércia: descobrimos que um sistema de computador em funcionamento tende a permanecer em movimento até que seja acionado por uma força externa, como uma mudança de configuração ou uma mudança no tipo de carga atendida. Mudanças recentes em um sistema podem ser um lugar produtivo para começar a identificar o que está errado.¹² Sistemas bem

projetados devem ter registro de produção extensivo para rastrear novas implementações de versão e mudanças de configuração em todas as camadas da pilha, desde os binários do servidor que lidam com o usuário tráfego para os pacotes instalados em nós individuais no cluster. Correlacionar mudanças no desempenho e comportamento de um sistema com outros eventos no sistema e no ambiente também pode ser útil na construção de painéis de monitoramento; por exemplo, você pode anotar um gráfico mostrando as taxas de erro do sistema com os horários de início e término de uma implantação de uma nova versão, conforme visto na [Figura 12-2](#).

¹¹ Ao contrário do RE2, o PCRE pode exigir tempo exponencial para avaliar algumas expressões regulares. RE2 está disponível em <https://github.com/google/re2>.

¹² [\[AI15\]](#) observa que esta é uma heurística frequentemente usada na resolução de interrupções.

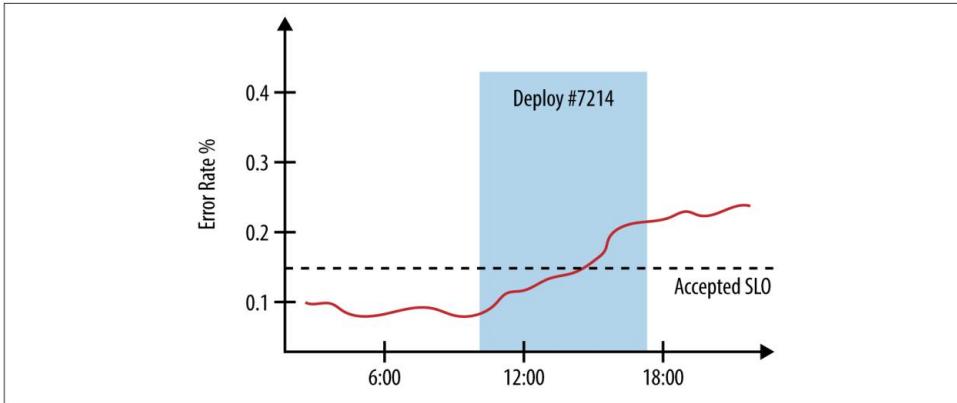


Figura 12-2. Taxas de erro representadas graficamente em relação aos horários de início e término da implantação

O envio manual de uma solicitação para o terminal /api/search (consulte “[Depurando o Shakespeare](#)” na página 139) e ver a falha listando os servidores de back-end que manipularam a resposta permite descontar a probabilidade de que o problema seja com o servidor de front-end da API e com os平衡adores de carga: a resposta provavelmente não teria incluído essas informações se a solicitação não tivesse pelo menos chegado aos back-ends de pesquisa e falhado lá. Agora você pode concentrar seus esforços nos back-ends, analisando seus logs, enviando consultas de teste para ver quais respostas eles retornam e examinando suas métricas exportadas.

Diagnósticos

específicos Embora as ferramentas genéricas descritas anteriormente sejam úteis em uma ampla gama de domínios de problemas, você provavelmente achará útil construir ferramentas e sistemas para ajudar no diagnóstico de seus serviços específicos. Os SREs do Google gastam muito tempo construindo essas ferramentas. Embora muitas dessas ferramentas sejam necessariamente específicas para um determinado sistema, certifique-se de procurar semelhanças entre serviços e equipes para evitar a duplicação de esforços.

Testar e Tratar

Depois de criar uma pequena lista de possíveis causas, é hora de tentar descobrir qual fator está na raiz do problema real. Usando o método experimental, podemos tentar incluir ou descartar nossas hipóteses. Por exemplo, suponha que pensamos que um problema é causado por uma falha de rede entre um servidor de lógica de aplicativo e um servidor de banco de dados ou pela recusa de conexões do banco de dados. Tentar se conectar ao banco de dados com as mesmas credenciais que o servidor de lógica de aplicação usa pode refutar a segunda hipótese, enquanto pingar o servidor de banco de dados pode refutar a primeira, dependendo

sobre topologia de rede, regras de firewall e outros fatores. Seguir o código e tentar imitar o fluxo de código, passo a passo, pode apontar exatamente o que está errado.

Há uma série de considerações a serem consideradas ao projetar testes (que podem ser tão simples quanto enviar um ping ou tão complicado quanto remover tráfego de um cluster e injetar solicitações especialmente formadas para encontrar uma condição de corrida):

- Um teste ideal deve ter alternativas mutuamente exclusivas, de modo que possa incluir um grupo de hipóteses e excluir outro. Na prática, isso pode ser difícil de conseguir.
- Considere primeiro o óbvio: realize os testes em ordem decrescente de probabilidade, considerando possíveis riscos para o sistema decorrentes do teste. Provavelmente faz mais sentido testar problemas de conectividade de rede entre duas máquinas antes de verificar se uma alteração recente na configuração removeu o acesso de um usuário à segunda máquina.
- Um experimento pode fornecer resultados enganosos devido a fatores de confusão. Por exemplo, uma regra de firewall pode permitir acesso apenas de um endereço IP específico, o que pode fazer com que o ping do banco de dados de sua estação de trabalho falhe, mesmo que o ping da máquina do servidor de lógica de aplicação tenha sido bem-sucedido. • Testes ativos podem ter efeitos colaterais que alteram resultados de testes futuros. Por exemplo, permitir que um processo use mais CPUs pode tornar as operações mais rápidas, mas pode aumentar a probabilidade de encontrar corridas de dados. Da mesma forma, ativar o registro detalhado pode piorar ainda mais o problema de latência e confundir seus resultados: o problema está piorando sozinho ou por causa do registro?
- Alguns testes podem não ser definitivos, apenas sugestivos. Pode ser muito difícil fazer com que condições de corrida ou impasses aconteçam de maneira oportuna e reproduzível, então você pode ter que se contentar com evidências menos certas de que essas são as causas.

Tome notas claras de quais ideias você teve, quais testes você executou e os resultados que você viu . repetir essas etapas.¹⁴ Se você executou testes ativos alterando um sistema - por exemplo, fornecendo mais recursos a um processo - fazer alterações de maneira sistemática e documentada o ajudará a retornar o sistema à configuração de pré-teste, em vez de executar em uma configuração de miscelânea desconhecida.

¹³ O uso de um documento compartilhado ou bate-papo em tempo real para anotações fornece um registro de data e hora de quando você fez algo, o que é útil para autópsias. Ele também compartilha essas informações com outras pessoas, para que estejam atualizadas com o estado atual do mundo e não precisem interromper sua solução de problemas.

¹⁴ Veja também "Resultados negativos são mágicos" na página 144 para mais informações sobre este ponto.

Resultados negativos são mágicos

Escrito por Randall Bosetti

Editado por Joan Wendt

Um resultado “negativo” é um resultado experimental no qual o efeito esperado está ausente – ou seja, qualquer experimento que não funcione como planejado. Isso inclui novos designs, heurísticas ou processos humanos que não melhoram os sistemas que substituem.

Os resultados negativos não devem ser ignorados ou descontados. Perceber que você está errado tem muito valor: um resultado negativo claro pode resolver algumas das questões de design mais difíceis. Muitas vezes, uma equipe tem dois projetos aparentemente razoáveis, mas o progresso em uma direção precisa abordar questões vagas e especulativas sobre se a outra direção pode ser melhor.

Experimentos com resultados negativos são conclusivos. Eles nos dizem algo certo sobre a produção, ou o espaço de design, ou os limites de desempenho de um sistema existente. Eles podem ajudar outros a determinar se seus próprios experimentos ou projetos valem a pena. Por exemplo, uma determinada equipe de desenvolvimento pode decidir não usar um servidor Web específico porque ele pode lidar com apenas cerca de 800 conexões das 8.000 conexões necessárias antes de falhar devido à contenção de bloqueio. Quando uma equipe de desenvolvimento subsequente decide avaliar servidores web, em vez de começar do zero, eles podem usar esse resultado negativo já bem documentado como ponto de partida para decidir rapidamente se (a) eles precisam de menos de 800 conexões ou (b) os problemas de contenção de bloqueio foram resolvidos.

Mesmo quando os resultados negativos não se aplicam diretamente ao experimento de outra pessoa, os dados suplementares coletados podem ajudar outros a escolher novos experimentos ou evitar armadilhas em projetos anteriores. Microbenchmarks, antipadrões documentados e postmortems de projetos se encaixam nessa categoria. Você deve considerar o escopo do resultado negativo ao projetar um experimento, porque um resultado negativo amplo ou especialmente robusto ajudará ainda mais seus colegas.

Ferramentas e métodos podem sobreviver ao experimento e informar trabalhos futuros. Como exemplo, ferramentas de benchmarking e geradores de carga podem resultar tão facilmente de um experimento que não confirma quanto de apoio. Muitos webmasters se beneficiaram do trabalho difícil e detalhado que produziu o Apache Bench, um teste de carga de servidor web, embora seus primeiros resultados provavelmente tenham sido decepcionantes.

A criação de ferramentas para experimentos repetíveis também pode ter benefícios indiretos: embora um aplicativo que você cria possa não se beneficiar de ter seu banco de dados em SSDs ou da criação de índices para chaves densas, o próximo talvez possa. Escrever um script que permita testar facilmente essas alterações de configuração garante que você não esqueça ou perca otimizações em seu próximo projeto.

A publicação de resultados negativos melhora a cultura orientada por dados do nosso setor. A contabilização de resultados negativos e insignificância estatística reduz o viés em nossas métricas e fornece um exemplo para outros de como aceitar a incerteza com maturidade. Ao publicar tudo, você incentiva os outros a fazerem o mesmo, e todos na indústria coletivamente aprendem muito mais rapidamente. A SRE já aprendeu essa lição com postmortems de alta qualidade, que tiveram um grande efeito positivo na estabilidade da produção.

Publique seus resultados. Se você estiver interessado nos resultados de um experimento, há uma boa chance de que outras pessoas também estejam. Quando você publica os resultados, essas pessoas não precisam projetar e executar um experimento semelhante. É tentador e comum evitar relatar resultados negativos porque é fácil perceber que o experimento “falhou”. Alguns experimentos estão condenados e tendem a ser pegos pela revisão.

Muitos outros experimentos simplesmente não são relatados porque as pessoas erroneamente acreditam que resultados negativos não são progresso.

Faça sua parte contando a todos sobre os designs, algoritmos e fluxos de trabalho da equipe que você descartou. Incentive seus colegas reconhecendo que resultados negativos fazem parte de uma tomada de risco ponderada e que todo experimento bem projetado tem mérito. Desconfie de qualquer documento de design, revisão de desempenho ou ensaio que não mencione falhas. Tal documento é potencialmente muito filtrado ou o autor não foi rigoroso em seus métodos.

Acima de tudo, publique os resultados que você achar surpreendentes para que outros – incluindo seu eu futuro – não fiquem surpresos.

Cura

Idealmente, agora você reduziu o conjunto de causas possíveis a uma. Em seguida, gostaríamos de provar que é a causa real. Provar definitivamente que um determinado fator causou um problema – reproduzindo-o à vontade – pode ser difícil de fazer em sistemas de produção; muitas vezes, só podemos encontrar fatores causais prováveis, pelos seguintes motivos:

- Os sistemas são complexos. É bem provável que existam vários fatores, cada um dos quais individualmente não é a causa, mas que, em conjunto, são causas.¹⁵ Sistemas reais também são frequentemente dependentes do caminho, de modo que devem estar em um estado específico antes que ocorra uma falha.
- Reproduzir o problema em um sistema de produção ativo pode não ser uma opção, seja devido à complexidade de colocar o sistema em um estado em que a falha possa ser acionada ou porque um tempo de inatividade adicional pode ser inaceitável. Ter um não

¹⁵ Veja [Mea08] sobre como pensar sobre sistemas, e também [Coo00] e [Dek14] sobre as limitações de encontrar uma única causa raiz em vez de examinar o sistema e seu ambiente em busca de fatores causais.

O ambiente de produção pode mitigar esses desafios, embora ao custo de ter outra cópia do sistema para executar.

Depois de encontrar os fatores que causaram o problema, é hora de escrever notas sobre o que deu errado com o sistema, como você rastreou o problema, como corrigiu o problema e como evitar que isso aconteça novamente. Em outras palavras, você precisa escrever uma autópsia (embora, idealmente, o sistema esteja vivo neste momento!).

Estudo de caso

O App Engine,¹⁶ parte da Cloud Platform do Google, é um produto de plataforma como serviço que permite aos desenvolvedores criar serviços na infraestrutura do Google. Um de nossos clientes internos apresentou um relatório de problemas indicando que eles viram recentemente um aumento dramático na latência, no uso da CPU e no número de processos em execução necessários para servir o tráfego para seu aplicativo, um sistema de gerenciamento de conteúdo usado para criar documentação para desenvolvedores.¹⁷ O cliente não conseguiu encontrar nenhuma alteração recente em seu código que se correlacionasse com o aumento de recursos, e não houve um aumento no tráfego para seu aplicativo (veja a [Figura 12-3](#)), então eles estavam se perguntando se um mudança no serviço do App Engine foi responsável.

Nossa investigação descobriu que a latência realmente aumentou em quase uma ordem de magnitude (como mostrado na [Figura 12-4](#)). Simultaneamente, a quantidade de tempo de CPU ([Figura 12-5](#)) e o número de processos de atendimento ([Figura 12-6](#)) quase quadruplicaram.

Claramente algo estava errado. Era hora de começar a solucionar problemas.

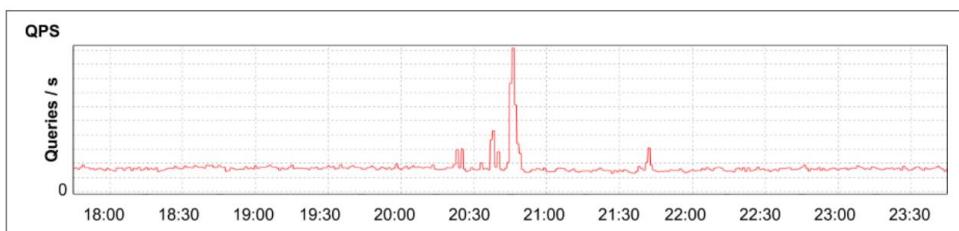


Figura 12-3. Solicitações do aplicativo recebidas por segundo, mostrando um breve pico e voltam ao normal

16 Consulte <https://cloud.google.com/appengine>.

17 Comprimimos e simplificamos este estudo de caso para ajudar na compreensão.

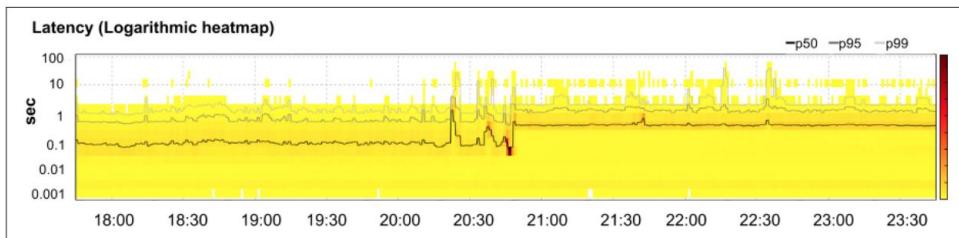


Figura 12-4. Latência do aplicativo, mostrando os percentis 50, 95 e 99 (linhas) com um mapa de calor mostrando quantas solicitações caíram em um determinado bucket de latência a qualquer momento (sombreado)

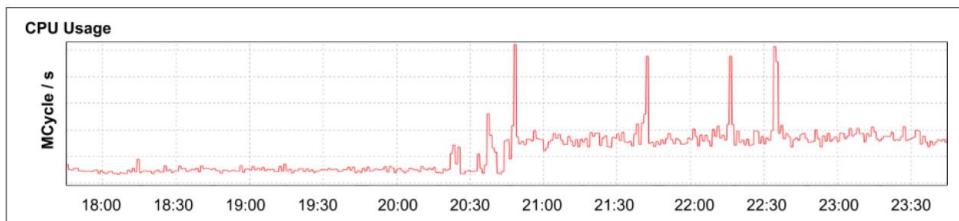


Figura 12-5. Uso agregado da CPU para o aplicativo

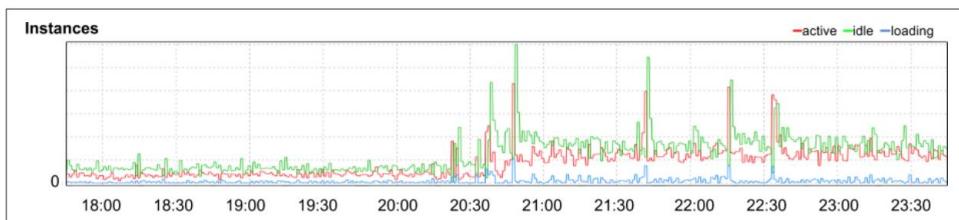


Figura 12-6. Número de instâncias para o aplicativo

Normalmente, um aumento repentino na latência e no uso de recursos indica um aumento no tráfego enviado ao sistema ou uma alteração na configuração do sistema. No entanto, podemos descartar facilmente essas duas causas possíveis: embora um pico no tráfego para o aplicativo por volta das 20h45 possa explicar um breve aumento no uso de recursos, esperamos que o tráfego retorne à linha de base logo após a normalização do volume de solicitações. Esse pico certamente não deveria ter continuado por vários dias, começando quando os desenvolvedores do aplicativo preencheram o relatório e começamos a investigar o problema. Em segundo lugar, a mudança no desempenho aconteceu no sábado, quando nem as mudanças no aplicativo nem no ambiente de produção estavam em andamento. Os pushes de código e os pushes de configuração mais recentes do serviço foram concluídos dias antes. Além disso, se o problema teve origem no serviço, esperaríamos ver efeitos semelhantes em outros aplicativos que usam a mesma infraestrutura. No entanto, nenhum outro aplicativo estava experimentando efeitos semelhantes.

Encaminhamos o relatório do problema para nossos colegas, os desenvolvedores do App Engine, para investigar se o cliente estava encontrando alguma idiossincrasia na infraestrutura de atendimento. Os desenvolvedores também não conseguiram encontrar nenhuma esquisitice. No entanto, um desenvolvedor notou uma correlação entre o aumento da latência e o aumento de uma chamada de API de armazenamento de dados específica, `merge_join`, que geralmente indica indexação abaixo do ideal ao ler do armazenamento de dados. Adicionar um índice composto às propriedades que o aplicativo usa para selecionar objetos do armazenamento de dados aceleraria essas solicitações e, em princípio, aceleraria o aplicativo como um todo, mas precisaríamos descobrir quais propriedades precisavam de indexação. Uma rápida olhada no código do aplicativo não revelou nenhum suspeito óbvio.

Era hora de retirar o maquinário pesado do nosso kit de ferramentas: usando o Dapper [Sig10], rastreamos as etapas de solicitações HTTP individuais - desde o recebimento por um proxy reverso de front-end até o ponto em que o código do aplicativo retornou uma resposta - e analisamos nos RPCs emitidos por cada servidor envolvido no tratamento dessa solicitação. Isso nos permitiria ver quais propriedades foram incluídas nas solicitações ao armazenamento de dados e, em seguida, criar os índices apropriados.

Durante a investigação, descobrimos que as solicitações de conteúdo estático, como imagens, que não eram atendidas pelo armazenamento de dados, também eram muito mais lentas do que o esperado. Observando gráficos com granularidade em nível de arquivo, vimos que suas respostas foram muito mais rápidas apenas alguns dias antes. Isso implicava que a correlação observada entre `merge_join` e o aumento da latência era espúria e que nossa teoria de indexação abaixo do ideal era fatalmente falha.

Examinando as solicitações inesperadamente lentas de conteúdo estático, a maioria dos RPCs enviados do aplicativo eram para um serviço de memcache, portanto, as solicitações deveriam ter sido muito rápidas — na ordem de alguns milissegundos. Esses pedidos acabaram sendo muito rápidos, então o problema não parecia se originar aí. No entanto, entre o momento em que o aplicativo começou a funcionar em uma solicitação e quando fez os primeiros RPCs, houve um período de cerca de 250 ms em que o aplicativo estava fazendo... bem, alguma coisa. Como o App Engine executa o código fornecido pelos usuários, sua equipe de SRE não cria o perfil nem inspeciona o código do aplicativo, portanto, não conseguimos saber o que o aplicativo estava fazendo nesse intervalo. Da mesma forma, o Dapper não pôde deixar de rastrear o que estava acontecendo, pois só pode rastrear chamadas RPC, e nenhuma foi feita durante esse período.

Diante do que era, a essa altura, um grande mistério, decidimos não resolvê-lo... ainda. O cliente tinha um lançamento público agendado para a semana seguinte e não tínhamos certeza de quando poderíamos identificar o problema e corrigi-lo. Em vez disso, recomendamos que o cliente aumente os recursos alocados para seu aplicativo para o tipo de instância mais rica em CPU disponível. Isso reduziu a latência do aplicativo para níveis aceitáveis, embora não tão baixo quanto gostaríamos. Concluímos que a mitigação da latência foi

bom o suficiente para que a equipe pudesse conduzir seu lançamento com sucesso e, em seguida, investigar à vontade.¹⁸

Nesse ponto, suspeitamos que o aplicativo fosse vítima de outra causa comum de aumentos repentinos na latência e no uso de recursos: uma mudança no tipo de trabalho. Vimos um aumento nas gravações no armazenamento de dados do aplicativo, pouco antes de sua latência aumentar, mas como esse aumento não foi muito grande — nem foi sustentado — nós o consideramos coincidência. No entanto, esse comportamento se assemelhava a um padrão comum: uma instância do aplicativo é inicializada lendo objetos do armazenamento de dados e armazenando-os na memória da instância. Ao fazer isso, a instância evita ler configurações raramente alteradas do armazenamento de dados em cada solicitação e, em vez disso, verifica os objetos na memória. Então, o tempo que leva para lidar com solicitações geralmente será dimensionado com a quantidade de dados de configuração.¹⁹ Não conseguimos provar que esse comportamento era a raiz do problema, mas é um antipadrão comum.

Os desenvolvedores de aplicativos adicionaram instrumentação para entender onde o aplicativo estava gastando seu tempo. Eles identificaram um método que era chamado em cada solicitação, que verificava se um usuário tinha acesso à lista de permissões a um determinado caminho. O método utilizava uma camada de cache que buscava minimizar os acessos tanto ao armazenamento de dados quanto ao serviço de memcache, mantendo objetos whitelist na memória das instâncias. Como um dos desenvolvedores do aplicativo observou na investigação: “Ainda não sei onde está o incêndio, mas estou cego pela fumaça que sai desse cache da lista de permissões”.

Algum tempo depois, a causa raiz foi encontrada: devido a um bug de longa data no sistema de controle de acesso do aplicativo, sempre que um caminho específico era acessado, um objeto whitelist era criado e armazenado no armazenamento de dados. No período que antecedeu o lançamento, um scanner de segurança automatizado estava testando o aplicativo em busca de vulnerabilidades e, como efeito colateral, sua verificação produziu milhares de objetos da lista de permissões ao longo de meia hora. Esses objetos supérfluos da lista branca precisavam ser verificados em todas as solicitações ao aplicativo, o que levava a respostas patologicamente lentas, sem causar chamadas RPC do aplicativo para outros serviços. A correção do bug e a remoção desses objetos retornaram o desempenho do aplicativo aos níveis esperados.

¹⁸ Embora o lançamento com um bug não identificado não seja o ideal, muitas vezes é impraticável eliminar todos os bugs conhecidos. Em vez disso, às vezes nos contentamos com as segundas melhores medidas e mitigamos os riscos da melhor maneira possível, usando o bom senso de engenharia.

¹⁹ A pesquisa de armazenamento de dados pode usar um índice para acelerar a comparação, mas uma implementação frequente na memória é uma comparação de loop for simples em todos os objetos armazenados em cache. Se houver apenas alguns objetos, não importará que isso leve um tempo linear, mas isso pode causar um aumento significativo na latência e no uso de recursos à medida que o número de objetos em cache aumenta.

Facilitando a solução de problemas

Há muitas maneiras de simplificar e acelerar a solução de problemas. Talvez os mais fundamentais sejam:

- Criando observabilidade — com métricas de caixa branca e logs estruturados — em cada componente desde o início.
- Projetar sistemas com interfaces bem compreendidas e observáveis entre componentes.

Garantir que as informações estejam disponíveis de maneira consistente em todo o sistema - por exemplo, usando um identificador de solicitação exclusivo em todo o intervalo de RPCs gerados por vários componentes - reduz a necessidade de descobrir qual entrada de log em um componente upstream corresponde a uma entrada de log em um componente a jusante, acelerando o tempo de diagnóstico e recuperação.

Problemas em representar corretamente o estado da realidade em uma mudança de código ou uma mudança de ambiente geralmente levam à necessidade de solucionar problemas. Simplificar, controlar e registrar essas alterações pode reduzir a necessidade de solução de problemas e facilitar quando isso acontecer.

Conclusão

Analisamos algumas etapas que você pode seguir para tornar o processo de solução de problemas claro e compreensível para os novatos, para que eles também possam se tornar eficazes na solução de problemas. Adotar uma abordagem sistemática para solução de problemas – em vez de confiar na sorte ou na experiência – pode ajudar a limitar o tempo de recuperação de seus serviços, levando a uma melhor experiência para seus usuários.

CAPÍTULO 13

Resposta de emergência

**Escrito por Corey Adam Baye
Editado por Diane Bates**

As coisas quebram; isso é vida.

Independentemente dos riscos envolvidos ou do tamanho de uma organização, uma característica vital para a saúde de longo prazo de uma organização e que, consequentemente, diferencia essa organização das outras, é como as pessoas envolvidas respondem a uma emergência. Poucos de nós naturalmente respondem bem durante uma emergência. Uma resposta adequada requer preparação e treinamento prático periódico e pertinente. Estabelecer e manter processos completos de treinamento e testes requer o apoio da diretoria e da administração, além da atenção cuidadosa da equipe. Todos esses elementos são essenciais para promover um ambiente no qual as equipes possam gastar dinheiro, tempo, energia e possivelmente até mesmo tempo de atividade para garantir que sistemas, processos e pessoas respondam de forma eficiente durante um processo.

emergência.

Observe que o capítulo sobre cultura postmortem discute as especificidades de como escrever postmortems para garantir que os incidentes que exigem resposta de emergência também se tornem uma oportunidade de aprendizado (veja o [Capítulo 15](#)). Este capítulo fornece exemplos mais concretos de tais incidentes.

O que fazer quando os sistemas quebram

Em primeiro lugar, não entre em pânico! Você não está sozinho, e o céu não está caindo. Você é um profissional e treinado para lidar com esse tipo de situação. Normalmente, ninguém está em perigo físico - apenas os pobres elétrons estão em perigo. Na pior das hipóteses, metade da Internet está fora do ar.

Então respire fundo... e continue.

Se você se sentir sobrecarregado, atraia mais pessoas. Às vezes pode até ser necessário paginar toda a empresa. Se sua empresa tiver um processo de resposta a incidentes (consulte o [Capítulo 14](#)), certifique-se de estar familiarizado com ele e siga esse processo.

Emergência induzida por teste

O Google adotou uma abordagem proativa para testes de desastres e emergências (consulte [\[Kri12\]](#)). Os SREs quebram nossos sistemas, observam como eles falham e fazem alterações para melhorar a confiabilidade e evitar que as falhas se repitam. Na maioria das vezes, essas falhas controladas acontecem conforme o planejado, e o sistema de destino e os sistemas dependentes se comportam aproximadamente da maneira que esperamos. Identificamos algumas fraquezas ou dependências ocultas e documentamos ações de acompanhamento para corrigir as falhas que descobrimos. No entanto, às vezes nossas suposições e os resultados reais são mundos à parte.

Aqui está um exemplo de um teste que descobriu várias dependências inesperadas.

Detalhes

Queríamos eliminar dependências ocultas em um banco de dados de teste em um dos nossos maiores bancos de dados MySQL distribuídos. O plano era bloquear todo o acesso a apenas um banco de dados em cem. Ninguém previu os resultados que se desenrolariam.

Resposta

Poucos minutos após o início do teste, vários serviços dependentes relataram que os usuários externos e internos não conseguiam acessar os principais sistemas. Alguns sistemas eram intermitentemente ou apenas parcialmente acessíveis.

Assumindo que o teste foi o responsável, SRE imediatamente abortou o exercício. Tentamos reverter a alteração de permissões, mas não obtivemos êxito. Em vez de entrar em pânico, imediatamente discutimos como restaurar o acesso adequado. Usando uma abordagem já testada, restauramos as permissões para as réplicas e failovers. Em um esforço paralelo, contatamos os principais desenvolvedores para corrigir a falha na biblioteca da camada de aplicativo de banco de dados.

Dentro de uma hora da decisão original, todo o acesso foi totalmente restaurado e todos os serviços puderam se conectar novamente. O amplo impacto desse teste motivou uma correção rápida e completa das bibliotecas e um plano de reteste periódico para evitar a recorrência de uma falha tão grande.

Descobertas

O que foi bem

Os serviços dependentes que foram afetados pelo incidente imediatamente aumentaram os problemas dentro da empresa. Presumimos, corretamente, que nosso experimento controlado havia saído do controle e imediatamente abortado o teste.

Conseguimos restaurar totalmente as permissões dentro de uma hora após o primeiro relatório, quando os sistemas começaram a se comportar corretamente. Algumas equipes adotaram uma abordagem diferente e reconfiguraram seus sistemas para evitar o banco de dados de teste. Esses esforços paralelos ajudaram a restaurar o serviço o mais rápido possível.

Os itens de ação de acompanhamento foram resolvidos de forma rápida e completa para evitar uma interrupção semelhante, e instituímos testes periódicos para garantir que falhas semelhantes não se repitam.

O que aprendemos

Embora esse teste tenha sido minuciosamente revisado e considerado bem definido, a realidade revelou que tínhamos uma compreensão insuficiente dessa interação específica entre os sistemas dependentes.

Deixamos de seguir o processo de resposta a incidentes, que havia sido implementado apenas algumas semanas antes e não havia sido amplamente divulgado. Esse processo teria assegurado que todos os serviços e clientes estivessem cientes da interrupção. Para evitar cenários semelhantes no futuro, o SRE refina e testa continuamente nossas ferramentas e processos de resposta a incidentes, além de garantir que as atualizações de nossos procedimentos de gerenciamento de incidentes sejam claramente comunicadas a todas as partes relevantes.

Como não testamos nossos procedimentos de reversão em um ambiente de teste, esses procedimentos apresentavam falhas, o que prolongava a interrupção. Agora, exigimos testes completos dos procedimentos de reversão antes desses testes em grande escala.

Emergência Induzida por Mudança

Como você pode imaginar, o Google tem muitas configurações — configurações complexas — e fazemos alterações constantemente nessa configuração. Para evitar a quebra total de nossos sistemas, realizamos vários testes nas alterações de configuração para garantir que não resultem em comportamento inesperado e indesejado. No entanto, a escala e a complexidade da infraestrutura do Google tornam impossível prever todas as dependências ou interações; às vezes, as alterações de configuração não saem inteiramente de acordo com o planejado.

O seguinte é um exemplo.

Detalhes

Uma mudança de configuração na infraestrutura que ajuda a proteger nossos serviços contra abusos foi lançada globalmente em uma sexta-feira. Essa infraestrutura interage essencialmente com todos os nossos sistemas voltados para o exterior, e a mudança desencadeou um bug de loop de falha nesses sistemas, o que fez com que toda a frota começasse a fazer loop de falha quase simultaneamente. Como a infraestrutura interna do Google também depende de nossos próprios serviços, muitos aplicativos internos também ficaram indisponíveis de repente.

Resposta

Em poucos segundos, os alertas de monitoramento começaram a ser disparados, indicando que alguns sites estavam inativos. Alguns engenheiros de plantão experimentaram simultaneamente o que acreditavam ser uma falha da rede corporativa e se mudaram para salas seguras dedicadas (salas do pânico) com acesso de backup ao ambiente de produção. Eles se juntaram a engenheiros adicionais que estavam lutando com seu acesso corporativo.

Dentro de cinco minutos desse primeiro push de configuração, o engenheiro responsável pelo push, sabendo da interrupção corporativa, mas ainda sem saber da interrupção mais ampla, empurrou outra alteração de configuração para reverter a primeira alteração. Nesse ponto, os serviços começaram a se recuperar.

Dentro de 10 minutos do primeiro impulso, os engenheiros de plantão declararam um incidente e passaram a seguir os procedimentos internos para resposta a incidentes. Eles começaram a notificar o resto da empresa sobre a situação. O engenheiro de push informou aos engenheiros de plantão que a interrupção provavelmente se devia à mudança que havia sido enviada e posteriormente revertida. No entanto, alguns serviços experimentaram bugs não relacionados ou configurações incorretas acionadas pelo evento original e não se recuperaram totalmente por até uma hora.

Descobertas

O que foi bem

Havia vários fatores em jogo que impediram que esse incidente resultasse em uma interrupção de longo prazo de muitos dos sistemas internos do Google.

Para começar, o monitoramento quase imediatamente detectou e nos alertou para o problema. No entanto, deve-se notar que, neste caso, nosso monitoramento não foi o ideal: alertas disparados repetidamente e constantemente, sobrecarregando os plantões e enviando spam para canais de comunicação regulares e de emergência.

Uma vez que o problema foi detectado, o gerenciamento de incidentes geralmente correu bem e as atualizações foram comunicadas com frequência e clareza. Nossos sistemas de comunicação fora de banda mantinham todos conectados mesmo quando algumas das pilhas de software mais complicadas estavam inutilizáveis. Essa experiência nos lembrou por que o SRE mantém sistemas de backup altamente confiáveis e de baixa sobrecarga, que usamos regularmente.

Além desses sistemas de comunicação fora de banda, o Google tem ferramentas de linha de comando e métodos de acesso alternativos que nos permitem realizar atualizações e reverter alterações mesmo quando outras interfaces estão inacessíveis. Essas ferramentas e métodos de acesso funcionaram bem durante a interrupção, com a ressalva de que os engenheiros precisavam estar mais familiarizados com as ferramentas e testá-las com mais rotina.

A infraestrutura do Google oferecia mais uma camada de proteção, pois o sistema afetado limitava a velocidade com que fornecia atualizações completas para novos clientes. Esse comportamento pode ter limitado o loop de falha e impedido uma interrupção completa, permitindo que os trabalhos permaneçam ativos por tempo suficiente para atender a algumas solicitações entre as falhas.

Finalmente, não devemos negligenciar o elemento de sorte na rápida resolução deste incidente: o engenheiro de envio estava seguindo canais de comunicação em tempo real – um nível adicional de diligência que não é uma parte normal do processo de liberação.

O engenheiro de push notou um grande número de reclamações sobre acesso corporativo diretamente após o push e reverteu a mudança quase imediatamente. Se essa rápida reversão não tivesse ocorrido, a interrupção poderia ter durado consideravelmente mais, tornando-se imensamente mais difícil de solucionar.

O que aprendemos

Um push anterior do novo recurso envolveu um canário completo, mas não acionou o mesmo bug, pois não havia exercido uma palavra-chave de configuração muito rara e específica em combinação com o novo recurso. A alteração específica que desencadeou esse bug não foi considerada arriscada e, portanto, seguiu um processo canário menos rigoroso. Quando a alteração foi enviada globalmente, ela usou a combinação de palavra-chave/recurso não testada que acionou a falha.

Ironicamente, as melhorias no canarying e na automação foram programadas para se tornarem mais prioritárias no trimestre seguinte. Este incidente imediatamente aumentou sua prioridade e reforçou a necessidade de canarying completo, independentemente do risco percebido.

Como seria de esperar, o alerta foi vocal durante esse incidente porque todos os locais ficaram essencialmente offline por alguns minutos. Isso interrompeu o trabalho real que estava sendo realizado pelos engenheiros de plantão e tornou a comunicação entre os envolvidos no incidente mais difícil.

O Google conta com nossas próprias ferramentas. Grande parte da pilha de software que usamos para solucionar problemas e comunicar está por trás de tarefas que estavam em loop de falha. Se essa interrupção tivesse durado mais, a depuração teria sido severamente prejudicada.

Emergência Induzida por Processo

Investimos uma quantidade considerável de tempo e energia na automação que gerencia nossa frota de máquinas. É incrível quantos trabalhos se pode iniciar, parar ou reequipar

toda a frota com muito pouco esforço. Às vezes, a eficiência de nossa automação pode ser um pouco assustadora quando as coisas não saem conforme o planejado.

Este é um exemplo em que se mover rápido não era uma coisa tão boa.

Detalhes

Como parte dos testes de automação de rotina, foram enviadas duas solicitações de desligamento consecutivas para a mesma instalação de servidor a ser desativada em breve. No caso da segunda solicitação de turndown, um bug sutil na automação enviou todas as máquinas em todas essas instalações globalmente para a fila do Diskerase, onde seus discos rígidos estavam destinados a serem limpos; consulte “[Automação: habilitando falha em escala](#)” na página 85 para obter mais detalhes.

Resposta

Logo após a emissão da segunda solicitação de desligamento, os engenheiros de plantão receberam uma mensagem quando a primeira instalação de servidor pequeno foi desativada para ser desativada. A investigação deles determinou que as máquinas haviam sido transferidas para a fila do Diskerase, então seguindo o procedimento normal, os engenheiros de plantão drenaram o tráfego do local. Como as máquinas naquele local foram apagadas, elas não puderam responder às solicitações. Para evitar a falha total dessas solicitações, os engenheiros de plantão drenaram o tráfego desse local. O tráfego foi redirecionado para locais que pudessem responder adequadamente às solicitações.

Em pouco tempo, pagers em todos os lugares estavam disparando para todas essas instalações de servidores ao redor do mundo. Em resposta, os engenheiros de plantão desativaram toda a automação da equipe para evitar mais danos. Eles pararam ou congelaram automação adicional e manutenção de produção logo em seguida.

Dentro de uma hora, todo o tráfego foi desviado para outros locais. Embora os usuários possam ter experimentado latências elevadas, suas solicitações foram atendidas. A interrupção foi oficialmente encerrada.

Agora começou a parte difícil: a recuperação. Alguns links de rede estavam relatando congestionamento intenso, então os engenheiros de rede implementaram mitigações à medida que os pontos de estrangulamento surgiam. A instalação de um servidor em um desses locais foi escolhida para ser a primeira de muitas a renascer das cinzas. Dentro de três horas da interrupção inicial, e graças à tenacidade de vários engenheiros, a instalação foi reconstruída e colocada novamente online, aceitando alegremente as solicitações dos usuários mais uma vez.

As equipes dos EUA passaram para seus colegas europeus, e a SRE elaborou um plano para priorizar as reinstalações usando um processo simplificado, mas manual. A equipe foi dividida em três partes, sendo cada parte responsável por uma etapa do processo de reinstalação manual. Dentro de três dias, a grande maioria da capacidade estava novamente online, enquanto quaisquer retardatários seriam recuperados no próximo mês ou dois.

Descobertas

O que foi bem

Os proxies reversos em grandes instalações de servidor são gerenciados de maneira muito diferente dos proxies reversos nessas pequenas instalações, portanto, as grandes instalações não foram afetadas. Engenheiros de plantão foram capazes de mover rapidamente o tráfego de instalações menores para grandes instalações. Por design, essas grandes instalações podem lidar com uma carga total sem dificuldade. No entanto, alguns links de rede ficaram congestionados e, portanto, exigiram que engenheiros de rede desenvolvessem soluções alternativas. Para reduzir o impacto nos usuários finais, os engenheiros de plantão direcionaram as redes congestionadas como sua maior prioridade.

O processo de abertura para as pequenas instalações funcionou de forma eficiente e bem. Do início ao fim, levou menos de uma hora para desligar e limpar com segurança um grande número dessas instalações.

Embora a automação de turndown tenha derrubado rapidamente o monitoramento para as pequenas instalações, os engenheiros de plantão foram capazes de reverter prontamente essas alterações de monitoramento. Isso os ajudou a avaliar a extensão dos danos.

Os engenheiros seguiram rapidamente os protocolos de resposta a incidentes, que amadureceram consideravelmente no ano desde a primeira interrupção descrita neste capítulo. A comunicação e a colaboração em toda a empresa e entre as equipes foram excelentes – uma prova real do programa e treinamento de gerenciamento de incidentes. Todas as mãos das respectivas equipes contribuíram, trazendo sua vasta experiência.

O que aprendemos

A causa raiz era que o servidor de automação de turndown não tinha as verificações de sanidade apropriadas nos comandos que enviava. Quando o servidor executou novamente em resposta ao desligamento inicial com falha, ele recebeu uma resposta vazia para o rack da máquina. Em vez de filtrar a resposta, ele passou o filtro vazio para o banco de dados da máquina, informando ao banco de dados da máquina para Diskerse todas as máquinas envolvidas. Sim, às vezes zero significa tudo. O banco de dados da máquina estava em conformidade, de modo que o fluxo de trabalho de turndown começou a passar pelas máquinas o mais rápido possível.

As reinstalações de máquinas eram lentas e não confiáveis. Esse comportamento deveu-se em grande parte ao uso do Trivial File Transfer Protocol (TFTP) na qualidade de serviço (QoS) de rede mais baixa dos locais distantes. O BIOS de cada máquina do sistema lidava mal com as falhas.¹ Dependendo das placas de rede envolvidas, o BIOS parava ou entrava em um ciclo de reinicialização constante. Eles não conseguiam transferir os arquivos de inicialização em cada ciclo e sobrecarregavam ainda mais os instaladores. De plantão

1 BIOS: Sistema básico de entrada/saída. BIOS é o software embutido em um computador para enviar instruções simples para o hardware, permitindo entrada e saída antes que o sistema operacional seja carregado.

os engenheiros conseguiram corrigir esses problemas de reinstalação reclassificando o tráfego de instalação com prioridade um pouco mais alta e usando a automação para reiniciar qualquer máquina que estivesse travada.

A infraestrutura de reinstalação de máquinas não conseguiu lidar com a configuração simultânea de milhares de máquinas. Essa incapacidade foi em parte devido a uma regressão que impediu que a infraestrutura executasse mais de duas tarefas de configuração por máquina de trabalho. A regressão também usou configurações de QoS impróprias para transferir arquivos e teve timeouts mal ajustados. Ele forçou a reinstalação do kernel, mesmo em máquinas que ainda tinham o kernel adequado e nas quais o Diskerase ainda não havia ocorrido. Para remediar essa situação, os engenheiros de plantão escalaram os responsáveis por essa infraestrutura, que conseguiram reajustá-la rapidamente para suportar essa carga incomum.

Todos os problemas têm soluções

O tempo e a experiência mostraram que os sistemas não apenas quebrarão, mas quebrarão de maneiras que nunca se poderia imaginar anteriormente. Uma das maiores lições que o Google aprendeu é que existe uma solução, mesmo que não seja óbvia, especialmente para a pessoa cujo pager está gritando. Se você não consegue pensar em uma solução, lance sua rede mais longe.

Envolve mais seus companheiros de equipe, procure ajuda, faça o que tiver que fazer, mas faça-o rapidamente. A maior prioridade é resolver o problema em questão rapidamente. Muitas vezes, a pessoa com mais estado é aquela cujas ações de alguma forma desencadearam o evento.

Utilize essa pessoa.

Muito importante, uma vez que a emergência tenha sido mitigada, não se esqueça de reservar um tempo para limpar, registrar o incidente e...

Aprender com o passado. Não o repita.

Mantenha um histórico de

interrupções Não há melhor maneira de aprender do que documentar o que quebrou no passado. A história é sobre aprender com os erros de todos. Seja completo, seja honesto, mas acima de tudo, faça perguntas difíceis. Procure ações específicas que possam impedir que tal interrupção ocorra, não apenas taticamente, mas também estrategicamente. Garanta que todos na empresa possam aprender o que você aprendeu publicando e organizando post-mortems.

Mantenha você e os outros responsáveis por acompanhar as ações específicas detalhadas nessas autópsias. Isso evitará uma interrupção futura quase idêntica e causada por quase os mesmos gatilhos que uma interrupção já documentada. Depois de ter um histórico sólido de aprendizado com interrupções passadas, veja o que você pode fazer para evitar futuras.

Faça as grandes, mesmo improváveis, perguntas: E se...?

Não há teste maior do que a realidade. Faça a si mesmo algumas perguntas grandes e abertas. E se a energia do prédio falhar...? E se os racks de equipamentos de rede estiverem em dois pés de água...? E se o datacenter primário de repente ficar escuro...? E se alguém comprometer seu servidor web...? O que você faz? Quem você chama? Quem vai preencher o cheque? Você tem um plano? Você sabe como reagir? Você sabe como seus sistemas vão reagir? Você poderia minimizar o impacto se isso acontecesse agora? A pessoa sentada ao seu lado poderia fazer o mesmo?

Incentive testes proativos Quando

se trata de falhas, teoria e realidade são duas esferas muito diferentes. Até que seu sistema realmente falhe, você não sabe realmente como esse sistema, seus sistemas dependentes ou seus usuários reagirão. Não confie em suposições ou no que você não pode ou não testou. Você preferiria que uma falha acontecesse às 2h da manhã de sábado, quando a maior parte da empresa ainda está fora do local de formação de equipe na Floresta Negra - ou quando você tem o seu melhor e mais brilhante à mão, monitorando o teste que eles revisaram meticulosamente nas semanas anteriores?

Conclusão

Analisamos três casos diferentes em que partes de nossos sistemas quebraram. Embora todas as três emergências tenham sido acionadas de forma diferente – uma por um teste proativo, outra por uma mudança de configuração e outra ainda por automação de desligamento – as respostas compartilhavam muitas características. Os respondentes não entraram em pânico. Atraíram outros quando acharam necessário. Os respondentes estudaram e aprenderam com interrupções anteriores.

Posteriormente, eles construíram seus sistemas para responder melhor a esses tipos de interrupções. Cada vez que novos modos de falha se apresentavam, os respondentes documentavam esses modos de falha. Esse acompanhamento ajudou outras equipes a aprender como solucionar melhor os problemas e fortalecer seus sistemas contra interrupções semelhantes. Os respondentes testaram proativamente seus sistemas. Esses testes garantiram que as mudanças corrigissem os problemas subjacentes e identificaram outras fraquezas antes que se tornassem interrupções.

E à medida que nossos sistemas evoluem, o ciclo continua, com cada interrupção ou teste resultando em melhorias incrementais nos processos e nos sistemas. Embora os estudos de caso neste capítulo sejam específicos do Google, essa abordagem de resposta a emergências pode ser aplicada ao longo do tempo a qualquer organização de qualquer tamanho.

CAPÍTULO 14**Gerenciando Incidentes**

Escrito por Andrew Stribblehill¹
Editado por Kavita Giuliani

O gerenciamento eficaz de incidentes é fundamental para limitar a interrupção causada por um incidente e restaurar as operações comerciais normais o mais rápido possível. Se você não tiver definido sua resposta a incidentes em potencial com antecedência, o gerenciamento de incidentes baseado em princípios pode sair pela janela em situações da vida real.

Este capítulo percorre um retrato de um incidente que fica fora de controle devido a práticas ad hoc de gerenciamento de incidentes, descreve uma abordagem bem gerenciada para o incidente e analisa como o mesmo incidente poderia ter acontecido se tratado com um incidente que funcionasse bem. gestão.

Incidentes não gerenciados

Coloque-se no lugar de Mary, a engenheira de plantão da The Firm. São duas da tarde de uma sexta-feira e seu pager acabou de explodir. O monitoramento de caixa preta informa que seu serviço parou de atender qualquer tráfego em um datacenter inteiro. Com um suspiro, você coloca seu café de lado e começa o trabalho de consertá-lo. Alguns minutos depois da tarefa, outro alerta informa que um segundo datacenter parou de servir. Em seguida, o terceiro dos cinco datacenters falha. Para agravar a situação, há mais tráfego do que os datacenters restantes podem lidar, então eles começam a sobrecarregar. Antes que você perceba, o serviço está sobrecarregado e incapaz de atender a nenhuma solicitação.

Você olha para os troncos pelo que parece uma eternidade. Milhares de linhas de registro sugerem que há um erro em um dos módulos atualizados recentemente, então você decide

¹ Uma versão anterior deste capítulo apareceu como um artigo em *:login:* (abril de 2015, vol. 40, no. 2).

reverter os servidores para a versão anterior. Quando você vê que a reversão não ajudou, você liga para Josephine, que escreveu a maior parte do código para o serviço agora com hemorragia.

Lembrando que são 3h30 no fuso horário dela, ela concorda em fazer login e dar uma olhada. Seus colegas Sabrina e Robin começam a bisbilhotar em seus próprios terminais. "Apenas olhando", eles dizem.

Agora, um dos processos ligou para o seu chefe e está exigindo com raiva saber por que ele não foi informado sobre o "colapso total desse serviço crítico para os negócios". Inde-

pendentemente, os vice-presidentes estão incomodando você por um ETA, repetidamente perguntando: "Como isso pode ter acontecido?" Você simpatizaria, mas isso exigiria um esforço cognitivo que você está mantendo em reserva para o seu trabalho. Os VPs recorrem à sua experiência anterior em engenharia e fazem comentários irrelevantes, mas difíceis de refutar, como "Aumente o tamanho da página!"

O tempo passa; os dois datacenters restantes falham completamente. Sem que você saiba, Josephine, sonolenta, ligou para Malcolm. Ele teve uma onda cerebral: algo sobre afinidade com a CPU. Ele tinha certeza de que poderia otimizar os processos restantes do servidor se pudesse implantar essa mudança simples no ambiente de produção, então o fez.

Em segundos, os servidores foram reiniciados, pegando a mudança. E então morreu.

A anatomia de um incidente não gerenciado

Observe que todos no cenário anterior estavam fazendo seu trabalho, como o viam.

Como as coisas podem dar tão errado? Alguns perigos comuns fizeram com que esse incidente ficasse fora de controle.

Foco Afiado no Problema Técnico

Tendemos a contratar pessoas como Mary por suas proezas técnicas. Portanto, não é de surpreender que ela estivesse ocupada fazendo mudanças operacionais no sistema, tentando corajosamente resolver o problema. Ela não estava em posição de pensar no quadro geral de como mitigar o problema porque a tarefa técnica em mãos era esmagadora.

Comunicação pobre

Pela mesma razão, Mary estava ocupada demais para se comunicar com clareza. Ninguém sabia quais ações seus colegas de trabalho estavam tomando. Os líderes de negócios ficaram irritados, os clientes ficaram frustrados e outros engenheiros que poderiam ter ajudado na depuração ou correção do problema não foram usados de forma eficaz.

O freelancer

Malcolm estava fazendo mudanças no sistema com a melhor das intenções. No entanto, ele não coordenou com seus colegas de trabalho – nem mesmo com Mary, que era tecnicamente responsável pela solução de problemas. Suas mudanças tornaram uma situação ruim muito pior.

Elementos do Processo de Gerenciamento de Incidentes

As habilidades e práticas de gerenciamento de incidentes existem para canalizar as energias de indivíduos entusiasmados. O sistema de gerenciamento de incidentes do Google é baseado no Incident Command System,² conhecido por sua clareza e escalabilidade.

Um processo de gerenciamento de incidentes bem projetado tem os seguintes recursos.

Separação recursiva de responsabilidades É importante

garantir que todos os envolvidos no incidente conheçam seu papel e não se desviem do território de outra pessoa. Um tanto contra-intuitivo, uma separação clara de responsabilidades permite aos indivíduos mais autonomia do que eles poderiam ter, uma vez que eles não precisam adivinhar seus colegas.

Se a carga de um determinado membro se tornar excessiva, essa pessoa precisa pedir mais funcionários ao líder de planejamento. Eles devem então delegar trabalho a outros, uma tarefa que pode implicar na criação de subincidentes. Alternativamente, um líder de função pode delegar componentes do sistema a colegas, que relatam informações de alto nível aos líderes.

Várias funções distintas devem ser delegadas a indivíduos específicos:

Comando do Incidente

O comandante do incidente detém o estado de alto nível sobre o incidente. Eles estruturam a força-tarefa de resposta a incidentes, atribuindo responsabilidades de acordo com a necessidade e prioridade. De fato, o comandante ocupa todos os cargos que não delegou. Se apropriado, eles podem remover os obstáculos que impedem que as operações funcionem com mais eficiência.

Trabalho operacional

O líder de operações trabalha com o comandante do incidente para responder ao incidente aplicando ferramentas operacionais à tarefa em questão. A equipe de operações deve ser o único grupo que modifica o sistema durante um incidente.

Comunicação

Essa pessoa é a face pública da força-tarefa de resposta a incidentes. Seus deveres definitivamente incluem a emissão de atualizações periódicas para a equipe de resposta a incidentes e partes interessadas (geralmente por e-mail) e podem se estender a tarefas como manter o documento de incidente preciso e atualizado.

² Consulte <http://www.fema.gov/national-incident-management-system> para mais detalhes.

Planejamento A função de planejamento dá suporte às operações ao lidar com problemas de longo prazo, como arquivar bugs, pedir jantar, organizar entregas e rastrear como o sistema divergiu da norma para que possa ser revertido assim que o incidente for resolvido.

Um Posto de Comando Reconhecido

As partes interessadas precisam entender onde podem interagir com o comandante do incidente. Em muitas situações, é apropriado localizar os membros da força-tarefa do incidente em uma central designada “Sala de Guerra”. Outras equipes podem preferir trabalhar em suas mesas, mantendo-se alertas para atualizações de incidentes por e-mail e IRC.

O Google descobriu que o IRC é um grande benefício na resposta a incidentes. O IRC é muito confiável e pode ser usado como um registro de comunicações sobre este evento, e esse registro é inestimável para manter em mente as mudanças de estado detalhadas. Também escrevemos bots que registram tráfego relacionado a incidentes (o que é útil para análise postmortem) e outros bots que registram eventos, como alertas para o canal. O IRC também é um meio conveniente sobre o qual as equipes geograficamente distribuídas podem se coordenar.

Documento do Estado do Incidente ao Vivo

A responsabilidade mais importante do comandante do incidente é manter um documento de incidente vivo. Isso pode ficar em um wiki, mas idealmente deve ser editado por várias pessoas simultaneamente. A maioria de nossas equipes usa o Google Docs, embora o Google Docs SRE use o Google Sites: afinal, dependendo do software que você está tentando corrigir como parte de seu sistema de gerenciamento de incidentes, dificilmente terminará bem.

Consulte o [Apêndice C](#) para obter um exemplo de documento de incidente. Este documento vivo pode ser confuso, mas deve ser funcional. O uso de um modelo facilita a geração dessa documentação e a manutenção das informações mais importantes no topo a torna mais útil. Guarde essa documentação para análise post-mortem e, se necessário, meta-análise.

Clear, Live Handoff É

essencial que o posto de comandante do incidente seja claramente entregue ao final do dia de trabalho. Se você estiver transferindo o comando para alguém em outro local, poderá atualizar de maneira simples e segura o novo comandante do incidente por telefone ou videochamada. Uma vez que o novo comandante do incidente esteja totalmente informado, o comandante de saída deve ser explícito em sua transferência, declarando especificamente: “Você agora é o comandante do incidente, ok?”, e não deve deixar a ligação até receber um firme reconhecimento da transferência. A transferência deve ser comunicada a outras pessoas que trabalham no incidente para que fique claro quem está liderando os esforços de gerenciamento de incidentes o tempo todo.

Um incidente gerenciado

Agora vamos examinar como esse incidente poderia ter acontecido se fosse tratado usando princípios de gerenciamento de incidentes.

São 14h, e Mary está tomando seu terceiro café do dia. O tom áspero do pager a surpreende, e ela engole a bebida. Problema: um datacenter parou de atender tráfego. Ela começa a investigar. Logo outro alerta é acionado e o segundo datacenter de cinco está fora de serviço. Como esse é um problema que cresce rapidamente, ela sabe que se beneficiará da estrutura de sua estrutura de gerenciamento de incidentes.

Mary pega Sabrina. "Você pode assumir o comando?" Assentindo com a cabeça, Sabrina rapidamente obtém um resumo do que ocorreu até agora com Mary. Ela captura esses detalhes em um e-mail que envia para uma lista de discussão pré-estabelecida. Sabrina reconhece que ainda não pode avaliar o impacto do incidente, então pede a avaliação de Mary.

Mary responde: "Os usuários ainda não foram impactados; vamos apenas esperar que não percamos um terceiro datacenter." Sabrina registra a resposta de Mary em um documento de incidente ao vivo.

Quando o terceiro alerta é acionado, Sabrina vê o alerta entre as conversas de depuração no IRC e rapidamente acompanha o tópico de e-mail com uma atualização. O encadeamento mantém os VPs a par do status de alto nível sem prendê-los em minúcias. Sabrina pede a um representante de comunicações externo para começar a redigir as mensagens do usuário. Ela então acompanha Mary para ver se eles devem entrar em contato com o desenvolvedor de plantão (atualmente Josephine). Recebendo a aprovação de Mary, Sabrina volta para Josephine.

Quando Josephine entra, Robin já se ofereceu para ajudar. Sabrina lembra Robin e Josephine que eles devem priorizar todas as tarefas delegadas a eles por Mary e que devem manter Mary informada sobre quaisquer ações adicionais que tomarem. Robin e Josephine rapidamente se familiarizam com a situação atual lendo o documento do incidente.

Até agora, Mary tentou a versão binária antiga e achou ruim: ela murmura isso para Robin, que atualiza o IRC para dizer que essa tentativa de correção não funcionou. Sabrina cola esta atualização no documento de gerenciamento de incidentes ao vivo.

Às 17h, Sabrina começa a encontrar funcionários substitutos para lidar com o incidente, já que ela e seus colegas estão prestes a ir para casa. Ela atualiza o documento do incidente. Uma breve conferência telefônica ocorre às 5h45 para que todos estejam cientes da situação atual. Às 18h, eles entregam suas responsabilidades aos colegas no escritório da irmã.

Mary volta ao trabalho na manhã seguinte e descobre que seus colegas transatlânticos assumiram a responsabilidade pelo bug, atenuaram o problema, encerraram o incidente e começaram a trabalhar na autópsia. Problema resolvido, ela faz um café fresco e se acomoda para planejar melhorias estruturais para que problemas dessa categoria não aflijam novamente a equipe.

Quando declarar um incidente

É melhor declarar um incidente com antecedência e, em seguida, encontrar uma solução simples e encerrar o incidente do que ter que transformar as horas da estrutura de gerenciamento de incidentes em um problema crescente. Defina condições claras para declarar um incidente. Minha equipe segue estas diretrizes amplas — se alguma das seguintes condições for verdadeira, o evento é um incidente:

- Você precisa envolver uma segunda equipe na solução do problema? • A interrupção é visível para os clientes?
- O problema não foi resolvido mesmo após uma hora de análise concentrada?

A proficiência em gerenciamento de incidentes diminui rapidamente quando não está em uso constante. Então, como os engenheiros podem manter suas habilidades de gerenciamento de incidentes atualizadas – lidar com mais incidentes? Felizmente, a estrutura de gerenciamento de incidentes pode ser aplicada a outras mudanças operacionais que precisam abranger fusos horários e/ou equipes. Se você usar a estrutura com frequência como parte regular de seus procedimentos de gerenciamento de mudanças, poderá seguir facilmente essa estrutura quando ocorrer um incidente real. Se sua organização realiza testes de recuperação de desastres (você deveria, é divertido: veja [Kri12]), o gerenciamento de incidentes deve fazer parte desse processo de teste. Muitas vezes interpretamos a resposta a um problema de plantão que já foi resolvido, talvez por colegas em outro local, para nos familiarizarmos ainda mais com o gerenciamento de incidentes.

Em suma

Descobrimos que, ao formular uma estratégia de gerenciamento de incidentes com antecedência, estruturar esse plano para ser dimensionado sem problemas e colocar o plano em prática regularmente, conseguimos reduzir nosso tempo médio de recuperação e fornecer à equipe uma maneira menos estressante de trabalhar em problemas emergentes. Qualquer organização preocupada com a confiabilidade se beneficiaria de uma estratégia semelhante.

Melhores Práticas para a Gestão de Incidentes Priorizar.

Pare o sangramento, restaure o serviço e preserve a evidência da causa raiz.

Preparar. Desenvolva e documente seus procedimentos de gerenciamento de incidentes com antecedência, em consulta com os participantes do incidente.

Confiar. Dê total autonomia dentro da função atribuída a todos os participantes do incidente.

Introspecção. Preste atenção ao seu estado emocional ao responder a um incidente. Se você começar a se sentir em pânico ou sobrecarregado, solicite mais apoio.

Considere alternativas. Considere periodicamente suas opções e reavalie se ainda faz sentido continuar o que você está fazendo ou se deve tomar outra atitude na resposta a incidentes.

Prática. Use o processo rotineiramente para que se torne uma segunda natureza.

Mude-o ao redor. Você foi o comandante do incidente da última vez? Assuma um papel diferente desta vez. Incentive cada membro da equipe a adquirir familiaridade com cada função.

CAPÍTULO 15

Cultura Postmortem: Aprendendo com o Fracasso

**Escrito por John Lunney e Sue Lueder
Editado por Gary O'Connor**

O custo do fracasso é a educação.

—Devin Carraway

Como SREs, trabalhamos com sistemas distribuídos de grande escala, complexos. Aprimoramos constantemente nossos serviços com novos recursos e adicionamos novos sistemas. Incidentes e interrupções são inevitáveis, dada a nossa escala e velocidade de mudança. Quando ocorre um incidente, corrigimos o problema subjacente e os serviços retornam às suas condições normais de operação. A menos que tenhamos algum processo formalizado de aprendizado com esses incidentes, eles podem se repetir ad infinitum. Se não forem controlados, os incidentes podem se multiplicar em complexidade ou mesmo em cascata, sobrecarregando um sistema e seus operadores e, por fim, impactando nossos usuários. Portanto, postmortems são uma ferramenta essencial para SRE.

O conceito postmortem é bem conhecido na indústria de tecnologia [All12]. Um post-mortem é um registro escrito de um incidente, seu impacto, as ações tomadas para mitigá-lo ou resolvê-lo, a(s) causa(s) raiz(es) e as ações de acompanhamento para evitar que o incidente se repita. Este capítulo descreve critérios para decidir quando realizar autópsias, algumas práticas recomendadas sobre autópsias e conselhos sobre como cultivar uma cultura postmortem com base na experiência que adquirimos ao longo dos anos.

A filosofia postmortem do Google

Os principais objetivos de escrever uma autópsia são garantir que o incidente seja documentado, que todas as causas-raiz contribuintes sejam bem compreendidas e, especialmente, que ações preventivas eficazes sejam implementadas para reduzir a probabilidade e/ou impacto de recorrência. Uma pesquisa detalhada das técnicas de análise de causa raiz está além do escopo deste capítulo (em vez disso, consulte [Roo04]); no entanto, artigos, práticas recomendadas e ferramentas são abundantes

no domínio da qualidade do sistema. Nossas equipes utilizam uma variedade de técnicas para análise de causa raiz e escolhem a técnica mais adequada aos seus serviços. Postmortems são esperados após qualquer evento indesejável significativo. Escrever uma autópsia não é punição – é uma oportunidade de aprendizado para toda a empresa. O processo post-mortem apresenta um custo inerente em termos de tempo ou esforço, por isso somos deliberados na escolha de quando escrever um. As equipes têm alguma flexibilidade interna, mas os gatilhos postmortem comuns incluem:

- Tempo de inatividade visível ao usuário ou degradação além de um determinado limite
- Perda de dados de qualquer tipo • Intervenção de engenheiro de plantão (reversão de liberação, redirecionamento de tráfego etc.) • Um tempo de resolução acima de algum limite
- Uma falha de monitoramento (que geralmente implica descoberta manual de incidentes)

É importante definir critérios postmortem antes que ocorra um incidente para que todos saibam quando uma autópsia é necessária. Além desses gatilhos objetivos, qualquer parte interessada pode solicitar uma autópsia para um evento.

Postmortems sem culpa são um princípio da cultura SRE. Para que uma autópsia seja realmente isenta de culpa, ela deve se concentrar na identificação das causas que contribuíram para o incidente sem indicar qualquer indivíduo ou equipe por comportamento ruim ou inadequado. Uma autópsia escrita sem culpa supõe que todos os envolvidos em um incidente tiveram boas intenções e fizeram a coisa certa com as informações que tinham. Se prevalecer uma cultura de apontar o dedo e envergonhar indivíduos ou equipes por fazer a coisa “errada”, as pessoas não trarão problemas à tona por medo de punição.

A cultura sem culpa se originou nas indústrias de saúde e aviônica, onde os erros podem ser fatais. Essas indústrias nutrem um ambiente onde cada “erro” é visto como uma oportunidade para fortalecer o sistema. Quando as autópsias deixam de atribuir culpas para investigar as razões sistemáticas pelas quais um indivíduo ou equipe tinha informações incompletas ou incorretas, planos de prevenção eficazes podem ser implementados. Você não pode “consertar” as pessoas, mas pode consertar sistemas e processos para melhor apoiar as pessoas que fazem as escolhas certas ao projetar e manter sistemas complexos.

Quando ocorre uma interrupção, uma autópsia não é escrita como uma formalidade a ser esquecida. Em vez disso, a autópsia é vista pelos engenheiros como uma oportunidade não apenas para corrigir uma fraqueza, mas para tornar o Google mais resiliente como um todo. Embora uma autópsia sem culpa não apenas desabafe a frustração apontando o dedo, ela deve apontar onde e como os serviços podem ser melhorados. Aqui estão dois exemplos:

Apontando dedos

"Precisamos reescrever todo o complicado sistema de back-end! Está quebrando semanalmente nos últimos três trimestres e tenho certeza de que estamos todos cansados de consertar as coisas de dois em dois. Sério, se eu for paginado mais uma vez, eu mesmo vou reescrever..."

Sem culpa

"Um item de ação para reescrever todo o sistema de back-end pode realmente impedir que essas páginas irritantes continuem a acontecer, e o manual de manutenção para esta versão é bastante longo e muito difícil de ser totalmente treinado. Tenho certeza que nossos futuros visitantes irão nos agradecer!"

Melhor prática: evitar culpa e mantê-la construtiva Postmortems

sem culpa podem ser difíceis de escrever, porque o formato postmortem identifica claramente as ações que levaram ao incidente. Remover a culpa de uma autópsia dá às pessoas a confiança para escalar os problemas sem medo. Também é importante não estigmatizar a produção frequente de necropsias por uma pessoa ou equipe. Uma atmosfera de culpa corre o risco de criar uma cultura na qual incidentes e problemas são varridos para debaixo do tapete, levando a um risco maior para a organização [Rapaz13].

Colabore e compartilhe conhecimento

Valorizamos a colaboração e o processo pós-morte não é exceção. O fluxo de trabalho pós-morte inclui colaboração e compartilhamento de conhecimento em todas as etapas.

Nossos documentos postmortem são Google Docs, com um modelo interno (ver [Apêndice D](#)). Independentemente da ferramenta específica que você usa, procure os seguintes recursos principais:

Colaboração em tempo real

Permite a coleta rápida de dados e ideias. Essencial durante a criação inicial de uma autópsia.

Um sistema aberto de comentários/anotações

Facilita as soluções de crowdsourcing e melhora a cobertura.

Notificações por e-mail

Podem ser direcionadas a colaboradores dentro do documento ou usadas para inserir outros em loop.

Escrever uma autópsia também envolve revisão formal e publicação. Na prática, as equipes compartilham o primeiro rascunho post-mortem internamente e solicitam a um grupo de engenheiros seniores que avaliem o rascunho quanto à completude. Os critérios de revisão podem incluir:

- Os principais dados de incidentes foram coletados para a posteridade? • As avaliações de impacto estão completas? • A causa raiz foi suficientemente profunda? • O plano de ação é apropriado e as correções de bugs resultantes têm prioridade apropriada?
- Compartilhamos o resultado com as partes interessadas relevantes?

Depois que a revisão inicial é concluída, a autópsia é compartilhada de forma mais ampla, normalmente com a equipe de engenharia maior ou em uma lista de discussão interna. Nossa objetivo é compartilhar postmortems para o público mais amplo possível que se beneficiaria do conhecimento ou das lições transmitidas. O Google tem regras rígidas sobre o acesso a qualquer informação que possa identificar um usuário,¹ e até mesmo documentos internos, como post-mortems, nunca incluem essas informações.

Melhor prática: nenhuma autópsia deixada sem revisão

Uma autópsia não revisada poderia muito bem nunca ter existido. Para garantir que cada rascunho concluído seja revisado, incentivamos sessões regulares de revisão para autópsias. Nessas reuniões, é importante encerrar quaisquer discussões e comentários em andamento, capturar ideias e finalizar o estado.

Uma vez que os envolvidos estejam satisfeitos com o documento e seus itens de ação, a autópsia é adicionada a um repositório de incidentes passados da equipe ou organização.² O compartilhamento transparente torna mais fácil para outros encontrar e aprender com a autópsia.

Apresentando uma cultura postmortem

É mais fácil falar do que fazer a introdução de uma cultura post-mortem em sua organização; tal esforço requer cultivo e reforço contínuos. Reforçamos uma cultura postmortem colaborativa por meio da participação ativa da alta administração no processo de revisão e colaboração. A administração pode encorajar essa cultura, mas post-mortems irrepreensíveis são, idealmente, o produto da automotivação do engenheiro. No espírito de nutrir a cultura postmortem, os SREs criam proativamente atividades que disseminam o que aprendemos sobre a infraestrutura do sistema. Alguns exemplos de atividades incluem:

Postmortem do mês Em um

boletim informativo mensal, um postmortem interessante e bem escrito é compartilhado com toda a organização.

1 Consulte <http://www.google.com/policies/privacy/>.

2 Se você deseja iniciar seu próprio repositório, o Etsy lançou o [Morgue](#), uma ferramenta para gerenciar postmortems.

Grupo pós-morte do Google+

Este grupo partilha e discute autópsias internas e externas, melhores práticas e comentários sobre autópsias.

Clubes de leitura pós-morte

As equipes organizam regularmente clubes de leitura pós-morte, nos quais uma autópsia interessante ou impactante é trazida à mesa (juntamente com alguns petiscos saborosos) para um diálogo aberto com participantes, não participantes e novos Googlers sobre o que aconteceu, quais lições incidente transmitido e as consequências do incidente. Muitas vezes, a autópsia que está sendo revisada tem meses ou anos!

Roda do Infortúnio

Novos SREs são frequentemente tratados com o exercício Roda do Infortúnio (consulte “[Desastre Role Playing](#)” na [página 401](#)), no qual uma autópsia anterior é reencenada com um elenco de engenheiros desempenhando papéis conforme estabelecido na autópsia. O comandante do incidente original atende para ajudar a tornar a experiência o mais “real” possível.

Um dos maiores desafios da introdução de postmortems em uma organização é que alguns podem questionar seu valor devido ao custo de sua preparação. As seguintes estratégias podem ajudar a enfrentar esse desafio:

- Facilite postmortems no fluxo de trabalho. Um período de teste com várias autópsias completas e bem-sucedidas pode ajudar a provar seu valor, além de ajudar a identificar quais critérios devem iniciar uma autópsia.
- Certifique-se de que escrever post-mortems eficazes seja uma prática recompensada e celebrada, tanto publicamente por meio dos métodos sociais mencionados anteriormente, quanto por meio de gerenciamento de desempenho individual e de equipe.
- Incentivar o reconhecimento e a participação da liderança sênior. Até Larry Página fala sobre o alto valor das autópsias!

Prática recomendada: recompensar visivelmente as pessoas por fazerem a coisa certa

Os fundadores do Google, Larry Page e Sergey Brin, apresentam o TGIF, um evento semanal realizado ao vivo em nossa sede em Mountain View, Califórnia, e transmitido para os escritórios do Google em todo o mundo. Um TGIF de 2014 concentrou-se em “The Art of the Postmortem”, que apresentou a discussão do SRE sobre incidentes de alto impacto. Um SRE discutiu um lançamento que ele havia empurrado recentemente; apesar dos testes completos, uma interação inesperada derrubou inadvertidamente um serviço crítico por quatro minutos. O incidente durou apenas quatro minutos porque o SRE teve a presença de espírito de reverter a mudança imediatamente, evitando uma interrupção muito mais longa e em maior escala. Esse engenheiro não apenas recebeu dois bônus de colegas³ imediatamente em reconhecimento ao tratamento rápido e sensato do incidente, mas também recebeu uma enorme salva de palmas do público da TGIF, que incluía os fundadores da empresa e uma audiência de Googlers aos milhares. Além de um fórum tão visível, o Google tem uma série de redes sociais internas que direcionam elogios de colegas para postmortems bem escritos e tratamento excepcional de incidentes. Este é um exemplo de muitos em que o reconhecimento dessas contribuições vem de colegas, CEOs e todos os demais.⁴

Melhor prática: peça feedback sobre a eficácia pós-morte

No Google, nos esforçamos para resolver os problemas à medida que surgem e compartilhar inovações internamente. Fazemos pesquisas regulares com nossas equipes sobre como o processo post-mortem está apoiando seus objetivos e como o processo pode ser melhorado. Fazemos perguntas como: A cultura está apoiando seu trabalho? Escrever uma autópsia envolve muito trabalho (ver [Capítulo 5](#))? Quais práticas recomendadas sua equipe recomenda para outras equipes? Que tipo de ferramentas você gostaria de ver desenvolvidas? Os resultados da pesquisa dão aos SREs nas trincheiras a oportunidade de pedir melhorias que aumentarão a eficácia da cultura post-mortem.

Além dos aspectos operacionais de gerenciamento e acompanhamento de incidentes, a prática pós-morte foi incorporada à cultura do Google: agora é uma norma cultural que qualquer incidente significativo seja seguido por uma autópsia abrangente.

³ O programa Peer Bonus do Google é uma forma de colegas Googlers reconhecerem colegas por esforços excepcionais e envolve uma recompensa simbólica em dinheiro.

⁴ Para uma discussão mais aprofundada deste incidente em particular, consulte o [Capítulo 13](#).

Conclusão e Melhorias Contínuas

Podemos dizer com confiança que, graças ao nosso investimento contínuo no cultivo de uma cultura postmortem, o Google resiste a menos interrupções e promove uma melhor experiência do usuário. Nosso grupo de trabalho “Postmortems no Google” é um exemplo de nosso compromisso com a cultura de autópsias sem culpa. Esse grupo coordena os esforços postmortem em toda a empresa: reunindo modelos postmortem, automatizando a criação postmortem com dados de ferramentas usadas durante um incidente e ajudando a automatizar a extração de dados postmortem para que possamos realizar análises de tendências. Conseguimos colaborar em práticas recomendadas de produtos tão diferentes quanto YouTube, Google Fiber, Gmail, Google Cloud, AdWords e Google Maps. Embora esses produtos sejam bastante diversos, todos eles realizam autópsias com o objetivo universal de aprender com nossas horas mais sombrias.

Com um grande número de postmortems produzidos a cada mês no Google, as ferramentas para agregar postmortems estão se tornando cada vez mais úteis. Essas ferramentas nos ajudam a identificar temas e áreas comuns para melhoria nos limites do produto. Para facilitar a compreensão e a análise automatizada, recentemente aprimoramos nosso modelo postmortem (consulte o [Apêndice D](#)) com campos de metadados adicionais. O trabalho futuro neste domínio inclui aprendizado de máquina para ajudar a prever nossas fraquezas, facilitar a investigação de incidentes em tempo real e reduzir incidentes duplicados.

CAPÍTULO 16

Rastreamento de interrupções

**Escrito por Gabe Krabbe
Editado por Lisa Carey**

Melhorar a confiabilidade ao longo do tempo só é possível se você começar a partir de uma linha de base conhecida e puder acompanhar o progresso. "Outalator", nosso rastreador de interrupções, é uma das ferramentas que usamos para fazer exatamente isso. O Outalator é um sistema que recebe passivamente todos os alertas enviados por nossos sistemas de monitoramento e nos permite anotar, agrupar e analisar esses dados.

Aprender sistematicamente com os problemas passados é essencial para uma gestão de serviço eficaz. Postmortems (veja o [Capítulo 15](#)) fornecem informações detalhadas para interrupções individuais, mas são apenas parte da resposta. Eles são escritos apenas para incidentes com um grande impacto, portanto, problemas que têm um impacto individualmente pequeno, mas são frequentes e amplos, não se enquadram em seu escopo. Da mesma forma, as autópsias tendem a fornecer insights úteis para melhorar um único serviço ou conjunto de serviços, mas podem perder oportunidades que teriam um efeito pequeno em casos individuais, ou oportunidades que têm uma relação custo/benefício ruim, mas que teriam grande impacto horizontal.¹ Também podemos obter informações úteis de perguntas como "Quantos alertas por turno de plantão essa equipe recebe?", "Qual é a proporção de alertas acionáveis/não acionáveis no último trimestre?", ou mesmo simplesmente "Qual dos os serviços que esta equipe gerencia gera mais trabalho?"

¹ Por exemplo, pode ser necessário um esforço significativo de engenharia para fazer uma alteração específica no Bigtable que tenha apenas um pequeno efeito de mitigação para uma interrupção. No entanto, se essa mesma mitigação estiver disponível em muitos eventos, o esforço de engenharia pode valer a pena.

Escada rolante

No Google, todas as notificações de alerta para SRE compartilham um sistema central replicado que rastreia se um humano confirmou o recebimento da notificação. Se nenhuma confirmação for recebida após um intervalo configurado, o sistema escala para o(s) próximo(s) destino(s) configurado(s) – por exemplo, de plantão primário para secundário. Esse sistema, chamado “The Escalator”, foi inicialmente projetado como uma ferramenta amplamente transparente que recebia cópias de e-mails enviados para aliases de plantão. Essa funcionalidade permitiu que o Escalator se integrasse facilmente aos fluxos de trabalho existentes sem exigir nenhuma alteração no comportamento do usuário (ou, no momento, monitorar o comportamento do sistema).

Outalador

Seguindo o exemplo do Escalator, onde adicionamos recursos úteis à infraestrutura existente, criamos um sistema que lidaria não apenas com as notificações de escalada individuais, mas com a próxima camada de abstração: interrupções.

O Outalator permite que os usuários visualizem uma lista de notificações intercaladas no tempo para várias filas de uma só vez, em vez de exigir que um usuário alterne manualmente entre as filas. A [Figura 16-1](#) mostra várias filas conforme aparecem na visualização de filas do Outalator. Essa funcionalidade é útil porque frequentemente uma única equipe de SRE é o principal ponto de contato para serviços com alvos de escalonamento secundários distintos, geralmente as equipes de desenvolvedores.

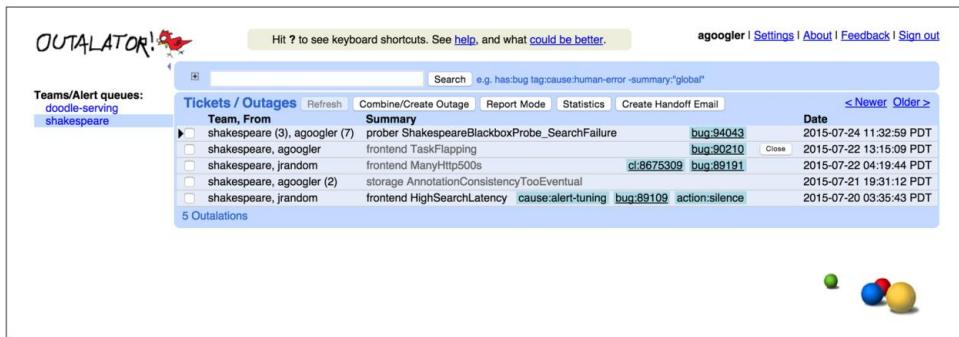


Figura 16-1. Visualização da fila do Outalator

O Outalator armazena uma cópia da notificação original e permite anotar incidentes.

Por conveniência, ele também recebe e salva silenciosamente uma cópia de qualquer resposta de e-mail.

Como alguns acompanhamentos são menos úteis do que outros (por exemplo, uma resposta a todos enviada com o único objetivo de adicionar mais destinatários à lista de cc), as anotações podem ser marcadas como “importantes”. Se uma anotação for importante, outras partes da mensagem serão recolhidas na interface para reduzir a confusão. Juntos, isso fornece mais contexto ao se referir a um incidente do que um encadeamento de e-mail possivelmente fragmentado.

Múltiplas notificações de escalonamento (“alertas”) podem ser combinadas em uma única entidade (“incidente”) no Outalator. Essas notificações podem estar relacionadas ao mesmo incidente único, podem ser eventos auditáveis não relacionados e desinteressantes, como acesso privilegiado ao banco de dados, ou podem ser falhas de monitoramento espúrias. Essa funcionalidade de agrupamento, mostrada na [Figura 16-2](#), organiza as exibições de visão geral e permite a análise separada de “incidentes por dia” versus “alertas por dia”.

The screenshot shows the Outalator ticket interface for ticket `m23bca7d408000005`. The ticket details include:

- Summary:** frontend ManyHttp500s
- Team:** shakespeare
- Escalation Level:** 1
- State:** closed
- Parent Outage:** None
- Short Link:** <http://o/e/m23bca7d408000005>
- Escalator Link:** <http://escalator/2575117616058204165>
- Tags:** bug:89191, c1:8675309

Content: Condition 'ManyHttp500s' was triggered.

Similar Tickets:

Created	Summary
2015-08-10 22:56:56 PDT	frontend ManyHttp500s
2015-08-10 22:56:05 PDT	mobile_frontend ManyHttp500s
2015-07-20 03:36:00 PDT	frontend HighSearchLatency
2015-07-17 00:33:06 PDT	frontend ManyHttp500s

Followups:

Time	From	To
2015-07-22 04:20:01 PDT	agoogler	agoogler
2015-07-22 05:10:52 PDT	jrandom	jrandom

Frontend task dropping requests into the bit bucket, fixed by <c1:8675309>. Will file a bug to deploy new build to production.

Ticket Events:

Time	Event Type	Creator	Source	Client	Team	Level
2015-07-22 04:19:46 PDT	CREATE	shakespeare-alerts	borgmon	mail	shakespeare	1
2015-07-22 04:20:01 PDT	ACK	jrandom-pager	16505551212	telebot	shakespeare	1

Add a Note: [Annotate]

Figura 16-2. Exibição do Outalator de um incidente

Construindo seu próprio Outalator

Muitas organizações usam sistemas de mensagens como Slack, Hipchat ou mesmo IRC para comunicação interna e/ou atualização de painéis de status. Esses sistemas são ótimos lugares para se conectar com um sistema como o Outalator.

Agregação Um

único evento pode, e muitas vezes irá, acionar vários alertas. Por exemplo, falhas de rede causam tempos limite e serviços de back-end inacessíveis para todos, de modo que todas as equipes afetadas recebem seus próprios alertas, incluindo os proprietários dos serviços de back-end; enquanto isso, o centro de operações da rede terá suas próprias buzinas tocando. No entanto, problemas ainda menores que afetam um único serviço podem acionar vários alertas devido ao diagnóstico de várias condições de erro. Embora valha a pena tentar minimizar o número de alertas acionados por um único evento, o acionamento de vários alertas é inevitável na maioria dos cálculos de compensação entre falsos positivos e falsos negativos.

A capacidade de agrupar vários alertas em um único incidente é fundamental para lidar com essa duplicação. Enviar um e-mail dizendo “isso é a mesma coisa que aquela outra coisa; são sintomas do mesmo incidente” funciona para um determinado alerta: pode evitar a duplicação de depuração ou pânico. Mas enviar um e-mail para cada alerta não é uma solução prática ou escalável para lidar com alertas duplicados dentro de uma equipe, muito menos entre equipes ou por longos períodos de tempo.

É claro

que nem todo evento de alerta é um incidente. Alertas falso-positivos ocorrem, bem como eventos de teste ou e-mails mal direcionados de humanos. O próprio Outalator não faz distinção entre esses eventos, mas permite a marcação de uso geral para adicionar metadados às notificações, em qualquer nível. As tags são principalmente “palavras” de forma livre e únicas. Os dois pontos, no entanto, são interpretados como separadores semânticos, o que sutilmente promove o uso de namespaces hierárquicos e permite algum tratamento automático. Esse namespace é compatível com prefixos de tags sugeridos, principalmente “causa” e “ação”, mas a lista é específica da equipe e gerada com base no uso histórico. Por exemplo, “causa:redes” pode ser informação suficiente para algumas equipes, enquanto outra equipe pode optar por tags mais específicas, como “causa:redes:comutador” versus “causa:redes:cabo”. Algumas equipes podem usar frequentemente tags no estilo “customer:123456”, então “customer” seria sugerido para essas equipes, mas não para outras.

As tags podem ser analisadas e transformadas em um link conveniente (links “bug:76543” para o sistema de rastreamento de bugs). Outras tags são apenas uma única palavra (“falso” é amplamente usado para falsos positivos). É claro que algumas tags são erros de digitação (“cause:netwrok”) e algumas tags não são particularmente úteis (“problem-went-away”), mas evitar uma lista predeterminada e permitir que as equipes encontrem suas próprias preferências e padrões resultar em uma ferramenta mais útil e melhores dados. No geral, as tags têm sido uma ferramenta extremamente poderosa para as equipes obterem e fornecerem uma visão geral dos pontos problemáticos de um determinado serviço, mesmo sem muita ou nenhuma análise formal. Por mais trivial que pareça a marcação, é provavelmente um dos recursos exclusivos mais úteis do Outalator.

Análise É

claro que o SRE faz muito mais do que apenas reagir a incidentes. Os dados históricos são úteis quando se está respondendo a um incidente – a pergunta “o que fizemos da última vez?” é sempre um bom ponto de partida. Mas a informação histórica é muito mais útil quando diz respeito a problemas sistêmicos, periódicos ou outros problemas mais amplos que possam existir. Habilitar essa análise é uma das funções mais importantes de uma ferramenta de rastreamento de interrupções.

A camada inferior de análise abrange contagem e estatísticas agregadas básicas para relatórios. Os detalhes dependem da equipe, mas incluem informações como incidentes por semana/mês/trimestre e alertas por incidente. A próxima camada é mais importante e fácil de fornecer: comparação entre equipes/serviços e ao longo do tempo para identificar os primeiros padrões e tendências. Essa camada permite que as equipes determinem se uma determinada carga de alerta é “normal” em relação ao seu próprio histórico e ao de outros serviços. “É a terceira vez esta semana” pode ser bom ou ruim, mas saber se “isso” costumava acontecer cinco vezes por dia ou cinco vezes por mês permite interpretação.

O próximo passo na análise de dados é encontrar questões mais amplas, que não são apenas contagens brutas, mas requerem alguma análise semântica. Por exemplo, identificar o componente de infraestrutura que causa a maioria dos incidentes e, portanto, o benefício potencial de aumentar a estabilidade ou o desempenho desse componente,² pressupõe que há uma maneira direta de fornecer essas informações junto com os registros de incidentes. Como um exemplo simples: diferentes equipes têm condições de alerta específicas do serviço, como “dados obsoletos” ou “alta latência”. Ambas as condições podem ser causadas pelo congestionamento da rede levando a atrasos na replicação do banco de dados e necessitam de intervenção. Ou podem estar dentro do objetivo de nível de serviço nominal, mas não estão atendendo às expectativas mais altas dos usuários. Examinar essas informações em várias equipes nos permite identificar problemas sistêmicos e escolher a solução correta, especialmente se a solução for a introdução de mais falhas artificiais para interromper o desempenho excessivo.

Relatórios e comunicação

De uso mais imediato para os SREs da linha de frente é a capacidade de selecionar zero ou mais outalações e incluir seus assuntos, tags e anotações “importantes” em um e-mail para o próximo engenheiro de plantão (e uma lista arbitrária de cc) para passar o estado recente entre os turnos. Para revisões periódicas dos serviços de produção (que ocorrem semanalmente para a maioria das equipes), o Outalator também oferece suporte a um “modo relatório”, no qual os importantes

² Por um lado, “a maioria dos incidentes causados” é um bom ponto de partida para reduzir o número de alertas acionados e melhorar o sistema geral. Por outro lado, essa métrica pode ser simplesmente um artefato de monitoramento supersensível ou um pequeno conjunto de sistemas clientes com mau comportamento ou executando fora do nível de serviço acordado. E, por outro lado, o número de incidentes por si só não indica a dificuldade de correção ou a gravidade do impacto.

as anotações são expandidas em linha com a lista principal para fornecer uma visão geral rápida dos lowlights.

Benefícios inesperados

Ser capaz de identificar que um alerta, ou uma enxurrada de alertas, coincide com uma determinada interrupção tem benefícios óbvios: aumenta a velocidade do diagnóstico e reduz a carga de outras equipes ao reconhecer que realmente há um incidente. Existem benefícios adicionais não óbvios. Para usar o Bigtable como exemplo, se um serviço tiver uma interrupção devido a um incidente aparente do Bigtable, mas você puder ver que a equipe do Bigtable SRE não foi alertada, alertar manualmente a equipe provavelmente é uma boa ideia. A visibilidade aprimorada entre equipes pode fazer uma grande diferença na resolução de incidentes, ou pelo menos na mitigação de incidentes.

Algumas equipes da empresa chegaram ao ponto de configurar configurações fictícias de escada rolante: nenhum humano recebe as notificações enviadas para lá, mas as notificações aparecem no Outalator e podem ser marcadas, anotadas e revisadas. Um exemplo para esse uso de “sistema de registro” é registrar e auditar o uso de contas privilegiadas ou de função (embora deva ser observado que essa funcionalidade é básica e usada para auditorias técnicas, e não legais). Outro uso é registrar e anotar automaticamente execuções de tarefas periódicas que podem não ser idempotentes – por exemplo, aplicação automática de mudanças de esquema de controle de versão para sistemas de banco de dados.

CAPÍTULO 17

Teste de confiabilidade

Escrito por Alex Perry e Max Luebbe
Editado por Diane Bates

Se você não experimentou, assuma que está quebrado.

-Desconhecido

Uma das principais responsabilidades dos Engenheiros de Confiabilidade do Site é quantificar a confiança nos sistemas que eles mantêm. Os SREs realizam essa tarefa adaptando técnicas clássicas de teste de software a sistemas em escala.¹ A confiança pode ser medida tanto pela confiabilidade passada quanto pela confiabilidade futura. A primeira é capturada pela análise de dados fornecidos pelo monitoramento do comportamento histórico do sistema, enquanto a segunda é quantificada fazendo previsões a partir de dados sobre o comportamento passado do sistema. Para que essas previsões sejam fortes o suficiente para serem úteis, uma das seguintes condições deve ocorrer:

- O site permanece completamente inalterado ao longo do tempo, sem lançamentos de software ou alterações na frota de servidores, o que significa que o comportamento futuro será semelhante ao comportamento passado.
- Você pode descrever com confiança todas as alterações no site, para que a análise permita a incerteza incorrida por cada uma dessas alterações.

1 Este capítulo explica como maximizar o valor derivado do investimento do esforço de engenharia em testes. Uma vez que um engenheiro define testes adequados (para um determinado sistema) de forma generalizada, o trabalho restante é comum a todas as equipes de SRE e, portanto, pode ser considerado infraestrutura compartilhada. Essa infraestrutura consiste em um agendador (para compartilhar recursos orçados em projetos não relacionados) e executores (esses binários de teste de sandbox para evitar que sejam considerados confiáveis). Esses dois componentes de infraestrutura podem ser considerados um serviço comum com suporte de SRE (muito parecido com armazenamento em escala de cluster) e, portanto, não serão discutidos aqui.

O teste é o mecanismo que você usa para demonstrar áreas específicas de equivalência quando ocorrem mudanças.² Cada teste que passa tanto antes quanto depois de uma mudança reduz a incerteza que a análise precisa permitir. Testes completos nos ajudam a prever a confiabilidade futura de um determinado site com detalhes suficientes para serem úteis na prática.

A quantidade de testes que você precisa realizar depende dos requisitos de confiabilidade do seu sistema. À medida que a porcentagem de sua base de código coberta por testes aumenta, você reduz a incerteza e a possível diminuição na confiabilidade de cada alteração. Cobertura de teste adequada significa que você pode fazer mais mudanças antes que a confiabilidade caia abaixo de um nível aceitável. Se você fizer muitas alterações muito rapidamente, a confiabilidade prevista se aproximará do limite de aceitabilidade. Neste ponto, você pode querer parar de fazer alterações enquanto novos dados de monitoramento se acumulam. Os dados acumulados complementam a cobertura testada, que valida a confiabilidade que está sendo afirmada para caminhos de execução revisados. Supondo que os clientes atendidos sejam distribuídos aleatoriamente [Woo96], as estatísticas de amostragem podem extrapolar das métricas monitoradas se o comportamento agregado está fazendo uso de novos caminhos. Essas estatísticas identificam as áreas que precisam de melhores testes ou outras adaptações.

Relações entre o teste e o tempo médio de reparo A aprovação em um teste

ou em uma série de testes não prova necessariamente a confiabilidade. No entanto, os testes que estão falhando geralmente comprovam a ausência de confiabilidade.

Um sistema de monitoramento pode descobrir bugs, mas apenas com a rapidez com que o pipeline de relatórios pode reagir. O Mean Time to Repair (MTTR) mede quanto tempo leva para a equipe de operações corrigir o bug, seja por meio de uma reversão ou outra ação.

É possível que um sistema de teste identifique um bug com MTTR zero. O MTTR zero ocorre quando um teste de nível de sistema é aplicado a um subsistema e esse teste detecta exatamente o mesmo problema que o monitoramento detectaria. Esse teste permite que o push seja bloqueado para que o bug nunca chegue à produção (embora ainda precise ser reparado no código-fonte). Reparar zero bugs MTTR bloqueando um push é rápido e conveniente. Quanto mais bugs você encontrar com zero MTTR, maior será o tempo médio entre falhas (MTBF) experimentado por seus usuários.

À medida que o MTBF aumenta em resposta a melhores testes, os desenvolvedores são incentivados a lançar recursos mais rapidamente. Alguns desses recursos, é claro, terão bugs. Novos bugs resultam em um ajuste oposto à velocidade de lançamento à medida que esses bugs são encontrados e corrigidos.

2 Para ler mais sobre equivalência, consulte [http://stackoverflow.com/questions/1909280/equivalence-class-testing vs-boundary-value-testing](http://stackoverflow.com/questions/1909280/equivalence-class-testing-vs-boundary-value-testing).

Os autores que escrevem sobre testes de software concordam amplamente sobre qual cobertura é necessária. A maioria dos conflitos de opinião deriva de terminologia conflitante, ênfase diferente no impacto do teste em cada uma das fases do ciclo de vida do software ou nas particularidades dos sistemas nos quais eles realizaram testes. Para uma discussão sobre testes no Google em geral, veja [Whi12]. As seções a seguir especificam como a terminologia relacionada ao teste de software é usada neste capítulo.

Tipos de teste de software

Os testes de software geralmente se dividem em duas categorias: tradicionais e de produção. Testes tradicionais são mais comuns no desenvolvimento de software para avaliar a correção do software offline, durante o desenvolvimento. Os testes de produção são executados em um serviço da Web ao vivo para avaliar se um sistema de software implantado está funcionando corretamente.

Testes tradicionais

Conforme mostrado na Figura 17-1, o teste de software tradicional começa com testes de unidade. O teste de funcionalidades mais complexas é colocado em camadas sobre os testes de unidade.

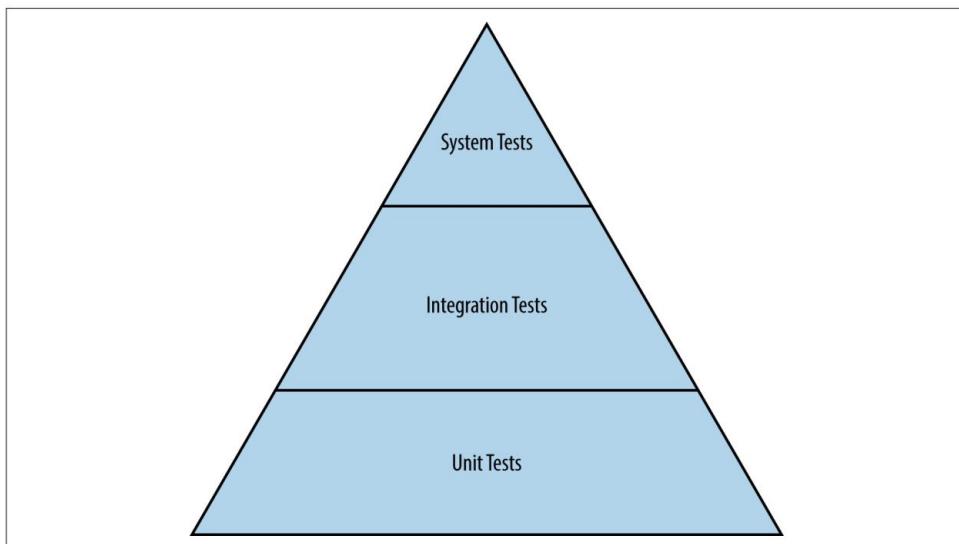


Figura 17-1. A hierarquia dos testes tradicionais

Testes de unidade

Um teste de unidade é a forma menor e mais simples de teste de software. Esses testes são empregados para avaliar uma unidade separável de software, como uma classe ou função, para correção independente do sistema de software maior que contém a unidade. Os testes unitários também são empregados como uma forma de especificação para garantir que uma função ou módulo

executa exatamente o comportamento exigido pelo sistema. Os testes de unidade são comumente usados para introduzir conceitos de desenvolvimento orientado a testes.

Testes de integração

Os componentes de software que passam por testes unitários individuais são montados em componentes maiores. Os engenheiros então executam um teste de integração em um componente montado para verificar se ele funciona corretamente. A injeção de dependência, que é realizada com ferramentas como o Dagger,³ é uma técnica extremamente poderosa para criar simulações de dependências complexas para que um engenheiro possa testar um componente de forma limpa. Um exemplo comum de injeção de dependência é substituir um banco de dados com estado por uma simulação leve que tenha um comportamento precisamente especificado.

Testes de sistema

Um teste de sistema é o teste de maior escala que os engenheiros executam para um sistema não implantado. Todos os módulos pertencentes a um componente específico, como um servidor que passou nos testes de integração, são montados no sistema. Em seguida, o engenheiro testa a funcionalidade de ponta a ponta do sistema. Os testes de sistema vêm em muitos sabores diferentes:

Testes de fumaça

Os testes de fumaça, nos quais os engenheiros testam um comportamento muito simples, mas crítico, estão entre os tipos mais simples de testes de sistema. Os testes de fumaça também são conhecidos como testes de sanidade e servem para curto-circuitar testes adicionais e mais caros.

Testes de desempenho

Uma vez que a correção básica é estabelecida por meio de um teste de fumaça, uma próxima etapa comum é escrever outra variante de um teste de sistema para garantir que o desempenho do sistema permaneça aceitável ao longo de seu ciclo de vida. Como os tempos de resposta para dependências ou requisitos de recursos podem mudar drasticamente durante o desenvolvimento, um sistema precisa ser testado para garantir que não se torne cada vez mais lento sem que ninguém perceba (antes de ser liberado para os usuários). Por exemplo, um determinado programa pode evoluir para precisar de 32 GB de memória quando antes precisava apenas de 8 GB, ou um tempo de resposta de 10 ms pode se transformar em 50 ms e depois em 100 ms. Um teste de desempenho garante que, com o tempo, um sistema não se degrade ou se torne muito caro.

Testes de regressão

Outro tipo de teste de sistema envolve a prevenção de bugs de volta para a base de código. Testes de regressão podem ser comparados a uma galeria de bugs invasores que historicamente causaram falhas no sistema ou produziram resultados incorretos. Ao documentar esses bugs como testes no sistema ou no nível de integração, os engenheiros que refatoraram o

³ Consulte <https://google.github.io/dagger/>.

codebase pode ter certeza de que eles não introduzem bugs acidentalmente que eles já investiram tempo e esforço para eliminar.

É importante notar que os testes têm um custo, tanto em termos de tempo quanto de recursos computacionais. Em um extremo, os testes unitários são muito baratos em ambas as dimensões, pois geralmente podem ser concluídos em milissegundos nos recursos disponíveis em um laptop. No outro extremo do espectro, trazer um servidor completo com dependências necessárias (ou equivalentes simulados) para executar testes relacionados pode levar muito mais tempo – de vários minutos a várias horas – e possivelmente exigir recursos de computação dedicados. A atenção a esses custos é essencial para desenvolver a produtividade e também incentiva o uso mais eficiente dos recursos de teste.

Testes de produção

Os testes de produção interagem com um sistema de produção ao vivo, em oposição a um sistema em um ambiente de teste hermético. Esses testes são, em muitos aspectos, semelhantes ao monitoramento de caixa preta (consulte o [Capítulo 6](#)) e, portanto, às vezes são chamados de teste de caixa preta.

Os testes de produção são essenciais para executar um serviço de produção confiável.

Rollouts Entangle Tests

Costuma-se dizer que os testes são (ou deveriam ser) realizados em um ambiente hermético [\[Nar12\]](#). Esta afirmação implica que a produção não é hermética. É claro que a produção geralmente não é hermética, porque as cadências de distribuição fazem mudanças ao vivo no ambiente de produção em partes pequenas e bem compreendidas.

Para gerenciar a incerteza e ocultar o risco dos usuários, as alterações podem não ser enviadas ao vivo na mesma ordem em que foram adicionadas ao controle do código-fonte. Os rollouts geralmente acontecem em estágios, usando mecanismos que gradualmente embaralham os usuários, além de monitorar em cada estágio para garantir que o novo ambiente não esteja atingindo problemas previstos, mas inesperados. Como resultado, todo o ambiente de produção não é intencionalmente representativo de uma determinada versão de um binário que foi verificado no controle de origem.

É possível que o controle de origem tenha mais de uma versão de um binário e seu arquivo de configuração associado esperando para ser ativado. Esse cenário pode causar problemas quando os testes são realizados no ambiente ao vivo. Por exemplo, o teste pode usar a versão mais recente de um arquivo de configuração localizado no controle do código-fonte junto com uma versão mais antiga do binário ativo. Ou pode testar uma versão mais antiga do arquivo de configuração e encontrar um bug que foi corrigido em uma versão mais recente do arquivo.

Da mesma forma, um teste de sistema pode usar os arquivos de configuração para montar seus módulos antes de executar o teste. Se o teste for aprovado, mas sua versão for aquela em que o teste de configuração (discutido na seção a seguir) falhar, o resultado do teste é válido hermeticamente, mas não operacionalmente. Tal resultado é inconveniente.

Teste de configuração

No Google, as configurações do serviço web são descritas em arquivos armazenados em nosso sistema de controle de versão. Para cada arquivo de configuração, um teste de configuração separado examina a produção para ver como um determinado binário está realmente configurado e relata discrepâncias em relação a esse arquivo. Esses testes não são inherentemente herméticos, pois operam fora da área restrita da infraestrutura de teste.

Os testes de configuração são criados e testados para uma versão específica do arquivo de configuração verificado. Comparar qual versão do teste está passando em relação à versão de meta para automação indica implicitamente até que ponto a produção real está atrasada em relação ao trabalho de engenharia em andamento.

Esses testes de configuração não hermética tendem a ser especialmente valiosos como parte de uma solução de monitoramento distribuído, pois o padrão de aprovação/reprovação na produção pode identificar caminhos na pilha de serviços que não possuem combinações sensatas das configurações locais. As regras da solução de monitoramento tentam corresponder os caminhos das solicitações reais do usuário (dos logs de rastreamento) com esse conjunto de caminhos indesejáveis. Quaisquer correspondências encontradas pelas regras tornam-se alertas de que liberações e/ou pushes em andamento não estão sendo executados com segurança e que ações corretivas são necessárias.

Os testes de configuração podem ser muito simples quando a implantação de produção usa o conteúdo real do arquivo e oferece uma consulta em tempo real para recuperar uma cópia do conteúdo. Nesse caso, o código de teste simplesmente emite essa consulta e diferencia a resposta em relação ao arquivo.

Os testes se tornam mais complexos quando a configuração faz um dos seguintes:

- Incorpora implicitamente padrões que são embutidos no binário (o que significa que os testes são versionados separadamente como resultado)
- Passa por um pré-processador como bash em sinalizadores de linha de comando (renderiza os testes sujeitos a regras de expansão)
- Especifica o contexto comportamental para um tempo de execução compartilhado (fazendo com que os testes dependam de

(cronograma de lançamento desse tempo de execução)

Teste de stress

Para operar um sistema com segurança, os SREs precisam entender os limites do sistema e de seus componentes. Em muitos casos, os componentes individuais não se degradam graciosamente além de um certo ponto - em vez disso, eles falham catastroficamente. Os engenheiros usam testes de estresse para encontrar os limites de um serviço da web. Os testes de estresse respondem a perguntas como:

- Quão cheio um banco de dados pode ficar antes que as gravações comecem a falhar?
- Quantas consultas por segundo podem ser enviadas para um servidor de aplicativos antes de ficar sobrecarregado, fazendo com que as solicitações falhem?

Teste canário

O teste canário está visivelmente ausente desta lista de testes de produção. O termo canário vem da frase "canário em uma mina de carvão" e se refere à prática de usar um pássaro vivo para detectar gases tóxicos antes que os humanos fossem envenenados.

Para realizar um teste canário, um subconjunto de servidores é atualizado para uma nova versão ou configuração e, em seguida, deixado em um período de incubação. Caso não ocorram variações inesperadas, o lançamento continua e o restante dos servidores é atualizado de forma progressiva.⁴ Se algo der errado, o único servidor modificado pode ser rapidamente revertido para um estado bom conhecido. Normalmente nos referimos ao período de incubação para o servidor atualizado como "preparando o binário".

Um teste canário não é realmente um teste; em vez disso, é uma aceitação estruturada do usuário. Enquanto a configuração e os testes de estresse confirmam a existência de uma condição específica sobre software determinístico, um teste canário é mais ad hoc. Ele apenas expõe o código em teste a tráfego de produção ao vivo menos previsível e, portanto, não é perfeito e nem sempre detecta falhas recém-introduzidas.

Para fornecer um exemplo concreto de como um canário pode proceder: considere uma determinada falha subjacente que relativamente raramente afeta o tráfego do usuário e está sendo implantada com uma distribuição de atualização exponencial. Esperamos um número cumulativo crescente de variações relatadas $CU = RK$, onde R é a taxa desses relatórios, U é a ordem da falha (definida posteriormente) e K é o período em que o tráfego cresce por um fator de e , ou 172%.⁵

Para evitar o impacto do usuário, uma distribuição que aciona variações indesejáveis precisa ser rapidamente revertida para a configuração anterior. No pouco tempo que a automação leva para observar as variações e responder, é provável que vários relatórios adicionais sejam gerados. Uma vez que a poeira tenha assentado, esses relatórios podem estimar tanto o número cumulativo C quanto a taxa R .

Dividindo e corrigindo para K dá uma estimativa de U , a ordem da falha subjacente.⁶ Alguns exemplos:

- $U=1$: A solicitação do usuário encontrou um código que está simplesmente quebrado.

⁴ Uma regra prática padrão é começar com o impacto da versão em 0,1% do tráfego do usuário e, em seguida, escalar por ordens de magnitude a cada 24 horas, variando a localização geográfica dos servidores que estão sendo atualizados (então no dia 2: 1%, dia 3: 10%, dia 4: 100%).

⁵ Por exemplo, assumindo um intervalo de 24 horas de crescimento exponencial contínuo entre 1% e 10%,

$$= 0.1 \times 0.1^{\frac{86400K}{24}} = 37523 \text{ segundos, ou cerca de 10 horas e 25 minutos.}$$

⁶ Estamos usando ordem aqui no sentido de "grande notação O" ordem de complexidade. Para mais contexto, consulte https://en.wikipedia.org/wiki/Big_O_notation.

- U=2: a solicitação deste usuário danifica aleatoriamente os dados que a solicitação de um usuário futuro pode Vejo.
- U=3: Os dados danificados aleatoriamente também são um identificador válido para uma solicitação anterior.

A maioria dos bugs são de primeira ordem: eles escalam linearmente com a quantidade de tráfego do usuário [Per07]. Geralmente, você pode rastrear esses bugs convertendo logs de todas as solicitações com respostas incomuns em novos testes de regressão. Essa estratégia não funciona para bugs de ordem superior; uma solicitação que falha repetidamente se todas as solicitações anteriores forem tentadas em ordem passará de repente se algumas solicitações forem omitidas. É importante detectar esses bugs de ordem superior durante o lançamento, porque, caso contrário, a carga de trabalho operacional pode aumentar muito rapidamente.

Tendo em mente a dinâmica dos bugs de ordem superior versus inferior, ao usar uma estratégia de distribuição exponencial, não é necessário tentar alcançar a justiça entre frações do tráfego de usuários. Contanto que cada método para estabelecer uma fração use o mesmo intervalo K , a estimativa de U será válida mesmo que você ainda não possa determinar qual método foi fundamental para iluminar a falha. Usar muitos métodos sequencialmente enquanto permite alguma sobreposição mantém o valor de K pequeno. Essa estratégia minimiza o número total de variações visíveis ao usuário C enquanto ainda permite uma estimativa antecipada de U (esperando por 1, é claro).

Criando um ambiente de teste e compilação

Embora seja maravilhoso pensar sobre esses tipos de testes e cenários de falha no primeiro dia de um projeto, frequentemente os SREs se juntam a uma equipe de desenvolvedores quando um projeto já está em andamento – uma vez que o projeto da equipe valida seu modelo de pesquisa, sua biblioteca prova que a base do projeto algoritmo é escalável, ou talvez quando todos os mocks da interface do usuário forem finalmente aceitáveis. A base de código da equipe ainda é um protótipo e testes abrangentes ainda não foram projetados ou implantados. Em tais situações, onde seus esforços de teste devem começar? A realização de testes de unidade para cada função e classe-chave é uma perspectiva completamente esmagadora se a cobertura de teste atual for baixa ou inexistente.

Em vez disso, comece com testes que proporcionem o maior impacto com o menor esforço.

Você pode iniciar sua abordagem fazendo as seguintes perguntas:

- Você pode priorizar a base de código de alguma forma? Para emprestar uma técnica do desenvolvimento de recursos e gerenciamento de projetos, se cada tarefa for de alta prioridade, nenhuma das tarefas será de alta prioridade. Você pode empilhar os componentes do sistema que está testando por qualquer medida de importância? • Existem funções ou classes específicas que são absolutamente críticas para a missão ou para os negócios? Por exemplo, o código que envolve cobrança é geralmente crítico para os negócios. O código de cobrança também é frequentemente separável de outras partes do sistema.

- Com quais APIs outras equipes estão se integrando? Mesmo o tipo de quebra que nunca passa dos testes de lançamento para um usuário pode ser extremamente prejudicial se confundir outra equipe de desenvolvedores, fazendo com que eles escrevam clientes errados (ou até mesmo abaixo do ideal) para sua API.

O software de envio que está obviamente quebrado está entre os pecados mais graves de um desenvolvedor. É preciso pouco esforço para criar uma série de testes de fumaça a serem executados em cada versão. Esse tipo de primeiro passo de baixo esforço e alto impacto pode levar a softwares altamente testados e confiáveis.

Uma maneira de estabelecer uma forte cultura de teste⁷ é começar a documentar todos os bugs relatados como casos de teste. Se cada bug for convertido em um teste, supõe-se que cada teste falhe inicialmente porque o bug ainda não foi corrigido. À medida que os engenheiros corrigem os bugs, o software passa no teste e você está no caminho para desenvolver um conjunto abrangente de testes de regressão.

Outra tarefa importante para criar software bem testado é configurar uma infraestrutura de teste. A base para uma infraestrutura de teste forte é um sistema de controle de origem com versão que rastreia todas as alterações na base de código.

Depois que o controle do código-fonte estiver em vigor, você poderá adicionar um sistema de compilação contínua que compila o software e executa testes sempre que o código é enviado. Achamos ótimo se o sistema de compilação notificar os engenheiros no momento em que uma alteração interrompe um projeto de software. Correndo o risco de parecer óbvio, é essencial que a versão mais recente de um projeto de software em controle de origem esteja funcionando completamente. Quando o sistema de compilação notifica os engenheiros sobre código quebrado, eles devem abandonar todas as suas outras tarefas e priorizar a correção do problema. É apropriado tratar os defeitos com seriedade por alguns motivos:

- Geralmente é mais difícil consertar o que está quebrado se houver alterações na base de código após a introdução do defeito.
- O software quebrado deixa a equipe mais lenta porque eles precisam contornar o problema quebra.
- As cadências de lançamento, como compilações noturnas e semanais, perdem seu valor. • A capacidade da equipe de responder a uma solicitação de liberação de emergência (por exemplo, em resposta a uma divulgação de vulnerabilidade de segurança) se torna muito mais complexa e difícil.

Os conceitos de estabilidade e agilidade estão tradicionalmente em tensão no mundo do SRE. O último ponto fornece um caso interessante em que a estabilidade realmente impulsiona a agilidade. Quando a compilação é previsivelmente sólida e confiável, os desenvolvedores podem iterar mais rapidamente!

⁷ Para saber mais sobre esse tópico, recomendamos [Bla14] do nosso ex-colega de trabalho e ex-Googler, Mike Bland.

Alguns sistemas de compilação, como o Bazel⁸, possuem recursos valiosos que permitem um controle mais preciso sobre os testes. Por exemplo, o Bazel cria gráficos de dependência para projetos de software.

Quando uma alteração é feita em um arquivo, o Bazel apenas reconstrói a parte do software que depende desse arquivo. Esses sistemas fornecem compilações reproduzíveis. Em vez de executar todos os testes em cada envio, os testes são executados apenas para o código alterado. Como resultado, os testes são executados de forma mais barata e rápida.

Há uma variedade de ferramentas para ajudá-lo a quantificar e visualizar o nível de cobertura de teste que você precisa [Cra10]. Use essas ferramentas para moldar o foco de seus testes: aborde a perspectiva de criar código altamente testado como um projeto de engenharia em vez de um exercício mental filosófico. Em vez de repetir o refrão ambíguo “Precisamos de mais testes”, defina metas e prazos explícitos.

Lembre-se de que nem todos os softwares são criados iguais. Sistemas críticos para a vida ou para a receita exigem níveis substancialmente mais altos de qualidade de teste e cobertura do que um script de não produção com vida útil curta.

Teste em escala

Agora que abordamos os fundamentos dos testes, vamos examinar como o SRE adota uma perspectiva de sistemas para testar a fim de impulsionar a confiabilidade em escala.

Um pequeno teste de unidade pode ter uma pequena lista de dependências: um arquivo de origem, a biblioteca de teste, as bibliotecas de tempo de execução, o compilador e o hardware local executando os testes.

Um ambiente de teste robusto determina que cada uma dessas dependências tenha sua própria cobertura de teste, com testes que abordam especificamente casos de uso que outras partes do ambiente esperam. Se a implementação desse teste de unidade depender de um caminho de código dentro de uma biblioteca de tempo de execução que não tenha cobertura de teste, uma alteração não relacionada no ambiente⁹ pode levar o teste de unidade a passar consistentemente no teste, independentemente de falhas no código em teste .

Em contraste, um teste de lançamento pode depender de tantas partes que tem uma dependência transitiva em cada objeto no repositório de código. Se o teste depende de uma cópia limpa do ambiente de produção, em princípio, cada pequeno patch requer a execução de uma iteração completa de recuperação de desastres.

Ambientes de testes práticos tentam selecionar pontos de ramificação entre as versões e fusões. Isso resolve a quantidade máxima de incerteza dependente para o número mínimo de iterações. É claro,

8 Consulte <https://github.com/google/bazel>.

9 Por exemplo, código em teste que envolve uma API não trivial para fornecer uma abstração mais simples e compatível com versões anteriores. A API que costumava ser síncrona retorna um futuro. A chamada de erros de argumento ainda entrega uma exceção, mas não até que o futuro seja avaliado. O código em teste passa o resultado da API diretamente de volta ao chamador. Muitos casos de uso indevido de argumentos podem não ser detectados.

quando uma área de incerteza se transforma em uma falha, você precisa selecionar pontos de ramificação adicionais.

Testando ferramentas escaláveis

Como peças de software, as ferramentas SRE também precisam de testes.¹⁰ As ferramentas desenvolvidas pelo SRE podem realizar tarefas como as seguintes:

- Recuperar e propagar métricas de desempenho de banco de dados
- Prever métricas de uso para planejar riscos de capacidade
- Refatorar dados em uma réplica de serviço que não é acessível ao usuário
- Alterar arquivos em um servidor

As ferramentas SRE compartilham duas características:

- Seus efeitos colaterais permanecem dentro da API mainstream testada
- Eles estão isolados da produção voltada para o usuário por uma validação existente e liberar barreira

Defesas de barreira contra software arriscado O

software que ignora a API habitualmente testada e pesada (mesmo que o faça por uma boa causa) pode causar estragos em um serviço ao vivo. Por exemplo, a implementação de um mecanismo de banco de dados pode permitir que os administradores desliguem temporariamente as transações para reduzir as janelas de manutenção. Se a implementação for usada por software de atualização em lote, o isolamento voltado para o usuário pode ser perdido se esse utilitário for iniciado acidentalmente em uma réplica voltada para o usuário. Evite esse risco de estragos com o design:

1. Use uma ferramenta separada para colocar uma barreira na configuração de replicação para que a réplica não passe em sua verificação de integridade. Como resultado, a réplica não é liberada para os usuários.
2. Configure o software arriscado para verificar a barreira na inicialização. Permitir que a software arriscado para acessar apenas réplicas não íntegras.
3. Use a ferramenta de validação de integridade de réplica que você usa para monitoramento de caixa preta para remover a barreira.

¹⁰ Esta seção fala especificamente sobre ferramentas usadas pelo SRE que precisam ser escaláveis. No entanto, o SRE também desenvolve e usa ferramentas que não precisam necessariamente ser escaláveis. As ferramentas que não precisam ser escaláveis também precisam ser testadas, mas essas ferramentas estão fora do escopo desta seção e, portanto, não serão discutidas aqui. Como sua pegada de risco é semelhante aos aplicativos voltados para o usuário, estratégias de teste semelhantes são aplicáveis a essas ferramentas desenvolvidas pelo SRE.

As ferramentas de automação também são software. Como sua pegada de risco parece fora de banda para uma camada diferente do serviço, suas necessidades de teste são mais sutis. As ferramentas de automação executam tarefas como as seguintes:

- Seleção de índice de banco de dados
 - Balanceamento de carga entre datacenters •
- Embaralhamento de logs de retransmissão para remasterização rápida

As ferramentas de automação compartilham duas características:

- A operação real realizada é contra um robusto, previsível e bem testado API
- O objetivo da operação é o efeito colateral que é uma descontinuidade invisível para outro cliente de API

O teste pode demonstrar o comportamento desejado da outra camada de serviço, antes e depois da mudança. Muitas vezes, é possível testar se o estado interno, conforme visto pela API, é constante em toda a operação. Por exemplo, os bancos de dados buscam respostas corretas, mesmo que um índice adequado não esteja disponível para a consulta. Por outro lado, algumas invariantes de API documentadas (como um cache DNS retido até o TTL) podem não se manter durante a operação. Por exemplo, se uma mudança no nível de execução substituir um servidor de nomes local por um proxy de armazenamento em cache, ambas as opções podem prometer manter as pesquisas concluídas por muitos segundos. É improvável que o estado do cache seja passado de um para o outro.

Dado que as ferramentas de automação implicam testes de liberação adicionais para outros binários para lidar com transientes ambientais, como você define o ambiente no qual essas ferramentas de automação são executadas? Afinal, a automação para embaralhar contêineres para melhorar o uso provavelmente tentará se embaralhar em algum momento se também for executada em um contêiner. Seria embarrasoso se uma nova versão de seu algoritmo interno produzisse páginas de memória sujas tão rapidamente que a largura de banda da rede do espelhamento associado acabasse impedindo o código de finalizar a migração ao vivo. Mesmo que haja um teste de integração para o qual o binário se embaralhe intencionalmente, o teste provavelmente não usa um modelo de tamanho de produção da frota de contêineres. É quase certo que não é permitido usar a escassa largura de banda intercontinental de alta latência para testar essas corridas.

Ainda mais divertido, uma ferramenta de automação pode estar mudando o ambiente em que outra ferramenta de automação é executada. Ou ambas as ferramentas podem estar alterando o ambiente da outra ferramenta de automação simultaneamente! Por exemplo, uma ferramenta de atualização de frota provavelmente consome mais recursos ao enviar atualizações. Como resultado, o rebalanceamento do contêiner seria tentado a mover a ferramenta. Por sua vez, a ferramenta de rebalanceamento de contêiner ocasionalmente precisa ser atualizada. Essa dependência circular é boa se as APIs associadas tiverem semântica de reinicialização, alguém se lembrar de implementar cobertura de teste para essas semânticas e a integridade do ponto de verificação for assegurada de forma independente.

Testando desastres

Muitas ferramentas de recuperação de desastres podem ser cuidadosamente projetadas para operar offline. Essas ferramentas fazem o seguinte:

- Calcular um estado de ponto de verificação que é equivalente a parar o serviço de forma limpa •

Empurrar o estado de ponto de verificação para ser carregável por ferramentas de validação não-desastre existentes • Suportar as ferramentas usuais de barreira de liberação, que acionam o procedimento de inicialização limpa

Em muitos casos, você pode implementar essas fases para que os testes associados sejam fáceis de escrever e ofereçam excelente cobertura. Se alguma das restrições (off-line, checkpoint, carregável, barreira ou inicialização limpa) precisar ser quebrada, é muito mais difícil mostrar confiança de que a implementação da ferramenta associada funcionará a qualquer momento em curto prazo.

As ferramentas de reparo online operam inherentemente fora da API principal e, portanto, tornam-se mais interessantes para testar. Um desafio que você enfrenta em um sistema distribuído é determinar se o comportamento normal, que pode eventualmente ser consistente por natureza, irá interagir mal com o reparo. Por exemplo, considere uma condição de corrida que você pode tentar analisar usando as ferramentas offline. Uma ferramenta offline geralmente é escrita para esperar consistência instantânea, em oposição à consistência eventual, porque a consistência instantânea é menos desafiadora para testar. Essa situação se torna complicada porque o binário de reparo geralmente é construído separadamente do binário de produção de serviço contra o qual está concorrendo. Conseqüentemente, pode ser necessário construir um binário instrumentado unificado para executar dentro desses testes para que as ferramentas possam observar as transações.

Usando testes estatísticos Técnicas

estatísticas, como Lemon [Ana07] para fuzzing e Chaos Monkey¹¹ e Jepsen¹² para estado distribuído, não são necessariamente testes repetíveis. A simples reexecução desses testes após uma alteração de código não prova definitivamente que a falha observada foi corrigida.¹³ No entanto, essas técnicas podem ser úteis:

- Eles podem fornecer um registro de todas as ações selecionadas aleatoriamente que são executadas em uma determinada execução – às vezes simplesmente registrando a semente do gerador de números aleatórios. • Se este log for refatorado imediatamente como um teste de lançamento, executá-lo algumas vezes antes de iniciar o relatório de bug geralmente é útil. A taxa de não-falha no replay informa o quão difícil será afirmar mais tarde que a falha foi corrigida.
- Variações na forma como a falha é expressa ajudam a identificar áreas suspeitas no código.
- Algumas dessas execuções posteriores podem demonstrar situações de falha mais graves do que as da execução original. Em resposta, você pode querer escalar a gravidade e o impacto do bug.

A necessidade de

velocidade Para cada versão (patch) no repositório de código, cada teste definido fornece uma indicação de aprovação ou reprovação. Essa indicação pode mudar para execuções repetidas e aparentemente idênticas. Você pode estimar a probabilidade real de um teste ser aprovado ou reprovado calculando a média dessas muitas execuções e calculando a incerteza estatística dessa probabilidade. No entanto, realizar este cálculo para cada teste em cada ponto de versão é computacionalmente inviável.

Em vez disso, você deve formar hipóteses sobre os muitos cenários de interesse e executar o número apropriado de repetições de cada teste e versão para permitir uma inferência razoável. Alguns desses cenários são benignos (no sentido de qualidade de código), enquanto outros são acionáveis. Esses cenários afetam todas as tentativas de teste em graus variados e, por serem acoplados, obter de forma confiável e rápida uma lista de hipóteses acionáveis (ou seja, componentes que estão realmente quebrados) significa estimar todos os cenários ao mesmo tempo.

Tempo.

11 Veja <https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey>.

12 Consulte <https://github.com/aphyr/jepsen>.

13 Mesmo que a execução de teste seja repetida com a mesma semente aleatória para que os kills da tarefa estejam na mesma ordem, não há serialização entre os kills e o tráfego de usuário falso. Portanto, não há garantia de que o caminho de código observado anteriormente será exercitado novamente.

Os engenheiros que usam a infraestrutura de teste querem saber se seu código - geralmente uma pequena fração de toda a fonte por trás de uma determinada execução de teste - está quebrado. Muitas vezes, não estar quebrado implica que quaisquer falhas observadas podem ser atribuídas ao código de outra pessoa. Em outras palavras, o engenheiro quer saber se seu código tem uma condição de corrida imprevista que torna o teste esquisito (ou mais esquisito do que o teste já era devido a outros fatores).

Prazos de teste

A maioria dos testes é simples, no sentido de que eles são executados como um binário hermético independente que cabe em um pequeno contêiner de computação por alguns segundos. Esses testes fornecem aos engenheiros feedback interativo sobre erros antes que o engenheiro mude o contexto para o próximo bug ou tarefa.

Testes que exigem orquestração em muitos binários e/ou em uma frota com muitos contêineres tendem a ter tempos de inicialização medidos em segundos. Esses testes geralmente são incapazes de oferecer feedback interativo, de modo que podem ser classificados como testes em lote. Em vez de dizer “não feche a guia do editor” para o engenheiro, essas falhas de teste estão dizendo “este código não está pronto para revisão” para o revisor de código.

O prazo informal para o teste é o ponto em que o engenheiro faz a próxima troca de contexto. Os resultados dos testes são melhor entregues ao engenheiro antes que ele mude o contexto, porque, caso contrário, o próximo contexto pode envolver a compilação do XKCD.¹⁴

Suponha que um engenheiro esteja trabalhando em um serviço com mais de 21.000 testes simples e ocasionalmente proponha um patch na base de código do serviço. Para testar o patch, você deseja comparar o vetor de resultados de aprovação/reprovação da base de código antes do patch com o vetor de resultados da base de código após o patch. Uma comparação favorável desses dois vetores qualifica provisoriamente a base de código como liberável. Essa qualificação cria um incentivo para executar os vários testes de liberação e integração, bem como outros testes binários distribuídos que examinam o dimensionamento do sistema (caso o patch use significativamente mais recursos de computação local) e a complexidade (caso o patch crie uma carga de trabalho superlinear em outro lugar).

A que taxa você pode sinalizar incorretamente o patch de um usuário como prejudicial ao calcular erroneamente a flakiness ambiental? Parece provável que os usuários reclamem veementemente se 1 em cada 10 patches for rejeitado. Mas uma rejeição de 1 patch entre 100 patches perfeitos pode passar sem comentários.

14 Consulte <http://xkcd.com/303/>.

Isso significa que você está interessado na 42.000^{a} raiz (uma para cada teste definido antes do patch e uma para cada teste definido após o patch) de 0,99 (a fração de patches que podem ser rejeitados). Este cálculo:

$$\frac{1}{21000} = 0.\overline{99}$$

sugere que esses testes individuais devem ser executados corretamente em 99,9999% do tempo.
Hum.

Envio para produção Embora o

gerenciamento de configuração de produção seja geralmente mantido em um repositório de controle de origem, a configuração geralmente é separada do código-fonte do desenvolvedor. Da mesma forma, a infraestrutura de teste de software geralmente não consegue ver a configuração de produção. Mesmo que os dois estejam localizados no mesmo repositório, as alterações para gerenciamento de configuração são feitas em branches e/ou em uma árvore de diretórios segregada que a automação de teste ignorou historicamente.

Em um ambiente corporativo legado onde engenheiros de software desenvolvem binários e os lançam por cima do muro para os administradores que atualizam os servidores, a segregação de infraestrutura de teste e configuração de produção é, na melhor das hipóteses, irritante e, na pior, pode prejudicar a confiabilidade e a agilidade. Essa segregação também pode levar à duplicação de ferramentas. Em um ambiente de operações nominalmente integrado, essa segregação degrada a resiliência porque cria inconsistências sutis entre o comportamento dos dois conjuntos de ferramentas.

Essa segregação também limita a velocidade do projeto devido às corridas de confirmação entre os sistemas de controle de versão.

No modelo SRE, o impacto de segregar a infraestrutura de teste da configuração de produção é sensivelmente pior, pois evita relacionar o modelo que descreve a produção ao modelo que descreve o comportamento da aplicação. Essa discrepância afeta os engenheiros que desejam encontrar inconsistências estatísticas nas expectativas no momento do desenvolvimento. No entanto, essa segregação não retarda o desenvolvimento tanto quanto impede que a arquitetura do sistema mude, porque não há como eliminar o risco de migração.

Considere um cenário de versionamento unificado e teste unificado, para que a metodologia SRE seja aplicável. Que impacto teria a falha de uma migração de arquitetura distribuída? Uma boa quantidade de testes provavelmente ocorrerá. Até agora, supõe-se que um engenheiro de software provavelmente aceitaria o sistema de teste dando a resposta errada 1 vez em 10 ou mais. Que risco você está disposto a correr com a migração se sabe que o teste pode retornar um falso negativo e a situação pode se tornar realmente empolgante, muito rapidamente? Claramente, algumas áreas de cobertura de teste precisam de um nível mais alto de paranóia

do que outros. Essa distinção pode ser generalizada: algumas falhas de teste são indicativas de um risco de impacto maior do que outras falhas de teste.

Esperar falha no teste

Não muito tempo atrás, um produto de software poderia ser lançado uma vez por ano. Seus binários foram gerados por uma cadeia de ferramentas do compilador durante muitas horas ou dias, e a maioria dos testes foi realizada por humanos contra instruções escritas manualmente. Esse processo de lançamento era ineficiente, mas havia pouca necessidade de automatizá-lo. O esforço de lançamento foi dominado por documentação, migração de dados, retreinamento de usuários e outros fatores.

O tempo médio entre falhas (MTBF) para esses lançamentos foi de um ano, não importando quantos testes fossem realizados. Tantas mudanças aconteceram por lançamento que alguma quebra visível do usuário estava fadada a estar escondida no software. Efetivamente, os dados de confiabilidade da versão anterior eram irrelevantes para a próxima versão.

Ferramentas eficazes de gerenciamento de API/ABI e linguagens interpretadas que escalam para grandes quantidades de código agora suportam a criação e execução de uma nova versão de software a cada poucos minutos. Em princípio, um exército suficientemente grande de humanos¹⁵ poderia completar os testes em cada nova versão usando os métodos descritos anteriormente e alcançar a mesma barra de qualidade para cada versão incremental. Embora, em última análise, apenas os mesmos testes sejam aplicados ao mesmo código, essa versão final do software tem maior qualidade na versão resultante que é enviada anualmente. Isso porque, além das versões anuais, também estão sendo testadas as versões intermediárias do código. Usando intermediários, você pode mapear inequivocamente os problemas encontrados durante o teste de volta às suas causas subjacentes e ter certeza de que todo o problema, e não apenas o sintoma limitado que foi exposto, foi corrigido. Este princípio de um ciclo de feedback mais curto é igualmente eficaz quando aplicado à cobertura de testes automatizados.

Se você permitir que os usuários experimentem mais versões do software durante o ano, o MTBF sofre porque há mais oportunidades de quebras visíveis ao usuário. No entanto, você também pode descobrir áreas que se beneficiariam de uma cobertura de teste adicional. Se esses testes forem implementados, cada melhoria protege contra alguma falha futura. O gerenciamento cuidadoso da confiabilidade combina os limites de incerteza devido à cobertura de teste com os limites de falhas visíveis ao usuário para ajustar a cadência de liberação. Essa combinação maximiza o conhecimento que você obtém das operações e dos usuários finais. Esses ganhos impulsionam a cobertura de teste e, por sua vez, a velocidade de lançamento do produto.

Se um SRE modifica um arquivo de configuração ou ajusta a estratégia de uma ferramenta de automação (em oposição à implementação de um recurso do usuário), o trabalho de engenharia corresponde ao mesmo modelo conceitual. Quando você está definindo uma cadência de lançamento com base na confiabilidade, geralmente faz sentido segmentar o orçamento de confiabilidade por funcionalidade ou (mais concretamente)

15 Talvez adquirido através da Mechanical Turk ou serviços similares.

veniente) por equipe. Nesse cenário, a equipe de engenharia de recursos visa atingir um determinado limite de incerteza que afeta sua cadênciā de lançamento de metas. A equipe SRE tem um orçamento separado com sua própria incerteza associada e, portanto, um limite superior em sua taxa de liberação.

Para permanecer confiável e evitar o dimensionamento do número de SREs que suportam um serviço linearmente, o ambiente de produção deve ser executado em grande parte sem supervisão. Para permanecer desacompanhado, o ambiente deve ser resiliente contra pequenas falhas. Quando ocorre um grande evento que exige intervenção manual do SRE, as ferramentas utilizadas pelo SRE devem ser devidamente testadas. Caso contrário, essa intervenção diminui a confiança de que os dados históricos são aplicáveis no futuro próximo. A redução da confiança requer a espera de uma análise dos dados de monitoramento para eliminar a incerteza incorrida. Enquanto a discussão anterior em ["Teste de ferramentas escaláveis"](#) na página 193 se concentrou em como aproveitar a oportunidade de cobertura de teste para uma ferramenta SRE, aqui você vê que o teste determina com que frequência é apropriado usar essa ferramenta em relação à produção.

Os arquivos de configuração geralmente existem porque alterar a configuração é mais rápido do que reconstruir uma ferramenta. Essa baixa latência geralmente é um fator para manter o MTTR baixo. No entanto, esses mesmos arquivos também são alterados com frequência por motivos que não precisam dessa latência reduzida. Quando visto do ponto de vista da confiabilidade:

- Um arquivo de configuração que existe para manter o MTTR baixo, e só é modificado quando há uma falha, tem uma cadênciā de liberação mais lenta que o MTBF. Pode haver uma quantidade razoável de incerteza sobre se uma determinada edição manual é realmente ideal sem que a edição afete a confiabilidade geral do site.
- Um arquivo de configuração que é alterado mais de uma vez por versão do aplicativo voltado para o usuário (por exemplo, porque mantém o estado da versão) pode ser um grande risco se essas alterações não forem tratadas da mesma forma que as versões do aplicativo. Se a cobertura de teste e monitoramento desse arquivo de configuração não for consideravelmente melhor do que a do aplicativo do usuário, esse arquivo dominará a confiabilidade do site de maneira negativa.

Um método de manipulação de arquivos de configuração é certificar-se de que cada arquivo de configuração seja categorizado em apenas uma das opções na lista com marcadores anterior e, de alguma forma, impor essa regra. Se você adotar a última estratégia, certifique-se do seguinte:

- Cada arquivo de configuração tem cobertura de teste suficiente para suportar rotinas regulares edição.
- Antes dos lançamentos, as edições de arquivo são um pouco atrasadas enquanto aguardam o teste de lançamento.
- Forneça um mecanismo de quebra de vidro para colocar o arquivo ao vivo antes de concluir o teste. Como quebrar o vidro prejudica a confiabilidade, geralmente é uma boa ideia tornar a quebra barulhenta (por exemplo) preenchendo um bug solicitando uma resolução mais robusta para a próxima vez.

Quebra de vidro e testes

Você pode implementar um mecanismo de quebra de vidro para desabilitar o teste de versão. Isso significa que quem fizer uma edição manual apressada não será informado sobre nenhum erro até que o impacto real do usuário seja relatado pelo monitoramento. É melhor deixar os testes rodando, associar o evento de push antecipado com o evento de teste pendente e (assim que possível) anotar de volta o push com quaisquer testes quebrados. Desta forma, um empurrão manual defeituoso pode ser rapidamente seguido por outro empurrão manual (espero que menos defeituoso). Idealmente, esse mecanismo de quebra de vidro aumenta automaticamente a prioridade desses testes de versão para que eles possam antecipar a validação incremental de rotina e a carga de trabalho de cobertura que a infraestrutura de teste já está processando.

Integração

Além de testar a unidade de um arquivo de configuração para mitigar seu risco à confiabilidade, também é importante considerar os arquivos de configuração de teste de integração. O conteúdo do arquivo de configuração é (para fins de teste) conteúdo potencialmente hostil ao interpretador que lê a configuração. Linguagens interpretadas, como Python, são comumente usadas para arquivos de configuração porque seus intérpretes podem ser incorporados e alguns sandboxing simples estão disponíveis para proteger contra erros de codificação não maliciosos.

Escrever seus arquivos de configuração em uma linguagem interpretada é arriscado, pois essa abordagem está repleta de falhas latentes que são difíceis de resolver definitivamente. Como o carregamento de conteúdo consiste na execução de um programa, não há limite superior inerente de quanto ineficiente o carregamento pode ser. Além de qualquer outro teste, você deve emparelhar esse tipo de teste de integração com uma verificação cuidadosa do prazo em todos os métodos de teste de integração para rotular os testes que não são executados até a conclusão em um período de tempo razoável como falhados.

Se a configuração for escrita como texto em uma sintaxe personalizada, cada categoria de teste precisará de uma cobertura separada do zero. O uso de uma sintaxe existente, como YAML, em combinação com um analisador altamente testado, como o `safe_load` do Python, remove parte do trabalho incorrido pelo arquivo de configuração. A escolha cuidadosa da sintaxe e do analisador pode garantir que haja um limite superior rígido de quanto tempo a operação de carregamento pode levar. No entanto, o implementador precisa resolver falhas de esquema, e as estratégias mais simples para fazer isso não têm um limite superior em tempo de execução. Pior ainda, essas estratégias tendem a não ser testadas de forma robusta por unidade.

A vantagem de usar buffers de protocolo¹⁶ é que o esquema é definido com antecedência e verificado automaticamente no tempo de carregamento, removendo ainda mais o trabalho, mas ainda oferecendo o tempo de execução limitado.

O papel do SRE geralmente inclui escrever ferramentas de engenharia de sistemas¹⁷ (se ninguém mais as estiver escrevendo) e adicionar validação robusta com cobertura de teste. Todas as ferramentas podem se comportar inesperadamente devido a bugs não detectados pelos testes, portanto, uma defesa em profundidade é aconselhável. Quando uma ferramenta se comporta de forma inesperada, os engenheiros precisam estar tão confiantes quanto possível de que a maioria de suas outras ferramentas está funcionando corretamente e, portanto, podem mitigar ou resolver os efeitos colaterais desse mau comportamento. Um elemento-chave para fornecer confiabilidade ao site é encontrar cada forma antecipada de mau comportamento e garantir que algum teste (ou o validador de entrada testado de outra ferramenta) relate esse mau comportamento. A ferramenta que encontra o problema pode não ser capaz de corrigi-lo ou mesmo pará-lo, mas deve pelo menos relatar o problema antes que ocorra uma interrupção catastrófica.

Por exemplo, considere a lista configurada de todos os usuários (como /etc/passwd em uma máquina no estilo Unix sem rede) e imagine uma edição que involuntariamente faz com que o analisador pare após analisar apenas metade do arquivo. Como os usuários recém-criados não foram carregados, a máquina provavelmente continuará funcionando sem problemas e muitos usuários podem não perceber a falha. A ferramenta que mantém os diretórios pessoais pode facilmente perceber a incompatibilidade entre os diretórios reais presentes e os implícitos na lista de usuários (parcial) e relatar com urgência a discrepância. O valor dessa ferramenta está em relatar o problema e deve evitar tentar remediar por conta própria (excluindo muitos dados do usuário).

Sondas de Produção

Dado que o teste especifica o comportamento aceitável em face de dados conhecidos, enquanto o monitoramento confirma o comportamento aceitável em face de dados desconhecidos do usuário, parece que as principais fontes de risco – tanto o conhecido quanto o desconhecido – são cobertas pela combinação de testes e monitoramento. Infelizmente, o risco real é mais complicado.

As solicitações boas conhecidas devem funcionar, enquanto as solicitações ruins conhecidas devem dar erro. Implementar ambos os tipos de cobertura como um teste de integração geralmente é uma boa ideia. Você pode reproduzir o mesmo banco de solicitações de teste como um teste de versão. Dividir as solicitações válidas conhecidas entre aquelas que podem ser reproduzidas na produção e aquelas que não podem gerar três conjuntos de solicitações:

16 Consulte <https://github.com/google/protobuf>.

17 Não porque os engenheiros de software não deveriam escrevê-los. Ferramentas que cruzam entre verticais de tecnologia e abrangem camadas de abstração tendem a ter associações fracas com muitas equipes de software e uma associação um pouco mais forte com equipes de sistemas.

- Solicitações ruins conhecidas
- Solicitações boas conhecidas que podem ser reproduzidas na produção
- Solicitações boas conhecidas que não podem ser reproduzidas na produção

Você pode usar cada conjunto como testes de integração e de lançamento. A maioria desses testes também pode ser usada como sondas de monitoramento.

Parece supérfluo e, em princípio, inútil implantar esse monitoramento porque essas mesmas solicitações já foram tentadas de duas outras maneiras. No entanto, essas duas maneiras eram diferentes por alguns motivos:

- O teste de lançamento provavelmente envolveu o servidor integrado com um frontend e um back-end falso.
- O teste de sonda provavelmente envolveu o binário de lançamento com um frontend de balanceamento de carga e um backend persistente escalável separado. • Frontends e backends provavelmente têm ciclos de lançamento independentes. É provável que os cronogramas para esses ciclos ocorram em taxas diferentes (devido às suas cadências de liberação adaptáveis).

Portanto, o probe de monitoramento em execução na produção é uma configuração que não foi testada anteriormente.

Essas sondas nunca devem falhar, mas o que significa se elas falharem? A API de front-end (do平衡ador de carga) ou a API de back-end (para o armazenamento persistente) não é equivalente entre os ambientes de produção e lançamento. A menos que você já saiba por que os ambientes de produção e lançamento não são equivalentes, o site provavelmente está quebrado.

O mesmo atualizador de produção que substitui gradualmente o aplicativo também substitui gradualmente os probes para que todas as quatro combinações de probes antigos ou novos que enviam solicitações para aplicativos antigos ou novos sejam geradas continuamente. Esse atualizador pode detectar quando uma das quatro combinações está gerando erros e reverter para o último estado válido conhecido. Normalmente, o atualizador espera que cada instância de aplicativo recém-iniciada não esteja íntegra por um curto período de tempo enquanto se prepara para começar a receber muito tráfego de usuários. Se os probes já forem inspecionados como parte da verificação de prontidão, a atualização falhará indefinidamente com segurança e nenhum tráfego de usuário será roteado para a nova versão. A atualização permanece pausada até que os engenheiros tenham tempo e disposição para diagnosticar a condição de falha e, em seguida, incentivar o atualizador de produção a reverter de forma limpa.

Este teste de produção por sonda de fato oferece proteção ao local, além de feedback claro para os engenheiros. Quanto mais cedo esse feedback for dado aos engenheiros, mais útil ele será. Também é preferível que o teste seja automatizado para que a entrega de avisos aos engenheiros seja escalável.

Suponha que cada componente tenha a versão de software mais antiga que está sendo substituída e a versão mais recente que está sendo lançada (agora ou muito em breve). A versão mais recente pode estar conversando com o par da versão antiga, o que a força a usar a API obsoleta. Ou a versão mais antiga pode estar conversando com a versão mais recente de um peer, usando a API que (no momento em que a versão mais antiga foi lançada) ainda não funcionava corretamente. Mas agora funciona, honestamente! É melhor esperar que esses testes de compatibilidade futura (que estão sendo executados como testes de monitoramento) tenham uma boa cobertura de API.

Versões de back-end falsas

Ao implementar testes de lançamento, o back-end falso geralmente é mantido pela equipe de engenharia do serviço de pares e meramente referenciado como uma dependência de compilação. O teste hermético que é executado pela infraestrutura de teste sempre combina o backend falso e o frontend de teste no mesmo ponto de construção no histórico de controle de revisão.

Essa dependência de compilação pode estar fornecendo um binário hermético executável e, idealmente, a equipe de engenharia que o mantém corta uma versão desse binário de back-end falso ao mesmo tempo em que corta seu aplicativo de back-end principal e seus testes. Se essa versão de back-end estiver disponível, pode valer a pena incluir testes herméticos de versão de front-end (sem o binário de back-end falso) no pacote de versão de front-end.

Seu monitoramento deve estar ciente de todas as versões de lançamento em ambos os lados de uma determinada interface de serviço entre dois peers. Essa configuração garante que recuperar cada combinação das duas versões e determinar se o teste ainda passa não exige muita configuração extra. Esse monitoramento não precisa acontecer continuamente — você só precisa executar novas combinações que são o resultado de uma das equipes cortando uma nova versão. Esses problemas não precisam bloquear esse novo lançamento em si.

Por outro lado, a automação de implantação deve bloquear idealmente a implantação de produção associada até que as combinações problemáticas não sejam mais possíveis. Da mesma forma, a automação da equipe de pares pode considerar drenar (e atualizar) as réplicas que ainda não foram movidas de uma combinação problemática.

Conclusão

O teste é um dos investimentos mais lucrativos que os engenheiros podem fazer para melhorar a confiabilidade de seu produto. O teste não é uma atividade que acontece uma ou duas vezes no ciclo de vida de um projeto; é contínuo. A quantidade de esforço necessária para escrever bons testes é substancial, assim como o esforço para construir e manter uma infraestrutura que promova uma forte cultura de teste. Você não pode resolver um problema até entendê-lo e, na engenharia, você só pode entender um problema medindo-o. As metodologias e técnicas neste capítulo fornecem uma base sólida para medir falhas e incertezas em um sistema de software e ajudam os engenheiros a raciocinar sobre a confiabilidade do software conforme ele é escrito e liberado para os usuários.

CAPÍTULO 18

Engenharia de Software em SRE

**Escrito por Dave Helstroom e Trisha Weir
com Evan Leonard e Kurt Delimon**

Editado por Kavita Guliani

Peça a alguém para citar um esforço de engenharia de software do Google e eles provavelmente listarão um produto voltado para o consumidor, como Gmail ou Maps; alguns podem até mencionar a infraestrutura subjacente, como Bigtable ou Colossus. Mas, na verdade, há uma enorme quantidade de engenharia de software nos bastidores que os consumidores nunca veem. Vários desses produtos são desenvolvidos dentro do SRE.

O ambiente de produção do Google é – em algumas medidas – uma das máquinas mais complexas que a humanidade já construiu. Os SREs têm experiência em primeira mão com os meandros da produção, tornando-os especialmente adequados para desenvolver as ferramentas apropriadas para resolver problemas internos e casos de uso relacionados à manutenção da produção. A maioria dessas ferramentas está relacionada à diretriva geral de manter o tempo de atividade e manter a latência baixa, mas assume muitas formas: os exemplos incluem mecanismos de distribuição binária, monitoramento ou um ambiente de desenvolvimento construído em composição dinâmica de servidor. No geral, essas ferramentas desenvolvidas pelo SRE são projetos de engenharia de software completos, distintos de soluções pontuais e hacks rápidos, e os SREs que as desenvolvem adotaram uma mentalidade baseada em produtos que leva os clientes internos e um roteiro para planos futuros em conta.

Por que a engenharia de software no SRE é importante?

De muitas maneiras, a vasta escala de produção do Google exigiu o desenvolvimento interno de software, porque poucas ferramentas de terceiros são projetadas em escala suficiente para as necessidades do Google. O histórico de projetos de software de sucesso da empresa nos levou a apreciar os benefícios de desenvolver diretamente dentro do SRE.

Os SREs estão em uma posição única para desenvolver efetivamente software interno para uma série de razões:

- A amplitude e profundidade do conhecimento de produção específico do Google dentro da organização SRE permite que seus engenheiros projetem e criem software com as considerações apropriadas para dimensões como escalabilidade, degradação graciosa durante falhas e a capacidade de interagir facilmente com outra infraestrutura ou Ferramentas.
- Como os SREs são incorporados ao assunto, eles entendem facilmente as necessidades e os requisitos da ferramenta que está sendo desenvolvida.
- Um relacionamento direto com o usuário pretendido – colegas SREs – resulta em feedback do usuário franco e de alto nível. Liberar uma ferramenta para um público interno com alta familiaridade com o espaço do problema significa que uma equipe de desenvolvimento pode lançar e iterar mais rapidamente. Os usuários internos geralmente são mais compreensivos quando se trata de IU mínima e outros problemas de produtos alfa.

De um ponto de vista puramente pragmático, o Google claramente se beneficia de ter engenheiros com experiência em SRE desenvolvendo software. Por design deliberado, a taxa de crescimento dos serviços suportados pelo SRE excede a taxa de crescimento da organização SRE; um dos princípios orientadores do SRE é que “o tamanho da equipe não deve ser dimensionado diretamente com o crescimento do serviço”. Alcançar o crescimento linear da equipe em face do crescimento exponencial do serviço requer trabalho de automação perpétuo e esforços para otimizar ferramentas, processos e outros aspectos de um serviço que introduzem inficiência na operação diária da produção.

Ter pessoas com experiência direta executando sistemas de produção desenvolvendo as ferramentas que acabarão contribuindo para as metas de tempo de atividade e latência faz muito sentido.

Por outro lado, SREs individuais, bem como a organização SRE mais ampla, também se beneficiam do desenvolvimento de software orientado por SRE.

Projetos de desenvolvimento de software completos no SRE oferecem oportunidades de desenvolvimento de carreira para SREs, bem como uma saída para engenheiros que não querem que suas habilidades de codificação fiquem enferrujadas. O trabalho de projeto de longo prazo fornece o equilíbrio necessário para interrupções e trabalho de plantão, e pode proporcionar satisfação no trabalho para engenheiros que desejam que suas carreiras mantenham um equilíbrio entre engenharia de software e engenharia de sistemas.

Além do design de ferramentas de automação e outros esforços para reduzir a carga de trabalho dos engenheiros no SRE, os projetos de desenvolvimento de software podem beneficiar ainda mais a organização do SRE, atraiendo e ajudando a reter engenheiros com uma ampla variedade de habilidades. A conveniência da diversidade da equipe é duplamente verdadeira para o SRE, onde uma variedade de origens e abordagens de solução de problemas podem ajudar a evitar pontos cegos. Para isso, o Google sempre se esforça para equipar suas equipes de SRE com uma mistura de engenheiros com experiência tradicional em desenvolvimento de software e engenheiros com experiência em engenharia de sistemas.

Estudo de Caso Auxon: Histórico do Projeto e Espaço do Problema

Este estudo de caso examina o Auxon, uma ferramenta poderosa desenvolvida no SRE para automatizar o planejamento de capacidade para serviços executados na produção do Google. Para entender melhor como o Auxon foi concebido e os problemas que ele aborda, examinaremos primeiro o espaço do problema associado ao planejamento de capacidade e as dificuldades que as abordagens tradicionais dessa tarefa apresentam para os serviços do Google e do setor como um todo. Para obter mais contexto sobre como o Google usa os termos serviço e cluster, consulte o [Capítulo 2](#).

Planejamento de Capacidade Tradicional

Existem inúmeras táticas para planejamento de capacidade de recursos de computação (consulte [\[Hix15a\]](#)), mas a maioria dessas abordagens se resume a um ciclo que pode ser aproximado da seguinte forma:

1) Colete previsões de demanda.

Quantos recursos são necessários? Quando e onde esses recursos são necessários?

- Usa os melhores dados disponíveis hoje para planejar o futuro • Normalmente cobre de vários trimestres a anos

2) Elaborar planos de construção e alocação.

Diante dessa perspectiva projetada, qual a melhor forma de atender essa demanda com oferta adicional de recursos? Quanta oferta e em quais locais?

3) Revise e assine o plano.

A previsão é razoável? O plano está alinhado com considerações orçamentárias, de nível de produto e técnicas?

4) Implante e configure recursos.

Uma vez que os recursos eventualmente chegam (potencialmente em fases ao longo de um período de tempo definido), quais serviços podem usar os recursos? Como faço para que os recursos de nível inferior (CPU, disco, etc.) sejam úteis para os serviços?

Vale ressaltar que o planejamento da capacidade é um ciclo sem fim: as premissas mudam, as implantações escorregam e os orçamentos são cortados, resultando em revisão após revisão do Plano. E cada revisão tem efeitos de gotejamento que devem se propagar pelos planos de todos os trimestres subsequentes. Por exemplo, um déficit neste trimestre deve ser compensado em trimestres futuros. O planejamento de capacidade tradicional usa a demanda como um fator-chave e molda manualmente a oferta para atender à demanda em resposta a cada mudança.

Frágil por

natureza O planejamento de capacidade tradicional produz um plano de alocação de recursos que pode ser interrompido por qualquer mudança aparentemente menor. Por exemplo:

- Um serviço sofre redução de eficiência e precisa de mais recursos do que o esperado para atender a mesma demanda.
 - As taxas de adoção de clientes aumentam, resultando em um aumento na demanda projetada.
 - A data de entrega de um novo cluster de recursos de computação é atrasada.
- Uma decisão de produto sobre uma meta de desempenho altera a forma da implantação de serviço necessária (a área de cobertura do serviço) e a quantidade de Recursos.

Pequenas mudanças exigem verificação cruzada de todo o plano de alocação para garantir que o plano ainda seja viável; alterações maiores (como entrega atrasada de recursos ou alterações na estratégia do produto) exigem potencialmente a recriação do plano do zero. Uma derrapagem de entrega em um único cluster pode afetar os requisitos de redundância ou latência de vários serviços: as alocações de recursos em outros clusters devem ser aumentadas para compensar a derrapagem, e essas e quaisquer outras alterações teriam que se propagar por todo o plano.

Além disso, considere que o plano de capacidade para qualquer trimestre (ou outro período de tempo) é baseado no resultado esperado dos planos de capacidade dos trimestres anteriores, o que significa que uma mudança em qualquer trimestre resulta em trabalho para atualizar trimestres subsequentes.

Trabalhoso e impreciso

Para muitas equipes, o processo de coleta dos dados necessários para gerar as previsões de demanda é lento e sujeito a erros. E quando é hora de encontrar capacidade para atender a essa demanda futura, nem todos os recursos são igualmente adequados. Por exemplo, se os requisitos de latência significam que um serviço deve se comprometer a atender a demanda do usuário no mesmo continente

como usuário, obter recursos adicionais na América do Norte não aliviará o déficit de capacidade na Ásia. Cada previsão tem restrições ou parâmetros sobre como pode ser cumprida; as restrições estão fundamentalmente relacionadas à intenção, que é discutida na próxima seção.

O mapeamento de solicitações de recursos restritos em alocações de recursos reais a partir da capacidade disponível é igualmente lento: é complexo e tedioso empacotar solicitações manualmente em um espaço limitado ou encontrar soluções que se ajustem a um orçamento limitado.

Esse processo já pode pintar um quadro sombrio, mas para piorar as coisas, as ferramentas necessárias geralmente não são confiáveis ou são complicadas. As planilhas sofrem severamente com problemas de escalabilidade e têm capacidades limitadas de verificação de erros. Os dados tornam-se obsoletos e o rastreamento de alterações torna-se difícil. As equipes geralmente são forçadas a simplificar

suposições e reduzir a complexidade de seus requisitos, simplesmente para tornar a manutenção da capacidade adequada um problema tratável.

Quando os proprietários de serviços enfrentam os desafios de ajustar uma série de solicitações de capacidade de vários serviços aos recursos disponíveis para eles, de uma maneira que atenda às várias restrições que um serviço pode ter, ocorre imprecisão adicional. Bin packing é um problema NP-difícil que é difícil para os seres humanos calcularem manualmente. Além disso, a solicitação de capacidade de um serviço geralmente é um conjunto inflexível de requisitos de demanda: X núcleos no cluster Y. As razões pelas quais X núcleos ou Y cluster são necessários e quaisquer graus de liberdade em torno desses parâmetros são perdidos há muito tempo pelo momento em que o pedido chega a um humano tentando encaixar uma lista de demandas no suprimento disponível.

O resultado líquido é um enorme dispêndio de esforço humano para criar uma embalagem de lixo que seja aproximada, na melhor das hipóteses. O processo é frágil para mudar e não há limites conhecidos para uma solução ótima.

Nossa solução: planejamento de capacidade baseado em

intenção Especifique os requisitos, não a implementação.

No Google, muitas equipes adotaram uma abordagem que chamamos de planejamento de capacidade baseado em intenção. A premissa básica dessa abordagem é codificar programaticamente as dependências e parâmetros (intenção) das necessidades de um serviço e usar essa codificação para gerar automaticamente um plano de alocação que detalha quais recursos vão para qual serviço e em qual cluster. Se a demanda, o fornecimento ou os requisitos de serviço mudarem, podemos simplesmente gerar automaticamente um novo plano em resposta aos parâmetros alterados, que agora é a nova melhor distribuição de recursos.

Com os verdadeiros requisitos e a flexibilidade de um serviço capturados, o plano de capacidade agora é muito mais ágil diante da mudança, e podemos chegar a uma solução ideal que atenda ao maior número possível de parâmetros. Com o bin packing delegado aos computadores, o trabalho humano é drasticamente reduzido e os proprietários de serviços podem se concentrar em prioridades de alta ordem, como SLOs, dependências de produção e requisitos de infraestrutura de serviço, em oposição à busca de recursos de baixo nível.

Como benefício adicional, o uso da otimização computacional para mapear da intenção à implementação alcança uma precisão muito maior, resultando em economia de custos para a organização. Bin packing ainda está longe de ser um problema resolvido, porque certos tipos ainda são considerados NP-hard; no entanto, os algoritmos de hoje podem resolver para uma solução ótima conhecida.

Planejamento de capacidade baseado em intenção

A intenção é a razão de como um proprietário de serviço deseja executar seu serviço. Passar de demandas concretas de recursos para motivos motivadores para chegar ao verdadeiro

a intenção de planejamento de capacidade geralmente requer várias camadas de abstração. Considere a seguinte cadeia de abstração:

- 1) "Quero 50 núcleos nos clusters X, Y e Z para o serviço Foo."

Esta é uma solicitação de recurso explícita. Mas... por que precisamos de tantos recursos especificamente nesses clusters específicos?

- 2) "Quero uma área de cobertura de 50 núcleos em quaisquer 3 clusters na região geográfica YYY para o serviço Foo."

Essa solicitação introduz mais graus de liberdade e é potencialmente mais fácil de cumprir, embora não explique o raciocínio por trás de seus requisitos. Mas... por que precisamos dessa quantidade de recursos e por que 3 pegadas?

- 3) "Quero atender a demanda do serviço Foo em cada região geográfica, e ter redundância N+2."

De repente, uma maior flexibilidade é introduzida e podemos entender em um nível mais "humano" o que acontece se o serviço Foo não receber esses recursos.

Mas... por que precisamos de N + 2 para o serviço Foo?

- 4) "Quero executar o serviço Foo com 5 noves de confiabilidade."

Este é um requisito mais abstrato, e a ramificação se o requisito não for atendido fica claro: a confiabilidade será prejudicada. E temos ainda maior flexibilidade aqui: talvez rodar em N + 2 não seja realmente suficiente ou ideal para este serviço, e algum outro plano de implantação seria mais adequado.

Então, qual nível de intenção deve ser usado pelo planejamento de capacidade orientado por intenção? Idealmente, todos os níveis de intenção devem ser suportados juntos, com os serviços se beneficiando quanto mais eles mudam para especificar a intenção versus a implementação. Na experiência do Google, os serviços tendem a obter as melhores vitórias à medida que passam para a etapa 3: bons graus de flexibilidade estão disponíveis e as ramificações dessa solicitação estão em termos de nível superior e compreensíveis. Serviços particularmente sofisticados podem apontar para a etapa 4.

Precursors da intenção

De que informações precisamos para capturar a intenção de um serviço? Insira dependências, métricas de desempenho e priorização.

Dependências

Os serviços do Google dependem de muitas outras infraestruturas e serviços voltados para o usuário, e essas dependências influenciam fortemente onde um serviço pode ser colocado. Por exemplo, imagine o serviço Foo voltado para o usuário, que depende do Bar, um serviço de armazenamento de infraestrutura. Foo expressa um requisito de que Bar deve estar localizado dentro de 30 milissegundos de latência de rede de Foo. Esse requisito tem repercussões importantes para onde colocamos Foo e Bar, e o planejamento de capacidade orientado à intenção deve levar essas restrições em consideração.

Além disso, as dependências de produção são aninhadas: para desenvolver o exemplo anterior, imagine que o serviço Bar tenha suas próprias dependências no Baz, um serviço de armazenamento distribuído de nível inferior, e no Qux, um serviço de gerenciamento de aplicativos. Portanto, onde agora podemos colocar Foo depende de onde podemos colocar Bar, Baz e Qux. Um determinado conjunto de dependências de produção pode ser compartilhado, possivelmente com diferentes estipulações em torno da intenção.

Métricas de desempenho

A demanda por um serviço diminui para resultar na demanda por um ou mais outros serviços. Compreender a cadeia de dependências ajuda a formular o escopo geral do problema de empacotamento, mas ainda precisamos de mais informações sobre o uso esperado de recursos. De quantos recursos de computação o serviço Foo precisa para atender N consultas de usuários? Para cada N consultas do serviço Foo, quantos Mbps de dados esperamos para o serviço Bar?

As métricas de desempenho são a cola entre as dependências. Eles convertem de um ou mais tipos de recursos de nível superior para um ou mais tipos de recursos de nível inferior.

A obtenção de métricas de desempenho apropriadas para um serviço pode envolver testes de carga e monitoramento de uso de recursos.

Priorização

Inevitavelmente, as restrições de recursos resultam em trocas e decisões difíceis: dos muitos requisitos que todos os serviços possuem, quais requisitos devem ser sacrificados diante da capacidade insuficiente?

Talvez a redundância N + 2 para o serviço Foo seja mais importante que a redundância N + 1 para o serviço Bar. Ou talvez o lançamento do recurso X seja menos importante que a redundância N + 0 para o serviço Baz.

O planejamento orientado à intenção força essas decisões a serem tomadas de forma transparente, aberta e consistente. As restrições de recursos implicam as mesmas compensações, mas com muita frequência a priorização pode ser ad hoc e opaca para os proprietários de serviços. O planejamento baseado em intenção permite que a priorização seja tão granular ou grosseira conforme necessário.

Introdução ao Auxon

Auxon é a implementação do Google de uma solução de planejamento de capacidade e alocação de recursos baseada em intenção e um excelente exemplo de um produto de engenharia de software projetado e desenvolvido pelo SRE: foi construído por um pequeno grupo de engenheiros de software e um gerente de programa técnico dentro do SRE ao longo de dois anos. Auxon é um estudo de caso perfeito para demonstrar como o desenvolvimento de software pode ser promovido dentro do SRE.

Auxon é usado ativamente para planejar o uso de muitos milhões de dólares em recursos de máquina no Google. Tornou-se um componente crítico do planejamento de capacidade para várias divisões importantes do Google.

Como produto, o Auxon fornece os meios para coletar descrições baseadas em intenção dos requisitos e dependências de recursos de um serviço. Essas intenções do usuário são expressas como requisitos de como o proprietário gostaria que o serviço fosse provisionado. Os requisitos podem ser especificados como uma solicitação como “Meu serviço deve ser N + 2 por continente” ou “Os servidores front-end não devem estar a mais de 50 ms dos servidores back-end”. O Auxon coleta essas informações por meio de uma linguagem de configuração do usuário ou por meio de uma API programática, traduzindo assim a intenção humana em restrições analisáveis por máquina. Os requisitos podem ser priorizados, um recurso que é útil se os recursos são insuficientes para atender a todos os requisitos e, portanto, compensações devem ser feitas. Esses requisitos – a intenção – são, em última análise, representados internamente como um programa linear ou inteiro misto gigante. Auxon resolve o programa linear e usa a solução de empacotamento resultante para formular um plano de alocação de recursos.

A Figura 18-1 e as explicações que a seguem descrevem os principais componentes do Auxon.

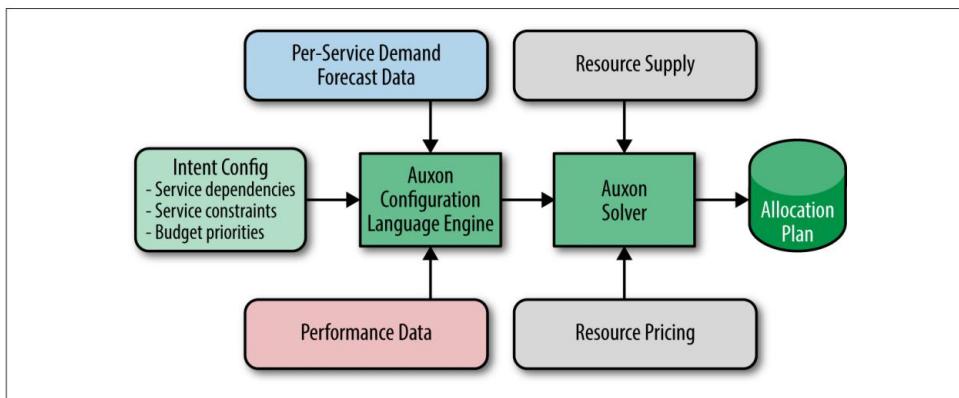


Figura 18-1. Os principais componentes do Auxon

Os dados de desempenho descrevem como um serviço é dimensionado: para cada unidade de demanda X no cluster Y, quantas unidades de dependência Z são usadas? Esses dados de dimensionamento podem ser derivados de várias maneiras, dependendo da maturidade do serviço em questão. Alguns serviços são testados quanto à carga, enquanto outros inferem sua escala com base no desempenho anterior.

Os dados de previsão de demanda por serviço descrevem a tendência de uso dos sinais de demanda previstos. Alguns serviços derivam seu uso futuro de previsões de demanda - uma previsão de consultas por segundo dividida por continente. Nem todos os serviços têm uma previsão de demanda: alguns serviços (por exemplo, um serviço de armazenamento como o Colossus) derivam sua demanda puramente de serviços que dependem deles.

O Resource Supply fornece dados sobre a disponibilidade de recursos fundamentais de nível básico: por exemplo, o número de máquinas que se espera que estejam disponíveis para uso em um determinado ponto no futuro. Na terminologia do programa linear, o fornecimento de recursos atua como um limite superior que limita como os serviços podem crescer e onde os serviços podem ser colocados. Em última análise, queremos fazer o melhor uso desse suprimento de recursos conforme a descrição baseada em intenção do grupo combinado de serviços permite.

A Precificação de Recursos fornece dados sobre quanto custam os recursos fundamentais de nível básico. Por exemplo, o custo das máquinas pode variar globalmente com base nas cargas de espaço/energia de uma determinada instalação. Na terminologia do programa linear, os preços informam os custos globais calculados, que atuam como o objetivo que queremos minimizar.

O Intent Config é a chave para como as informações baseadas em intenção são alimentadas ao Auxon. Ele define o que constitui um serviço e como os serviços se relacionam entre si. A configuração atua como uma camada de configuração que permite que todos os outros componentes sejam conectados. Ele foi projetado para ser legível e configurável por humanos.

O Auxon Configuration Language Engine atua com base nas informações que recebe do Intent Config. Este componente formula uma solicitação legível por máquina (um buffer de protocolo que pode ser entendido pelo Auxon Solver. Ele aplica a verificação de sanidade leve à configuração e é projetado para atuar como o gateway entre a definição de intenção configurável por humanos e a definição de intenção analisável por máquina solicitação de otimização.

Auxon Solver é o cérebro da ferramenta. Ele formula o programa de inteiro misto ou linear gigante com base na solicitação de otimização recebida do Configuration Language Engine. Ele foi projetado para ser muito escalável, o que permite que o solver seja executado em paralelo em centenas ou mesmo milhares de máquinas em execução nos clusters do Google.

Além dos kits de ferramentas de programação linear de inteiros mistos, também há componentes no Auxon Solver que lidam com tarefas como agendamento, gerenciamento de um pool de trabalhadores e árvores de decisão descendentes.

O Plano de Alocação é a saída do Auxon Solver. Ele prescreve quais recursos devem ser alocados para quais serviços e em quais locais. São os detalhes computados de implementação da definição baseada em intenção dos requisitos do problema de planejamento de capacidade. O Plano de Alocação também inclui informações sobre quaisquer requisitos que não puderam ser atendidos — por exemplo, se um requisito não pôde ser atendido devido à falta de recursos ou requisitos concorrentes que, de outra forma, eram muito rígidos.

Requisitos e implementação: sucessos e lições aprendidas O Auxon foi imaginado pela primeira vez por um SRE e um gerente de programa técnico que haviam sido incumbidos separadamente por suas respectivas equipes de planejar a capacidade de grandes porções da infraestrutura do Google. Tendo realizado o planejamento manual de capacidade em planilhas,

eles estavam bem posicionados para entender as ineficiências e oportunidades de melhoria por meio da automação e os recursos que tal ferramenta poderia exigir.

Ao longo do desenvolvimento da Auxon, a equipe SRE por trás do produto continuou profundamente envolvida no mundo da produção. A equipe manteve um papel em rodízios de plantão para vários serviços do Google e participou de discussões de design e liderança técnica desses serviços. Por meio dessas interações contínuas, a equipe conseguiu se manter no mundo da produção: eles agiam como consumidores e desenvolvedores de seu próprio produto. Quando o produto falhou, a equipe foi diretamente impactada. As solicitações de recursos foram informadas por meio de experiências em primeira mão da própria equipe. A experiência em primeira mão do espaço do problema não apenas trouxe um enorme senso de propriedade no sucesso do produto, mas também ajudou a dar credibilidade e legitimidade ao produto dentro do SRE.

Aproximação

Não foque na perfeição e pureza da solução, especialmente se os limites do problema não forem bem conhecidos. Iniciar e iterar.

Qualquer esforço de engenharia de software suficientemente complexo está fadado a encontrar incertezas sobre como um componente deve ser projetado ou como um problema deve ser abordado. Auxon enfrentou essa incerteza no início de seu desenvolvimento porque o mundo da programação linear era um território desconhecido para os membros da equipe. As limitações da programação linear, que pareciam ser uma parte central de como o produto provavelmente funcionaria, não eram bem compreendidas. Para resolver a consternação da equipe sobre essa dependência insuficientemente compreendida, optamos por construir inicialmente um mecanismo de resolução simplificado (o chamado "Stupid Solver") que aplicava algumas heurísticas simples sobre como os serviços deveriam ser organizados com base nos requisitos especificados pelo usuário. Embora o Stupid Solver nunca produzisse uma solução verdadeiramente ideal, deu à equipe a sensação de que nossa visão para o Auxon era alcançável, mesmo que não construíssemos algo perfeito desde o primeiro dia.

Ao implantar a aproximação para ajudar a acelerar o desenvolvimento, é importante realizar o trabalho de uma forma que permita que a equipe faça melhorias futuras e revisite a aproximação. No caso do Stupid Solver, toda a interface do solver foi abstraída dentro do Auxon, de modo que os componentes internos do solver pudessem ser trocados posteriormente. Eventualmente, à medida que criamos confiança em um modelo de programação linear unificado, foi uma operação simples trocar o Stupid Solver por algo, bem, mais esperto.

Os requisitos de produto da Auxon também tinham algumas incógnitas. Construir software com requisitos difusos pode ser um desafio frustrante, mas algum grau de incerteza não precisa ser um obstáculo. Use essa imprecisão como um incentivo para garantir que o software seja projetado para ser geral e modular. Por exemplo, um dos objetivos do projeto Auxon era integrar com sistemas de automação dentro do Google para permitir uma

Plano de alocação a ser executado diretamente na produção (atribuição de recursos e ativação/desativação/redimensionamento de serviços conforme apropriado). No entanto, na época, o mundo dos sistemas de automação estava em grande fluxo, pois uma enorme variedade de abordagens estava em uso. Em vez de tentar projetar soluções exclusivas para permitir que o Auxon trabalhe com cada ferramenta individual, moldamos o Plano de Alocação para ser universalmente útil, de modo que esses sistemas de automação pudessem trabalhar em seus próprios pontos de integração. Essa abordagem “agnóstica” tornou-se fundamental para o processo de integração de novos clientes da Auxon, porque permitiu que os clientes começassem a usar o Auxon sem alternar para uma ferramenta de automação de ativação específica, ferramenta de previsão ou ferramenta de dados de desempenho.

Também aproveitamos projetos modulares para lidar com requisitos difusos ao construir um modelo de desempenho de máquina no Auxon. Os dados sobre o desempenho futuro da plataforma da máquina (por exemplo, CPU) eram escassos, mas nossos usuários queriam uma maneira de modelar vários cenários de potência da máquina. Abstraímos os dados da máquina por trás de uma única interface, permitindo que o usuário troque em diferentes modelos de desempenho futuro da máquina. Mais tarde, estendemos ainda mais essa modularidade, com base em requisitos cada vez mais definidos, para fornecer uma biblioteca simples de modelagem de desempenho de máquina que funcionasse nessa interface.

Se há um tema a ser extraído de nosso estudo de caso Auxon, é que o velho lema de “lançar e iterar” é particularmente relevante em projetos de desenvolvimento de software SRE.

Não espere pelo design perfeito; em vez disso, mantenha a visão geral em mente enquanto avança com o design e o desenvolvimento. Quando você encontrar áreas de incerteza, projete o software para ser flexível o suficiente para que, se o processo ou a estratégia mudarem em um nível mais alto, você não incorra em um enorme custo de retrabalho. Mas, ao mesmo tempo, mantenha-se firme, certificando-se de que as soluções gerais tenham uma implementação específica do mundo real que demonstre a utilidade do design.

Aumentando a conscientização e impulsionando a

adoção Como acontece com qualquer produto, o software desenvolvido pela SRE deve ser projetado com conhecimento de seus usuários e requisitos. Ele precisa impulsionar a adoção por meio de utilidade, desempenho e capacidade demonstrada para beneficiar as metas de confiabilidade de produção do Google e melhorar a vida dos SREs. O processo de socialização de um produto e obtenção de adesão em toda a organização é a chave para o sucesso do projeto.

Não subestime o esforço necessário para aumentar a conscientização e o interesse em seu produto de software – uma única apresentação ou anúncio por e-mail não é suficiente. Socializar ferramentas de software internas para um grande público exige o seguinte:

- Uma abordagem consistente e coerente •

Defesa do usuário

- O patrocínio de engenheiros e gerentes seniores, a quem você terá que demonstrar a utilidade do seu produto

É importante considerar a perspectiva do cliente ao tornar seu produto utilizável. Um engenheiro pode não ter tempo ou inclinação para investigar o código-fonte para descobrir como usar uma ferramenta. Embora os clientes internos geralmente sejam mais tolerantes com arestas e alfas iniciais do que os clientes externos, ainda é necessário fornecer documentação. Os SREs estão ocupados e, se sua solução for muito difícil ou confusa, eles escreverão sua própria solução.

Defina expectativas

Quando um engenheiro com anos de familiaridade com um problema começa a projetar um produto, é fácil imaginar um estado final utópico para o trabalho. No entanto, é importante diferenciar os objetivos aspiracionais do produto dos critérios mínimos de sucesso (ou Produto Mínimo Viável). Os projetos podem perder credibilidade e falhar prometendo demais, cedo demais; ao mesmo tempo, se um produto não promete um resultado suficientemente recompensador, pode ser difícil superar a energia de ativação necessária para convencer as equipes internas a tentar algo novo. Demonstrar progresso constante e incremental por meio de pequenas versões aumenta a confiança do usuário na capacidade de sua equipe de fornecer software útil.

No caso da Auxon, alcançamos um equilíbrio planejando um roteiro de longo prazo juntamente com correções de curto prazo. As equipes foram prometidas que:

- Quaisquer esforços de integração e configuração forneceria o benefício imediato de aliviar a dor de empacotar manualmente as solicitações de recursos de curto prazo. • À medida que recursos adicionais fossem desenvolvidos para o Auxon, os mesmos arquivos de configuração seriam transferidos e forneceria novas e muito mais amplas economias de custo de longo prazo e outros benefícios. O roteiro do projeto permitiu que os serviços determinassem rapidamente se seus casos de uso ou recursos necessários não foram implementados nas versões anteriores.
- Enquanto isso, a abordagem de desenvolvimento iterativo da Auxon alimentou as prioridades de desenvolvimento e novos marcos para o roteiro.

Identifique os clientes adequados

A equipe que desenvolveu o Auxon percebeu que uma solução de tamanho único pode não servir para todos; muitas equipes maiores já tinham soluções domésticas para planejamento de capacidade que funcionavam razoavelmente bem. Embora suas ferramentas personalizadas não fossem perfeitas, essas equipes não tiveram problemas suficientes no processo de planejamento de capacidade para experimentar uma nova ferramenta, especialmente uma versão alfa com arestas.

As versões iniciais do Auxon visavam intencionalmente equipes que não possuíam processos de planejamento de capacidade existentes. Porque essas equipes teriam que investir cony

esforço de configuração, quer adotassem uma ferramenta existente ou nossa nova abordagem, eles estavam interessados em adotar a ferramenta mais nova. Os primeiros sucessos alcançados pela Auxon com essas equipes demonstraram a utilidade do projeto e transformaram os próprios clientes em defensores da ferramenta. Quantificar a utilidade do produto provou ser ainda mais benéfico; quando integrarmos uma das áreas de negócios do Google, a equipe criou um estudo de caso detalhando o processo e comparando os resultados antes e depois. A economia de tempo e a redução da labuta humana por si só representaram um grande incentivo para outras equipes experimentarem o Auxon.

Atendimento ao Cliente

Embora o software desenvolvido dentro do SRE tenha como alvo um público de TPMs e engenheiros com alta proficiência técnica, qualquer software suficientemente inovador ainda apresenta uma curva de aprendizado para novos usuários. Não tenha medo de fornecer suporte ao cliente de luva branca para os adotantes iniciais para ajudá-los no processo de integração.

Às vezes, a automação também envolve uma série de preocupações emocionais, como o medo de que o trabalho de alguém seja substituído por um script de shell. Ao trabalhar individualmente com os primeiros usuários, você pode lidar com esses medos pessoalmente e demonstrar que, em vez de assumir o trabalho de realizar uma tarefa tediosa manualmente, a equipe possui as configurações, processos e resultados finais de seu trabalho técnico. Os adotantes posteriores são convencidos pelos exemplos felizes dos adotantes iniciais.

Além disso, como as equipes de SRE do Google estão distribuídas em todo o mundo, os defensores dos primeiros adeptos de um projeto são particularmente benéficos, pois podem servir como especialistas locais para outras equipes interessadas em experimentar o projeto.

Projetando no nível certo

Uma ideia que chamamos de agnosticismo – escrever o software para ser generalizado para permitir uma miríade de fontes de dados como entrada – foi um princípio fundamental do projeto de Auxon. O agnosticismo significava que os clientes não precisavam se comprometer com nenhuma ferramenta para usar a estrutura Auxon. Essa abordagem permitiu que o Auxon permanecesse com utilidade geral suficiente, mesmo quando equipes com casos de uso divergentes começaram a usá-lo.

Abordamos potenciais usuários com a mensagem “venha como você é; vamos trabalhar com o que você tem.” Ao evitar a personalização excessiva para um ou dois grandes usuários, alcançamos uma adoção mais ampla em toda a organização e reduzimos a barreira de entrada para novos serviços.

Também nos esforçamos conscientemente para evitar a armadilha de definir o sucesso como 100% de adoção em toda a organização. Em muitos casos, há retornos decrescentes ao fechar a última milha para habilitar um conjunto de recursos suficiente para todos os serviços da cauda longa do Google.

Dinâmica da equipe

Ao selecionar engenheiros para trabalhar em um produto de desenvolvimento de software SRE, descobrimos um grande benefício em criar uma equipe inicial que combina generalistas capazes de se atualizar rapidamente em um novo tópico com engenheiros que possuem amplo conhecimento e experiência . Uma diversidade de experiências abrange pontos cegos, bem como as armadilhas de assumir que o caso de uso de cada equipe é o mesmo que o seu.

É essencial que sua equipe estabeleça uma relação de trabalho com os especialistas necessários e que seus engenheiros se sintam à vontade para trabalhar em um novo espaço de problemas. Para as equipes de SRE na maioria das empresas, aventurar-se nesse novo espaço de problemas exige terceirizar tarefas ou trabalhar com consultores, mas as equipes de SRE em organizações maiores podem fazer parcerias com especialistas internos. Durante as fases iniciais de conceituação e design do Auxon, apresentamos nosso documento de design para as equipes internas do Google especializadas em Pesquisa Operacional e Análise Quantitativa para aproveitar sua experiência no campo e impulsionar o conhecimento da equipe do Auxon sobre capacidade planejamento.

À medida que o desenvolvimento do projeto continuava e o conjunto de recursos da Auxon se tornava mais amplo e complexo, a equipe adquiriu membros com experiência em estatística e otimização matemática, o que em uma empresa menor pode ser semelhante a trazer um consultor externo para dentro da empresa. Esses novos membros da equipe foram capazes de identificar áreas de melhoria quando a funcionalidade básica do projeto foi concluída e adicionar sutiliza tornou-se nossa principal prioridade.

O momento certo para contratar especialistas varia, é claro, de projeto para projeto. Como uma diretriz aproximada, o projeto deve ser bem-sucedido e comprovadamente bem-sucedido, de modo que as habilidades da equipe atual sejam significativamente reforçadas pelo conhecimento adicional.

Fomentando a Engenharia de Software no SRE

O que torna um projeto um bom candidato para dar o salto de uma ferramenta única para um esforço de engenharia de software completo? Sinais positivos fortes incluem engenheiros com experiência em primeira mão no domínio relativo que estão interessados em trabalhar no projeto e uma base de usuários alvo que é altamente técnica (e, portanto, capaz de fornecer relatórios de bugs de alto sinal durante as fases iniciais de desenvolvimento). O projeto deve fornecer benefícios visíveis, como reduzir o trabalho dos SREs, melhorar uma infraestrutura existente ou simplificar um processo complexo.

É importante que o projeto se encaixe no conjunto geral de objetivos da organização, para que os líderes de engenharia possam avaliar seu impacto potencial e, posteriormente, defender seu projeto, tanto com suas equipes de relatórios quanto com outras equipes que possam interagir com suas equipes. . Socialização entre organizações e ajuda de revisão

evitar esforços desarticulados ou sobrepostos, e um produto que pode ser facilmente estabelecido como a promoção de um objetivo de todo o departamento é mais fácil para a equipe e o suporte.

O que torna um projeto candidato ruim? Muitas das mesmas bandeiras vermelhas que você pode identificar instintivamente em qualquer projeto de software, como software que toca muitas partes móveis ao mesmo tempo, ou design de software que requer uma abordagem tudo ou nada que impeça o desenvolvimento iterativo. Como as equipes do Google SRE estão atualmente organizadas em torno dos serviços que executam, os projetos desenvolvidos pelo SRE correm o risco de serem trabalhos excessivamente específicos que beneficiam apenas uma pequena porcentagem da organização. Como os incentivos da equipe são alinhados principalmente para fornecer uma ótima experiência para os usuários de um serviço específico, os projetos geralmente não conseguem generalizar para um caso de uso mais amplo, pois a padronização entre as equipes de SRE vem em segundo lugar. No extremo oposto do espectro, frameworks excessivamente genéricos podem ser igualmente problemáticos; se uma ferramenta se esforça para ser muito flexível e muito universal, ela corre o risco de não se encaixar em nenhum caso de uso e, portanto, ter valor insuficiente por si só. Projetos com grande escopo e objetivos abstratos geralmente exigem um esforço de desenvolvimento significativo, mas não possuem os casos de uso concretos necessários para fornecer benefícios ao usuário final em um prazo razoável.

Como exemplo de um caso de uso amplo: um平衡ador de carga de camada 3 desenvolvido pelos SREs do Google provou ser tão bem-sucedido ao longo dos anos que foi reutilizado como uma oferta de produto voltada para o cliente por meio do Google Cloud Load Balancer [Eis16].

Construindo com sucesso uma cultura de engenharia de software em SRE: espaço e tempo de desenvolvimento SREs são geralmente generalistas, pois o desejo de aprender primeiro em amplitude em vez de em profundidade se presta bem a entender o quadro geral (e há poucos quadros maiores do que o intrincado funcionamento interno da infraestrutura técnica moderna). Esses engenheiros geralmente têm fortes habilidades de codificação e desenvolvimento de software, mas podem não ter a experiência tradicional de SWE de fazer parte de uma equipe de produto ou ter que pensar nas solicitações de recursos do cliente. Uma citação de um engenheiro em um projeto inicial de desenvolvimento de software SRE resume a abordagem SRE convencional para software: “Eu tenho um documento de design; por que precisamos de requisitos?” A parceria com engenheiros, TPMs ou PMs que estão familiarizados com o desenvolvimento de software voltado para o usuário pode ajudar a construir uma cultura de desenvolvimento de software em equipe que reúna o melhor do desenvolvimento de produtos de software e da experiência prática de produção.

O tempo de trabalho dedicado e ininterrupto do projeto é essencial para qualquer esforço de desenvolvimento de software. O tempo de projeto dedicado é necessário para permitir o progresso em um projeto, porque é quase impossível escrever código – muito menos se concentrar em projetos maiores e mais impactantes – quando você está se debatendo entre várias tarefas no decorrer de uma hora. Portanto, a capacidade de trabalhar em um projeto de software sem interrupções costuma ser uma razão atraente para os engenheiros começarem a trabalhar em um projeto de desenvolvimento. Esse tempo deve ser defendido agressivamente.

A maioria dos produtos de software desenvolvidos dentro do SRE começa como projetos paralelos cuja utilidade os leva a crescer e se formalizar. Neste ponto, um produto pode se ramificar em uma das várias direções possíveis:

- Permanecer como um esforço de base desenvolvido no tempo livre dos engenheiros • Estabelecer-se como um projeto formal por meio de processos estruturados (ver “[Getýting Lá](#)”)
- Obter patrocínio executivo da liderança da SRE para expandir para um esforço de desenvolvimento de software com equipe completa

No entanto, em qualquer um desses cenários – e este é um ponto que vale a pena enfatizar – é essencial que os SREs envolvidos em qualquer esforço de desenvolvimento continuem trabalhando como SREs em vez de se tornarem desenvolvedores em tempo integral incorporados à organização SRE. A imersão no mundo da produção dá aos SREs que realizam o trabalho de desenvolvimento uma perspectiva inestimável, pois eles são o criador e o cliente de qualquer produto.

Chegando Iá Se

você gosta da ideia de desenvolvimento organizado de software em SRE, provavelmente está se perguntando como introduzir um modelo de desenvolvimento de software em uma organização de SRE focada no suporte à produção.

Primeiro, reconheça que esse objetivo é tanto uma mudança organizacional quanto um desafio técnico. Os SREs estão acostumados a trabalhar em estreita colaboração com seus colegas de equipe, analisando e reagindo rapidamente aos problemas. Portanto, você está trabalhando contra o instinto natural de um SRE para escrever rapidamente algum código para atender às suas necessidades imediatas. Se sua equipe de SRE for pequena, essa abordagem pode não ser problemática. No entanto, à medida que sua organização cresce, essa abordagem ad hoc não será dimensionada, resultando em soluções de software amplamente funcionais, mas restritas ou de propósito único, que não podem ser compartilhadas, o que inevitavelmente leva a esforços duplicados e perda de tempo.

Em seguida, pense no que você deseja alcançar desenvolvendo software em SRE. Você quer apenas promover melhores práticas de desenvolvimento de software em sua equipe ou está interessado em desenvolvimento de software que produza resultados que possam ser usados em todas as equipes, possivelmente como um padrão para a organização? Em organizações maiores estabelecidas, a última mudança levará tempo, possivelmente abrangendo vários anos. Tal mudança precisa ser enfrentada em várias frentes, mas tem um retorno maior. A seguir estão algumas diretrizes da experiência do Google:

Crie e comunique uma mensagem clara

É importante definir e comunicar sua estratégia, planos e – o mais importante – os benefícios que o SRE obtém com esse esforço. Os SREs são muito céticos (na verdade, o ceticismo é uma característica para a qual contratamos especificamente); a resposta inicial de um SRE a tal esforço provavelmente será “isso soa como muita sobrecarga” ou “vai

nunca funciona". Comece apresentando um caso convincente de como essa estratégia ajudará o SRE; por exemplo:

- Soluções de software consistentes e com suporte aceleram a aceleração para novos SREs.
- Reduzir o número de maneiras de executar a mesma tarefa permite que todo o departamento se beneficie das habilidades que uma única equipe desenvolveu, tornando assim o conhecimento e o esforço portáteis entre as equipes.

Quando os SREs começam a fazer perguntas sobre como sua estratégia funcionará, em vez de se a estratégia deve ser seguida, você sabe que ultrapassou o primeiro obstáculo.

Avalie as capacidades da sua organização

Os SREs têm muitas habilidades, mas é relativamente comum que um SRE não tenha experiência como parte de uma equipe que construiu e enviou um produto para um conjunto de usuários. Para desenvolver um software útil, você está efetivamente criando uma equipe de produto. Essa equipe inclui funções e habilidades necessárias que sua organização de SRE pode não ter exigido anteriormente. Alguém desempenhará o papel de gerente de produto, atuando como defensor do cliente? Seu líder de tecnologia ou gerente de projeto tem as habilidades e/ou experiência para executar um processo de desenvolvimento ágil?

Comece a preencher essas lacunas aproveitando as habilidades já presentes em sua empresa. Peça à sua equipe de desenvolvimento de produto para ajudá-lo a estabelecer práticas ágeis por meio de treinamento ou coaching. Solicite tempo de consultoria de um gerente de produto para ajudá-lo a definir os requisitos do produto e priorizar o trabalho de recursos. Dada uma oportunidade de desenvolvimento de software grande o suficiente, pode haver um caso para contratar pessoas dedicadas para essas funções. Fazer o caso de contratar para essas funções é mais fácil quando você tem alguns resultados positivos do experimento.

Iniciar e iterar Ao

iniciar um programa de desenvolvimento de software SRE, seus esforços serão seguidos por muitos olhares atentos. É importante estabelecer credibilidade entregando algum produto de valor em um período de tempo razoável. Sua primeira rodada de produtos deve visar metas relativamente diretas e alcançáveis – sem controvérsias ou soluções existentes. Também obtivemos sucesso ao combinar essa abordagem com um ritmo de seis meses de lançamentos de atualização de produtos que forneceram recursos úteis adicionais. Esse ciclo de lançamento permitiu que as equipes se concentrassem na identificação do conjunto certo de recursos a serem construídos e, em seguida, construíssem esses recursos enquanto aprendiam simultaneamente como ser uma equipe produtiva de desenvolvimento de software. Após o lançamento inicial, algumas equipes do Google mudaram para um modelo push-on-green para entrega e feedback ainda mais rápidos.

Não diminua seus padrões Ao

começar a desenvolver software, você pode ficar tentado a cortar custos. Resista a esse impulso mantendo-se nos mesmos padrões aos quais suas equipes de desenvolvimento de produtos são mantidas. Por exemplo:

- Pergunte a si mesmo: se este produto fosse criado por uma equipe de desenvolvimento separada, você integraria o produto?
- Se a sua solução tiver ampla adoção, ela pode se tornar crítica para os SREs realizarem seus trabalhos com sucesso. Portanto, a confiabilidade é de extrema importância. Você tem práticas adequadas de revisão de código? Você tem testes de ponta a ponta ou de integração? Peça a outra equipe de SRE que revise o produto quanto à prontidão para produção, como faria se estivesse integrando qualquer outro serviço.

Leva muito tempo para construir credibilidade para seus esforços de desenvolvimento de software, mas pouco tempo para perder credibilidade devido a um passo em falso.

Conclusões

Os projetos de engenharia de software no Google SRE floresceram à medida que a organização cresceu e, em muitos casos, as lições aprendidas e a execução bem-sucedida de projetos de desenvolvimento de software anteriores abriram o caminho para empreendimentos subsequentes.

A experiência prática única de produção que os SREs trazem para o desenvolvimento de ferramentas pode levar a abordagens inovadoras para problemas antigos, como visto com o desenvolvimento do Auxon para resolver o complexo problema de planejamento de capacidade. Projetos de software orientados a SRE também são notavelmente benéficos para a empresa no desenvolvimento de um modelo sustentável para serviços de suporte em escala. Como os SREs geralmente desenvolvem software para simplificar processos ineficientes ou automatizar tarefas comuns, esses projetos significam que a equipe de SRE não precisa escalar linearmente com o tamanho dos serviços que eles suportam.

Em última análise, os benefícios de ter SREs dedicando parte de seu tempo ao desenvolvimento de software são colhidos pela empresa, pela organização SRE e pelos próprios SREs.

CAPÍTULO 19

Balanceamento de carga no frontend

Escrito por Piotr Lewandowski
Editado por Sarah Chavis

Atendemos muitos milhões de solicitações a cada segundo e, como você já deve ter adivinhado, usamos mais de um único computador para atender a essa demanda. Mas mesmo que tivéssemos um supercomputador que fosse de alguma forma capaz de lidar com todas essas solicitações (imagine a conectividade de rede que tal configuração exigiria!), ainda não empregariamos uma estratégia que dependesse de um único ponto de falha; quando você está lidando com sistemas de grande escala, colocar todos os ovos na mesma cesta é uma receita para o desastre.

Este capítulo se concentra no balanceamento de carga de alto nível — como equilibrarmos o tráfego de usuários entre datacenters. O capítulo a seguir amplia para explorar como implementamos o balanceamento de carga dentro de um datacenter.

Poder não é a resposta

Para fins de argumentação, vamos supor que temos uma máquina incrivelmente poderosa e uma rede que nunca falha. Essa configuração seria suficiente para atender às necessidades do Google? Não. Mesmo essa configuração ainda seria limitada pelas restrições físicas associadas à nossa infraestrutura de rede. Por exemplo, a velocidade da luz é um fator limitante nas velocidades de comunicação do cabo de fibra óptica, o que cria um limite superior na rapidez com que podemos fornecer dados com base na distância que ele precisa percorrer. Mesmo em um mundo ideal, contar com uma infraestrutura com um único ponto de falha é uma má ideia.

Na realidade, o Google tem milhares de máquinas e ainda mais usuários, muitos dos quais emitem várias solicitações ao mesmo tempo. O balanceamento de carga de tráfego é como decidimos qual das muitas máquinas em nossos datacenters atenderá a uma solicitação específica. Idealmente, o tráfego é distribuído em vários links de rede, datacenters e máquinas em um

mal” moda. Mas o que significa “ótimo” neste contexto? Na verdade, não há uma resposta única, porque a solução ideal depende muito de vários fatores:

- O nível hierárquico em que avaliamos o problema (global versus local)
- O nível técnico em que avaliamos o problema (hardware versus software)
- A natureza do tráfego com o qual estamos lidando

Vamos começar analisando dois cenários de tráfego comuns: uma solicitação de pesquisa básica e uma solicitação de upload de vídeo. Os usuários desejam obter os resultados da consulta rapidamente, portanto, a variável mais importante para a solicitação de pesquisa é a latência. Por outro lado, os usuários esperam que os uploads de vídeo demorem um tempo não negligenciável, mas também desejam que essas solicitações sejam bem-sucedidas na primeira vez, portanto, a variável mais importante para o upload de vídeo é a taxa de transferência. As diferentes necessidades das duas solicitações desempenham um papel em como determinamos a distribuição ideal para cada solicitação em nível global:

- A solicitação de pesquisa é enviada para o datacenter disponível mais próximo — conforme medido em tempo de ida e volta (RTT) — porque queremos minimizar a latência na solicitação.
- O fluxo de upload de vídeo é roteado por um caminho diferente—talvez para um link que esteja subutilizado no momento— para maximizar a taxa de transferência em detrimento da latência.

Mas no nível local, dentro de um determinado datacenter, geralmente assumimos que todas as máquinas dentro do prédio estão igualmente distantes do usuário e conectadas à mesma rede. Portanto, a distribuição ideal da carga se concentra na utilização ideal dos recursos e na proteção de um único servidor contra sobrecarga.

Claro, este exemplo apresenta uma imagem muito simplificada. Na realidade, muitas outras considerações levam em consideração a distribuição de carga ideal: algumas solicitações podem ser direcionadas para um datacenter um pouco mais distante para manter os caches aquecidos ou o tráfego não interativo pode ser roteado para uma região completamente diferente para evitar o congestionamento da rede. O balanceamento de carga, especialmente para grandes sistemas, é tudo menos simples e estático. No Google, abordamos o problema por meio do balanceamento de carga em vários níveis, dois dos quais são descritos nas seções a seguir. Para apresentar uma discussão concreta, consideraremos as solicitações HTTP enviadas por TCP. O balanceamento de carga de serviços sem estado (como DNS sobre UDP) difere um pouco, mas a maioria dos mecanismos descritos aqui também deve ser aplicável a serviços sem estado.

Balanceamento de carga usando DNS

Antes que um cliente possa enviar uma solicitação HTTP, muitas vezes ele precisa procurar um endereço IP usando o DNS. Isso oferece a oportunidade perfeita para apresentar nossa primeira camada de balanceamento de carga: balanceamento de carga DNS. A solução mais simples é retornar vários registros A ou AAAA na resposta do DNS e permitir que o cliente escolha um endereço IP arbitrariamente. Embora conceitualmente simples e trivial de implementar, essa solução apresenta vários desafios.

O primeiro problema é que ele fornece muito pouco controle sobre o comportamento do cliente: os registros são selecionados aleatoriamente e cada um atrairá uma quantidade aproximadamente igual de tráfego.

Podemos mitigar esse problema? Em teoria, poderíamos usar registros SRV para especificar pesos e prioridades de registro, mas os registros SRV ainda não foram adotados para HTTP.

Outro problema potencial decorre do fato de que normalmente o cliente não consegue determinar o endereço mais próximo. Podemos mitigar esse cenário usando um endereço anycast para servidores de nomes autorizados e aproveitar o fato de que as consultas DNS fluirão para o endereço mais próximo. Em sua resposta, o servidor pode retornar endereços roteados para o datacenter mais próximo. Uma melhoria adicional cria um mapa de todas as redes e suas localizações físicas aproximadas e fornece respostas de DNS com base nesse mapeamento. No entanto, essa solução tem o custo de ter uma implementação de servidor DNS muito mais complexa e manter um pipeline que manterá o mapeamento de local atualizado.

É claro que nenhuma dessas soluções é trivial, devido a uma característica fundamental do DNS: os usuários finais raramente conversam diretamente com servidores de nomes autorizados. Em vez disso, um servidor DNS recursivo geralmente fica em algum lugar entre os usuários finais e os servidores de nomes. Esse servidor faz proxy de consultas entre um usuário e um servidor e geralmente fornece uma camada de cache. O intermediário do DNS tem três implicações muito importantes no gerenciamento de tráfego:

- Resolução recursiva de endereços IP
- Caminhos de resposta não determinísticos •

Complicações adicionais de armazenamento em cache

A resolução recursiva de endereços IP é problemática, pois o endereço IP visto pelo servidor de nomes autoritativo não pertence a um usuário; em vez disso, é o resolvidor recursivo. Esta é uma limitação séria, porque só permite a otimização da resposta para a distância mais curta entre o resolvidor e o servidor de nomes. Uma possível solução é usar a extensão EDNS0 proposta em [Con15], que inclui informações sobre a sub-rede do cliente na consulta DNS enviada por um resolvidor recursivo. Dessa forma, um servidor de nomes autoritário retorna uma resposta que é ótima da perspectiva do usuário, em vez da perspectiva do resolvidor. Embora este ainda não seja o padrão oficial, suas vantagens óbvias levaram os maiores resolvidores de DNS (como OpenDNS e Google¹) a suportá-lo já.

Não só é difícil encontrar o endereço IP ideal para retornar ao servidor de nomes para a solicitação de um determinado usuário, mas esse servidor de nomes pode ser responsável por atender milhares ou milhões de usuários, em regiões que variam de um único escritório a um continente inteiro.

Por exemplo, um grande ISP nacional pode executar servidores de nomes para toda a sua rede a partir de um datacenter, mas ter interconexões de rede em cada área metropolitana. Os ISPs

¹ Consulte <https://groups.google.com/forum/#topic/public-dns-announce/67oxFjSLeUM>.

os servidores de nomes retornariam uma resposta com o endereço IP mais adequado para seu datacenter, apesar de haver melhores caminhos de rede para todos os usuários!

Por fim, os resolvidores recursivos geralmente armazenam em cache as respostas e encaminham essas respostas dentro dos limites indicados pelo campo TTL (tempo de vida útil) no registro DNS. O resultado final é que é difícil estimar o impacto de uma determinada resposta: uma única resposta autorizada pode atingir um único usuário ou vários milhares de usuários. Resolvemos este problema de duas maneiras:

- Analisamos as alterações de tráfego e atualizamos continuamente nossa lista de resolvidores de DNS conhecidos com o tamanho aproximado da base de usuários por trás de um determinado resolvedor, o que nos permite rastrear o impacto potencial de um determinado resolvedor.
- Estimamos a distribuição geográfica dos usuários por trás de cada resolvedor rastreado para aumentar a chance de direcionar esses usuários para o melhor local.

Estimar a distribuição geográfica é particularmente complicado se a base de usuários estiver distribuída em grandes regiões. Nesses casos, fazemos trocas para selecionar a melhor localização e otimizar a experiência para a maioria dos usuários.

Mas o que realmente significa “melhor localização” no contexto do balanceamento de carga DNS? A resposta mais óbvia é o local mais próximo do usuário. No entanto (como se determinar a localização dos usuários não fosse difícil por si só), existem critérios adicionais. O balanceador de carga DNS precisa garantir que o datacenter selecionado tenha capacidade suficiente para atender a solicitações de usuários que provavelmente receberão sua resposta. Ele também precisa saber que o datacenter selecionado e sua conectividade de rede estão em boas condições, porque direcionar as solicitações do usuário para um datacenter que está com problemas de energia ou de rede não é o ideal. Felizmente, podemos integrar o servidor DNS autoritativo com nossos sistemas de controle global que rastreiam o tráfego, a capacidade e o estado de nossa infraestrutura.

A terceira implicação do intermediário DNS está relacionada ao armazenamento em cache. Dado que servidores de nomes autorizados não podem liberar os caches dos resolvidores, os registros DNS precisam de um TTL relativamente baixo. Isso efetivamente define um limite inferior para a rapidez com que as alterações de DNS podem ser propagadas para os usuários.² Infelizmente, há pouco que podemos fazer além de manter isso em mente enquanto tomamos decisões de balanceamento de carga.

Apesar de todos esses problemas, o DNS ainda é a maneira mais simples e eficaz de equilibrar a carga antes mesmo de a conexão do usuário ser iniciada. Por outro lado, deve ficar claro que o balanceamento de carga com DNS por si só não é suficiente. Tenha em mente que todas as respostas DNS servidas devem caber dentro do limite de 512 bytes³ definido pela RFC 1035 [Moc87].

² Infelizmente, nem todos os resolvidores de DNS respeitam o valor TTL definido por servidores de nomes autorizados.

³ Caso contrário, os usuários devem estabelecer uma conexão TCP apenas para obter uma lista de endereços IP.

Esse limite define um limite superior para o número de endereços que podemos espremer em uma única resposta DNS, e esse número é quase certamente menor que o número de servidores.

Para realmente resolver o problema de平衡amento de carga de front-end, esse nível inicial de balanceamento de carga DNS deve ser seguido por um nível que aproveite os endereços IP virtuais.

Balanceamento de carga no endereço IP virtual

Os endereços IP virtuais (VIPs) não são atribuídos a nenhuma interface de rede específica.

Em vez disso, eles geralmente são compartilhados em vários dispositivos. No entanto, do ponto de vista do usuário, o VIP continua sendo um único endereço IP regular. Em teoria, essa prática permite ocultar detalhes de implementação (como o número de máquinas por trás de um determinado VIP) e facilita a manutenção, pois podemos agendar atualizações ou adicionar mais máquinas ao pool sem que o usuário saiba.

Na prática, a parte mais importante da implementação do VIP é um dispositivo chamado balanceador de carga de rede. O balanceador recebe os pacotes e os encaminha para uma das máquinas atrás do VIP. Esses back-ends podem processar ainda mais a solicitação.

Existem várias abordagens possíveis que o balanceador pode adotar para decidir qual back-end deve receber a solicitação. A primeira (e talvez a mais intuitiva) abordagem é sempre preferir o back-end menos carregado. Em teoria, essa abordagem deve resultar na melhor experiência do usuário final, pois as solicitações sempre são roteadas para a máquina menos ocupada. Infelizmente, essa lógica se quebra rapidamente no caso de protocolos com estado, que devem usar o mesmo backend durante uma requisição. Esse requisito significa que o balanceador deve acompanhar todas as conexões enviadas por ele para garantir que todos os pacotes subsequentes sejam enviados ao backend correto. A alternativa é usar algumas partes de um pacote para criar um ID de conexão (possivelmente usando uma função de hash e algumas informações do pacote) e usar o ID de conexão para selecionar um back-end. Por exemplo, o ID da conexão pode ser expresso como:

$$\text{id(packet)} \bmod N$$

onde id é uma função que recebe o pacote como entrada e produz um ID de conexão e N é o número de backends configurados.

Isso evita o estado de armazenamento e todos os pacotes pertencentes a uma única conexão são sempre encaminhados para o mesmo back-end. Sucesso? Ainda não. O que acontece se um back-end falhar e precisar ser removido da lista de back-end? De repente, N torna-se N-1 e, em seguida, $\text{id(packet)} \bmod N$ torna-se $\text{id(packet)} \bmod N-1$. Quase todos os pacotes de repente são mapeados para um backend diferente! Se os backends não compartilham nenhum estado entre si, esse remapeamento força um reset de quase todas as conexões existentes. Esse cenário definitivamente não é a melhor experiência do usuário, mesmo que esses eventos sejam pouco frequentes.

Felizmente, existe uma solução alternativa que não requer manter o estado de cada conexão na memória, mas não forçará todas as conexões a serem redefinidas quando um único

máquina cai: hash consistente. Proposto em 1997, o hash consistente [Kar97] descreve uma maneira de fornecer um algoritmo de mapeamento que permanece relativamente estável mesmo quando novos back-ends são adicionados ou removidos da lista. Essa abordagem minimiza a interrupção das conexões existentes quando o pool de back-ends é alterado. Como resultado, geralmente podemos usar o rastreamento de conexão simples, mas retornar ao hash consistente quando o sistema estiver sob pressão (por exemplo, durante um ataque de negação de serviço contínuo).

Voltando à questão maior: como exatamente um平衡ador de carga de rede deve encaminhar pacotes para um back-end VIP selecionado? Uma solução é realizar uma tradução de endereço de rede. No entanto, isso requer manter uma entrada de cada conexão na tabela de rastreamento, o que impede ter um mecanismo de fallback completamente sem estado.

Outra solução é modificar as informações da camada de enlace de dados (camada 2 do modelo de rede OSI). Ao alterar o endereço MAC de destino de um pacote encaminhado, o balanceador pode deixar todas as informações nas camadas superiores intactas, de modo que o back-end receba os endereços IP de origem e destino originais. O back-end pode então enviar uma resposta diretamente ao remetente original — uma técnica conhecida como Direct Server Response (DSR). Se as solicitações do usuário forem pequenas e as respostas forem grandes (por exemplo, a maioria das solicitações HTTP), o DSR oferece uma economia enorme, porque apenas uma pequena fração do tráfego precisa passar pelo balanceador de carga. Melhor ainda, o DSR não exige que mantenhamos o estado no dispositivo do balanceador de carga. Infelizmente, o uso da camada 2 para balanceamento de carga interno apresenta sérias desvantagens quando implantado em escala: todas as máquinas (ou seja, todos os平衡adores de carga e todos os seus back-ends) devem ser capazes de alcançar uns aos outros na camada de enlace de dados. Isso não é um problema se essa conectividade puder ser suportada pela rede e o número de máquinas não crescer excessivamente, porque todas as máquinas precisam residir em um único domínio de broadcast. Como você pode imaginar, o Google superou essa solução há algum tempo e teve que encontrar uma abordagem alternativa.

Nossa solução de balanceamento de carga VIP atual [Eis16] usa encapsulamento de pacotes. Um平衡ador de carga de rede coloca o pacote encaminhado em outro pacote IP com Encapsulamento de Roteamento Genérico (GRE) [Han94] e usa o endereço de um backend como destino. Um back-end que recebe o pacote retira a camada IP+GRE externa e processa o pacote IP interno como se fosse entregue diretamente à sua interface de rede. O平衡ador de carga de rede e o back-end não precisam mais existir no mesmo domínio de transmissão; eles podem até estar em continentes separados, desde que uma rota entre os dois existe.

O encapsulamento de pacotes é um mecanismo poderoso que oferece grande flexibilidade na maneira como nossas redes são projetadas e evoluem. Infelizmente, o encapsulamento também tem um preço: tamanho do pacote inflado. O encapsulamento introduz sobrecarga (24 bytes no caso de IPv4+GRE, para ser preciso), o que pode fazer com que o pacote exceda o tamanho da Unidade Máxima de Transmissão (MTU) disponível e exija fragmentação.

Quando o pacote chega ao datacenter, a fragmentação pode ser evitada usando uma MTU maior dentro do datacenter; no entanto, essa abordagem requer uma rede que ofereça suporte a grandes unidades de dados de protocolo. Tal como acontece com muitas coisas em escala, o balanceamento de carga parece simples na superfície – balanceamento de carga antecipado e balanceamento de carga com frequência – mas a dificuldade está nos detalhes, tanto para balanceamento de carga de front-end quanto para manipulação de pacotes quando eles chegam ao datacenter.

CAPÍTULO 20**Balanceamento de carga no datacenter**

Escrito por Alejandro Forero Cuervo
Editado por Sarah Chavis

Este capítulo se concentra no balanceamento de carga no datacenter. Especificamente, ele discute algoritmos para distribuição de trabalho em um determinado datacenter para um fluxo de consultas. Cobrimos as políticas no nível do aplicativo para rotear solicitações para servidores individuais que podem processá-las. Princípios de rede de nível inferior (por exemplo, switches, roteamento de pacotes) e seleção de datacenter estão fora do escopo deste capítulo.

Suponha que haja um fluxo de consultas chegando ao datacenter – elas podem vir do próprio datacenter, datacenters remotos ou uma mistura de ambos – a uma taxa que não excede os recursos que o datacenter tem para processá-los (ou apenas o excede por períodos de tempo muito curtos). Assuma também que existem serviços dentro do datacenter, contra os quais essas consultas operam. Esses serviços são implementados como muitos processos de servidor homogêneos e intercambiáveis, executados principalmente em máquinas diferentes. Os serviços menores normalmente têm pelo menos três desses processos (usar menos processos significa perder 50% ou mais de sua capacidade se você perder uma única máquina) e o maior pode ter mais de 10.000 processos (dependendo do tamanho do datacenter). No caso típico, os serviços são compostos por entre 100 e 1.000 processos. Chamamos esses processos de tarefas de back-end (ou apenas back-ends). Outras tarefas, conhecidas como tarefas de cliente, mantêm conexões com as tarefas de back-end. Para cada consulta recebida, uma tarefa de cliente deve decidir qual tarefa de back-end deve lidar com a consulta. Os clientes se comunicam com os back-ends usando um protocolo implementado em uma combinação de TCP e UDP.

Devemos observar que os datacenters do Google abrigam um conjunto muito diversificado de serviços que implementam diferentes combinações das políticas discutidas neste capítulo. Nosso exemplo de trabalho, como acabamos de descrever, não se encaixa em nenhum serviço diretamente. É um cenário generalizado que nos permite discutir as várias técnicas que consideraremos úteis para

nossos serviços. Algumas dessas técnicas podem ser mais (ou menos) aplicáveis a casos de uso específicos, mas essas técnicas foram projetadas e implementadas por vários engenheiros do Google ao longo de muitos anos.

Essas técnicas são aplicadas em muitas partes de nossa pilha. Por exemplo, a maioria das solicitações HTTP externas chegam ao GFE (Google Frontend), nosso sistema de proxy reverso HTTP. O GFE usa esses algoritmos, juntamente com os algoritmos descritos no [Capítulo 19](#), para rotear as cargas úteis e os metadados da solicitação para os processos individuais que executam os aplicativos que podem processar essas informações. Isso se baseia em uma configuração que mapeia vários padrões de URL para aplicativos individuais sob o controle de diferentes equipes. Para produzir os payloads de resposta (que eles retornam ao GFE, para serem devolvidos aos navegadores), esses aplicativos geralmente usam esses mesmos algoritmos, por sua vez, para se comunicar com a infraestrutura ou serviços complementares dos quais dependem. Às vezes, a pilha de dependências pode ficar relativamente profunda, onde uma única solicitação HTTP recebida pode desencadear uma longa cadeia transitiva de solicitações dependentes para vários sistemas, potencialmente com alta distribuição em vários pontos.

O Caso Ideal

Em um caso ideal, a carga de um determinado serviço é distribuída perfeitamente por todas as suas tarefas de back-end e, a qualquer momento, as tarefas de back-end menos e mais carregadas consomem exatamente a mesma quantidade de CPU.

Só podemos enviar tráfego para um datacenter até o ponto em que a tarefa mais carregada atinja seu limite de capacidade; isso está representado na [Figura 20-1](#) para dois cenários no mesmo intervalo de tempo. Durante esse período, o algoritmo de平衡amento de carga entre datacenters deve evitar o envio de tráfego adicional para o datacenter, pois isso corre o risco de sobrecarregar algumas tarefas.

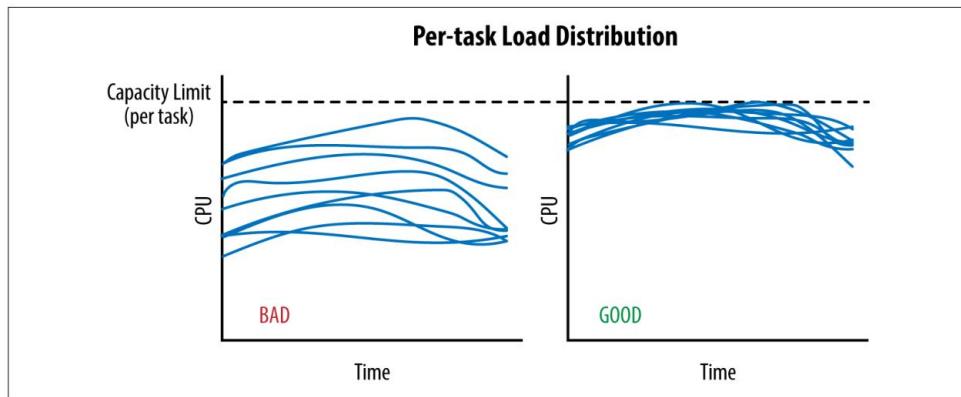


Figura 20-1. Dois cenários de distribuição de carga por tarefa ao longo do tempo

Conforme mostrado no gráfico à esquerda na [Figura 20-2](#), uma quantidade significativa de capacidade é desperdiçada: a capacidade ociosa de cada tarefa, exceto a tarefa mais carregada.

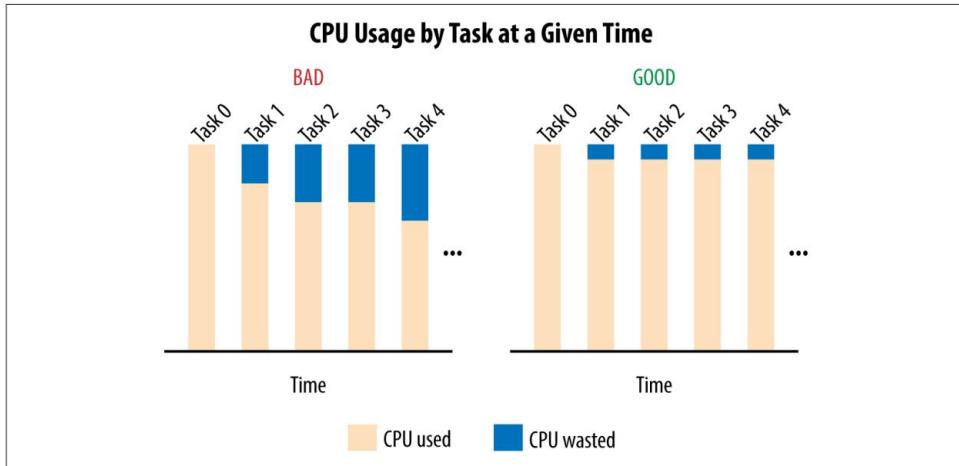


Figura 20-2. Histograma de CPU usado e desperdiçado em dois cenários

Mais formalmente, seja $CPU[i]$ a taxa de CPU consumida pela tarefa i em um determinado ponto de tempo e suponha que a tarefa 0 seja a tarefa mais carregada. Então, no caso de um grande spread, estamos desperdiçando a soma das diferenças na CPU de qualquer tarefa para $CPU[0]$: ou seja, a soma sobre todas as tarefas i de $(CPU[0] - CPU[i])$ será desperdiçado. Neste caso, “desperdiçado” significa reservado, mas não utilizado.

Este exemplo ilustra como práticas ruins de平衡amento de carga no datacenter limitam artificialmente a disponibilidade de recursos: você pode estar reservando 1.000 CPUs para seu serviço em um determinado datacenter, mas não conseguir usar mais do que, digamos, 700 CPUs.

Identificando Tarefas Ruins: Controle de Fluxo e Patos Mancos

Antes de podermos decidir qual tarefa de back-end deve receber uma solicitação de cliente, precisamos identificar — e evitar — tarefas insalubres em nosso pool de back-ends.

Uma abordagem simples para tarefas insalubres: controle de fluxo

Suponha que nossas tarefas de cliente rastreiem o número de solicitações ativas que enviaram em cada conexão a uma tarefa de back-end. Quando essa contagem de solicitações ativas atinge um limite configurado, o cliente trata o back-end como não íntegro e não envia mais solicitações. Para a maioria dos back-ends, 100 é um limite razoável; no caso médio, as solicitações tendem a terminar rápido o suficiente para que seja muito raro que o número de solicitações ativas de um determinado cliente atinja esse limite em condições normais de operação. Essa forma (muito básica!) de controle de fluxo também serve como uma forma simplista de平衡amento de carga: se uma determinada tarefa de back-end

fica sobrecarregado e as solicitações começam a se acumular, os clientes evitam esse back-end e a carga de trabalho se espalha organicamente entre as outras tarefas de back-end.

Infelizmente, essa abordagem muito simplista apenas protege as tarefas de back-end contra formas muito extremas de sobrecarga e é muito fácil para os back-ends ficarem sobrecarregados bem antes que esse limite seja atingido. O inverso também é verdadeiro: em alguns casos, os clientes podem atingir esse limite quando seus back-ends ainda têm muitos recursos sobressalentes. Por exemplo, alguns back-ends podem ter solicitações de longa duração que proíbem respostas rápidas.

Vimos casos em que esse limite padrão falhou, fazendo com que todas as tarefas de back-end se tornassem inacessíveis, com solicitações bloqueadas nos clientes até atingirem o tempo limite e falharem.

Aumentar o limite de solicitações ativas pode evitar essa situação, mas não resolve o problema subjacente de saber se uma tarefa é realmente insalubre ou simplesmente lenta para responder.

Uma abordagem robusta para tarefas insalubres: estado de pato manco

Do ponto de vista do cliente, uma determinada tarefa de back-end pode estar em qualquer um dos seguintes estados:

Saudável

A tarefa de back-end foi inicializada corretamente e está processando solicitações.

Recusando conexões A

tarefa de back-end não responde. Isso pode acontecer porque a tarefa está iniciando ou desligando, ou porque o back-end está em um estado anormal (embora seja raro um back-end parar de escutar em sua porta se não estiver desligando).

Pato manco

A tarefa de back-end está escutando em sua porta e pode servir, mas está pedindo explicitamente aos clientes que parem de enviar solicitações.

Quando uma tarefa entra no estado de pato manco, ela transmite esse fato para todos os seus clientes ativos. Mas e os clientes inativos? Com a implementação de RPC do Google, clientes inativos (ou seja, clientes sem conexões TCP ativas) ainda enviam verificações periódicas de integridade do UDP. O resultado é que as informações do pato lame são propagadas rapidamente para todos os clientes - normalmente em 1 ou 2 RTT - independentemente de seu estado atual.

A principal vantagem de permitir que uma tarefa exista em um estado de pato lame quase operacional é que ele simplifica o desligamento limpo, o que evita servir erros para todas as solicitações desafortunadas que estavam ativas em tarefas de back-end que estão sendo desligadas. Desativar uma tarefa de back-end que tenha solicitações ativas sem atender a nenhum erro facilita pushes de código, atividades de manutenção ou falhas de máquina que podem exigir a reinicialização de todas as tarefas relacionadas. Esse desligamento seguiria estas etapas gerais:

1. O agendador de tarefas envia um sinal SIGTERM para a tarefa de backend.
2. A tarefa de back-end entra no estado de lame duck e solicita que seus clientes enviem novas solicitações para outras tarefas de back-end. Isso é feito através de uma chamada de API na implementação RPC que é explicitamente chamada no manipulador SIGTERM.
3. Qualquer solicitação em andamento iniciada antes da tarefa de back-end entrar no estado de lame duck (ou depois de entrar no estado de lame duck, mas antes de um cliente detectá-la) é executada normalmente.
4. À medida que as respostas fluem de volta para os clientes, o número de solicitações ativas no back-end diminui gradualmente para zero.
5. Após um intervalo configurado, a tarefa de backend é encerrada de forma limpa ou o agendador de tarefas a elimina. O intervalo deve ser definido com um valor grande o suficiente para que todas as solicitações típicas tenham tempo suficiente para serem concluídas. Esse valor depende do serviço, mas uma boa regra geral é entre 10s e 150s, dependendo da complexidade do cliente.

Essa estratégia também permite que um cliente estabeleça conexões com tarefas de backend enquanto executa procedimentos de inicialização potencialmente de longa duração (e, portanto, ainda não está pronto para começar a servir). As tarefas de back-end poderiam começar a escutar conexões somente quando estivessem prontas para servir, mas isso atrasaria a negociação das conexões desnecessariamente. Assim que a tarefa de back-end estiver pronta para começar a ser veiculada, ela sinalizará isso explicitamente para os clientes.

Limitando o pool de conexões com subconjuntos

Além do gerenciamento de integridade, outra consideração para balanceamento de carga é o subconjunto: limitar o conjunto de tarefas de back-end em potencial com as quais uma tarefa de cliente interage.

Cada cliente em nosso sistema RPC mantém um pool de conexões de longa duração com seus back-ends que ele usa para enviar novas solicitações. Essas conexões geralmente são estabelecidas no início, quando o cliente está iniciando e geralmente permanecem abertas, com solicitações fluindo por elas, até a morte do cliente. Um modelo alternativo seria estabelecer e encerrar uma conexão para cada solicitação, mas esse modelo tem custos significativos de recursos e latência. No caso de uma conexão que permanece inativa por muito tempo, nossa implementação de RPC possui uma otimização que alterna a conexão para um modo "inativo" barato onde, por exemplo, a frequência de verificações de integridade é reduzida e a conexão TCP subjacente é caíu em favor do UDP.

Cada conexão requer alguma memória e CPU (devido à verificação periódica de integridade) em ambas as extremidades. Embora essa sobrecarga seja pequena em teoria, ela pode se tornar rapidamente significativa quando ocorre em muitas máquinas. A subconfiguração evita a situação em que um único cliente se conecta a um número muito grande de tarefas de back-end ou uma única tarefa de back-end recebe conexões de um número muito grande de tarefas de cliente. Em ambos os casos, você potencialmente desperdiça uma quantidade muito grande de recursos para um ganho muito pequeno.

Escolhendo o subconjunto certo

Escolher o subconjunto certo se resume a escolher a quantas tarefas de back-end cada cliente se conecta — o tamanho do subconjunto — e o algoritmo de seleção. Normalmente, usamos um tamanho de subconjunto de 20 a 100 tarefas de back-end, mas o tamanho de subconjunto “certo” para um sistema depende muito do comportamento típico de seu serviço. Por exemplo, você pode querer usar um tamanho de subconjunto maior se:

- O número de clientes é significativamente menor que o número de back-ends. Nesse caso, você deseja que o número de back-ends por cliente seja grande o suficiente para que você não acabe com tarefas de back-end que nunca receberão tráfego.
- Há desequilíbrios de carga frequentes nas tarefas do cliente (ou seja, uma tarefa do cliente envia mais solicitações do que outras). Esse cenário é típico em situações em que os clientes ocasionalmente enviam rajadas de solicitações. Nesse caso, os próprios clientes recebem solicitações de outros clientes que ocasionalmente têm um grande fan-out (por exemplo, “ler todas as informações de todos os seguidores de um determinado usuário”). Como uma explosão de solicitações será concentrada no subconjunto atribuído ao cliente, você precisa de um tamanho de subconjunto maior para garantir que a carga seja distribuída uniformemente pelo conjunto maior de tarefas de back-end disponíveis.

Uma vez que o tamanho do subconjunto é determinado, precisamos de um algoritmo para definir o subconjunto de tarefas de back-end que cada tarefa do cliente usará. Isso pode parecer uma tarefa simples, mas se torna complexa rapidamente ao trabalhar com sistemas de grande escala, onde o provisionamento eficiente é crucial e as reinicializações do sistema são garantidas.

O algoritmo de seleção para clientes deve atribuir back-ends uniformemente para otimizar o provisionamento de recursos. Por exemplo, se a subconjunto sobrecarregar um back-end em 10%, todo o conjunto de back-ends precisará ser superprovisionado em 10%. O algoritmo também deve lidar com reinicializações e falhas de maneira elegante e robusta, continuando a carregar back-ends da maneira mais uniforme possível, minimizando a rotatividade. Nesse caso, “churn” refere-se à seleção de substituição de back-end. Por exemplo, quando uma tarefa de back-end fica indisponível, seus clientes podem precisar escolher temporariamente um back-end substituto. Quando um back-end de substituição é selecionado, os clientes devem criar novas conexões TCP (e provavelmente realizar uma negociação no nível do aplicativo), o que cria sobrecarga adicional. Da mesma forma, quando uma tarefa de cliente é reiniciada, ela precisa reabrir as conexões com todos os seus back-ends.

O algoritmo também deve lidar com redimensionamentos no número de clientes e/ou número de backends, com churn de conexão mínimo e sem conhecer esses números antecipadamente. Essa funcionalidade é particularmente importante (e complicada) quando todo o conjunto de tarefas de cliente ou back-end é reiniciado uma de cada vez (por exemplo, para enviar uma nova versão). À medida que os back-ends são enviados, queremos que os clientes continuem atendendo, de forma transparente, com a menor perda de conexão possível.

Um algoritmo de seleção de subconjunto: subconjunto aleatório

Uma implementação ingênuas de um algoritmo de seleção de subconjunto pode fazer com que cada cliente embaralhe aleatoriamente a lista de back-ends uma vez e preencha seu subconjunto selecionando back-ends resolvíveis/saudáveis da lista. Embaralhando uma vez e, em seguida, escolhendo back-ends desde o início da lista trata de reinicializações e falhas de forma robusta (por exemplo, com relativamente pouca rotatividade) porque explicitamente os limita de consideração. No entanto, descobrimos que essa estratégia funciona muito mal na maioria dos cenários práticos porque distribui a carga de maneira muito desigual.

Durante o trabalho inicial de balanceamento de carga, implementamos subconjuntos aleatórios e calculamos a carga esperada para vários casos. Como exemplo, considere:

- 300 clientes
- 300 back-ends
- Um tamanho de subconjunto de 30% (cada cliente se conecta a 90 back-ends)

Como mostra a [Figura 20-3](#), o backend menos carregado tem apenas 63% da carga média (57 conexões, onde a média é de 90 conexões) e o mais carregado tem 121% (109 conexões). Na maioria dos casos, um tamanho de subconjunto de 30% já é maior do que gostaríamos de usar na prática. A distribuição de carga calculada muda toda vez que executamos a simulação enquanto o padrão geral permanece.

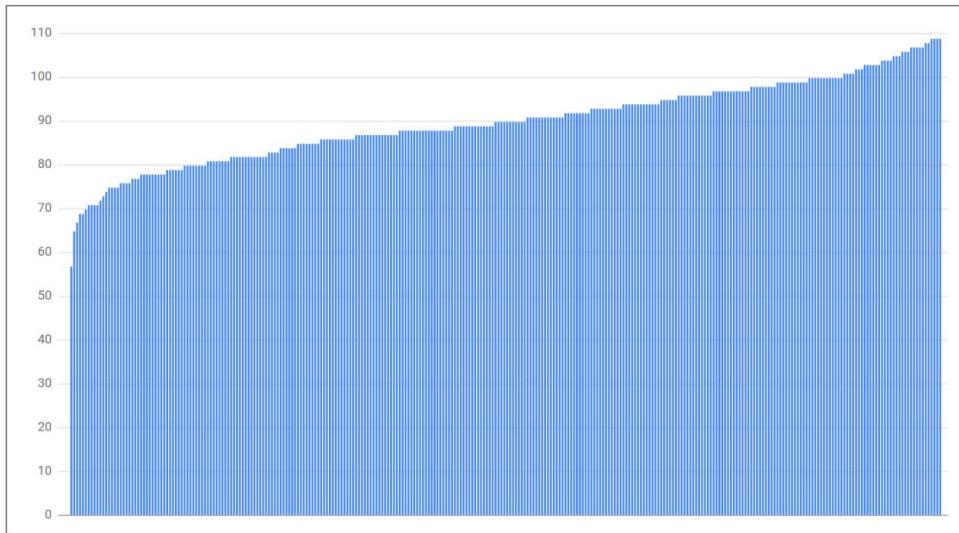


Figura 20-3. Distribuição de conexão com 300 clientes, 300 back-ends e um tamanho de subconjunto de 30%

Infelizmente, tamanhos de subconjuntos menores levam a desequilíbrios ainda piores. Por exemplo, a [Figura 20-4](#) mostra os resultados se o tamanho do subconjunto for reduzido para 10% (30 back-ends por cliente). Nesse caso, o backend menos carregado recebe 50% da carga média (15 conexões) e o mais carregado recebe 150% (45 conexões).

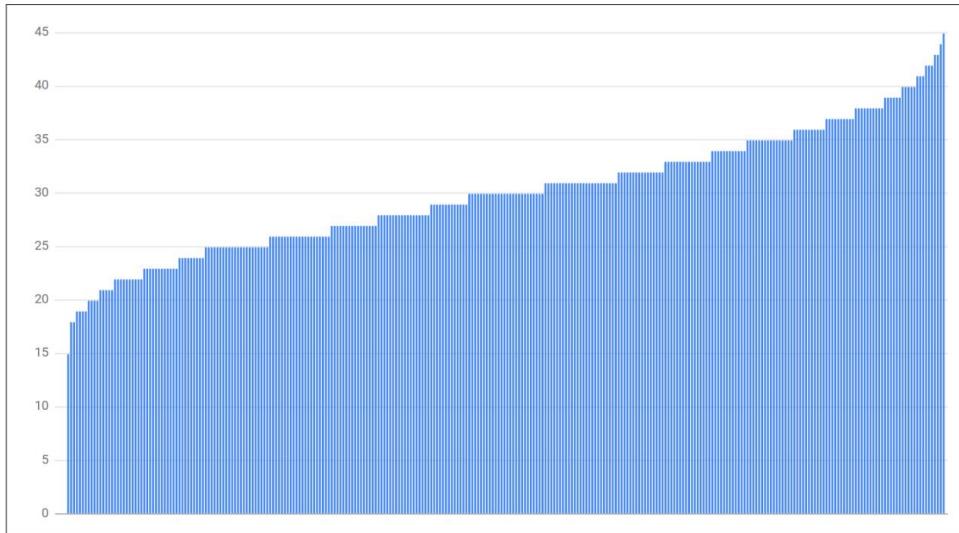


Figura 20-4. Distribuição de conexão com 300 clientes, 300 back-ends e um tamanho de subconjunto de 10%

Concluímos que, para que o subconjunto aleatório distribua a carga de maneira relativamente uniforme em todas as tarefas disponíveis, precisaríamos de tamanhos de subconjunto de até 75%. Um subconjunto tão grande é simplesmente impraticável; a variação no número de clientes que se conectam a uma tarefa é muito grande para considerar a subconjunto aleatório de uma boa política de seleção de subconjunto em escala.

Um algoritmo de seleção de subconjunto: subconjunto determinístico

A solução

do Google para as limitações de subconjunto aleatório é o subconjunto determinístico.

O código a seguir implementa esse algoritmo, descrito em detalhes a seguir:

```
def Subset(backends, client_id, subset_size):
    subset_count = len(backends) / subset_size

    # Agrupar clientes em rodadas; cada rodada usa a mesma lista embaralhada:
    rodada = client_id / subset_count random.seed(round) random.shuffle(backends)

    # O id do subconjunto correspondente ao cliente atual: subset_id
    = client_id % subset_count

    start = subset_id * subset_size return
    backends[start:start + subset_size]
```

Dividimos as tarefas do cliente em “rodadas”, onde a rodada i consiste em subconjunto_contagem de tarefas consecutivas do cliente, começando na tarefa subconjunto_contagem $\times i$, e subconjunto_contagem é o número de subconjuntos (ou seja, o número de tarefas de backend dividido pelo subconjunto desejado Tamanho). Dentro de cada rodada, cada backend é atribuído a exatamente um cliente (exceto possivelmente a última rodada, que pode não conter clientes suficientes, portanto, alguns backends podem não ser atribuídos).

Por exemplo, se tivermos 12 tarefas de backend [0, 11] e um tamanho de subconjunto desejado de 3, teremos rodadas contendo 4 clientes cada (`subset_count = 12/3`). Se tivéssemos 10 clientes, o algoritmo anterior poderia render as seguintes rodadas:

- Rodada 0: [0, 6, 3, 5, 1, 7, 11, 9, 2, 4, 8, 10] • Rodada 1: [8, 11, 4, 0, 5, 6, 10, 3, 2, 7, 9, 1]
- Rodada 2: [8, 3, 7, 2, 1, 4, 9, 10, 6, 5, 0, 11]

O ponto-chave a ser observado é que cada rodada apenas atribui cada backend em toda a lista a um cliente (exceto o último, onde ficamos sem clientes). Neste exemplo, cada back-end é atribuído a exatamente dois ou três clientes.

A lista deve ser embaralhada; caso contrário, os clientes recebem um grupo de tarefas de back-end consecutivas que podem ficar temporariamente indisponíveis (por exemplo, porque o trabalho de back-end está sendo atualizado gradualmente em ordem, da primeira à última tarefa). Rodadas diferentes usam uma semente diferente para embaralhar. Caso contrário, quando um back-end falha, a carga que estava recebendo é distribuída apenas entre os back-ends restantes em seu subconjunto. Se back-ends adicionais no subconjunto falharem, o efeito se compõe e a situação pode piorar significativamente rapidamente: se N back-ends em um subconjunto estiverem inativos, sua carga correspondente será distribuída pelos back-ends restantes ($subset_size - N$). Uma abordagem muito melhor é distribuir essa carga por todos os backends restantes usando um shuffle diferente para cada rodada.

Quando usamos um embaralhamento diferente para cada rodada, os clientes na mesma rodada começarão com a mesma lista embaralhada, mas os clientes nas rodadas terão listas embaralhadas diferentes.

A partir daqui, o algoritmo cria definições de subconjunto com base na lista embaralhada de back-ends e no tamanho de subconjunto desejado. Por exemplo:

- Subconjunto[0] = shuffled_backends[0] até shuffled_backends[2] • Subconjunto[1] = shuffled_backends[3] até shuffled_backends[5] • Subconjunto[2] = shuffled_backends[6] até shuffled_backends[8] • Subconjunto[3] = shuffled_backends[9] a shuffled_backends[11]

onde `shuffled_backend` é a lista embaralhada criada por cada cliente. Para atribuir um subconjunto a uma tarefa do cliente, basta pegar o subconjunto que corresponde à sua posição dentro de sua rodada (por exemplo, $(i \% 4)$ para `client[i]` com quatro subconjuntos):

- cliente[0], cliente[4], cliente[8] usará subconjunto[0]
- cliente[1], cliente[5], cliente[9] usará subconjunto[1]
- cliente[2], cliente[6], cliente[10] usará subconjunto[2]
- cliente[3], cliente[7], cliente[11] usará subconjunto[3]

Como os clientes nas rodadas usarão um valor diferente para shuffled_backends (e, portanto, para subconjunto) e os clientes nas rodadas usarão subconjuntos diferentes, a carga da conexão é distribuída uniformemente. Nos casos em que o número total de back-ends não é divisível pelo tamanho do subconjunto desejado, permitimos que alguns subconjuntos sejam ligeiramente maiores do que outros, mas na maioria dos casos o número de clientes atribuídos a um back-end diferirá em no máximo 1.

Como mostra a [Figura 20-5](#), a distribuição para o exemplo anterior de 300 clientes cada um se conectando a 10 de 300 back-ends produz resultados muito bons: cada back-end recebe exatamente o mesmo número de conexões.

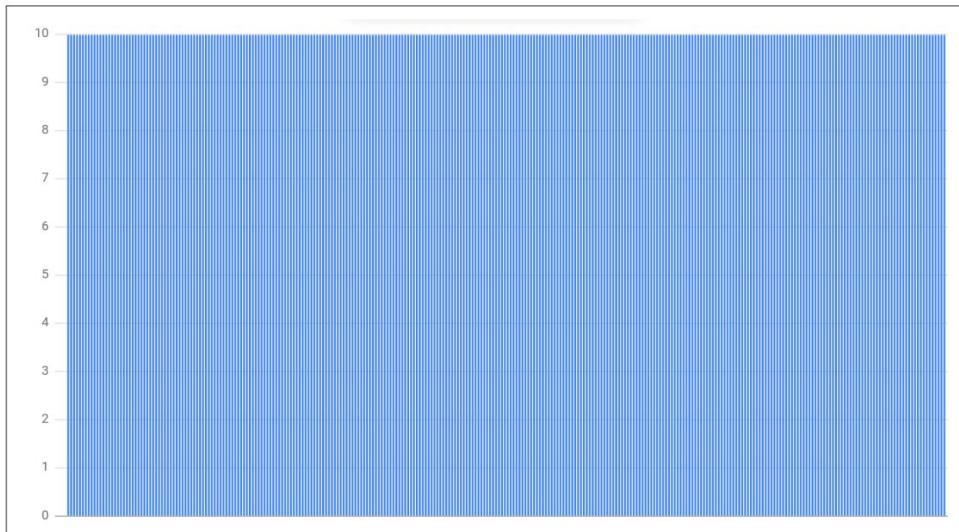


Figura 20-5. Distribuição de conexão com 300 clientes e subconjunto determinístico para 10 de 300 back-ends

Políticas de balanceamento de carga

Agora que estabelecemos as bases de como uma determinada tarefa de cliente mantém um conjunto de conexões que são conhecidas como íntegras, vamos examinar as políticas de平衡amento de carga. Esses são os mecanismos usados pelas tarefas do cliente para selecionar qual tarefa de backend em seu subconjunto recebe uma solicitação do cliente. Muitas das complexidades nas políticas de balanceamento de carga decorrem da natureza distribuída do processo de tomada de decisão no qual os clientes precisam

decidir, em tempo real (e apenas com informações parciais e/ou obsoletas do estado do back-end), qual back-end deve ser usado para cada solicitação.

As políticas de balanceamento de carga podem ser muito simples e não levar em conta nenhuma informação sobre o estado dos backends (por exemplo, Round Robin) ou podem atuar com mais informações sobre os backends (por exemplo, Round Robin Least-Loaded ou Round Robin Ponderado).

Round Robin Simples Uma

abordagem muito simples para balanceamento de carga faz com que cada cliente envie solicitações em modo round robin para cada tarefa de back-end em seu subconjunto ao qual ele pode se conectar com sucesso e que não está no estado de lame duck. Por muitos anos, essa foi nossa abordagem mais comum e ainda é usada por muitos serviços.

Infelizmente, embora o Round Robin tenha a vantagem de ser muito simples e ter um desempenho significativamente melhor do que apenas selecionar tarefas de back-end aleatoriamente, os resultados dessa política podem ser muito ruins. Embora os números reais dependam de muitos fatores, como custo variável de consulta e diversidade de máquina, descobrimos que o Round Robin pode resultar em uma distribuição de até 2x no consumo de CPU da tarefa menos carregada para a mais carregada.

Esse spread é extremamente dispendioso e ocorre por vários motivos, incluindo:

- Subconjunto pequeno
- Custos de consulta
- variáveis • Diversidade de máquinas
- Fatores de desempenho imprevisíveis

Subconjuntos

pequenos Uma das razões mais simples pelas quais o Round Robin distribui a carga mal é que todos os seus clientes podem não emitir requisições na mesma taxa. Diferentes taxas de solicitações entre clientes são especialmente prováveis quando processos muito diferentes compartilham os mesmos back-ends.

Nesse caso, e especialmente se você estiver usando tamanhos de subconjuntos relativamente pequenos, os backends nos subconjuntos dos clientes que geram mais tráfego naturalmente tenderão a ser mais carregados.

Custos de consulta

variáveis Muitos serviços lidam com solicitações que exigem quantidades muito diferentes de recursos para processamento. Na prática, descobrimos que a semântica de muitos serviços no Google é tal que as solicitações mais caras consomem 1.000 vezes (ou mais) CPU do que as solicitações mais baratas. O balanceamento de carga usando Round Robin é ainda mais difícil quando o custo da consulta não pode ser previsto com antecedência. Por exemplo, uma consulta como "devolver todos os e-mails recebidos pelo usuário XYZ no último dia" pode ser muito barata (se o usuário recebeu poucos e-mails ao longo do dia) ou extremamente cara.

O balanceamento de carga em um sistema com grandes discrepâncias no custo potencial da consulta é muito problemático. Pode ser necessário ajustar as interfaces de serviço para limitar funcionalmente a quantidade de trabalho realizado por solicitação. Por exemplo, no caso da consulta de e-mail descrita anteriormente, você pode introduzir uma interface de paginação e alterar a semântica da solicitação para “retornar os 100 e-mails mais recentes (ou menos) recebidos pelo usuário XYZ no último dia”. Infelizmente, muitas vezes é difícil introduzir essas mudanças semânticas. Isso não apenas requer alterações em todo o código do cliente, mas também envolve considerações adicionais de consistência. Por exemplo, o usuário pode estar recebendo novos e-mails ou excluindo e-mails enquanto o cliente busca e-mails página por página. Para este caso de uso, um cliente que itera ingenuamente pelos resultados e concatena as respostas (em vez de paginar com base em uma visualização fixa dos dados) provavelmente produzirá uma visualização inconsistente, repetindo algumas mensagens e/ou ignorando outras.

Para manter as interfaces (e suas implementações) simples, os serviços geralmente são definidos para permitir que as solicitações mais caras consumam 100, 1.000 ou até 10.000 vezes mais recursos do que as solicitações mais baratas. No entanto, os requisitos de recursos variados por solicitação naturalmente significam que algumas tarefas de back-end terão azar e, ocasionalmente, receberão solicitações mais caras do que outras. A extensão em que essa situação afeta o balanceamento de carga depende de quão caras são as solicitações mais caras. Por exemplo, para um de nossos backends Java, as consultas consomem cerca de 15 ms de CPU em média, mas algumas consultas podem facilmente exigir até 10 segundos. Cada tarefa neste backend reserva vários núcleos de CPU, o que reduz a latência, permitindo que alguns dos cálculos ocorram em paralelo. Mas, apesar desses núcleos reservados, quando um back-end recebe uma dessas grandes consultas, sua carga aumenta significativamente por alguns segundos. Uma tarefa mal comportada pode ficar sem memória ou até mesmo parar de responder completamente (por exemplo, devido ao thrashing de memória), mas mesmo no caso normal (ou seja, o backend tem recursos suficientes e sua carga normaliza quando a consulta grande é concluída), a latência de outras solicitações sofre devido à competição de recursos com a solicitação cara.

Diversidade de

máquinas Outro desafio para o Simple Round Robin é o fato de que nem todas as máquinas no mesmo datacenter são necessariamente as mesmas. Um determinado datacenter pode ter máquinas com CPUs de desempenho variável e, portanto, a mesma solicitação pode representar uma quantidade de trabalho significativamente diferente para máquinas diferentes.

Lidar com a diversidade de máquinas – sem exigir homogeneidade estrita – foi um desafio por muitos anos no Google. Em teoria, a solução para trabalhar com capacidade heterogênea de recursos em uma frota é simples: dimensionar as reservas de CPU dependendo do tipo de processador/máquina. No entanto, na prática, a implementação dessa solução exigia um esforço significativo porque exigia que nosso agendador de tarefas levasse em conta as equivalências de recursos com base no desempenho médio da máquina em uma amostra de serviços. Por exemplo, 2 unidades de CPU na máquina X (uma máquina “lenta”) é equivalente a 0,8 unidades de CPU na máquina Y (uma máquina “rápida”). Com essas informações, o agendador de tarefas é então

necessário para ajustar as reservas de CPU para um processo com base no fator de equivalência e no tipo de máquina na qual o processo foi agendado. Na tentativa de mitigar essa complexidade, criamos uma unidade virtual para taxa de CPU chamada "GCU" (Google Compute Units). As GCUs tornaram-se o padrão para modelagem de taxas de CPU e foram usadas para manter um mapeamento de cada arquitetura de CPU em nossos datacenters para sua GCU correspondente com base em seu desempenho.

Fatores de desempenho imprevisíveis

Talvez o maior fator complicador para o Simple Round Robin seja que as máquinas - ou, mais precisamente, o desempenho das tarefas de back-end - podem diferir muito devido a vários aspectos imprevisíveis que não podem ser contabilizados estaticamente.

Dois dos muitos fatores imprevisíveis que contribuem para o desempenho incluem:

Vizinhos antagônicos

Outros processos (geralmente não relacionados e executados por equipes diferentes) podem ter um impacto significativo no desempenho de seus processos. Vimos diferenças de desempenho dessa natureza de até 20%. Essa diferença decorre principalmente da competição por recursos compartilhados, como espaço em caches de memória ou largura de banda, de maneiras que podem não ser diretamente óbvias. Por exemplo, se a latência de solicitações de saída de uma tarefa de backend aumentar (por causa da competição por recursos de rede com um vizinho antagônico), o número de solicitações ativas também aumentará, o que pode desencadear um aumento na coleta de lixo.

A tarefa é reiniciada

Quando uma tarefa é reiniciada, geralmente requer muito mais recursos por alguns minutos. Como apenas um exemplo, vimos essa condição afetar plataformas como Java que otimizam o código dinamicamente mais do que outras. Em resposta, na verdade adicionamos à lógica de algum código de servidor — mantemos os servidores no estado de pato manco e os pré-aquecemos (acionando essas otimizações) por um período de tempo após o início, até que seu desempenho seja nominal. O efeito de reinicializações de tarefas pode se tornar um problema considerável quando você considera que atualizamos muitos servidores (por exemplo, enviar novas compilações, o que requer a reinicialização dessas tarefas) todos os dias.

Se sua política de balanceamento de carga não puder se adaptar a limitações de desempenho imprevistas, você acabará inherentemente com uma distribuição de carga abaixo do ideal ao trabalhar em escala.

Round Robin menos carregado

Uma abordagem alternativa ao Simple Round Robin é fazer com que cada tarefa do cliente acompanhe o número de solicitações ativas que possui para cada tarefa de back-end em seu subconjunto e use o Round Robin entre o conjunto de tarefas com um número mínimo de solicitações ativas.

Por exemplo, suponha que um cliente use um subconjunto de tarefas de back-end t0 a t9 e atualmente tenha o seguinte número de solicitações ativas em cada back-end:

t0 t1 t2 t3 t4 t5 t6 t7 t8 t9

2 1 0 0 1 0 2 0 0 1

Para uma nova solicitação, o cliente filtraria a lista de possíveis tarefas de back-end apenas para aquelas tarefas com o menor número de conexões (t2, t3, t5, t7 e t8) e escolheria um back-end dessa lista.

Vamos supor que ele escolha t2. A tabela de estado de conexão do cliente agora teria a seguinte aparência:

t0 t1 t2 t3 t4 t5 t6 t7 t8 t9

2 1 1 0 1 0 2 0 0 1

Supondo que nenhuma das solicitações atuais tenha sido concluída, na próxima solicitação, o pool de candidatos de back-end se torna t3, t5, t7 e t8.

Vamos avançar rapidamente até emitirmos quatro novas solicitações. Ainda supondo que nenhuma solicitação seja concluída nesse meio tempo, a tabela de estado de conexão teria a seguinte aparência:

t0 t1 t2 t3 t4 t5 t6 t7 t8 t9

2 1 1 1 1 1 2 1 1 1

Neste ponto, o conjunto de candidatos de back-end são todas as tarefas, exceto t0 e t6. No entanto, se a solicitação em relação à tarefa t4 for concluída, seu estado atual se tornará "0 solicitações ativas" e uma nova solicitação será atribuída a t4.

Essa implementação, na verdade, usa Round Robin, mas é aplicada em todo o conjunto de tarefas com solicitações ativas mínimas. Sem essa filtragem, a política pode não conseguir distribuir as solicitações o suficiente para evitar uma situação em que parte das tarefas de back-end disponíveis não sejam utilizadas. A ideia por trás da política menos carregada é que as tarefas carregadas tenderão a ter maior latência do que aquelas com capacidade ociosa, e essa estratégia naturalmente tirará a carga dessas tarefas carregadas.

Dito tudo isso, aprendemos (da maneira mais difícil!) sobre uma armadilha muito perigosa da abordagem Round Robin menos carregada: se uma tarefa não estiver muito saudável, ela pode começar a apresentar 100% de erros. Dependendo da natureza desses erros, eles podem ter latência muito baixa; muitas vezes é significativamente mais rápido apenas retornar um "Não estou saudável!" erro do que realmente processar uma solicitação. Como resultado, os clientes podem começar a enviar uma quantidade muito grande de tráfego para a tarefa não íntegra, pensando erroneamente que a tarefa está disponível, em vez de falhá-la rapidamente! Dizemos que a tarefa insalubre agora está afundando o tráfego. Felizmente, essa armadilha pode ser resolvida com relativa facilidade modificando a política para contar erros recentes como se fossem solicitações ativas. Dessa forma, se uma tarefa de back-end

torna-se insalubre, a política de balanceamento de carga começa a desviar a carga da mesma forma que desviaria a carga de uma tarefa sobrecarregada.

O Round Robin menos carregado tem duas limitações importantes:

A contagem de solicitações ativas pode não ser um proxy muito bom para a capacidade de um determinado back-end. Muitas solicitações passam uma parte significativa de sua vida apenas

aguardando uma resposta da rede (ou seja, aguardando respostas para solicitações iniciadas em outros back-ends) e muito pouco tempo no processamento real. Por exemplo, uma tarefa de back-end pode processar duas vezes mais solicitações que outra (por exemplo, porque está sendo executada em uma máquina com uma CPU duas vezes mais rápida que o restante), mas a latência de suas solicitações ainda pode ser aproximadamente a mesma como a latência das requisições na outra tarefa (porque as requisições passam a maior parte de sua vida apenas esperando que a rede responda). Nesse caso, como o bloqueio de E/S geralmente consome zero de CPU, muito pouca RAM e nenhuma largura de banda, ainda desejariamos enviar o dobro de solicitações para o back-end mais rápido. No entanto, o Round Robin Least-Loaded considerará ambas as tarefas de back-end igualmente carregadas.

A contagem de solicitações ativas em cada cliente não inclui solicitações de outros clientes para os mesmos backends. Ou seja, cada tarefa de cliente tem apenas uma visão muito limitada do estado de suas tarefas de backend: a visão de suas próprias solicitações.

Na prática, descobrimos que grandes serviços usando o Round Robin menos carregado verão sua tarefa de backend mais carregada usando duas vezes mais CPU do que a menos carregada, tendo um desempenho tão ruim quanto o Round Robin.

Round Robin ponderado Round

Robin ponderado é uma importante política de平衡amento de carga que melhora o Round Robin Simples e Menos Carregado, incorporando informações fornecidas pelo backend no processo de decisão.

O Round Robin ponderado é bastante simples em princípio: cada tarefa do cliente mantém uma pontuação de "capacidade" para cada backend em seu subconjunto. As solicitações são distribuídas no modo Round-Robin, mas os clientes pesam as distribuições de solicitações para back-ends proporcionalmente. Em cada resposta (incluindo respostas a verificações de integridade), os backends incluem as taxas observadas atuais de consultas e erros por segundo, além da utilização (normalmente, uso da CPU). Os clientes ajustam as pontuações de capacidade periodicamente para escolher tarefas de back-end com base em seu número atual de solicitações bem-sucedidas tratadas e em qual custo de utilização; solicitações com falha resultam em uma penalidade que afeta decisões futuras.

Na prática, o Round Robin ponderado funcionou muito bem e reduziu significativamente a diferença entre as tarefas mais e menos utilizadas. A [Figura 20-6](#) mostra as taxas de CPU para um subconjunto aleatório de tarefas de back-end no momento em que seus clientes trocaram

de Round Robin menos carregado a ponderado. O spread das tarefas menos carregadas para as tarefas mais carregadas diminuiu drasticamente.

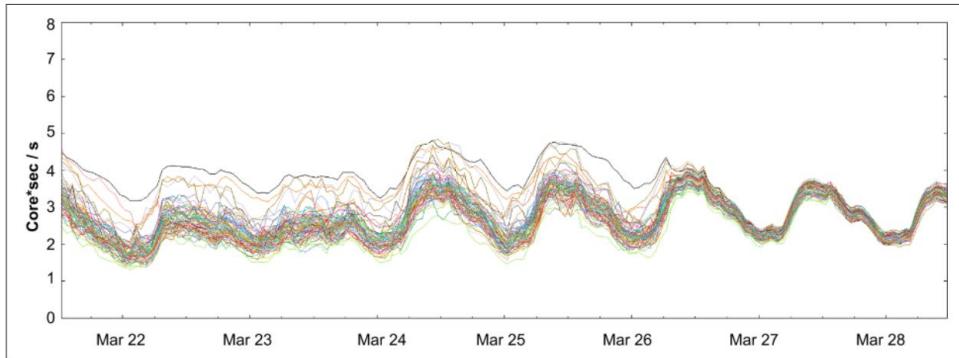


Figura 20-6. Distribuição da CPU antes e depois de habilitar o Round Robin ponderado

CAPÍTULO 21

Como lidar com sobrecarga

**Escrito por Alejandro Forero Cuervo
Editado por Sarah Chavis**

Evitar a sobrecarga é um objetivo das políticas de平衡amento de carga. Mas não importa quão eficiente seja sua política de balanceamento de carga, eventualmente alguma parte do seu sistema ficará sobrecarregada. O tratamento adequado das condições de sobrecarga é fundamental para a execução de um sistema de atendimento confiável.

Uma opção para lidar com a sobrecarga é fornecer respostas degradadas: respostas que não são tão precisas ou que contêm menos dados do que as respostas normais, mas que são mais fáceis de calcular. Por exemplo:

- Em vez de pesquisar um corpus inteiro para fornecer os melhores resultados disponíveis para um consulta de pesquisa, pesquise apenas uma pequena porcentagem do conjunto de candidatos.
- Confie em uma cópia local dos resultados que podem não estar totalmente atualizados, mas que serão mais barato de usar do que ir contra o armazenamento canônico.

No entanto, sob sobrecarga extrema, o serviço pode nem mesmo ser capaz de calcular e fornecer respostas degradadas. Neste ponto, pode não ter outra opção imediata a não ser veicular erros. Uma maneira de mitigar esse cenário é equilibrar o tráfego entre os datacenters de forma que nenhum datacenter receba mais tráfego do que tem capacidade de processar. Por exemplo, se um datacenter executar 100 tarefas de backend e cada tarefa puder processar até 500 solicitações por segundo, o algoritmo de balanceamento de carga não permitirá que mais de 50.000 consultas por segundo sejam enviadas a esse datacenter. No entanto, mesmo essa restrição pode ser insuficiente para evitar sobrecarga quando você está operando em escala. No final das contas, é melhor construir clientes e back-ends para lidar com restrições de recursos normalmente: redirecionar quando possível, fornecer resultados degradados quando necessário e lidar com erros de recursos de forma transparente quando tudo mais falhar.

As armadilhas das “consultas por segundo”

Consultas diferentes podem ter requisitos de recursos muito diferentes. O custo de uma consulta pode variar com base em fatores arbitrários, como o código no cliente que a emite (para serviços que têm muitos clientes diferentes) ou até mesmo a hora do dia (por exemplo, usuários domésticos versus usuários de trabalho; ou tráfego interativo do usuário final versus tráfego em lote).

Aprendemos esta lição da maneira mais difícil: modelar a capacidade como “consultas por segundo” ou usar recursos estáticos das solicitações que se acredita serem um proxy para os recursos que consomem (por exemplo, “quantas chaves as solicitações estão lendo”) geralmente torna para uma métrica ruim. Mesmo que essas métricas tenham um desempenho adequado em um determinado momento, os índices podem mudar. Às vezes, a mudança é gradual, mas às vezes a mudança é drástica (por exemplo, uma nova versão do software de repente fez com que alguns recursos de algumas solicitações exigissem significativamente menos recursos). Um destino móvel é uma métrica ruim para projetar e implementar o balanceamento de carga.

Uma solução melhor é medir a capacidade diretamente nos recursos disponíveis. Por exemplo, você pode ter um total de 500 núcleos de CPU e 1 TB de memória reservados para um determinado serviço em um determinado datacenter. Naturalmente, funciona muito melhor usar esses números diretamente para modelar a capacidade de um datacenter. Costumamos falar sobre o custo de uma solicitação para se referir a uma medida normalizada de quanto tempo de CPU foi consumido (em diferentes arquiteturas de CPU, levando em consideração as diferenças de desempenho).

Na maioria dos casos (embora certamente não em todos), descobrimos que simplesmente usar O consumo de CPU como sinal para provisionamento funciona bem, pelos seguintes motivos filhos:

- Em plataformas com coleta de lixo, a pressão da memória se traduz naturalmente em maior consumo de CPU.
- Em outras plataformas, é possível provisionar os recursos restantes de forma que seja muito improvável que eles se esgotem antes que a CPU se esgote.

Nos casos em que o provisionamento excessivo de recursos não CPU é proibitivamente caro, levamos em consideração cada recurso do sistema separadamente ao considerar o consumo de recursos.

Limites por cliente

Um componente para lidar com sobrecarga é decidir o que fazer no caso de sobrecarga global. Em um mundo perfeito, onde as equipes coordenam seus lançamentos cuidadosamente com os proprietários de suas dependências de back-end, a sobrecarga global nunca acontece e os serviços de back-end sempre têm capacidade suficiente para atender seus clientes. Infelizmente, não vivemos em um mundo perfeito. Aqui, na realidade, a sobrecarga global ocorre com bastante frequência (especialmente para serviços internos que tendem a ter muitos clientes executados por muitas equipes).

Quando ocorre uma sobrecarga global, é vital que o serviço apenas forneça respostas de erro para clientes que se comportam mal, enquanto outros clientes permanecem inalterados. Para alcançar esse resultado, os proprietários de serviços provisionam sua capacidade com base no uso negociado com seus clientes e definem cotas por cliente de acordo com esses acordos.

Por exemplo, se um serviço de back-end tiver 10.000 CPUs alocadas em todo o mundo (em vários datacenters), seus limites por cliente podem ser parecidos com o seguinte:

- O Gmail pode consumir até 4.000 segundos de CPU por segundo.
- O calendário pode consumir até 4.000 segundos de CPU por segundo.
- O Android pode consumir até 3.000 segundos de CPU por segundo.
- O Google+ pode consumir até 2.000 segundos de CPU por segundo.
- Todos os outros usuários podem consumir até 500 segundos de CPU por segundo.

Observe que esses números podem somar mais de 10.000 CPUs alocadas ao serviço de back-end. O proprietário do serviço está confiando no fato de que é improvável que todos os seus clientes atinjam seus limites de recursos simultaneamente.

Agregamos informações de uso global em tempo real de todas as tarefas de back-end e usamos esses dados para aumentar os limites efetivos de tarefas de back-end individuais. Uma análise mais detalhada do sistema que implementa essa lógica está fora do escopo desta discussão, mas escrevemos um código significativo para implementar isso em nossas tarefas de back-end. Uma parte interessante do quebra-cabeça é calcular em tempo real a quantidade de recursos — especificamente CPU — consumidos por cada solicitação individual. Essa computação é particularmente complicada para servidores que não implementam um modelo de thread por solicitação, onde um pool de threads apenas executa diferentes partes de todas as solicitações à medida que elas chegam, usando APIs sem bloqueio.

Limitação do lado do cliente

Quando um cliente está fora da cota, uma tarefa de back-end deve rejeitar solicitações rapidamente com a expectativa de que retornar um erro "cliente está fora da cota" consome significativamente menos recursos do que realmente processar a solicitação e fornecer uma resposta correta. No entanto, essa lógica não é válida para todos os serviços. Por exemplo, é quase tão caro rejeitar uma solicitação que requer uma pesquisa simples de RAM (onde a sobrecarga do tratamento do protocolo de solicitação/resposta é significativamente maior do que a sobrecarga de produzir a resposta) quanto aceitar e executar essa solicitação.

E mesmo no caso em que a rejeição de solicitações economiza recursos significativos, essas solicitações ainda consomem alguns recursos. Se a quantidade de solicitações rejeitadas for significativa, esses números se somam rapidamente. Nesses casos, o backend pode ficar sobrecarregado mesmo que a grande maioria de sua CPU seja gasta apenas rejeitando solicitações!

A limitação do lado do cliente resolve esse problema.¹ Quando um cliente detecta que uma parte significativa de suas solicitações recentes foi rejeitada devido a erros "fora de cota", ele inicia a autorregulação e limita a quantidade de tráfego de saída que gera. Solicitações acima do limite falham localmente sem chegar à rede.

Implementamos a limitação do lado do cliente por meio de uma técnica que chamamos de limitação adaptativa. Especificamente, cada tarefa do cliente mantém as seguintes informações nos últimos dois minutos de seu histórico:

solicitações de

O número de solicitações tentadas pela camada de aplicativo (no cliente, em cima do sistema de limitação adaptável)

aceita

O número de solicitações aceitas pelo back-end

Em condições normais, os dois valores são iguais. À medida que o back-end começa a rejeitar o tráfego, o número de aceitações se torna menor que o número de solicitações. Os clientes podem continuar a emitir solicitações para o back-end até que as solicitações sejam K vezes maiores que as aceitas. Uma vez atingido esse ponto de corte, o cliente começa a se autorregular e novas solicitações são rejeitadas localmente (ou seja, no cliente) com a probabilidade calculada na [Equação 21-1](#).

Equação 21-1. Probabilidade de rejeição de solicitação do cliente

$$\max 0, \left(\frac{\text{pedidos} \times K}{\text{aceita}} \right)$$

À medida que o próprio cliente começa a rejeitar solicitações, as solicitações continuarão a exceder as aceitações. Embora possa parecer contra-intuitivo, dado que as solicitações rejeitadas localmente não são propagadas para o back-end, esse é o comportamento preferencial. À medida que a taxa na qual o aplicativo tenta solicitações ao cliente cresce (em relação à taxa na qual o back-end as aceita), queremos aumentar a probabilidade de descartar novas solicitações.

Descobrimos que a limitação adaptativa funciona bem na prática, levando a taxas estáveis de solicitações em geral. Mesmo em grandes situações de sobrecarga, os back-ends acabam rejeitando uma solicitação para cada solicitação que realmente processam. Uma grande vantagem dessa abordagem é que a decisão é tomada pela tarefa do cliente com base inteiramente em informações locais e usando uma implementação relativamente simples: não há dependências adicionais ou penalidades de latência.

Para serviços em que o custo de processamento de uma solicitação é muito próximo ao custo de rejeição dessa solicitação, permitindo que cerca de metade dos recursos de back-end sejam consumidos pela rejeição

¹ Por exemplo, veja [Porteiro](#), que fornece um sistema de limitação do lado do cliente distribuído e cooperativo.

pedidos de ted podem ser inaceitáveis. Nesse caso, a solução é simples: modifique o multiplicador de aceitações K (por exemplo, 2) na probabilidade de rejeição do pedido do cliente ([Equação 21-1](#)). Nesse caminho:

- Reduzir o multiplicador fará com que a limitação adaptativa se comporte de forma mais agressiva •

Aumentar o multiplicador fará com que a limitação adaptativa se comporte de forma menos agressiva

Por exemplo, em vez de fazer com que o cliente se autorregula quando solicitações = $2 * \text{aceita}$, faça com que ele se autorregula quando solicitações = $1,1 * \text{aceita}$. Reduzir o multiplicador para solicitações aceitadas pelo backend para cada 10 solicitações aceitas.

Geralmente preferimos o multiplicador 2x. Ao permitir que mais solicitações cheguem ao back-end do que o esperado, desperdiçamos mais recursos no back-end, mas também aceleramos a propagação do estado do back-end para os clientes. Por exemplo, se o backend decidir parar de rejeitar o tráfego das tarefas do cliente, o atraso até que todas as tarefas do cliente tenham detectado essa mudança de estado é menor.

Uma consideração adicional é que a limitação do lado do cliente pode não funcionar bem com clientes que enviam solicitações muito esporadicamente para seus back-ends. Nesse caso, a visão que cada cliente tem do estado do backend é reduzida drasticamente, e abordagens para incrementar essa visibilidade tendem a ser caras.

Criticamente

A criticidade é outra noção que achamos muito útil no contexto de cotas globais e limitação. Uma solicitação feita a um back-end é associada a um dos quatro possíveis valores de criticidade, dependendo de quão crítica consideramos essa solicitação:

CRITICAL_PLUS

Reservado para as solicitações mais críticas, aquelas que resultarão em sério impacto visível ao usuário se falharem.

CRÍTICO

O valor padrão para solicitações enviadas de trabalhos de produção. Essas solicitações resultarão em impacto visível ao usuário, mas o impacto pode ser menos grave do que os de CRITICAL_PLUS. Espera-se que os serviços forneçam capacidade suficiente para todo o tráfego CRITICAL e CRITICAL_PLUS esperado.

SCHEDDABLE_PLUS

Tráfego para o qual é esperada indisponibilidade parcial. Esse é o padrão para trabalhos em lote, que podem repetir as solicitações minutos ou até horas depois.

DERRAMÁVEL

Tráfego para o qual se espera indisponibilidade parcial frequente e indisponibilidade total ocasional.

Descobrimos que quatro valores eram suficientemente robustos para modelar quase todos os serviços. Tivemos várias discussões sobre propostas para agregar mais valores, porque isso nos permitiria classificar as solicitações com mais precisão. No entanto, definir valores adicionais exigiria mais recursos para operar vários sistemas com reconhecimento de criticidade.

Fizemos da criticidade uma noção de primeira classe do nosso sistema RPC e trabalhamos duro para integrá-la em muitos de nossos mecanismos de controle para que ela possa ser levada em consideração ao reagir a situações de sobrecarga. Por exemplo:

- Quando um cliente fica sem cota global, uma tarefa de back-end só rejeitará solicitações de uma determinada criticidade se já estiver rejeitando todas as solicitações de todas as críticas mais baixas (na verdade, os limites por cliente que nosso sistema suporta, descritos anteriormente). Ier, pode ser definido por criticidade).
- Quando uma tarefa está sobrecarregada, ela rejeitará solicitações de menor criticidade mais cedo. •

O sistema de limitação adaptável também mantém estatísticas separadas para cada criticidade.

A criticidade de uma solicitação é ortogonal aos seus requisitos de latência e, portanto, à qualidade de serviço (QoS) da rede subjacente usada. Por exemplo, quando um sistema exibe resultados de pesquisa ou sugestões enquanto o usuário está digitando uma consulta de pesquisa, as solicitações subjacentes são altamente descartáveis (se o sistema estiver sobrecarregado, é aceitável não exibir esses resultados), mas tendem a ter requisitos de latência rigorosos.

Também ampliamos significativamente nosso sistema RPC para propagar a criticidade automaticamente. Se um backend recebe a solicitação A e, como parte da execução dessa solicitação, emite a solicitação B e a solicitação C para outros backends, a solicitação B e a solicitação C usarão a mesma criticidade que a solicitação A por padrão.

No passado, muitos sistemas do Google desenvolveram suas próprias noções ad hoc de criticidade que muitas vezes eram incompatíveis entre os serviços. Ao padronizar e propagar a criticidade como parte de nosso sistema RPC, agora podemos definir a criticidade de forma consistente em pontos específicos. Isso significa que podemos ter certeza de que as dependências sobrecarregadas cumprirão a criticidade de alto nível desejada à medida que rejeitam o tráfego, independentemente da profundidade da pilha RPC em que estejam. Nossa prática é, portanto, definir a criticidade o mais próximo possível dos navegadores ou clientes móveis - normalmente nos frontends HTTP que produzem o HTML a ser retornado - e apenas substituir a criticidade em casos específicos em que faz sentido em pontos específicos da pilha.

Sinais de utilização

Nossa implementação de proteção de sobrecarga em nível de tarefa é baseada na noção de utilização. Em muitos casos, a utilização é apenas uma medida da taxa de CPU (ou seja, a taxa de CPU atual dividida pelo total de CPUs reservadas para a tarefa), mas em alguns casos também levamos em consideração medidas como a porção de memória reservada que está sendo usado atualmente. À medida que a utilização se aproxima dos limites configurados, começamos a rejeitar solicitações com base em sua criticidade (limites mais altos para criticidades mais altas).

Os sinais de utilização que usamos são baseados no estado local da tarefa (já que o objetivo dos sinais é proteger a tarefa) e temos implementações para vários sinais.

O sinal mais útil geralmente é baseado na “carga” no processo, que é determinada usando um sistema que chamamos de média de carga do executor.

Para encontrar a média de carga do executor, contamos o número de threads ativos no processo. Nesse caso, “ativo” refere-se a threads que estão atualmente em execução ou prontas para serem executadas e aguardando um processador livre. Suavizamos esse valor com decaimento exponencial e começamos a rejeitar solicitações à medida que o número de threads ativos cresce além do número de processadores disponíveis para a tarefa. Isso significa que uma solicitação de entrada que tem um fan-out muito grande (ou seja, uma que agenda uma rajada de um número muito grande de operações de curta duração) fará com que a carga aumente muito brevemente, mas a suavização irá principalmente engolir esse pico. No entanto, se as operações não forem de curta duração (ou seja, a carga aumentar e permanecer alta por um período significativo de tempo), a tarefa começará a rejeitar solicitações.

Embora a média de carga do executor tenha provado ser um sinal muito útil, nosso sistema pode conectar qualquer sinal de utilização que um determinado back-end possa precisar. Por exemplo, podemos usar a pressão de memória – que indica se o uso de memória em uma tarefa de back-end cresceu além dos parâmetros operacionais normais – como outro possível sinal de utilização. O sistema também pode ser configurado para combinar vários sinais e rejeitar solicitações que ultrapassem os limites de utilização combinados (ou individuais).

Lidando com Erros de Sobrecarga

Além de lidar com a carga normalmente, pensamos bastante em como os clientes devem reagir quando receberem uma resposta de erro relacionada à carga. No caso de erros de sobrecarga, distinguimos duas situações possíveis.

Um grande subconjunto de tarefas de back-end no datacenter está sobrecarregado.

Se o sistema de平衡amento de carga entre datacenters estiver funcionando perfeitamente (ou seja, ele pode propagar o estado e reagir instantaneamente a mudanças no tráfego), essa condição não ocorre.

Um pequeno subconjunto de tarefas de back-end no datacenter está sobrecarregado.

Essa situação geralmente é causada por imperfeições no balanceamento de carga dentro do datacenter. Por exemplo, uma tarefa pode ter recebido muito recentemente um pedido muito caro. Nesse caso, é muito provável que o datacenter tenha capacidade restante em outras tarefas para lidar com a solicitação.

Se um grande subconjunto de tarefas de back-end no datacenter estiver sobrecarregado, as solicitações não devem ser repetidas e os erros devem aparecer até o chamador (por exemplo, retornar um erro ao usuário final). É muito mais comum que apenas uma pequena parte das tarefas fique sobrecarregada e, nesse caso, a resposta preferencial é tentar novamente a solicitação imediatamente. Em geral, nosso sistema de balanceamento de carga entre datacenters tenta direcionar o tráfego dos clientes para os datacenters de back-end disponíveis mais próximos. Em alguns casos, o datacenter mais próximo está longe (por exemplo, um cliente pode ter seu back-end disponível mais próximo em um continente diferente), mas geralmente conseguimos situar os clientes perto de seus back-ends.

Dessa forma, a latência adicional de tentar novamente uma solicitação - apenas algumas viagens de ida e volta da rede - tende a ser insignificante.

Do ponto de vista de nossas políticas de balanceamento de carga, novas tentativas de requisições são indistinguíveis de novas requisições. Ou seja, não usamos nenhuma lógica explícita para garantir que uma nova tentativa vá para uma tarefa de back-end diferente; contamos apenas com a probabilidade provável de que a nova tentativa chegue a uma tarefa de back-end diferente simplesmente em virtude do número de back-ends participantes no subconjunto. Garantir que todas as tentativas realmente vão para uma tarefa diferente incorreria em mais complexidade em nossas APIs do que vale a pena.

Mesmo que um back-end esteja apenas levemente sobrecarregado, uma solicitação de cliente geralmente é melhor atendida se o back-end rejeitar novas tentativas e novas solicitações de maneira igual e rápida. Essas solicitações podem ser repetidas imediatamente em uma tarefa de backend diferente que pode ter recursos sobressalentes. A consequência de tratar tentativas e novas solicitações de forma idêntica no back-end é que repetir solicitações em diferentes tarefas se torna uma forma de balanceamento de carga orgânico: redireciona a carga para tarefas que podem ser mais adequadas para essas solicitações.

Decidindo tentar novamente

Quando um cliente recebe uma resposta de erro de “tarefa sobrecarregada”, ele precisa decidir se tenta novamente a solicitação. Temos alguns mecanismos para evitar novas tentativas quando uma parte significativa das tarefas em um cluster está sobrecarregada.

Primeiro, implementamos um orçamento de repetição por solicitação de até três tentativas. Se uma solicitação já falhou três vezes, deixamos a falha borbulhar para o chamador. A lógica é que, se uma solicitação já foi recebida em tarefas sobrecarregadas três vezes, é relativamente improvável que tentar novamente ajude porque todo o datacenter provavelmente está sobrecarregado.

Em segundo lugar, implementamos um orçamento de repetição por cliente. Cada cliente acompanha a proporção de solicitações que correspondem a novas tentativas. Uma solicitação só será repetida enquanto isso

proporção é inferior a 10%. A lógica é que, se apenas um pequeno subconjunto de tarefas estiver sobrecarregado, haverá relativamente pouca necessidade de tentar novamente.

Como exemplo concreto (do pior cenário), vamos supor que um datacenter está aceitando uma pequena quantidade de solicitações e rejeitando uma grande parte das solicitações. Seja X a taxa total de solicitações tentadas contra o datacenter de acordo com a lógica do lado do cliente. Devido ao número de tentativas que ocorrerão, o número de solicitações aumentará significativamente, para um pouco abaixo de $3X$. Embora tenhamos efetivamente limitado o crescimento causado por novas tentativas, um aumento de três vezes nas solicitações é significativo, especialmente se o custo de rejeitar versus processar uma solicitação for considerável. No entanto, colocar camadas no orçamento de repetição por cliente (uma taxa de repetição de 10%) reduz o crescimento para apenas 1,1x no caso geral – uma melhoria significativa.

Uma terceira abordagem faz com que os clientes incluam um contador de quantas vezes a solicitação já foi tentada nos metadados da solicitação. Por exemplo, o contador começa em 0 na primeira tentativa e é incrementado a cada nova tentativa até atingir 2, quando o orçamento por solicitação faz com que ele pare de ser repetido. Os back-ends mantêm histogramas desses valores no histórico recente. Quando um back-end precisa rejeitar uma solicitação, ele consulta esses histogramas para determinar a probabilidade de que outras tarefas de back-end também estejam sobrecarregadas.

Se esses histogramas revelarem uma quantidade significativa de tentativas (indicando que outras tarefas de back-end provavelmente também estão sobrecarregadas), eles retornarão um “overloaded; não tente novamente” em vez do erro padrão “tarefa sobrecarregada” que aciona novas tentativas.

A [Figura 21-1](#) mostra o número de tentativas em cada solicitação recebida por uma determinada tarefa de backend em várias situações de exemplo, em uma janela deslizante (correspondendo a 1.000 solicitações iniciais, sem contar as tentativas). Para simplificar, o orçamento de repetição por cliente é ignorado (ou seja, esses números assumem que o único limite para novas tentativas é o orçamento de repetição de três tentativas por solicitação), e a subconfiguração pode alterar um pouco esses números.

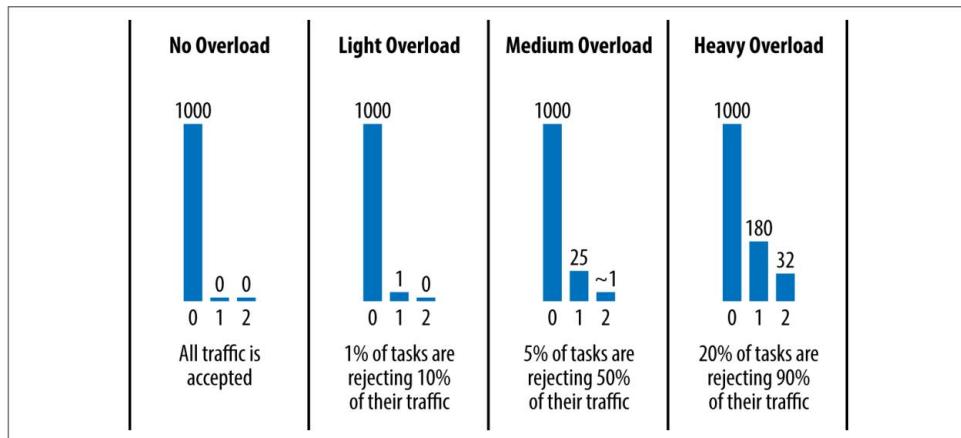


Figura 21-1. Histogramas de tentativas em várias condições

Nossos serviços maiores tendem a ser pilhas profundas de sistemas, que podem, por sua vez, ter dependências entre si. Nessa arquitetura, as solicitações devem ser repetidas apenas na camada imediatamente acima da camada que está rejeitando. Quando decidimos que uma determinada solicitação não pode ser atendida e não deve ser repetida, usamos um método “overloaded; não tente novamente” e, assim, evite uma explosão de novas tentativas combinatórias.

Considere o exemplo da [Figura 21-2](#) (na prática, nossas pilhas costumam ser significativamente mais complexas). Imagine que o DB Frontend esteja sobrecarregado e rejeite uma solicitação. Nesse caso:

- O back-end B tentará novamente a solicitação de acordo com as diretrizes anteriores. • No entanto, uma vez que o Backend B determina que a solicitação para o DB Frontend não pode ser atendida (por exemplo, porque a solicitação já foi tentada e rejeitada três vezes), o Backend B deve retornar ao Backend A um “overloaded ; não tente novamente” ou uma resposta degradada (assumindo que pode produzir alguma resposta moderadamente útil mesmo quando sua solicitação ao DB Frontend falhou).
- O backend A tem exatamente as mesmas opções para a solicitação que recebeu do frontend e procede de acordo.

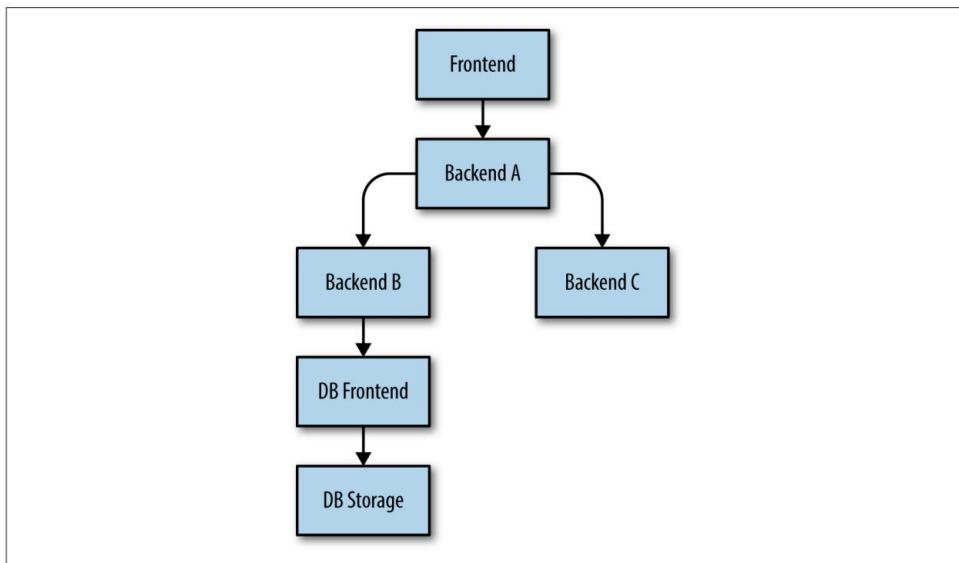


Figura 21-2. Uma pilha de dependências

O ponto chave é que uma requisição com falha do DB Frontend só deve ser tentada novamente pelo Backend B, a camada imediatamente acima dele. Se várias camadas tentassem novamente, teríamos uma explosão combinatória.

Carregar das Conexões

A carga associada às conexões é um último fator que merece destaque. Às vezes, levamos em consideração apenas a carga nos back-ends que é causada diretamente pelas solicitações que eles recebem (o que é um dos problemas com abordagens que modelam a carga com base em consultas por segundo). No entanto, isso ignora os custos de CPU e memória de manter um grande conjunto de conexões ou o custo de uma taxa rápida de rotatividade de conexões. Esses problemas são insignificantes em sistemas pequenos, mas rapidamente se tornam problemáticos ao executar sistemas RPC de grande escala.

Conforme mencionado anteriormente, nosso protocolo RPC exige que clientes inativos realizem verificações periódicas de integridade. Depois que uma conexão estiver ociosa por um período configurável, o cliente descarta sua conexão TCP e alterna para UDP para verificação de integridade. Infelizmente, esse comportamento é problemático quando você tem um número muito grande de tarefas de cliente que emitem uma taxa muito baixa de solicitações: a verificação de integridade das conexões pode exigir mais recursos do que realmente atender às solicitações. Abordagens como ajustar cuidadosamente os parâmetros de conexão (por exemplo, diminuir significativamente a frequência das verificações de integridade) ou mesmo criar e destruir as conexões dinamicamente podem melhorar significativamente essa situação.

O tratamento de rajadas de novas solicitações de conexão é um segundo problema (mas relacionado). Vimos rajadas desse tipo acontecerem no caso de trabalhos em lote muito grandes que criam um número muito grande de tarefas do cliente do trabalhador de uma só vez. A necessidade de negociar e manter um número excessivo de novas conexões simultaneamente pode sobrecarregar facilmente um grupo de back-ends. Em nossa experiência, existem algumas estratégias que podem ajudar a mitigar essa carga:

- Exponha a carga ao algoritmo de balanceamento de carga entre datacenters (por exemplo, balanceamento de carga base na utilização do cluster, em vez de apenas no número de solicitações). Nesse caso, a carga das solicitações é efetivamente rebalanceada para outros datacenters que têm capacidade ociosa. • Obrigar que os trabalhos de cliente em lote usem um conjunto separado de tarefas de back-end de proxy em lote que não fazem nada além de encaminhar solicitações para os back-ends subjacentes e devolver suas respostas aos clientes de maneira controlada. Portanto, em vez de “cliente em lote → back-end”, você tem “cliente em lote → proxy em lote → back-end”. Nesse caso, quando o trabalho muito grande é iniciado, apenas o trabalho de proxy em lote sofre, protegendo os back-ends reais (e clientes de prioridade mais alta). Efetivamente, o proxy de lote atua como um fusível. Outra vantagem de usar o proxy é que ele normalmente reduz o número de conexões contra o back-end, o que pode melhorar o balanceamento de carga contra o back-end (por exemplo, as tarefas do proxy podem usar subconjuntos maiores e provavelmente ter uma visão melhor do estado da rede). as tarefas de back-end).

Conclusões

Este capítulo e o [Capítulo 20](#) discutiram como várias técnicas (subconjunto determinístico, Round Robin ponderado, limitação do lado do cliente, cotas do cliente etc.) podem ajudar a distribuir a carga pelas tarefas em um datacenter de maneira relativamente uniforme. Entretanto, esses mecanismos dependem da propagação do estado sobre um sistema distribuído. Embora eles tenham um desempenho razoavelmente bom no caso geral, a aplicação no mundo real resultou em um pequeno número de situações em que eles funcionam de maneira imperfeita.

Como resultado, consideramos fundamental garantir que as tarefas individuais sejam protegidas contra sobrecarga. Para simplificar: uma tarefa de back-end provisionada para atender a uma determinada taxa de tráfego deve continuar a atender o tráfego a essa taxa sem nenhum impacto significativo na latência, independentemente de quanto tráfego em excesso seja lançado na tarefa. Como corolário, a tarefa de back-end não deve cair e falhar sob a carga. Essas declarações devem ser verdadeiras até uma determinada taxa de tráfego – algo acima de 2x ou até 10x do que a tarefa está provisionada para processar. Aceitamos que pode haver um certo ponto em que um sistema começa a entrar em colapso, e aumentar o limite em que esse colapso ocorre torna-se relativamente difícil de alcançar.

A chave é levar a sério essas condições de degradação. Quando essas condições de degradação são ignoradas, muitos sistemas apresentarão um comportamento terrível. E, à medida que o trabalho se acumula e as tarefas acabam ficando sem memória e travam (ou acabam queimando quase toda a CPU em thrashing de memória), a latência sofre à medida que o tráfego é descartado e as tarefas competem por recursos. Deixada desmarcada, a falha em um subconjunto de um sistema (como uma tarefa de back-end individual) pode desencadear a falha de outros componentes do sistema, potencialmente causando a falha de todo o sistema (ou um subconjunto considerável). O impacto desse tipo de falha em cascata pode ser tão grave que é fundamental para qualquer sistema que opere em escala se proteger contra isso; veja o [Capítulo 22](#).

É um erro comum supor que um back-end sobrecarregado deve ser desativado e parar de aceitar todo o tráfego. No entanto, essa suposição na verdade vai contra o objetivo de balanceamento de carga robusto. Na verdade, queremos que o back-end continue aceitando o máximo de tráfego possível, mas apenas aceite essa carga à medida que a capacidade for liberada. Um back-end bem comportado, suportado por políticas robustas de balanceamento de carga, deve aceitar apenas as solicitações que pode processar e rejeitar o restante normalmente.

Embora tenhamos uma vasta gama de ferramentas para implementar um bom balanceamento de carga e proteções contra sobrecarga, não existe uma bala mágica: o balanceamento de carga geralmente requer um profundo conhecimento de um sistema e da semântica de suas solicitações. As técnicas descritas neste capítulo evoluíram de acordo com as necessidades de muitos sistemas do Google e provavelmente continuarão a evoluir à medida que a natureza de nossos sistemas continuar a mudar.

CAPÍTULO 22

Lidando com falhas em cascata

Escrito por Mike Ulrich

Se no início você não conseguir, recue exponencialmente.

—Dan Sandler, engenheiro de software do Google

Por que as pessoas sempre esquecem que você precisa adicionar um pouco de jitter?

—Ade Oshieye, advogado do desenvolvedor do Google

Uma falha em cascata é uma falha que cresce ao longo do tempo como resultado de feedback positivo.¹ Pode ocorrer quando uma parte de um sistema geral falha, aumentando a probabilidade de que outras partes do sistema falhem. Por exemplo, uma única réplica de um serviço pode falhar devido à sobrecarga, aumentando a carga nas réplicas restantes e aumentando sua probabilidade de falha, causando um efeito dominó que desativa todas as réplicas de um serviço.

Usaremos o serviço de pesquisa Shakespeare discutido em “[Shakespeare: um serviço de amostra](#)” na [página 20](#) como exemplo ao longo deste capítulo. Sua configuração de produção pode se parecer com a [Figura 22-1](#).

¹ Veja Wikipedia, “Comentários positivos,” https://en.wikipedia.org/wiki/Positive_feedback.

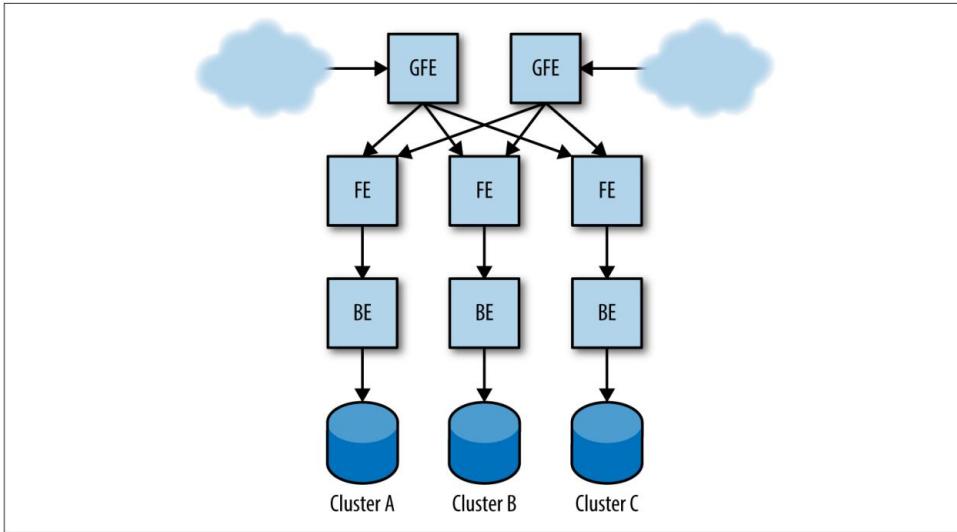


Figura 22-1. Exemplo de configuração de produção para o serviço de pesquisa Shakespeare

Causas de falhas em cascata e projeto para evitá-las

O design de sistema bem pensado deve levar em consideração alguns cenários típicos que respondem pela maioria das falhas em cascata.

Servidor sobrecarregado

A causa mais comum de falhas em cascata é a sobrecarga. A maioria das falhas em cascata descritas aqui são diretamente devido à sobrecarga do servidor ou devido a extensões ou variações deste cenário.

Suponha que o front-end no cluster A esteja processando 1.000 solicitações por segundo (QPS), como na Figura 22-2.

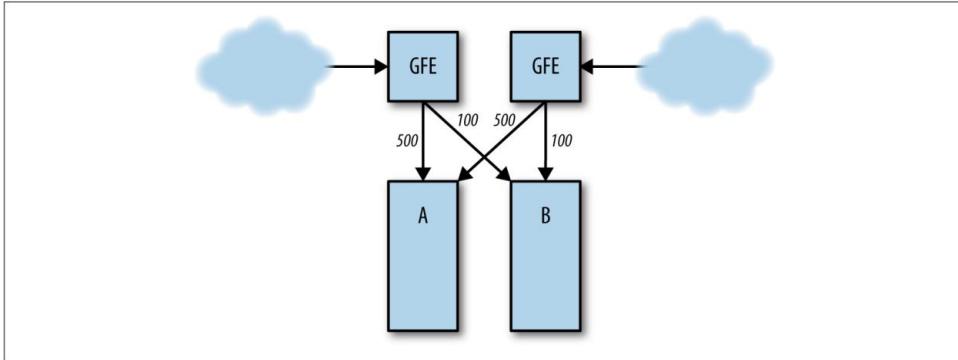


Figura 22-2. Distribuição normal de carga do servidor entre os clusters A e B

Se o cluster B falhar ([Figura 22-3](#)), as solicitações para o cluster A aumentam para 1.200 QPS. Os frontends em A não são capazes de lidar com solicitações em 1.200 QPS e, portanto, começam a ficar sem recursos, o que os faz travar, perder prazos ou se comportar mal. Como resultado, a taxa de solicitações tratadas com sucesso em A cai bem abaixo de 1.000 QPS.

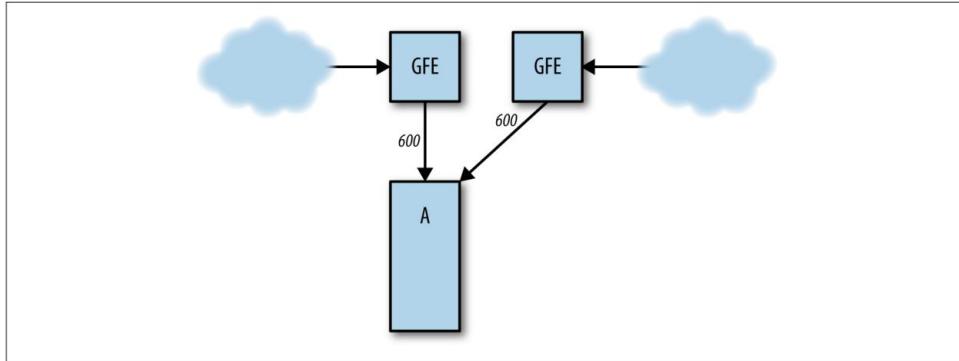


Figura 22-3. O cluster B falha, enviando todo o tráfego para o cluster A

Essa redução na taxa de trabalho útil que está sendo feito pode se espalhar para outros domínios de falha, potencialmente se espalhando globalmente. Por exemplo, a sobrecarga local em um cluster pode levar à falha de seus servidores; em resposta, o controlador de平衡amento de carga envia solicitações para outros clusters, sobrecarregando seus servidores, levando a uma falha de sobrecarga em todo o serviço. Pode não demorar muito para que esses eventos ocorram (por exemplo, na ordem de alguns minutos), porque o balanceador de carga e os sistemas de agendamento de tarefas envolvidos podem agir muito rapidamente.

Esgotamento de recursos

A falta de um recurso pode resultar em maior latência, taxas de erro elevadas ou substituição de resultados de qualidade inferior. Esses são, na verdade, efeitos desejados da falta de recursos: algo eventualmente precisa ceder à medida que a carga aumenta além do que um servidor pode suportar.

Dependendo de qual recurso se esgota em um servidor e de como o servidor é construído, o esgotamento de recursos pode tornar o servidor menos eficiente ou fazer com que o servidor falhe, solicitando que o balanceador de carga distribua os problemas de recursos para outros servidores. Quando isso acontece, a taxa de solicitações tratadas com êxito pode cair e possivelmente enviar o cluster ou um serviço inteiro para uma falha em cascata.

Diferentes tipos de recursos podem ser esgotados, resultando em efeitos variados nos servidores.

CPU

Se houver CPU insuficiente para lidar com a carga da solicitação, normalmente todas as solicitações ficam mais lentas. Este cenário pode resultar em vários efeitos secundários, incluindo o seguinte:

Aumento do número de solicitações em voo

Como as solicitações demoram mais para serem tratadas, mais solicitações são tratadas simultaneamente (até uma capacidade máxima possível na qual o enfileiramento pode ocorrer). Isso afeta quase todos os recursos, incluindo memória, número de threads ativos (em um modelo de servidor de thread por solicitação), número de descritores de arquivo e recursos de back-end (que, por sua vez, podem ter outros efeitos).

Comprimentos de fila excessivamente longos

Se não houver capacidade suficiente para lidar com todas as solicitações em estado estável, o servidor saturará suas filas. Isso significa que a latência aumenta (as solicitações são enfileiradas por mais tempo) e a fila usa mais memória. Consulte "[Gerenciamento de filas](#)" na [página 286](#) para obter uma discussão sobre estratégias de mitigação.

Fome de thread

Quando um thread não pode progredir porque está aguardando um bloqueio, as verificações de integridade podem falhar se o ponto de extremidade da verificação de integridade não puder ser atendido a tempo.

CPU ou solicitação de fome

Watchdogs2 internos no servidor detectam que o servidor não está progredindo, fazendo com que os servidores travem devido à falta de CPU ou devido à falta de solicitação se os eventos de watchdog forem acionados remotamente e processados como parte da solicitação fila.

Prazos de RPC perdidos

À medida que um servidor fica sobrecarregado, suas respostas aos RPCs de seus clientes chegam mais tarde, o que pode exceder os prazos definidos por esses clientes. O trabalho que o servidor fez para responder é desperdiçado e os clientes podem tentar novamente os RPCs, levando a uma sobrecarga ainda maior.

Benefícios reduzidos de cache da CPU

À medida que mais CPU é usada, a chance de se espalhar para mais núcleos aumenta, resultando em menor uso de caches locais e diminuição da eficiência da CPU.

2 Um watchdog é frequentemente implementado como um thread que acorda periodicamente para ver se o trabalho foi feito desde a última vez que foi verificado. Caso contrário, ele assume que o servidor está travado e o mata. Por exemplo, solicitações de um tipo conhecido podem ser enviadas ao servidor em intervalos regulares; se um não foi recebido ou processado quando esperado, isso pode indicar falha – do servidor, do sistema que envia solicitações ou da rede intermediária.

Memória

Se nada mais, mais solicitações em andamento consomem mais RAM da alocação da solicitação, resposta e objetos RPC. O esgotamento da memória pode causar os seguintes efeitos:

Tarefas em extinção

Por exemplo, uma tarefa pode ser despejada pelo gerenciador de contêiner (VM ou outro) por exceder os limites de recursos disponíveis, ou travamentos específicos do aplicativo podem fazer com que as tarefas sejam interrompidas.

Aumento da taxa de coleta de lixo (GC) em Java, resultando em maior uso da CPU

Um ciclo vicioso pode ocorrer neste cenário: menos CPU está disponível, resultando em solicitações mais lentas, resultando em maior uso de RAM, resultando em mais GC, resultando em disponibilidade ainda menor da CPU. Isso é conhecido coloquialmente como a “espiral da morte da GC”.

Redução nas taxas de acerto do cache

A redução na RAM disponível pode reduzir as taxas de acerto do cache no nível do aplicativo, resultando em mais RPCs para os back-ends, o que possivelmente pode fazer com que os back-ends fiquem sobrecarregados.

Tópicos

A fome de thread pode causar erros diretamente ou levar a falhas na verificação de integridade. Se o servidor adicionar threads conforme necessário, a sobrecarga de thread pode usar muita RAM. Em casos extremos, a fome de encadeamento também pode fazer com que você fique sem IDs de processo.

Descritores de arquivo

A falta de descritores de arquivo pode levar à incapacidade de inicializar conexões de rede, o que, por sua vez, pode causar falhas nas verificações de integridade.

Dependências entre recursos

Observe que muitos desses cenários de esgotamento de recursos alimentam-se uns dos outros—um serviço com sobrecarga geralmente tem uma série de sintomas secundários que podem parecer a causa raiz, dificultando a depuração.

Por exemplo, imagine o seguinte cenário:

1. Um frontend Java tem parâmetros de coleta de lixo (GC) mal ajustados.
2. Sob carga alta (mas esperada), o frontend fica sem CPU devido ao GC.
3. O esgotamento da CPU retarda a conclusão das solicitações.
4. O aumento do número de solicitações em andamento faz com que mais RAM seja usada para processar os pedidos.

5. A pressão de memória devido a solicitações, em combinação com uma alocação de memória fixa para o processo frontend como um todo, deixa menos RAM disponível para armazenamento em cache.
6. O tamanho reduzido do cache significa menos entradas no cache, além de um menor taxa de acerto.
7. O aumento nas faltas de cache significa que mais solicitações chegam ao back-end para atendimento.
8. O backend, por sua vez, fica sem CPU ou threads.
9. Finalmente, a falta de CPU faz com que as verificações básicas de integridade falhem, iniciando uma cascata falha.

Em situações tão complexas quanto o cenário anterior, é improvável que a cadeia causal seja totalmente diagnosticada durante uma interrupção. Pode ser muito difícil determinar que a falha de back-end foi causada por uma diminuição na taxa de cache no front-end, principalmente se os componentes de front-end e back-end tiverem proprietários diferentes.

Indisponibilidade de serviço O

esgotamento de recursos pode causar falhas nos servidores; por exemplo, os servidores podem travar quando muita RAM é alocada para um contêiner. Uma vez que alguns servidores travam por sobrecarga, a carga nos servidores restantes pode aumentar, fazendo com que eles também travem. O problema tende a se tornar uma bola de neve e logo todos os servidores começam a travar. Muitas vezes, é difícil escapar desse cenário porque, assim que os servidores voltam a ficar online, são bombardeados com uma taxa extremamente alta de solicitações e falham quase que imediatamente.

Por exemplo, se um serviço estava íntegro em 10.000 QPS, mas iniciou uma falha em cascata devido a falhas em 11.000 QPS, reduzir a carga para 9.000 QPS quase certamente não interromperá as falhas. Isso porque o serviço atenderá ao aumento da demanda com capacidade reduzida; apenas uma pequena fração de servidores normalmente estará íntegro o suficiente para lidar com solicitações. A fração de servidores que estarão íntegros depende de alguns fatores: a rapidez com que o sistema é capaz de iniciar as tarefas, a rapidez com que o binário pode começar a servir com capacidade total e por quanto tempo uma tarefa recém-iniciada é capaz de sobreviver ao carregar.

Neste exemplo, se 10% dos servidores estiverem íntegros o suficiente para lidar com solicitações, a taxa de solicitação precisaria cair para cerca de 1.000 QPS para que o sistema se estabilizasse e se recuperasse.

Da mesma forma, os servidores podem parecer insalubres para a camada de balanceamento de carga, resultando em capacidade reduzida de balanceamento de carga: os servidores podem entrar no estado “pato manco” (consulte “[Uma abordagem robusta para tarefas insalubres: estado do pato lame](#)” na página 234) ou falhar nas verificações de integridade sem bater. O efeito pode ser muito semelhante ao travamento: mais servidores parecem íntegros, os servidores íntegros tendem a aceitar solicitações por um período muito breve antes de se tornarem íntegros e menos servidores participam do tratamento de solicitações.

As políticas de balanceamento de carga que evitam servidores que atenderam a erros podem agravar ainda mais os problemas — alguns back-ends atendem a alguns erros, portanto, não contribuem para a capacidade disponível para o serviço. Isso aumenta a carga nos servidores restantes, iniciando o efeito bola de neve.

Prevenindo a sobrecarga do servidor

A lista a seguir apresenta estratégias para evitar a sobrecarga do servidor em ordem aproximada de prioridade:

Teste de carga os limites de capacidade do servidor e teste o modo de falha para sobrecarga

Este é o exercício mais importante que você deve realizar para evitar a sobrecarga do servidor. A menos que você teste em um ambiente realista, é muito difícil prever exatamente qual recurso será esgotado e como esse esgotamento de recurso se manifestará. Para obter detalhes, consulte “[Teste de falhas em cascata](#)” na página 278.

Exiba resultados degradados

Exiba resultados de qualidade inferior e mais baratos para o usuário. Sua estratégia aqui será específica do serviço. Consulte “[Descarga de Carga e Degradação Graciosa](#)” na página 267.

Instrumentar o servidor para rejeitar solicitações quando sobrecarregado

Os servidores devem se proteger de ficarem sobrecarregados e travarem.

Quando sobrecarregado nas camadas front-end ou back-end, falhe cedo e de forma barata.

Para obter detalhes, consulte “[Descarga de Carga e Degradação Graciosa](#)” na página 267.

Instrumentar sistemas de nível superior para rejeitar solicitações, em vez de sobrecarregar os servidores

Observe que, como a limitação de taxa geralmente não leva em consideração a integridade geral do serviço, talvez não seja possível interromper uma falha que já começou. As implementações simples de limitação de taxa também podem deixar a capacidade não utilizada. A limitação de taxa pode ser implementada em vários lugares:

- Nos proxies reversos, limitando o volume de solicitações por critérios como endereço IP para mitigar tentativas de ataques de negação de serviço e clientes abusivos.
- Nos平衡adores de carga, descartando solicitações quando o serviço entra em sobrecarga global. Dependendo da natureza e complexidade do serviço, essa limitação de taxa pode ser indiscriminada (“descartar todo o tráfego acima de X solicitações por segundo”) ou mais seletiva (“descartar solicitações que não são de usuários que interagiram recentemente com o serviço” ou “descarte solicitações para operações de baixa prioridade, como sincronização em segundo plano, mas continue atendendo a sessões interativas de usuário”).

- Em tarefas individuais, para evitar que flutuações aleatórias no balanceamento de carga sobrecarreguem o servidor.

Realizar planejamento de

capacidade Um bom planejamento de capacidade pode reduzir a probabilidade de ocorrência de uma falha em cascata. O planejamento de capacidade deve ser acoplado a testes de desempenho para determinar a carga na qual o serviço falhará. Por exemplo, se o ponto de interrupção de cada cluster for 5.000 QPS, a carga for distribuída uniformemente pelos clusters³ e a carga de pico do serviço for 19.000 QPS, então aproximadamente seis clusters serão necessários para executar o serviço em N + 2.

O planejamento de capacidade reduz a probabilidade de acionar uma falha em cascata, mas não é suficiente para proteger o serviço de falhas em cascata. Quando você perde partes importantes de sua infraestrutura durante um evento planejado ou não planejado, nenhuma quantidade de planejamento de capacidade pode ser suficiente para evitar falhas em cascata. Problemas de balanceamento de carga, partições de rede ou aumentos inesperados de tráfego podem criar bolsões de carga alta além do planejado. Alguns sistemas podem aumentar o número de tarefas para seu serviço sob demanda, o que pode evitar sobrecarga; no entanto, ainda é necessário um planejamento de capacidade adequado.

Gerenciamento de filas A

maioria dos servidores thread-per-request usa uma fila na frente de um pool de threads para lidar com as solicitações. As solicitações chegam, ficam em uma fila e, em seguida, as threads retiram as solicitações da fila e executam o trabalho real (quaisquer ações exigidas pelo servidor).

Normalmente, se a fila estiver cheia, o servidor rejeitará novas solicitações.

Se a taxa de solicitação e a latência de uma determinada tarefa forem constantes, não há motivo para enfileirar solicitações: um número constante de threads deve ser ocupado. Sob esse cenário idealizado, as requisições só serão enfileiradas se a taxa de estado estável de requisições recebidas exceder a taxa na qual o servidor pode processar requisições, o que resulta na saturação do pool de threads e da fila.

As solicitações enfileiradas consomem memória e aumentam a latência. Por exemplo, se o tamanho da fila for 10x o número de encadeamentos, o tempo para tratar a solicitação em um encadeamento será de 100 milissegundos. Se a fila estiver cheia, uma solicitação levará 1,1 segundo para ser processada, a maior parte do tempo gasto na fila.

Para um sistema com tráfego bastante estável ao longo do tempo, geralmente é melhor ter pequenos comprimentos de fila em relação ao tamanho do pool de encadeamentos (por exemplo, 50% ou menos), o que resulta na rejeição antecipada de solicitações pelo servidor quando não pode sustentar a taxa de solicitações recebidas. Por exemplo, o Gmail geralmente usa servidores sem fila, contando com failover para outros

³ Isso geralmente não é uma boa suposição devido à geografia; consulte também “Job and Data Organization” na página 22.

tarefas do servidor quando os encadeamentos estão cheios. Na outra extremidade do espectro, sistemas com carga “bursty” para os quais os padrões de tráfego flutuam drasticamente podem se sair melhor com um tamanho de fila baseado no número atual de threads em uso, tempo de processamento para cada solicitação e o tamanho e a frequência dos bursts .

Descarte de carga e degradação graciosa

O descarte de carga reduz uma parte da carga ao eliminar o tráfego à medida que o servidor se aproxima das condições de sobrecarga. O objetivo é evitar que o servidor fique sem RAM, falhe nas verificações de integridade, atenda com latência extremamente alta ou qualquer outro sintoma associado à sobrecarga, enquanto ainda faz o máximo de trabalho útil possível.

Uma maneira direta de reduzir a carga é fazer a limitação por tarefa com base na CPU, na memória ou no comprimento da fila; limitar o comprimento da fila conforme discutido em “[Gerenciamento de filas](#)” na [página 276](#) é uma forma dessa estratégia. Por exemplo, uma abordagem eficaz é retornar um HTTP 503 (serviço indisponível) para qualquer solicitação recebida quando houver mais de um determinado número de solicitações de clientes em andamento.

Alterar o método de enfileiramento do padrão first-in, first-out (FIFO) para last-in, first-out (LIFO) ou usar o algoritmo de atraso controlado (CoDel) [Nic12] ou abordagens semelhantes podem reduzir a carga removendo solicitações que provavelmente não vale a pena processar [Mau15]. Se a pesquisa na web de um usuário estiver lenta porque um RPC ficou na fila por 10 segundos, há uma boa chance de o usuário ter desistido e atualizado seu navegador, emitindo outra solicitação: não adianta responder à primeira, pois ela será ignorada ! Essa estratégia funciona bem quando combinada com a propagação de prazos RPC em toda a pilha, descritos em “[Latência e prazos](#)” na [página 271](#).

Abordagens mais sofisticadas incluem identificar clientes para serem mais seletivos sobre qual trabalho é descartado ou selecionar solicitações mais importantes e priorizar.

Essas estratégias são mais prováveis de serem necessárias para serviços compartilhados.

A degradação graciosa leva o conceito de redução de carga um passo adiante, reduzindo a quantidade de trabalho que precisa ser realizada. Em alguns aplicativos, é possível diminuir significativamente a quantidade de trabalho ou tempo necessário, diminuindo a qualidade das respostas. Por exemplo, um aplicativo de pesquisa pode pesquisar apenas um subconjunto de dados armazenados em um cache de memória em vez do banco de dados completo em disco ou usar um algoritmo de classificação menos preciso (mas mais rápido) quando sobrecarregado.

Ao avaliar as opções de redução de carga ou degradação suave para seu serviço, considere o seguinte:

- Quais métricas você deve usar para determinar quando o corte de carga ou degradação graciosa deve ser ativado (por exemplo, uso da CPU, latência, comprimento da fila, número de threads usados, se o seu serviço entra no modo degradado automaticamente ou se a intervenção manual é necessária) ?

- Que ações devem ser tomadas quando o servidor está em modo degradado? • Em qual camada deve ser implementado o descarte de carga e a degradação graciosa?
- Faz sentido implementar essas estratégias em todas as camadas da pilha ou é suficiente ter um ponto de estrangulamento de alto nível?

Ao avaliar as opções e implantar, lembre-se do seguinte:

- A degradação gradual não deve ser acionada com muita frequência, geralmente em casos de falha no planejamento de capacidade ou mudança inesperada de carga. Mantenha o sistema simples e compreensível, principalmente se não for usado com frequência.
- Lembre-se de que o caminho de código que você nunca usa é o caminho de código que (frequentemente) não funciona. Na operação em estado estável, o modo de degradação normal não será usado, o que implica que você terá muito menos experiência operacional com esse modo e qualquer uma de suas peculiaridades, o que aumenta o nível de risco. Você pode garantir que a degradação normal continue funcionando executando regularmente um pequeno subconjunto de servidores próximo à sobrecarga para exercitar esse caminho de código.
- Monitore e alerte quando muitos servidores entrarem nesses modos.
- Rejeição de carga complexa e degradação suave podem causar problemas por si mesmos — a complexidade excessiva pode fazer com que o servidor entre em um modo degradado quando não for desejado ou entre em ciclos de feedback em momentos indesejados. Projete uma maneira de desativar rapidamente a degradação graciosa complexa ou ajustar os parâmetros, se necessário.

Armazenar essa configuração em um sistema consistente que cada servidor possa observar quanto a alterações, como Chubby, pode aumentar a velocidade de implantação, mas também apresenta seus próprios riscos de falha sincronizada.

Novas tentativas

Suponha que o código no frontend que fala com o backend implemente novas tentativas ingenuamente. Ele tenta novamente após encontrar uma falha e limita o número de RPCs de back-end por solicitação lógica para 10. Considere este código no front-end, usando gRPC em Go:

```
func exampleRpcCall(client pb.ExampleClient, request pb.Request) *pb.Response {
    // Define o tempo limite de RPC para 5
    segundos. opts := grpc.WithTimeout(5 * time.Second)

    // Tente até 20 vezes para fazer a chamada RPC.
    tentativas := 20 para tentativas > 0 { conn, err :=
        grpc.Dial(*serverAddr, opts...) if err != nil { // Algo deu
            errado na configuração da conexão. Tente novamente.
            tentativas -- continuar
    }
}
```

```

        } adiar conn.Close()

        // Cria um stub de cliente e faz a chamada RPC. client :=  

        pb.NewBackendClient(conn) response, err :=  

        client.MakeRequest(context.Background, request) if err != nil { // Algo deu  

        errado ao fazer a chamada. Tente novamente. tentativas -- continuar

    }

    resposta de retorno
}

grpclog.Fatalf("foram sem tentativas")
}

```

Este sistema pode cascatear da seguinte maneira:

1. Suponha que nosso back-end tenha um limite conhecido de 10.000 QPS por tarefa, após o qual todas as solicitações adicionais são rejeitadas em uma tentativa de degradação normal.
2. O front-end chama MakeRequest a uma taxa constante de 10.100 QPS e sobrecarrega o back-end em 100 QPS, que o back-end rejeita.
3. Esses 100 QPS com falha são repetidos em MakeRequest a cada 1.000 ms e provavelmente são bem-sucedidos. Mas as tentativas estão se somando às solicitações enviadas ao back-end, que agora recebe 10.200 QPS — 200 QPS dos quais estão falhando devido à sobrecarga.
4. O volume de novas tentativas aumenta: 100 QPS de tentativas no primeiro segundo levam a 200 QPS, depois a 300 QPS e assim por diante. Cada vez menos solicitações são bem-sucedidas na primeira tentativa, de modo que o trabalho menos útil está sendo executado como uma fração das solicitações para o back-end.
5. Se a tarefa de back-end não for capaz de lidar com o aumento na carga — que está consumindo descritores de arquivo, memória e tempo de CPU no back-end — ela pode derreter e travar sob a simples carga de solicitações e novas tentativas. Essa falha redistribui as solicitações recebidas pelas tarefas de back-end restantes, sobrecarregando ainda mais essas tarefas.

Algumas suposições simplificadoras foram feitas aqui para ilustrar esse cenário,⁴ mas permanece o fato de que novas tentativas podem desestabilizar um sistema. Observe que picos de carga temporários e aumentos lentos no uso podem causar esse efeito.

⁴ Um exercício instrutivo, deixado para o leitor: escreva um simulador simples e veja como a quantidade de trabalho útil que o backend pode fazer varia com o quanto ele está sobrecarregado e quantas tentativas são permitidas.

Mesmo que a taxa de chamadas para MakeRequest diminua para os níveis anteriores ao colapso (9.000 QPS, por exemplo), dependendo de quanto o retorno de uma falha custa ao back-end, o problema pode não desaparecer. Dois fatores estão em jogo aqui:

- Se o back-end gastar uma quantidade significativa de recursos processando solicitações que acabarão por falhar devido à sobrecarga, as próprias tentativas podem estar mantendo o back-end em um modo sobrecarregado.
- Os próprios servidores back-end podem não ser estáveis. As novas tentativas podem amplificar os efeitos vistos em “[Sobrecarga do servidor](#)” na [página 260](#).

Se qualquer uma dessas condições for verdadeira, para sair dessa interrupção, você deve reduzir ou eliminar drasticamente a carga nos front-ends até que as tentativas parem e os back-ends se estabilizem.

Este padrão tem contribuído para várias falhas em cascata, quer os frontends e backends se comuniquem por meio de mensagens RPC, o “frontend” é o código JavaScript do cliente que emite chamadas XMLHttpRequest para um endpoint e tenta novamente em caso de falha, ou as tentativas originam-se de um protocolo de sincronização offline que tenta novamente agressivamente quando encontra uma falha clara.

Ao emitir novas tentativas automáticas, tenha em mente as seguintes considerações:

- A maioria das estratégias de proteção de back-end descritas em “[Evitando sobrecarga do servidor](#)” na [página 265](#) se aplicam. Em particular, testar o sistema pode destacar problemas e a degradação suave pode reduzir o efeito das novas tentativas no back-end. • Sempre use a retirada exponencial aleatória ao agendar novas tentativas. Veja também “[Recuo Exponencial e Jitter](#)” no blog de arquitetura da AWS [Bro15]. Se as novas tentativas não forem distribuídas aleatoriamente pela janela de novas tentativas, uma pequena perturbação (por exemplo, um blipe na rede) pode fazer com que ondulações de novas tentativas sejam agendadas ao mesmo tempo, que podem então se ampliar [Flo94].
- Limite as tentativas por solicitação. Não repita uma determinada solicitação indefinidamente. • Considere ter um orçamento de repetição em todo o servidor. Por exemplo, permita apenas 60 tentativas por minuto em um processo e, se o orçamento de novas tentativas for excedido, não tente novamente; apenas falhe o pedido. Essa estratégia pode conter o efeito de repetição e ser a diferença entre uma falha de planejamento de capacidade que leva a algumas consultas descartadas e uma falha global em cascata. • Pense no serviço de forma holística e decida se você realmente precisa realizar novas tentativas em um determinado nível. Em particular, evite ampliar novas tentativas emitindo novas tentativas em vários níveis: uma única solicitação na camada mais alta pode produzir um número de tentativas tão grande quanto o produto do número de tentativas em cada camada para a camada mais baixa. Se o banco de dados não puder atender às solicitações porque está sobrecarregado e as camadas de back-end, front-end e JavaScript emitirem 3 tentativas (4 tentativas), um

uma única ação do usuário pode criar 64 tentativas (4³) no banco de dados. Este comportamento é indesejáveis quando o banco de dados está retornando esses erros porque está sobrecarregado. • Use códigos de resposta claros e considere como os diferentes modos de falha devem ser tratados. Por exemplo, separe as condições de erro passíveis de repetição e não-recuperáveis. Não tente novamente erros permanentes ou solicitações malformadas em um cliente, porque nenhum dos dois terá sucesso. Retornar um status específico quando sobrecarregado para que clientes e outras camadas recuem e não tentem novamente.

Em uma emergência, pode não ser óbvio que uma interrupção é devido a um mau comportamento de repetição. Gráficos de taxas de repetição podem ser uma indicação de mau comportamento de repetição, mas podem ser confundidos como um sintoma em vez de uma causa composta. Em termos de mitigação, este é um caso especial do problema de capacidade insuficiente, com a ressalva adicional de que você deve corrigir o comportamento de repetição (geralmente exigindo um push de código), reduzir a carga significativamente ou cortar as solicitações completamente.

Latência e prazos

Quando um frontend envia um RPC para um servidor backend, o frontend consome recursos aguardando uma resposta. Os prazos de RPC definem quanto tempo uma solicitação pode esperar antes que o frontend desista, limitando o tempo que o backend pode consumir Recursos.

Escolhendo um

prazo Geralmente é aconselhável definir um prazo. A definição de nenhum prazo ou um prazo extremamente alto pode causar problemas de curto prazo que já passaram há muito tempo para continuar a consumir recursos do servidor até que o servidor seja reiniciado.

Prazos altos podem resultar em consumo de recursos em níveis mais altos da pilha quando os níveis mais baixos da pilha estão tendo problemas. Prazos curtos podem fazer com que algumas solicitações mais caras falhem de forma consistente. Equilibrar essas restrições para escolher um bom prazo pode ser uma arte.

Prazos perdidos

Um tema comum em muitas interrupções em cascata é que os servidores gastam recursos tratando de solicitações que excederão seus prazos no cliente. Como resultado, os recursos são gastos enquanto nenhum progresso é feito: você não recebe crédito por atribuições atrasadas com RPCs.

Suponha que um RPC tenha um prazo de 10 segundos, conforme definido pelo cliente. O servidor está muito sobrecarregado e, como resultado, leva 11 segundos para passar de uma fila para um pool de threads. Neste ponto, o cliente já desistiu do pedido. Na maioria das circunstâncias, seria imprudente para o servidor tentar lidar com essa solicitação, porque estaria fazendo um trabalho para o qual nenhum crédito seria concedido - o cliente não se importa com o que

trabalho que o servidor faz após o término do prazo, porque ele já desistiu da solicitação.

Se o tratamento de uma solicitação for realizado em vários estágios (por exemplo, há alguns retornos de chamada e chamadas RPC), o servidor deve verificar o prazo restante em cada estágio antes de tentar executar mais trabalho na solicitação. Por exemplo, se uma solicitação for dividida em etapas de análise, solicitação de back-end e processamento, pode fazer sentido verificar se há tempo suficiente para lidar com a solicitação antes de cada etapa.

Propagação de prazo

Em vez de inventar um prazo ao enviar RPCs para back-ends, os servidores devem empregar propagação de prazo e propagação de cancelamento.

Com a propagação de prazo, um prazo é definido no topo da pilha (por exemplo, no frontend).

A árvore de RPCs que emanam de uma solicitação inicial terão todos o mesmo deadline absoluto. Por exemplo, se o servidor A selecionar um prazo de 30 segundos e processar a solicitação por 7 segundos antes de enviar um RPC ao servidor B, o RPC de A para B terá um prazo de 23 segundos. Se o servidor B demorar 4 segundos para processar a solicitação e enviar um RPC para o servidor C, o RPC de B para C terá um prazo de 19 segundos e assim por diante.

Idealmente, cada servidor na árvore de solicitações implementa a propagação de prazos.

Sem propagação de prazo, pode ocorrer o seguinte cenário:

1. O servidor A envia um RPC ao servidor B com um prazo de 10 segundos.
2. O servidor B leva 8 segundos para iniciar o processamento da solicitação e, em seguida, envia um RPC para servidor C.
3. Se o servidor B usa propagação de prazo, ele deve definir um prazo de 2 segundos, mas suponha que, em vez disso, use um prazo de 20 segundos codificado para o RPC para o servidor C.
4. O servidor C retira a solicitação de sua fila após 5 segundos.

Se o servidor B tivesse usado a propagação de prazo, o servidor C poderia desistir imediatamente da solicitação porque o prazo de 2 segundos foi excedido. No entanto, neste cenário, o servidor C processa a solicitação pensando que tem 15 segundos de sobra, mas não está fazendo um trabalho útil, pois a solicitação do servidor A para o servidor B já excedeu seu prazo.

Você pode querer reduzir um pouco o prazo de saída (por exemplo, algumas centenas de milissegundos) para levar em conta os tempos de trânsito da rede e o pós-processamento no cliente.

Considere também definir um limite superior para prazos de saída. Você pode querer limitar quanto tempo o servidor espera por RPCs de saída para back-ends não críticos ou por RPCs para back-ends que normalmente são concluídos em um curto período. No entanto, certifique-se de entender seu mix de tráfego, pois você pode inadvertidamente criar tipos específicos de

solicitações falham o tempo todo (por exemplo, solicitações com grandes cargas ou solicitações que exigem resposta a muita computação).

Existem algumas exceções para as quais os servidores podem desejar continuar processando uma solicitação após o término do prazo. Por exemplo, se um servidor recebe uma solicitação que envolve a execução de alguma operação de recuperação cara e periodicamente verifica o andamento da recuperação, seria uma boa ideia verificar o prazo somente após escrever o ponto de verificação, em vez de após a operação cara.

A propagação de cancelamentos evita o potencial vazamento de RPC que ocorre se um RPC inicial tiver um prazo longo, mas RPCs entre camadas mais profundas da pilha têm prazos curtos e tempo limite. Usando a propagação de prazo simples, o RPC inicial continua a usar os recursos do servidor até atingir o tempo limite, apesar de não conseguir progredir.

Latência bimodal

Suponha que o frontend do exemplo anterior consista em 10 servidores, cada um com 100 threads de trabalho. Isso significa que o frontend tem um total de 1.000 threads de capacidade. Durante a operação normal, os frontends executam 1.000 QPS e as solicitações são concluídas em 100 ms. Isso significa que os frontends geralmente têm 100 threads de trabalho ocupados dos 1.000 threads de trabalho configurados ($1.000 \text{ QPS} * 0,1 \text{ segundos}$).

Suponha que um evento faça com que 5% das solicitações nunca sejam concluídas. Isso pode ser o resultado da indisponibilidade de alguns intervalos de linhas do Bigtable, o que torna as solicitações correspondentes a esse keyspace do Bigtable inservíveis. Como resultado, 5% das solicitações atingiram o prazo, enquanto os 95% restantes das solicitações demoraram os habituais 100 ms.

Com um prazo de 100 segundos, 5% das solicitações consumiriam 5.000 threads ($50 \text{ QPS} * 100 \text{ segundos}$), mas o frontend não tem tantos threads disponíveis. Assumindo que não há outros efeitos secundários, o frontend só será capaz de lidar com 19,6% das solicitações ($1.000 \text{ threads disponíveis} / (5.000 + 95) \text{ threads de trabalho}$), resultando em 80,4% taxa de erro.

Portanto, em vez de apenas 5% das solicitações receberem um erro (aqueles que não foram concluídas devido à indisponibilidade do keyspace), a maioria das solicitações recebe um erro.

As diretrizes a seguir podem ajudar a resolver essa classe de problemas:

- Detectar esse problema pode ser muito difícil. Em particular, pode não estar claro que a latência bimodal é a causa de uma interrupção quando você está analisando a latência média.
- Quando você observar um aumento de latência, tente observar a distribuição de latências além das médias.
- Esse problema pode ser evitado se as solicitações não concluídas retornarem com um erro antes, ao invés de esperar o prazo completo. Por exemplo, se um back-end não estiver disponível, geralmente é melhor retornar imediatamente um erro para esse back-end,

em vez de consumir recursos até que o backend esteja disponível. Se a sua camada RPC suportar uma opção fail-fast, use-a.

- Ter prazos várias ordens de magnitude mais longos do que a latência média de solicitação geralmente é ruim. No exemplo anterior, um pequeno número de solicitações atingiu inicialmente o prazo, mas o prazo foi três ordens de grandeza maior do que a latência média normal, levando à exaustão do encadeamento.
- Ao usar recursos compartilhados que podem ser esgotados por algum keyspace, considere limitar as solicitações em andamento por esse keyspace ou usar outros tipos de rastreamento de abuso. Suponha que seu back-end processe solicitações para clientes diferentes que têm desempenho e características de solicitação muito diferentes. Você pode considerar permitir que apenas 25% de seus encadeamentos sejam ocupados por qualquer cliente para fornecer justiça diante da carga pesada de qualquer cliente que se comporte mal.

Inicialização lenta e cache a frio

Os processos geralmente são mais lentos para responder às solicitações imediatamente após o início do que estarão em estado estável. Esta lentidão pode ser causada por um ou ambos os seguintes:

Inicialização necessária

Configurando conexões ao receber a primeira solicitação que precisa de um determinado back-end

Melhorias de desempenho de tempo de execução em algumas linguagens, principalmente Java
Compilação Just-In-Time, otimização de hotspot e carregamento de classe adiado

Da mesma forma, alguns binários são menos eficientes quando os caches não são preenchidos. Por exemplo, no caso de alguns serviços do Google, a maioria das solicitações é atendida fora dos caches, portanto, as solicitações que perdem o cache são significativamente mais caras. Na operação de estado estável com um cache quente, ocorrem apenas algumas falhas de cache, mas quando o cache está completamente vazio, 100% das solicitações são caras. Outros serviços podem empregar caches para manter o estado de um usuário na RAM. Isso pode ser feito por meio de aderência rígida ou suave entre proxies reversos e front-ends de serviço.

Se o serviço não for provisionado para lidar com solicitações em um cache frio, ele corre um risco maior de interrupções e deve tomar medidas para evitá-las.

Os seguintes cenários podem levar a um cache frio:

Ativando um novo cluster Um cluster adicionado recentemente terá um cache vazio.

Retornando um cluster para serviço após manutenção O cache pode estar obsoleto.

Reinicializações Se uma tarefa com um cache foi reiniciada recentemente, o preenchimento do cache levará algum tempo. Pode valer a pena mover o cache de um servidor para um binário separado como o memcache, que também permite o compartilhamento de cache entre muitos servidores, embora ao custo de introduzir outro RPC e uma pequena latência adicional.

Se o armazenamento em cache tiver um efeito significativo no serviço,⁵ convém usar uma ou algumas das seguintes estratégias:

- Superprovisionar o serviço. É importante observar a distinção entre um cache de latência e um cache de capacidade: quando um cache de latência é empregado, o serviço pode sustentar sua carga esperada com um cache vazio, mas um serviço que usa um cache de capacidade não pode sustentar sua carga esperada com um cache vazio. Os proprietários de serviço devem estar atentos ao adicionar caches ao serviço e certificar-se de que quaisquer novos caches sejam caches de latência ou sejam suficientemente bem projetados para funcionar com segurança como caches de capacidade. Às vezes, os caches são adicionados a um serviço para melhorar o desempenho, mas na verdade acabam sendo dependências rígidas.
- Empregar técnicas gerais de prevenção de falhas em cascata. Em particular, os servidores devem rejeitar solicitações quando estiverem sobrecarregados ou entrar em modos degradados, e testes devem ser realizados para ver como o serviço se comporta após eventos como uma grande reinicialização.
- Ao adicionar carga a um cluster, aumente lentamente a carga. A taxa de solicitação inicialmente pequena aquece o cache; uma vez que o cache está quente, mais tráfego pode ser adicionado. É uma boa ideia garantir que todos os clusters carreguem carga nominal e que os caches sejam mantidos aquecidos.

Sempre vá para baixo na pilha

No exemplo do serviço Shakespeare, o frontend fala com um backend, que por sua vez fala com a camada de armazenamento. Um problema que se manifesta na camada de armazenamento pode causar problemas para os servidores que se comunicam com ela, mas a correção da camada de armazenamento geralmente repara as camadas de back-end e front-end.

No entanto, suponha que os back-ends se comuniquem entre si. Por exemplo, os back-ends podem fazer proxy de solicitações entre si para alterar quem é o proprietário de um usuário quando a camada de armazenamento não pode atender a uma solicitação. Essa comunicação intra-camada pode ser problemática por vários motivos:

⁵ Às vezes, você descobre que uma proporção significativa de sua capacidade real de atendimento é uma função do atendimento a partir de um cache e, se você perdesse o acesso a esse cache, não seria capaz de atender a tantas consultas. Uma observação semelhante é válida para a latência: um cache pode ajudá-lo a atingir metas de latência (reduzindo o tempo médio de resposta quando a consulta pode ser veiculada a partir do cache) que você possivelmente não poderia atender sem esse cache.

- A comunicação é suscetível a um deadlock distribuído. Os back-ends podem usar o mesmo pool de threads para aguardar RPCs enviados para back-ends remotos que estão recebendo simultaneamente solicitações de back-ends remotos. Suponha que o pool de threads do back-end A esteja cheio. O back-end B envia uma solicitação ao back-end A e usa um encadeamento no back-end B até que o pool de encadeamentos do back-end A seja limpo. Esse comportamento pode fazer com que a saturação do pool de threads se espalhe.
- Se a comunicação intracamada aumentar em resposta a algum tipo de falha ou condição de carga pesada (por exemplo, rebalanceamento de carga que é mais ativo sob alta carga), a comunicação intracamada pode alternar rapidamente de um modo de solicitação intracamada baixa para alta quando a carga aumenta o suficiente.

Por exemplo, suponha que um usuário tenha um back-end primário e um back-end secundário de hot standby predeterminado em um cluster diferente que possa assumir o controle do usuário. Os proxies de back-end primários solicitam ao back-end secundário como resultado de erros da camada inferior ou em resposta a uma carga pesada no mestre. Se todo o sistema estiver sobrecarregado, o proxy primário para secundário provavelmente aumentará e adicionará ainda mais carga ao sistema, devido ao custo adicional de analisar e aguardar a solicitação para o secundário no primário.

- Dependendo da criticidade da comunicação entre camadas, a inicialização do sistema pode se tornar mais complexa.

Geralmente, é melhor evitar a comunicação intra-camada — ou seja, possíveis ciclos no caminho de comunicação — no caminho de solicitação do usuário. Em vez disso, faça com que o cliente faça a comunicação. Por exemplo, se um front-end falar com um back-end, mas adivinhar o back-end errado, o back-end não deve fazer proxy para o back-end correto. Em vez disso, o back-end deve informar ao front-end para tentar novamente sua solicitação no back-end correto.

Condições de acionamento para falhas em cascata

Quando um serviço é suscetível a falhas em cascata, existem vários distúrbios possíveis que podem iniciar o efeito dominó. Esta seção identifica alguns dos fatores que desencadeiam falhas em cascata.

Processo de morte

Algumas tarefas do servidor podem morrer, reduzindo a quantidade de capacidade disponível. As tarefas podem morrer devido a uma consulta de morte (um RPC cujo conteúdo desencadeia uma falha no processo), problemas de cluster, falhas de asserção ou vários outros motivos. Um evento muito pequeno (por exemplo, algumas falhas ou tarefas reprogramadas para outras máquinas) pode fazer com que um serviço prestes a cair seja interrompido.

Atualizações de

processo O push de uma nova versão do binário ou a atualização de sua configuração pode iniciar uma falha em cascata se um grande número de tarefas for afetado simultaneamente. Para evitar esse cenário, considere a sobrecarga de capacidade necessária ao configurar a infraestrutura de atualização do serviço ou empurre fora do pico. Ajustar dinamicamente o número de atualizações de tarefas em voo com base no volume de solicitações e na capacidade disponível pode ser uma abordagem viável.

Novos lançamentos

Um novo binário, alterações de configuração ou uma alteração na pilha de infraestrutura subjacente podem resultar em alterações nos perfis de solicitação, uso e limites de recursos, back-ends ou vários outros componentes do sistema que podem desencadear uma falha em cascata.

Durante uma falha em cascata, geralmente é aconselhável verificar as alterações recentes e revertê-las, principalmente se essas alterações afetaram a capacidade ou alteraram o perfil da solicitação.

Seu serviço deve implementar algum tipo de registro de alterações, que pode ajudar a identificar rapidamente as alterações recentes.

Crescimento orgânico

Em muitos casos, uma falha em cascata não é desencadeada por uma mudança de serviço específica, mas porque um crescimento no uso não foi acompanhado por um ajuste de capacidade.

Mudanças, drenos ou desligamentos planejados Se

seu serviço for multihomed, parte de sua capacidade pode estar indisponível devido a manutenção ou interrupções em um cluster. Da mesma forma, uma das dependências críticas do serviço pode ser drenada, resultando em uma redução na capacidade do serviço upstream devido a dependências de drenagem, ou um aumento na latência devido à necessidade de enviar as solicitações para um cluster mais distante.

Solicitar alterações no

perfil Um serviço de back-end pode receber solicitações de clusters diferentes porque um serviço de front-end mudou seu tráfego devido a alterações na configuração do平衡amento de carga, alterações na combinação de tráfego ou plenitude do cluster. Além disso, o custo médio para lidar com uma carga útil individual pode ter mudado devido a alterações no código do front-end ou na configuração. Da mesma forma, os dados tratados pelo serviço podem ter mudado organicamente devido ao uso crescente ou diferente pelos usuários existentes: por exemplo, o número e o tamanho das imagens, por usuário, para um serviço de armazenamento de fotos tendem a aumentar com o tempo.

Limites de recursos

Alguns sistemas operacionais de cluster permitem o comprometimento excessivo de recursos. CPU é um recurso fungível; muitas vezes, algumas máquinas têm uma certa quantidade de CPU disponível, o que fornece uma rede de segurança contra picos de CPU. A disponibilidade dessa CPU de folga difere entre as células e também entre as máquinas dentro da célula.

Depender dessa CPU frouxa como sua rede de segurança é perigoso. Sua disponibilidade depende inteiramente do comportamento dos outros trabalhos no cluster, portanto, ele pode cair repentinamente a qualquer momento. Por exemplo, se uma equipe iniciar um MapReduce que consome muita CPU e agenda em muitas máquinas, a quantidade agregada de CPU de folga pode diminuir repentinamente e acionar condições de fome de CPU para trabalhos não relacionados.

Ao realizar testes de carga, certifique-se de permanecer dentro dos limites de recursos confirmados.

Teste de falhas em cascata

As maneiras específicas pelas quais um serviço falhará podem ser muito difíceis de prever a partir dos primeiros princípios. Esta seção discute estratégias de teste que podem detectar se os serviços são suscetíveis a falhas em cascata.

Você deve testar seu serviço para determinar como ele se comporta sob carga pesada para ganhar confiança de que ele não entrará em uma falha em cascata em várias circunstâncias.

Teste até a falha e além Entender o

comportamento do serviço sob carga pesada talvez seja o primeiro passo mais importante para evitar falhas em cascata. Saber como seu sistema se comporta quando está sobrecarregado ajuda a identificar quais tarefas de engenharia são as mais importantes para correções de longo prazo; no mínimo, esse conhecimento pode ajudar a inicializar o processo de depuração para engenheiros de plantão quando surgir uma emergência.

Carregue os componentes de teste até que eles quebrem. À medida que a carga aumenta, um componente normalmente trata as solicitações com sucesso até atingir um ponto em que não pode mais lidar com mais solicitações. Nesse ponto, o componente deve começar a fornecer erros ou resultados degradados em resposta a uma carga adicional, mas não reduzir significativamente a taxa na qual ele trata as solicitações com êxito. Um componente que é altamente suscetível a uma falha em cascata começará a travar ou apresentar uma taxa muito alta de erros quando estiver sobrecarregado; um componente melhor projetado será capaz de rejeitar algumas solicitações e sobreviver.

O teste de carga também revela onde está o ponto de ruptura, conhecimento fundamental para o processo de planejamento de capacidade. Ele permite que você teste regressões, provisione os limites do pior caso e negocie a utilização versus as margens de segurança.

Devido aos efeitos do cache, aumentar gradualmente a carga pode gerar resultados diferentes do que aumentar imediatamente para os níveis de carga esperados. Portanto, considere testar os padrões de carga gradual e de impulso.

Você também deve testar e entender como o componente se comporta ao retornar à carga nominal depois de ter sido empurrado muito além dessa carga. Esses testes podem responder a perguntas como:

- Se um componente entrar em modo degradado com carga pesada, ele é capaz de sair do modo degradado sem intervenção humana?
- Se alguns servidores falharem sob carga pesada, quanto a carga precisa cair para que o sistema se estabilize?

Se você estiver testando a carga de um serviço com estado ou de um serviço que emprega armazenamento em cache, seu teste de carga deve rastrear o estado entre várias interações e verificar a correção em carga alta, que geralmente é onde ocorrem bugs sutis de simultaneidade.

Lembre-se de que componentes individuais podem ter pontos de ruptura diferentes, portanto, teste cada componente separadamente. Você não saberá de antemão qual componente pode atingir a parede primeiro e deseja saber como seu sistema se comportará quando isso acontecer.

Se você acredita que seu sistema possui proteções adequadas contra sobrecarga, considere realizar testes de falha em uma pequena fatia da produção para encontrar o ponto em que os componentes do sistema falham sob tráfego real. Esses limites podem não ser refletidos adequadamente pelo tráfego de teste de carga sintético, portanto, os testes de tráfego reais podem fornecer resultados mais realistas do que os testes de carga, com o risco de causar problemas visíveis ao usuário. Tenha cuidado ao testar em tráfego real: certifique-se de ter capacidade extra disponível caso suas proteções automáticas não funcionem e você precise fazer failover manualmente. Você pode considerar alguns dos seguintes testes de produção:

- Reduzir a contagem de tarefas de forma rápida ou lenta ao longo do tempo, além do tráfego esperado padrões
- Perdendo rapidamente a capacidade de um cluster •

Inibindo vários back-ends

Teste clientes populares

Entenda como clientes grandes usam seu serviço. Por exemplo, você quer saber se os clientes:

- Pode enfileirar o trabalho enquanto o serviço está inativo • Usar recuo exponencial aleatório em erros

- São vulneráveis a acionadores externos que podem criar grandes quantidades de carga (por exemplo, uma atualização de software acionada externamente pode limpar o cache de um cliente offline)

Dependendo do seu serviço, você pode ou não estar no controle de todo o código do cliente que fala com o seu serviço. No entanto, ainda é uma boa ideia entender como os clientes grandes que interagem com seu serviço se comportarão.

Os mesmos princípios se aplicam a grandes clientes internos. Teste as falhas do sistema com os maiores clientes para ver como eles reagem. Pergunte aos clientes internos como eles acessam seu serviço e quais mecanismos eles usam para lidar com falhas de back-end.

Testar back-ends não críticos

Teste seus back-ends não críticos e certifique-se de que a indisponibilidade deles não interfira nos componentes críticos do seu serviço.

Por exemplo, suponha que seu front-end tenha back-ends críticos e não críticos. Muitas vezes, uma determinada solicitação inclui componentes críticos (por exemplo, resultados da consulta) e componentes não críticos (por exemplo, sugestões de ortografia). Suas solicitações podem diminuir significativamente e consumir recursos aguardando a conclusão de back-ends não críticos.

Além de testar o comportamento quando o back-end não crítico estiver indisponível, teste como o front-end se comporta se o back-end não crítico nunca responder (por exemplo, se houver solicitações de blackholing). Os back-ends anunciados como não críticos ainda podem causar problemas nos front-ends quando as solicitações têm prazos longos. O front-end não deve começar a rejeitar muitas solicitações, ficar sem recursos ou servir com latência muito alta quando um back-end não crítico entra em buracos.

Etapas imediatas para solucionar falhas em cascata

Depois de identificar que seu serviço está passando por uma falha em cascata, você pode usar algumas estratégias diferentes para remediar a situação – e, é claro, uma falha em cascata é uma boa oportunidade para usar seu protocolo de gerenciamento de incidentes ([Capítulo 14](#)).

Aumentar recursos

Se o seu sistema estiver funcionando com capacidade reduzida e você tiver recursos ociosos, adicionar tarefas pode ser a maneira mais conveniente de se recuperar da interrupção. No entanto, se o serviço entrou em algum tipo de espiral de morte, adicionar mais recursos pode não ser suficiente para se recuperar.

Interromper falhas/mortes de verificação de

integridade Alguns sistemas de agendamento de cluster, como o Borg, verificam a integridade de tarefas em um trabalho e reiniciam tarefas que não estão íntegras. Essa prática pode criar um modo de falha no qual a própria verificação de integridade torna o serviço não íntegro. Por exemplo, se metade das tarefas não puder realizar nenhum trabalho porque estão sendo iniciadas e a outra metade será eliminada em breve porque estão sobrecarregadas e falhando nas verificações de integridade, desabilitar temporariamente as verificações de integridade pode permitir que o sistema se estabilize até todas as tarefas estão em execução.

A verificação da integridade do processo (“este binário está respondendo?”) e a verificação da integridade do serviço (“esse binário é capaz de responder a essa classe de solicitações agora?”) são duas operações conceitualmente distintas. A verificação de integridade do processo é relevante para o agendador de cluster, enquanto a verificação de integridade do serviço é relevante para o平衡ador de carga. A distinção clara entre os dois tipos de verificações de integridade pode ajudar a evitar esse cenário.

Reiniciar servidores

Se os servidores estiverem de alguma forma bloqueados e não estiverem progredindo, reiniciá-los pode ajudar. Tente reiniciar os servidores quando:

- Os servidores Java estão em uma espiral de morte do GC • Algumas solicitações em andamento não têm prazos, mas estão consumindo recursos, levando-os a bloquear threads, por exemplo
- Os servidores estão travados

Certifique-se de identificar a origem da falha em cascata antes de reiniciar seus servidores. Certifique-se de que essa ação não irá simplesmente deslocar a carga. Canário esta mudança, e fazê-lo lentamente. Suas ações podem ampliar uma falha em cascata existente se a interrupção for realmente devido a um problema como um cache frio.

Drop Tra c A

queda de carga é um grande martelo, geralmente reservado para situações em que você tem uma verdadeira falha em cascata em suas mãos e não pode consertá-la por outros meios. Por exemplo, se uma carga pesada fizer com que a maioria dos servidores falhe assim que eles se tornarem íntegros, você poderá colocar o serviço em funcionamento novamente:

1. Abordando a condição de acionamento inicial (adicionando capacidade, por exemplo).
2. Reduzir a carga o suficiente para que o travamento pare. Considere ser agressivo aqui — se todo o serviço estiver em loop de falha, permita apenas, digamos, 1% do tráfego.
3. Permitindo que a maioria dos servidores se tornem íntegros.
4. Aumentar gradualmente a carga.

Essa estratégia permite o aquecimento dos caches, o estabelecimento de conexões etc., antes que a carga retorne aos níveis normais.

Obviamente, essa tática causará muitos danos visíveis ao usuário. Se você pode ou não (ou se deve) descartar o tráfego indiscriminadamente depende de como o serviço está configurado. Se você tiver algum mecanismo para descartar tráfego menos importante (por exemplo, pré-busca), use esse mecanismo primeiro.

É importante ter em mente que essa estratégia permite que você se recupere de uma interrupção em cascata assim que o problema subjacente for corrigido. Se o problema que iniciou a falha em cascata não for corrigido (por exemplo, capacidade global insuficiente), a falha em cascata pode ocorrer logo após o retorno de todo o tráfego. Portanto, antes de usar essa estratégia, considere corrigir (ou pelo menos ocultar) a causa raiz ou a condição desencadeante. Por exemplo, se o serviço ficou sem memória e agora está em uma espiral de morte, adicionar mais memória ou tarefas deve ser o primeiro passo.

Entre em Modos Degradados

Exiba resultados degradados fazendo menos trabalho ou eliminando tráfego sem importância. Essa estratégia deve ser projetada em seu serviço e pode ser implementada somente se você souber qual tráfego pode ser degradado e tiver a capacidade de diferenciar entre as várias cargas úteis.

Eliminar a carga em lote

Alguns serviços têm carga importante, mas não crítica. Considere desligar essas fontes de carga. Por exemplo, se atualizações de índice, cópias de dados ou coleta de estatísticas consumirem recursos do caminho de serviço, considere desligar essas fontes de carga durante uma interrupção.

Elimine o Mau Traç

Se algumas consultas estiverem criando carga pesada ou travamentos (por exemplo, consultas de morte), considere bloqueá-las ou eliminá-las por outros meios.

Falha em cascata e Shakespeare Um

documentário sobre as obras de Shakespeare vai ao ar no Japão e aponta explicitamente para o nosso serviço Shakespeare como um excelente lugar para realizar pesquisas adicionais. Após a transmissão, o tráfego para nosso datacenter asiático aumenta além da capacidade do serviço. Esse problema de capacidade é ainda agravado por uma grande atualização do serviço Shakespeare que ocorre simultaneamente nesse datacenter.

Felizmente, existem várias salvaguardas que ajudam a mitigar o potencial de falha. O processo de Revisão de Prontidão de Produção identificou alguns problemas que a equipe já abordou. Por exemplo, os desenvolvedores criaram uma degradação graciosa no serviço. À medida que a capacidade se torna escassa, o serviço não retorna mais imagens ao lado de texto ou pequenos mapas que ilustram onde uma história acontece. E dependendo de sua finalidade, um RPC que expira ou não é repetido (por exemplo, no caso das imagens acima mencionadas) ou é repetido com uma retirada exponencial aleatória.

Apesar dessas salvaguardas, as tarefas falham uma a uma e são reiniciadas pelo Borg, o que reduz ainda mais o número de tarefas de trabalho.

Como resultado, alguns gráficos no painel de serviço ficam com um tom alarmante de vermelho e o SRE é paginado. Em resposta, os SREs adicionam temporariamente capacidade ao datacenter asiático aumentando o número de tarefas disponíveis para o trabalho de Shakespeare. Ao fazer isso, eles podem restaurar o serviço Shakespeare no cluster asiático.

Depois, a equipe de SRE escreve uma autópsia detalhando a cadeia de eventos, o que correu bem, o que poderia ter corrido melhor e vários itens de ação para evitar que esse cenário ocorra novamente. Por exemplo, no caso de uma sobrecarga de serviço, o平衡ador de carga GSLB redirecionará parte do tráfego para datacenters vizinhos. Além disso, a equipe do SRE ativa o escalonamento automático, para que o número de tarefas aumente automaticamente com o tráfego, para que eles não precisem se preocupar com esse tipo de problema novamente.

Observações Finais

Quando os sistemas estão sobrecarregados, algo precisa dar para remediar a situação. Depois que um serviço passa do ponto de interrupção, é melhor permitir que alguns erros visíveis ao usuário ou resultados de qualidade inferior passem do que tentar atender totalmente a todas as solicitações. Entender onde estão esses pontos de interrupção e como o sistema se comporta além deles é fundamental para os proprietários de serviços que desejam evitar falhas em cascata.

Sem os devidos cuidados, algumas mudanças no sistema destinadas a reduzir erros de fundo ou melhorar o estado estável podem expor o serviço a um risco maior de interrupção total. Tentar novamente em caso de falhas, deslocar a carga de servidores não saudáveis, eliminar servidores insalubres, adicionar caches para melhorar o desempenho ou reduzir a latência: tudo isso pode ser implementado para melhorar o caso normal, mas pode aumentar a chance de causar uma falha em grande escala . Tenha cuidado ao avaliar as alterações para garantir que uma interrupção não seja trocada por outra.

CAPÍTULO 23

Gerenciando o estado crítico: distribuído Consenso para Confiabilidade

**Escrito por Laura Nolan
Editado por Tim Harvey**

Os processos falham ou podem precisar ser reiniciados. Os discos rígidos falham. Desastres naturais podem destruir vários datacenters em uma região. Os engenheiros de confiabilidade do site precisam antecipar esses tipos de falhas e desenvolver estratégias para manter os sistemas funcionando apesar delas. Essas estratégias geralmente envolvem a execução desses sistemas em vários sites. Distribuir geograficamente um sistema é relativamente simples, mas também introduz a necessidade de manter uma visão consistente do estado do sistema, que é uma tarefa mais complexa e difícil.

Grupos de processos podem querer concordar de forma confiável em questões como:

- Qual processo é o líder de um grupo de processos? • Qual é o conjunto de processos em um grupo? • Uma mensagem foi confirmada com sucesso em uma fila distribuída? • Um processo possui um arrendamento ou não? • O que é um valor em um armazenamento de dados para uma determinada chave?

Descobrimos que o consenso distribuído é eficaz na construção de sistemas confiáveis e altamente disponíveis que exigem uma visão consistente de algum estado do sistema. O problema do consenso distribuído trata de chegar a um acordo entre um grupo de processos conectados por uma rede de comunicação não confiável. Por exemplo, vários processos em um sistema distribuído podem precisar ser capazes de formar uma visão consistente de uma parte crítica da configuração, se um bloqueio distribuído é mantido ou não, ou se uma mensagem em uma fila foi processada. É um dos conceitos mais fundamentais em comunicação distribuída.

puting e um em que confiamos para praticamente todos os serviços que oferecemos. A Figura 23-1 ilustra um modelo simples de como um grupo de processos pode obter uma visão consistente do estado do sistema por meio de um consenso distribuído.

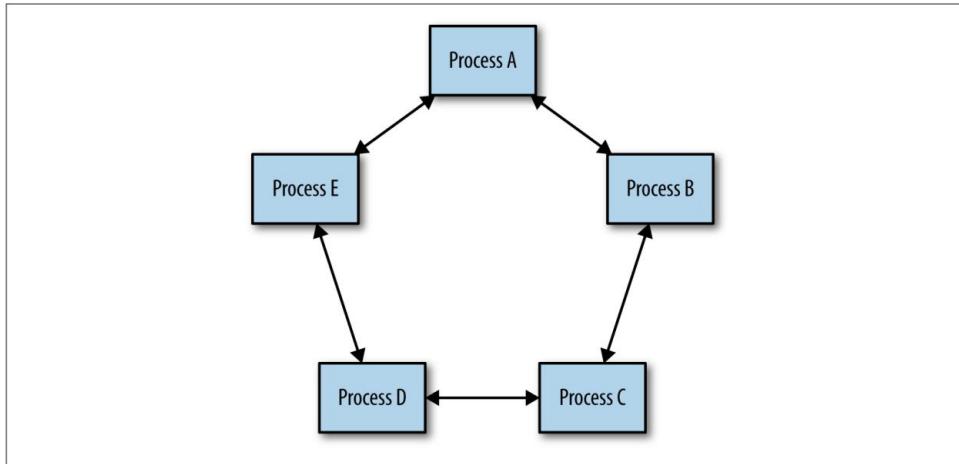


Figura 23-1. Consenso distribuído: acordo entre um grupo de processos

Sempre que você vir a eleição do líder, estado compartilhado crítico ou bloqueio distribuído, recomendamos o uso de sistemas de consenso distribuído que foram formalmente comprovados e testados exaustivamente. Abordagens informais para resolver esse problema podem levar a interrupções e, mais insidiosamente, a problemas de consistência de dados sutis e difíceis de corrigir que podem prolongar interrupções em seu sistema desnecessariamente.

Teorema CAP

O teorema CAP ([\[Fox99\]](#), [\[Bre12\]](#)) sustenta que um sistema distribuído não pode ter simultaneamente todas as três propriedades a seguir:

- Visualizações consistentes dos dados em cada nó
- Disponibilidade dos dados em cada nó •

Tolerância a partições de rede [\[Gil02\]](#)

A lógica é intuitiva: se dois nós não podem se comunicar (porque a rede é particionada), então o sistema como um todo pode parar de atender algumas ou todas as solicitações em alguns ou todos os nós (reduzindo assim a disponibilidade), ou pode atender solicitações como de costume, o que resulta em visualizações inconsistentes dos dados em cada nó.

Como as partições de rede são inevitáveis (cabos cortados, pacotes perdidos ou atrasados devido a congestionamentos, quebras de hardware, componentes de rede mal configurados etc.), entender o consenso distribuído realmente equivale a entender como a consistência e a disponibilidade funcionam para seu aplicativo específico. As pressões comerciais geralmente exigem altos níveis de disponibilidade e muitos aplicativos exigem visualizações consistentes de seus dados.

Engenheiros de sistemas e software geralmente estão familiarizados com a semântica tradicional de armazenamento de dados ACID (Atomicidade, Consistência, Isolamento e Durabilidade), mas um número crescente de tecnologias de armazenamento de dados distribuído fornece um conjunto diferente de semântica conhecido como BASE (Basicamente Disponível, Soft state, e Consistência eventual). Datastores que suportam a semântica BASE têm aplicações úteis para certos tipos de dados e podem lidar com grandes volumes de dados e transações que seriam muito mais caras, e talvez totalmente inviáveis, com datastores que suportam a semântica ACID.

A maioria desses sistemas que suportam a semântica BASE depende da replicação multimaster, onde as gravações podem ser confirmadas para diferentes processos simultaneamente, e há algum mecanismo para resolver conflitos (geralmente tão simples quanto “o último timestamp vence”). Essa abordagem é geralmente conhecida como consistência eventual. No entanto, a consistência eventual pode levar a resultados surpreendentes [Lu15], principalmente no caso de clock drift (o que é inevitável em sistemas distribuídos) ou particionamento de rede [Kin15].¹

Também é difícil para os desenvolvedores projetar sistemas que funcionem bem com datastores que suportem apenas a semântica BASE. Jeff Shute [Shu13], por exemplo, afirmou: “descobrimos que os desenvolvedores gastam uma fração significativa de seu tempo construindo mecanismos extremamente complexos e propensos a erros para lidar com a consistência eventual e lidar com dados que podem estar desatualizados. Achamos que esse é um fardo inaceitável para os desenvolvedores e que os problemas de consistência devem ser resolvidos no nível do banco de dados.”

Os projetistas de sistemas não podem sacrificar a correção para obter confiabilidade ou desempenho, particularmente em torno do estado crítico. Por exemplo, considere um sistema que lida com transações financeiras: requisitos de confiabilidade ou desempenho não fornecem muito valor se os dados financeiros não estiverem corretos. Os sistemas precisam ser capazes de sincronizar de forma confiável o estado crítico em vários processos. Algoritmos de consenso distribuído fornecem essa funcionalidade.

¹ Kyle Kingsbury escreveu uma extensa série de artigos sobre correção de sistemas distribuídos, que contém muitos exemplos de comportamento inesperado e incorreto nesses tipos de armazenamentos de dados. Consulte <https://aphyr.com/tags/jepsen>.

Motivando o Uso do Consenso: Sistemas Distribuídos Falha de coordenação

Os sistemas distribuídos são complexos e sutis para entender, monitorar e solucionar problemas. Os engenheiros que executam esses sistemas são frequentemente surpreendidos pelo comportamento na presença de falhas. As falhas são eventos relativamente raros e não é uma prática comum testar sistemas nessas condições. É muito difícil raciocinar sobre o comportamento do sistema durante falhas. As partições de rede são particularmente desafiadoras - um problema que parece ser causado por uma partição completa pode ser o resultado de:

- Uma rede muito lenta •

Algumas, mas não todas as mensagens sendo descartadas • Aceleração ocorrendo em uma direção, mas não na outra

As seções a seguir fornecem exemplos de problemas que ocorreram em sistemas distribuídos do mundo real e discutem como a eleição de líderes e algoritmos de consenso distribuído podem ser usados para evitar tais problemas.

Estudo de caso 1: O problema do cérebro dividido

Um serviço é um repositório de conteúdo que permite a colaboração entre vários usuários. Ele usa conjuntos de dois servidores de arquivos replicados em diferentes racks para confiabilidade. O serviço precisa evitar gravar dados simultaneamente em ambos os servidores de arquivos em um conjunto, pois isso pode resultar em corrupção de dados (e possivelmente dados irrecuperáveis).

Cada par de servidores de arquivos tem um líder e um seguidor. Os servidores monitoram uns aos outros por meio de pulsações. Se um servidor de arquivos não puder contatar seu parceiro, ele emite um comando STONITH (Atire no outro nó na cabeça) para seu nó parceiro para encerrar o nó e, em seguida, assume o controle de seus arquivos. Essa prática é um método padrão da indústria para reduzir instâncias de cérebro dividido, embora, como veremos, seja conceitualmente infundado.

O que acontece se a rede ficar lenta ou começar a descartar pacotes? Nesse cenário, os servidores de arquivos excedem seus tempos limite de pulsação e, conforme projetado, enviam comandos STONITH para seus nós parceiros e assumem o controle. No entanto, alguns comandos podem não ser entregues devido à rede comprometida. Os pares de servidores de arquivos agora podem estar em um estado no qual ambos os nós devem estar ativos para o mesmo recurso ou em que ambos estão inativos porque ambos emitiram e receberam comandos STONITH. Isso resulta em corrupção ou indisponibilidade de dados.

O problema aqui é que o sistema está tentando resolver um problema de eleição de líder usando tempos limite simples. A eleição do líder é uma reformulação do problema do consenso assíncrono distribuído, que não pode ser resolvido corretamente usando pulsações.

Estudo de caso 2: Failover requer intervenção humana

sistema de banco de dados altamente fragmentado tem um primário para cada fragmento, que replica sincronamente para um secundário em outro datacenter. Um sistema externo verifica a integridade das primárias e, se elas não estiverem mais íntegras, promove a secundária para primária. Se o primário não puder determinar a saúde de seu secundário, ele se torna indisponível e escala para um humano para evitar o cenário de cérebro dividido visto no Estudo de Caso 1.

Essa solução não apresenta risco de perda de dados, mas afeta negativamente a disponibilidade de dados. Também aumenta desnecessariamente a carga operacional dos engenheiros que executam o sistema, e a intervenção humana é escalonada. Esse tipo de evento, em que um primário e um secundário têm problemas de comunicação, é altamente provável de ocorrer no caso de um problema de infraestrutura maior, quando os engenheiros de resposta já podem estar sobrecarregados com outras tarefas. Se a rede é tão afetada que um sistema de consenso distribuído não pode eleger um mestre, um humano provavelmente não está melhor posicionado para fazê-lo.

Estudo de caso 3: Algoritmos de associação de grupo

defeituosos Um sistema tem um componente que executa serviços de indexação e pesquisa. Ao iniciar, os nós usam um protocolo de fofocas para descobrir uns aos outros e ingressar no cluster. O cluster elege um líder, que realiza a coordenação. No caso de uma partição de rede que divide o cluster, cada lado (incorrectamente) elege um mestre e aceita gravações e exclusões, levando a um cenário de divisão cerebral e corrupção de dados.

O problema de determinar uma visão consistente da associação de grupo em um grupo de processos é outra instância do problema de consenso distribuído.

De fato, muitos problemas de sistemas distribuídos são versões diferentes de consenso distribuído, incluindo eleição de mestre, associação de grupo, todos os tipos de bloqueio e concessão distribuídos, enfileiramento e mensagens distribuídas confiáveis e manutenção de qualquer tipo de rede compartilhada crítica. estado que deve ser visto de forma consistente em um grupo de processos. Todos esses problemas devem ser resolvidos apenas usando algoritmos de consenso distribuído que foram provados formalmente corretos e cujas implementações foram testadas extensivamente. Meios ad hoc de resolver esses tipos de problemas (como batimentos cardíacos e protocolos de fofocas) sempre terão problemas de confiabilidade na prática.

Como funciona o consenso distribuído

O problema do consenso tem múltiplas variantes. Ao lidar com sistemas de software distribuídos, estamos interessados no consenso distribuído assíncrono, que se aplica a ambientes com atrasos potencialmente ilimitados na passagem de mensagens. (O consenso síncrono se aplica a sistemas de tempo real, nos quais hardware dedicado significa que as mensagens sempre serão passadas com garantias de tempo específicas.)

Os algoritmos de consenso distribuídos podem ser de falha de falha (que pressupõe que os nós com falha nunca retornam ao sistema) ou recuperação de falha. Os algoritmos de recuperação de falhas são muito mais úteis, porque a maioria dos problemas em sistemas reais são transitórios por natureza devido a uma rede lenta, reinicializações e assim por diante.

Os algoritmos podem lidar com falhas bizantinas ou não bizantinas. A falha bizantina ocorre quando um processo passa mensagens incorretas devido a um bug ou atividade maliciosa, e são comparativamente dispendiosos de manusear e encontrados com menos frequência.

Tecnicamente, resolver o problema de consenso distribuído assíncrono em tempo limitado é impossível. Conforme comprovado pelo resultado de impossibilidade FLP vencedor do Prêmio Dijkstra [Fis85], nenhum algoritmo de consenso distribuído assíncrono pode garantir progresso na presença de uma rede não confiável.

Na prática, abordamos o problema do consenso distribuído em tempo limitado, garantindo que o sistema tenha réplicas saudáveis e conectividade de rede suficientes para progredir de forma confiável na maior parte do tempo. Além disso, o sistema deve ter backoffs com atrasos aleatórios. Essa configuração evita que as tentativas causem efeitos em cascata e evita o problema dos proponentes de duelo descrito mais adiante neste capítulo.

Os protocolos garantem a segurança e a redundância adequada no sistema estimula a vivacidade.

A solução original para o problema do consenso distribuído foi o protocolo Paxos de Lamport [Lam98], mas existem outros protocolos que resolvem o problema, incluindo Raft [Ong14], Zab [Jun11] e Mencius [Mao08]. O próprio Paxos tem muitas variações destinadas a aumentar o desempenho [Zoo14]. Eles geralmente variam apenas em um único detalhe, como dar uma função de líder especial a um processo para simplificar o protocolo.

Visão Geral do Paxos: Um Exemplo de Protocolo O

Paxos funciona como uma sequência de propostas, que podem ou não ser aceitas pela maioria dos processos do sistema. Se uma proposta não for aceita, ela falhará. Cada proposta tem um número de sequência, que impõe uma ordenação estrita a todas as operações do sistema.

Na primeira fase do protocolo, o proponente envia um número de sequência aos aceitadores. Cada aceitante concordará em aceitar a proposta somente se ainda não tiver visto uma proposta com um número de sequência superior. Os proponentes podem tentar novamente com um número de sequência mais alto, se necessário. Os proponentes devem usar números de sequência exclusivos (desenhando de conjuntos disjuntos ou incorporando seu nome de host no número de sequência, por exemplo).

Se um proponente receber a concordância da maioria dos aceitantes, ele poderá confirmar a proposta enviando uma mensagem de confirmação com um valor.

A sequência estrita das propostas resolve quaisquer problemas relacionados com a ordenação das mensagens no sistema. A exigência de uma maioria para se comprometer significa que dois valores diferentes não podem ser comprometidos para a mesma proposta, porque quaisquer duas maiorias se sobrepõem em pelo menos um nó. Os aceitantes devem escrever um diário sobre armazenamento persistente sempre que concordarem em aceitar uma proposta, porque os aceitantes precisam honrar essas garantias após o reinício.

O Paxos por si só não é tão útil: tudo o que ele permite é concordar com um valor e um número de proposta uma vez. Como apenas um quorum de nós precisa concordar com um valor, qualquer nó pode não ter uma visão completa do conjunto de valores que foram acordados. Essa limitação é verdadeira para a maioria dos algoritmos de consenso distribuídos.

Padrões de Arquitetura de Sistema para Consenso Distribuído

Os algoritmos de consenso distribuído são de baixo nível e primitivos: eles simplesmente permitem que um conjunto de nós concorde com um valor, uma vez. Eles não mapeiam bem as tarefas de design reais. O que torna o consenso distribuído útil é a adição de componentes de sistema de nível superior, como armazenamentos de dados, armazenamentos de configuração, filas, bloqueio e serviços de eleição de líderes para fornecer a funcionalidade prática do sistema que os algoritmos de consenso distribuído não abordam. O uso de componentes de nível superior reduz a complexidade para projetistas de sistemas. Ele também permite que algoritmos de consenso distribuído subjacentes sejam alterados, se necessário, em resposta a mudanças no ambiente no qual o sistema é executado ou mudanças em requisitos não funcionais.

Muitos sistemas que usam algoritmos de consenso com sucesso, na verdade o fazem como clientes de algum serviço que implementa esses algoritmos, como Zookeeper, Consul e etcd.

O Zookeeper [Hun10] foi o primeiro sistema de consenso de código aberto a ganhar força na indústria porque era fácil de usar, mesmo com aplicativos que não foram projetados para usar o consenso distribuído. O serviço Chubby preenche um nicho semelhante no Google. Seus autores apontam [Bur06] que fornecer primitivas de consenso como um serviço, em vez de bibliotecas que os engenheiros criam em seus aplicativos, libera os mantenedores de aplicativos de ter que implantar seus sistemas de maneira compatível com um serviço de consenso altamente disponível (executando o número certo) de réplicas, lidar com a participação no grupo, lidar com o desempenho, etc.).

Máquinas de estado replicadas confiáveis Uma

máquina de estado replicada (RSM) é um sistema que executa o mesmo conjunto de operações, na mesma ordem, em vários processos. Os RSMs são o bloco de construção fundamental de componentes e serviços úteis de sistemas distribuídos, como armazenamento de dados ou configuração, bloqueio e eleição de líderes (descritos com mais detalhes posteriormente).

As operações em um RSM são ordenadas globalmente por meio de um algoritmo de consenso. Este é um conceito poderoso: vários artigos ([Agu10], [Kir08], [Sch90]) mostram que qualquer

O programa ministerio pode ser implementado como um serviço replicado de alta disponibilidade sendo implementado como um RSM.

Conforme mostrado na [Figura 23-2](#), as máquinas de estado replicadas são um sistema implementado em uma camada lógica acima do algoritmo de consenso. O algoritmo de consenso lida com o acordo sobre a sequência de operações, e o RSM executa as operações nessa ordem. Como nem todos os membros do grupo de consenso são necessariamente membros de cada quórum de consenso, os RSMs podem precisar sincronizar o estado dos pares. Conforme descrito por Kirsch e Amir [\[Kir08\]](#), você pode usar um protocolo de janela deslizante para reconciliar o estado entre processos pares em um RSM.

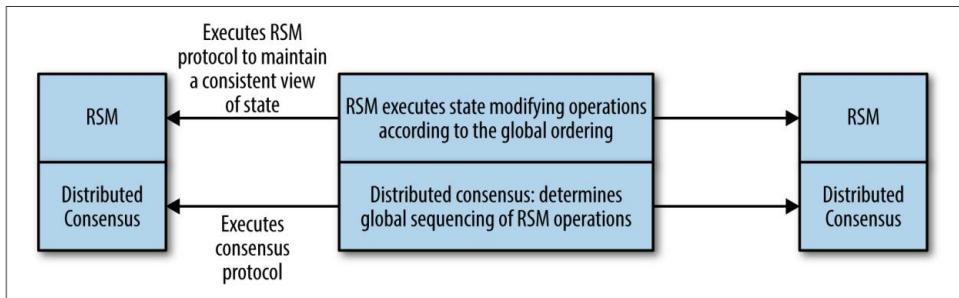


Figura 23-2. A relação entre algoritmos de consenso e máquinas de estado replicadas

Armazenamentos de dados replicados confiáveis e armazenamentos de

configuração Os armazenamentos de dados replicados confiáveis são uma aplicação de máquinas de estado replicadas. Datastóres replicados usam algoritmos de consenso no caminho crítico de seu trabalho. Assim, o desempenho, a taxa de transferência e a capacidade de escala são muito importantes nesse tipo de projeto. Assim como os armazenamentos de dados criados com outras tecnologias subjacentes, os armazenamentos de dados baseados em consenso podem fornecer uma variedade de semânticas de consistência para operações de leitura, o que faz uma grande diferença no dimensionamento do armazenamento de dados. Essas compensações são discutidas em “[Desempenho de consenso distribuído](#)” na página 296.

Outros sistemas (baseados em consenso não distribuído) geralmente dependem de carimbos de data/hora para fornecer limites sobre a idade dos dados retornados. Os carimbos de data/hora são altamente problemáticos em sistemas distribuídos porque é impossível garantir que os relógios sejam sincronizados em várias máquinas. Spanner [Cor12] aborda esse problema modelando a incerteza de pior caso envolvida e desacelerando o processamento quando necessário para resolver essa incerteza.

Processamento de Alta Disponibilidade Usando Eleição de Líder A eleição

de líder em sistemas distribuídos é um problema equivalente ao consenso distribuído. Serviços replicados que usam um único líder para realizar algum tipo específico de trabalho

no sistema são muito comuns; o mecanismo de líder único é uma forma de garantir a exclusão mútua em um nível grosso.

Esse tipo de design é apropriado quando o trabalho do líder de serviço pode ser realizado por um processo ou é fragmentado. Os projetistas de sistemas podem construir um serviço altamente disponível escrevendo-o como se fosse um programa simples, replicando esse processo e usando a eleição do líder para garantir que apenas um líder esteja trabalhando a qualquer momento (como mostrado na Figura 23-3). Muitas vezes, o trabalho do líder é coordenar algum grupo de trabalhadores no sistema. Esse padrão foi usado no GFS [Ghe03] (que foi substituído pelo Colossus) e no armazenamento de valores-chave do Bigtable [Cha06].

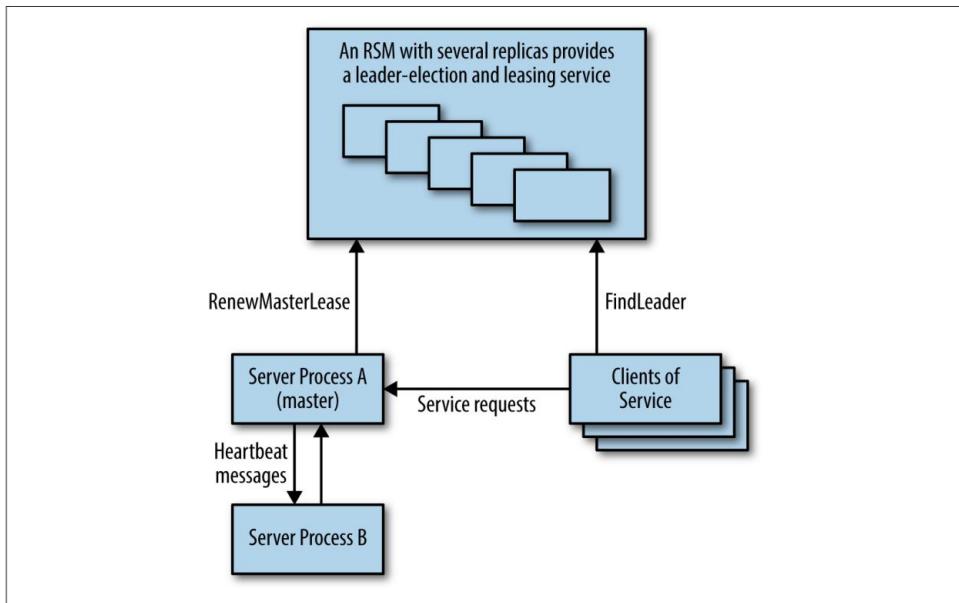


Figura 23-3. Sistema altamente disponível usando um serviço replicado para eleição de mestre

Nesse tipo de componente, diferentemente do armazenamento de dados replicado, o algoritmo de consenso não está no caminho crítico do trabalho principal que o sistema está realizando, portanto, a taxa de transferência geralmente não é uma grande preocupação.

Coordenação Distribuída e Serviços de Bloqueio Uma barreira em uma computação distribuída é uma primitiva que bloqueia um grupo de processos de prosseguir até que alguma condição seja atendida (por exemplo, até que todas as partes de uma fase de uma computação sejam concluídas). O uso de uma barreira divide efetivamente uma computação distribuída em fases lógicas. Por exemplo, como mostrado na Figura 23-4, uma barreira pode ser usada na implementação do modelo MapReduce [Dea04] para garantir que toda a fase Map seja concluída antes que a parte Reduzir do cálculo continue.

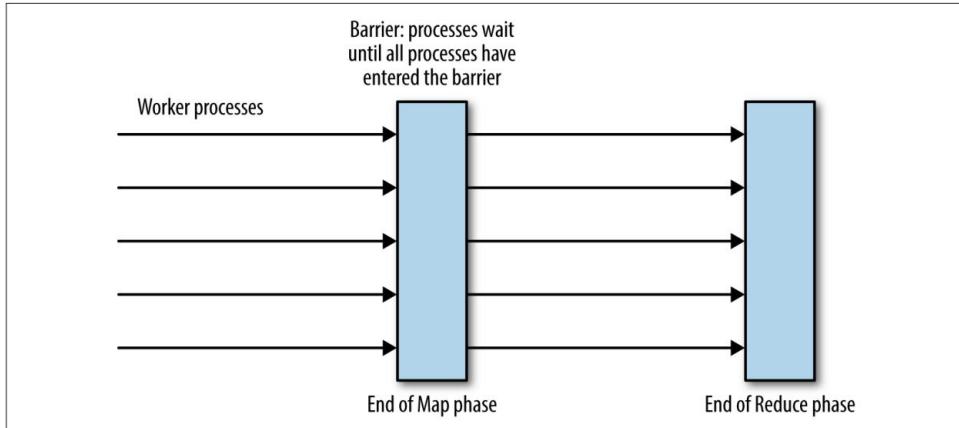


Figura 23-4. Barreiras para coordenação de processos na computação MapReduce

A barreira poderia ser implementada por um único processo coordenador, mas essa implementação adiciona um único ponto de falha que geralmente é inaceitável. A barreira também pode ser implementada como um RSM. O serviço de consenso Zookeeper pode implementar o padrão de barreira: veja [Hun10] e [Zoo14].

Os bloqueios são outra primitiva de coordenação útil que pode ser implementada como um RSM. Considere um sistema distribuído no qual os processos de trabalho consomem atomicamente alguns arquivos de entrada e gravam resultados. Bloqueios distribuídos podem ser usados para evitar que vários trabalhadores processem o mesmo arquivo de entrada. Na prática, é essencial usar concessões renováveis com timeouts em vez de bloqueios indefinidos, pois isso evita que os bloqueios sejam mantidos indefinidamente por processos que travam. O bloqueio distribuído está além do escopo deste capítulo, mas lembre-se de que os bloqueios distribuídos são uma primitiva de sistema de baixo nível que deve ser usada com cuidado. A maioria dos aplicativos deve usar um sistema de nível superior que forneça transações distribuídas.

Filas Distribuídas Confiáveis e Filas de Mensagens são uma

estrutura de dados comum, geralmente usada como uma forma de distribuir tarefas entre vários processos de trabalho.

Os sistemas baseados em filas podem tolerar falhas e perdas de nós de trabalho com relativa facilidade. No entanto, o sistema deve garantir que as tarefas reivindicadas sejam processadas com êxito. Para isso, recomenda-se um sistema de concessão (discutido anteriormente em relação aos bloqueios) em vez de uma remoção total da fila. A desvantagem dos sistemas baseados em filas é que a perda da fila impede que todo o sistema funcione. Implementar a fila como um RSM pode minimizar o risco e tornar todo o sistema muito mais robusto.

A difusão atômica é uma primitiva de sistemas distribuídos em que as mensagens são recebidas de forma confiável e na mesma ordem por todos os participantes. Este é um conceito de sistemas distribuídos incrivelmente poderoso e muito útil na concepção de sistemas práticos. Existe um grande número de infra-estruturas de mensagens de publicação-assinatura para uso de projetistas de sistemas, embora nem todas forneçam garantias atômicas.

Chandra e Toueg [Cha96] demonstram a equivalência de transmissão atômica e consenso.

O padrão de enfileiramento como distribuição de trabalho, que usa a fila como um dispositivo de平衡amento de carga, conforme mostrado na [Figura 23-5](#), pode ser considerado um sistema de mensagens ponto a ponto.

Os sistemas de mensagens geralmente também implementam uma fila de publicação-assinatura, onde as mensagens podem ser consumidas por muitos clientes que se inscrevem em um canal ou tópico. Nesse caso de um para muitos, as mensagens na fila são armazenadas como uma lista ordenada persistente.

Os sistemas de publicação-assinatura podem ser usados para muitos tipos de aplicativos que exigem que os clientes se inscrevam para receber notificações de algum tipo de evento. Os sistemas de publicação-assinatura também podem ser usados para implementar caches distribuídos coerentes.

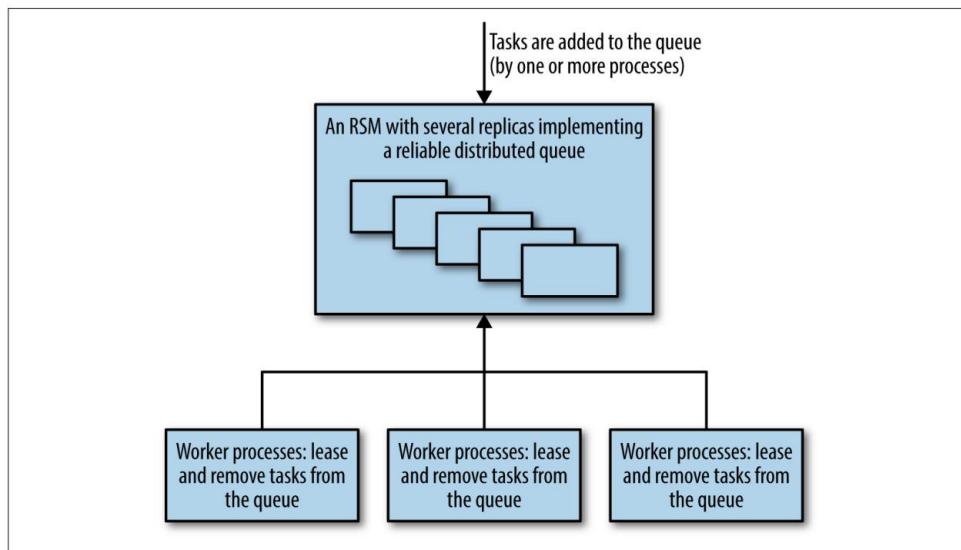


Figura 23-5. Um sistema de distribuição de trabalho orientado a filas usando um componente de filas baseado em consenso confiável

Os sistemas de enfileiramento e mensagens geralmente precisam de excelente taxa de transferência, mas não precisam de latência extremamente baixa (devido ao fato de raramente serem diretamente voltados para o usuário). No entanto, latências muito altas em um sistema como o que acabamos de descrever, que tem vários trabalhadores solicitando tarefas de uma fila, podem se tornar um problema se a porcentagem de tempo de processamento de cada tarefa aumentar significativamente.

Desempenho de consenso distribuído

A sabedoria convencional geralmente sustenta que os algoritmos de consenso são muito lentos e caros para usar em muitos sistemas que exigem alto rendimento e baixa latência [Bol11].

Essa concepção simplesmente não é verdadeira - embora as implementações possam ser lentas, há vários truques que podem melhorar o desempenho. Os algoritmos de consenso distribuído estão no centro de muitos dos sistemas críticos do Google, descritos em [Ana13], [Bur06], [Cor12] e [Shu13], e provaram ser extremamente eficazes na prática. A escala do Google não é uma vantagem aqui: na verdade, nossa escala é mais uma desvantagem porque apresenta dois desafios principais: nossos conjuntos de dados tendem a ser grandes e nossos sistemas funcionam em uma grande distância geográfica. Conjuntos de dados maiores multiplicados por várias réplicas representam custos de computação significativos e distâncias geográficas maiores aumentam a latência entre as réplicas, o que, por sua vez, reduz o desempenho.

Não há um consenso distribuído “melhor” e algoritmo de replicação de máquina de estado para desempenho, porque o desempenho depende de vários fatores relacionados à carga de trabalho, objetivos de desempenho do sistema e como o sistema deve ser implantado. Seções apresentam pesquisas, com o objetivo de aumentar a compreensão do que é possível alcançar com o consenso distribuído, muitos dos sistemas descritos estão disponíveis e estão em uso agora.

As cargas de trabalho podem variar de várias maneiras e entender como elas podem variar é fundamental para discutir o desempenho. No caso de um sistema de consenso, a carga de trabalho pode variar em termos de:

- Taxa de transferência: o número de propostas sendo feitas por unidade de tempo no pico de carga
- O tipo de solicitações: proporção de operações que mudam de estado
- A semântica de consistência necessária para operações de leitura
- Tamanhos de solicitação, se o tamanho da carga de dados puder variar

As estratégias de implantação também variam. Por exemplo:

- A implantação é local ou ampla?
- Que tipos de quórum são usados e onde estão a maioria dos processos?
- O sistema usa sharding, pipelining e batching?

Muitos sistemas de consenso usam um processo de líder distinto e exigem que todas as solicitações sejam direcionadas a esse nó especial. Como mostrado na Figura 23-6, como resultado, o desempenho do sistema percebido pelos clientes em diferentes localizações geográficas pode variar consideravelmente.

2 Em particular, o desempenho do algoritmo original do Paxos não é ideal, mas foi bastante aprimorado nos anos.

bly, simplesmente porque nós mais distantes têm tempos de ida e volta mais longos para o processo líder.

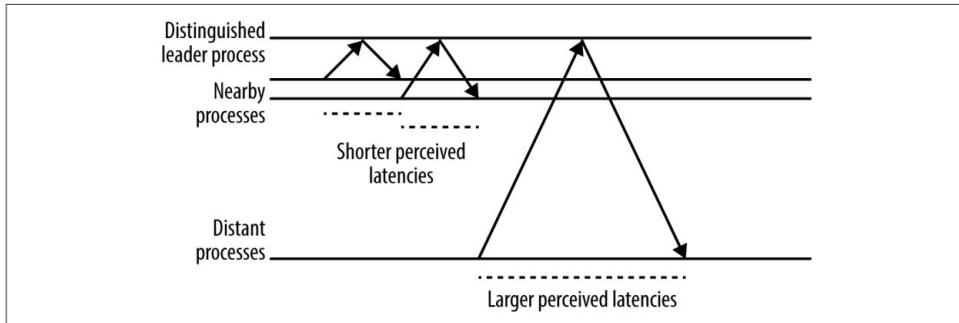


Figura 23-6. O efeito da distância de um processo do servidor na latência percebida no cliente

Multi-Paxos: Fluxo de Mensagens Detalhado O

protocolo Multi-Paxos usa um processo de líder forte: a menos que um líder ainda não tenha sido eleito ou ocorra alguma falha, é necessário apenas uma viagem de ida e volta do proponente a um quórum de aceitantes para chegar a um consenso. O uso de um processo líder forte é ótimo em termos do número de mensagens a serem passadas e é típico de muitos protocolos de consenso.

A [Figura 23-7](#) mostra um estado inicial com um novo proponente executando a primeira fase Preparar/ Prometer do protocolo. A execução desta fase estabelece uma nova visão numerada, ou termo líder. Nas execuções subsequentes do protocolo, enquanto a visão permanece a mesma, a primeira fase é desnecessária porque o proponente que estabeleceu a visão pode simplesmente enviar mensagens de Aceitação , e o consenso é alcançado quando um quorum de respostas é recebido (incluindo o próprio proponente).

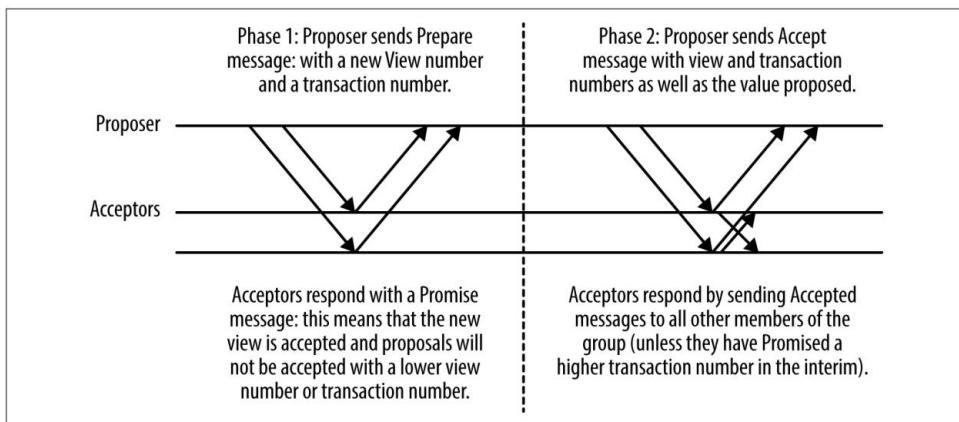


Figura 23-7. Fluxo de mensagens Multi-Paxos básico

Outro processo no grupo pode assumir o papel de proponente para propor mensagens a qualquer momento, mas mudar o proponente tem um custo de desempenho. Ela exige a viagem de ida e volta extra para executar a Fase 1 do protocolo, mas, mais importante, pode causar uma situação de duelo de proponentes na qual as propostas se interrompem repetidamente e nenhuma proposta pode ser aceita, conforme mostrado na [Figura 23-8](#). Como esse cenário é uma forma de livelock, ele pode continuar indefinidamente.

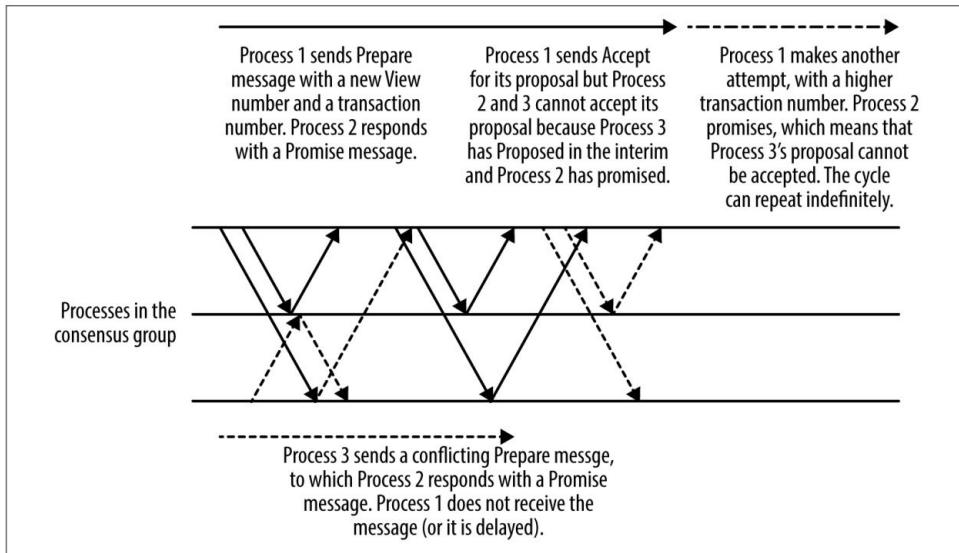


Figura 23-8. Proponentes de duelo em Multi-Paxos

Todos os sistemas de consenso prático abordam essa questão de colisões, geralmente elegendo um processo proponente, que faz todas as propostas no sistema, ou usando um proponente rotativo que aloca a cada processo slots específicos para suas propostas.

Para sistemas que usam um processo de líder, o processo de eleição de líder deve ser ajustado cuidadosamente para equilibrar a indisponibilidade do sistema que ocorre quando nenhum líder está presente com o risco de duelo de proponentes. É importante implementar os tempos limite e as estratégias de recuo corretos. Se vários processos detectarem que não há líder e todos tentarem se tornar líderes ao mesmo tempo, nenhum dos processos provavelmente terá sucesso (novamente, proponentes duelando). Introduzir aleatoriedade é a melhor abordagem. Raft [Ong14], por exemplo, tem um método bem pensado de abordar o processo de eleição do líder.

Escalonamento de cargas de trabalho pesadas de

leitura O dimensionamento da carga de trabalho de leitura geralmente é fundamental porque muitas cargas de trabalho são pesadas de leitura. Os armazenamentos de dados replicados têm a vantagem de que os dados estão disponíveis em vários locais, o que significa que, se uma consistência forte não for necessária para todas as leituras, os dados podem ser lidos de qualquer

réplica. Essa técnica de leitura de réplicas funciona bem para certas aplicações, como o sistema Photon do Google [Ana13], que usa consenso distribuído para coordenar o trabalho de vários pipelines. Photon usa uma operação atômica compare-and-set para modificação de estado (inspirada em registros atômicos), que deve ser absolutamente consistente; mas as operações de leitura podem ser atendidas a partir de qualquer réplica, porque dados obsoletos resultam em trabalho extra sendo executado, mas não em resultados incorretos [Gup15]. A troca vale a pena.

Para garantir que os dados que estão sendo lidos estejam atualizados e consistentes com quaisquer alterações feitas antes da execução da leitura, é necessário fazer o seguinte:

- Execute uma operação de consenso somente leitura. •

Leia os dados de uma réplica com garantia de ser a mais atualizada. Em um sistema que usa um processo líder estável (como fazem muitas implementações de consenso distribuído), o líder pode fornecer essa garantia. • Use concessões de quorum, nas quais algumas réplicas recebem uma concessão de todos ou parte dos dados no sistema, permitindo leituras locais fortemente consistentes ao custo de algum desempenho de gravação. Essa técnica é discutida em detalhes na seção a seguir.

Arrendamentos de Quórum

As concessões de quórum [Mor14] são uma otimização de desempenho de consenso distribuído recentemente desenvolvida com o objetivo de reduzir a latência e aumentar a taxa de transferência para operações de leitura. Como mencionado anteriormente, no caso do Paxos clássico e da maioria dos outros protocolos de consenso distribuído, realizar uma leitura fortemente consistente (ou seja, que tenha a garantia de ter a visão mais atualizada do estado) requer uma operação de consenso distribuído que lê a partir de um quorum de réplicas ou uma réplica líder estável que garante ter visto todas as operações recentes de alteração de estado. Em muitos sistemas, as operações de leitura superam amplamente as gravações, portanto, essa dependência de uma operação distribuída ou de uma única réplica prejudica a latência e a taxa de transferência do sistema.

A técnica de concessão de quorum simplesmente concede uma concessão de leitura em algum subconjunto do estado do armazenamento de dados replicado para um quórum de réplicas. A locação é por um período de tempo específico (geralmente breve). Qualquer operação que altere o estado desses dados deve ser reconhecida por todas as réplicas no quorum de leitura. Se alguma dessas réplicas ficar indisponível, os dados não poderão ser modificados até que a concessão expire.

As concessões de quorum são particularmente úteis para cargas de trabalho de leitura intensa nas quais as leituras de subconjuntos específicos dos dados estão concentradas em uma única região geográfica.

Os sistemas Distributed Consensus Performance e Network Latency

Consensus enfrentam duas grandes restrições físicas no desempenho ao confirmar mudanças de estado. Um é o tempo de ida e volta da rede e o outro é o tempo necessário para gravar dados no armazenamento persistente, que será examinado posteriormente.

Os tempos de ida e volta da rede variam enormemente dependendo do local de origem e destino, que são afetados tanto pela distância física entre a origem e o destino quanto pela quantidade de congestionamento na rede. Dentro de um único datacenter, os tempos de ida e volta entre as máquinas devem ser da ordem de um milissegundo. Um tempo de ida e volta típico (RTT) dentro dos Estados Unidos é de 45 milissegundos e de Nova York a Londres é de 70 milissegundos.

O desempenho do sistema de consenso em uma rede local pode ser comparável ao de um sistema de replicação assíncrona líder-seguidor [Bol11], como muitos bancos de dados tradicionais usam para replicação. No entanto, muitos dos benefícios de disponibilidade dos sistemas de consenso distribuído exigem que as réplicas sejam “distantes”umas das outras, para estarem em diferentes domínios de falha.

Muitos sistemas de consenso usam TCP/IP como protocolo de comunicação. O TCP/IP é orientado à conexão e fornece algumas garantias de confiabilidade em relação ao seqüenciamento FIFO de mensagens. No entanto, configurar uma nova conexão TCP/IP requer uma viagem de ida e volta de rede para executar o handshake de três vias que configura uma conexão antes que qualquer dado possa ser enviado ou recebido. O início lento do TCP/IP inicialmente limita a largura de banda da conexão até que seus limites sejam estabelecidos. Os tamanhos iniciais da janela TCP/IP variam de 4 a 15 KB.

O início lento do TCP/IP provavelmente não é um problema para os processos que formam um grupo de consenso: eles estabelecerão conexões entre si e manterão essas conexões abertas para reutilização porque estarão em comunicação frequente. No entanto, para sistemas com um número muito alto de clientes, pode não ser prático para todos os clientes manter uma conexão persistente com os clusters de consenso aberta, porque as conexões TCP/IP abertas consomem alguns recursos, por exemplo, descritores de arquivo, além de gerando tráfego de manutenção. Essa sobrecarga pode ser um problema importante para aplicativos que usam armazenamentos de dados baseados em consenso altamente fragmentados contendo milhares de réplicas e um número ainda maior de clientes. Uma solução é usar um conjunto de proxies regionais, conforme mostrado na [Figura 23-9](#), que mantém conexões TCP/IP persistentes com o grupo de consenso para evitar a sobrecarga de configuração em longas distâncias. Os proxies também podem ser uma boa maneira de encapsular estratégias de fragmentação e balanceamento de carga, bem como a descoberta de membros e líderes de cluster.

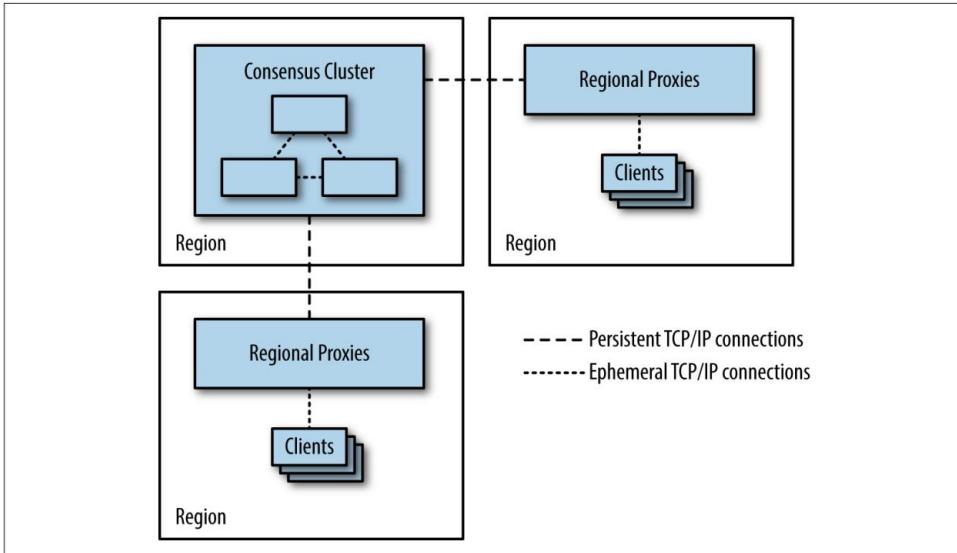


Figura 23-9. Usando proxies para reduzir a necessidade de clientes abrirem conexões TCP/IP entre regiões

Raciocínio sobre desempenho: Fast Paxos

[Lam06] é uma versão do algoritmo Paxos projetada para melhorar seu desempenho em redes de longa distância. Usando Fast Paxos, cada cliente pode enviar mensagens de Proposta diretamente para cada membro de um grupo de aceitantes, ao invés de através de um líder, como no Classic Paxos ou Multi-Paxos. A ideia é substituir um envio de mensagem paralelo do cliente para todos os aceitadores no Fast Paxos por duas operações de envio de mensagem no Classic Paxos:

- Uma mensagem do cliente para um único proponente •

Uma operação de envio de mensagem paralela do proponente para as outras réplicas

Intuitivamente, parece que o Fast Paxos deve ser sempre mais rápido que o Classic Paxos. No entanto, isso não é verdade: se o cliente no sistema Fast Paxos tiver um RTT (tempo de ida e volta) alto para os aceitadores e os aceitadores tiverem conexões rápidas entre si, substituímos N mensagens paralelas nos links de rede mais lentos (em Fast Paxos) para uma mensagem no link mais lento mais N mensagens paralelas nos links mais rápidos (Classic Paxos). Devido ao efeito de cauda de latência, na maioria das vezes, uma única viagem de ida e volta em um link lento com uma distribuição de latências é mais rápida que um quorum (como mostrado em [Jun07]) e, portanto, Fast Paxos é mais lento que Classic Paxos nesse caso.

Muitos sistemas agrupam várias operações em uma única transação no aceitador para aumentar o rendimento. Ter clientes atuando como proponentes também torna muito mais difícil

culto para propostas de lote. A razão para isso é que as propostas chegam independentemente aos aceitantes, de modo que você não pode agrupá-las de maneira consistente.

Líderes estáveis

Vimos como a Multi-Paxos elege um líder estável para melhorar o desempenho. Zab [Jun11] e Raft [Ong14] também são exemplos de protocolos que elegem um líder estável por questões de desempenho. Essa abordagem pode permitir otimizações de leitura, pois o líder possui o estado mais atualizado, mas também apresenta vários problemas:

- Todas as operações que mudam de estado devem ser enviadas via líder, requisito que adiciona latência de rede para clientes que não estão localizados perto do líder.
- A largura de banda da rede de saída do processo líder é um gargalo do sistema [Mao08], porque a mensagem Accept do líder contém todos os dados relacionados a qualquer proposta, enquanto outras mensagens contêm apenas reconhecimentos de uma transação numerada sem carga útil de dados.
- Se o líder estiver em uma máquina com problemas de desempenho, então o a taxa de transferência de todo o sistema será reduzida.

Quase todos os sistemas de consenso distribuído que foram projetados com o desempenho em mente usam o padrão de líder estável único ou um sistema de liderança rotativa em que cada algoritmo de consenso distribuído numerado é pré-atribuído a uma réplica (geralmente por um módulo simples do ID da transação). Algoritmos que usam essa abordagem incluem Mencius [Mao08] e Paxos Igualitário [Mor12a].

Em uma rede de longa distância com clientes espalhados geograficamente e réplicas do grupo de consenso localizadas razoavelmente perto dos clientes, essa eleição de líder leva a uma menor latência percebida para os clientes porque o RTT de sua rede para a réplica mais próxima será, em média, menor do que isso para um líder arbitrário.

Em lotes Em

Iotes, conforme descrito em “Raciocínio sobre desempenho: Paxos rápidos” na página 301, aumenta o rendimento do sistema, mas ainda deixa as réplicas ociosas enquanto aguardam respostas às mensagens que enviaram. As ineficiências apresentadas pelas réplicas ociosas podem ser resolvidas pelo pipeline, que permite que várias propostas sejam executadas de uma só vez. Essa otimização é muito semelhante ao caso do TCP/IP, no qual o protocolo tenta “manter o canal cheio” usando uma abordagem de janela deslizante. Pipelining é normalmente usado em combinação com batching.

Os lotes de solicitações no pipeline ainda são ordenados globalmente com um número de exibição e um número de transação, portanto, esse método não viola as propriedades de ordenação global necessárias para executar uma máquina de estado replicada. Este método de otimização é discutido em [Bol11] e [San11].

Acesso ao disco

O registro no armazenamento persistente é necessário para que um nó, tendo travado e retornado ao cluster, honre quaisquer compromissos anteriores que tenha feito em relação às transações de consenso em andamento. No protocolo Paxos, por exemplo, os aceitantes não podem concordar com uma proposta quando já concordaram com uma proposta com um número de sequência maior. Se os detalhes das propostas acordadas e confirmadas não forem registrados no armazenamento persistente, um aceitador poderá violar o protocolo se ele travar e for reiniciado, levando a estado inconsistente.

O tempo necessário para gravar uma entrada em um log no disco varia muito dependendo de qual hardware ou ambiente virtualizado é usado, mas provavelmente levará entre um e vários milissegundos.

O fluxo de mensagens para Multi-Paxos foi discutido em “[Multi-Paxos: Fluxo de Mensagens Detalhado](#)” na [página 297](#), mas esta seção não mostrou onde o protocolo deve registrar mudanças de estado no disco. Uma gravação em disco deve acontecer sempre que um processo faz um compromisso que deve honrar. Na segunda fase de desempenho crítico do Multi-Paxos, esses pontos ocorrem antes que um aceitante envie uma mensagem Accepted em resposta a uma proposta, e antes que o proponente envie a mensagem Accept, porque essa mensagem Accept também é um Accepted implícito . mensagem [\[Lam98\]](#).

Isso significa que a latência para uma única operação de consenso envolve o seguinte:

- Uma gravação em disco no proponente
- Mensagens paralelas para os aceitadores
- Gravações em disco paralelas nos aceitadores
- As mensagens de retorno

Existe uma versão do protocolo Multi-Paxos que é útil para casos em que o tempo de gravação do disco domina: essa variante não considera a mensagem Accept do proponente como uma mensagem Accept implícita . Em vez disso, o proponente grava no disco em paralelo com os outros processos e envia uma mensagem de aceitação explícita. A latência torna-se então proporcional ao tempo necessário para enviar duas mensagens e para um quórum de processos executar uma gravação síncrona em disco em paralelo.

Se a latência para realizar uma pequena gravação aleatória no disco for da ordem de 10 milissegundos, a taxa de operações de consenso será limitada a aproximadamente 100 por minuto. Esses tempos assumem que os tempos de ida e volta da rede são desprezíveis e o proponente realiza seu registro em paralelo com os aceitadores.

Como já vimos, algoritmos de consenso distribuído são frequentemente usados como base para construir uma máquina de estado replicada. Os RSMs também precisam manter logs de transações para fins de recuperação (pelos mesmos motivos que qualquer armazenamento de dados). O algoritmo de consenso

log e o log de transações do RSM podem ser combinados em um único log. A combinação desses logs evita a necessidade de alternar constantemente entre a gravação em dois locais físicos diferentes no disco [Bol11], reduzindo o tempo gasto nas operações de busca. Os discos podem sustentar mais operações por segundo e, portanto, o sistema como um todo pode realizar mais transações.

Em um armazenamento de dados, os discos têm outros propósitos além da manutenção de logs: o estado do sistema geralmente é mantido no disco. As gravações de log devem ser descarregadas diretamente no disco, mas as gravações para alterações de estado podem ser gravadas em um cache de memória e descarregadas no disco posteriormente, reordenadas para usar o agendamento mais eficiente [Bol11].

Outra otimização possível é agrupar várias operações do cliente em uma operação no proponente ([Ana13], [Bol11], [Cha07], [Jun11], [Mao08], [Mor12a]). Isso amortiza os custos fixos do registro em disco e a latência da rede no maior número de operações, aumentando o rendimento.

Implantação de Sistemas Distribuídos Baseados em Consenso

As decisões mais críticas que os projetistas de sistemas devem tomar ao implantar um sistema baseado em consenso dizem respeito ao número de réplicas a serem implantadas e à localização dessas réplicas.

Número de Réplicas Em

geral, sistemas baseados em consenso operam usando quorums majoritários, ou seja, um grupo de $2f + 1$ réplicas pode tolerar falhas f (se tolerância a falhas bizantinas, em que o sistema é resistente a réplicas retornando resultados incorretos, é necessário, então $3f + 1$ réplicas podem tolerar f falhas [Cas99]). Para falhas não bizantinas, o número mínimo de réplicas que podem ser implantadas é três — se duas forem implantadas, não haverá tolerância para falha de nenhum processo. Três réplicas podem tolerar uma falha. A maior parte do tempo de inatividade do sistema é resultado da manutenção planejada [Ken12]: três réplicas permitem que um sistema opere normalmente quando uma réplica está inativa para manutenção (assumindo que as duas réplicas restantes podem lidar com a carga do sistema com um desempenho aceitável).

Se ocorrer uma falha não planejada durante uma janela de manutenção, o sistema de consenso fica indisponível. A indisponibilidade do sistema de consenso geralmente é inaceitável e, portanto, cinco réplicas devem ser executadas, permitindo que o sistema opere com até duas falhas. Nenhuma intervenção é necessariamente necessária se quatro de cinco réplicas em um sistema de consenso permanecerem, mas se três forem deixadas, uma ou duas réplicas adicionais devem ser adicionadas.

Se um sistema de consenso perde tantas de suas réplicas que não pode formar um quorum, então esse sistema está, em teoria, em um estado irrecuperável porque os logs duráveis de pelo menos uma das réplicas ausentes não podem ser acessados. Se não houver quorum, é possível que tenha sido tomada uma decisão que foi vista apenas pelas réplicas ausentes. Administradores

pode ser capaz de forçar uma mudança na associação do grupo e adicionar novas réplicas que alcancem a existente para continuar, mas a possibilidade de perda de dados sempre permanece - uma situação que deve ser evitada, se possível.

Em um desastre, os administradores precisam decidir se devem executar uma reconfiguração forçada ou aguardar algum período de tempo para que as máquinas com estado do sistema fiquem disponíveis. Quando tais decisões estão sendo tomadas, o tratamento do log do sistema (além do monitoramento) torna-se crítico. Os artigos teóricos geralmente apontam que o consenso pode ser usado para construir um log replicado, mas não discutem como lidar com réplicas que podem falhar e se recuperar (e, portanto, perder alguma sequência de decisões de consenso) ou atrasar devido à lentidão. Para manter a robustez do sistema, é importante que essas réplicas sejam atualizadas.

O log replicado nem sempre é um cidadão de primeira classe na teoria do consenso distribuído, mas é um aspecto muito importante dos sistemas de produção. Raft descreve um método para gerenciar a consistência de logs replicados [Ong14] definindo explicitamente como quaisquer lacunas no log de uma réplica são preenchidas. Se um sistema Raft de cinco instâncias perder todos os seus membros, exceto seu líder, o líder ainda terá total conhecimento de todas as decisões comprometidas. Por outro lado, se a maioria ausente dos membros incluía o líder, não há garantias fortes sobre a atualização das réplicas restantes são.

Existe uma relação entre o desempenho e o número de réplicas em um sistema que não precisam fazer parte de um quorum: uma minoria de réplicas mais lentas pode ficar para trás, permitindo que o quorum de réplicas de melhor desempenho seja executado mais rapidamente (desde que o líder tem um bom desempenho). Se o desempenho da réplica variar significativamente, cada falha poderá reduzir o desempenho geral do sistema, pois serão necessários outliers lentos para formar um quorum. Quanto mais falhas ou réplicas atrasadas um sistema puder tolerar, melhor será o desempenho geral do sistema.

A questão do custo também deve ser considerada no gerenciamento de réplicas: cada réplica usa recursos de computação dispendiosos. Se o sistema em questão for um único cluster de processos, o custo de execução de réplicas provavelmente não será uma grande consideração. No entanto, o custo das réplicas pode ser uma consideração séria para sistemas como o Photon [Ana13], que usa uma configuração shard na qual cada shard é um grupo completo de processos executando um algoritmo de consenso. À medida que o número de shards cresce, o mesmo acontece com o custo de cada réplica adicional, porque um número de processos igual ao número de shards deve ser adicionado ao sistema.

A decisão sobre o número de réplicas para qualquer sistema é, portanto, um trade-off entre os seguintes fatores:

- A necessidade de confiabilidade
- Frequência de manutenção planejada que afeta o sistema

- Risco
- Atuação
- Custo

Esse cálculo será diferente para cada sistema: os sistemas têm objetivos de nível de serviço diferentes para disponibilidade; algumas organizações realizam manutenção com mais regularidade do que outras; e organizações usam hardware de custo, qualidade e confiabilidade variados.

Localização das réplicas

As decisões sobre onde implantar os processos que compõem um cluster de consenso são feitas com base em dois fatores: uma compensação entre os domínios de falha que o sistema deve manipular e os requisitos de latência para o sistema. Vários problemas complexos estão em jogo para decidir onde localizar réplicas.

Um domínio de falha é o conjunto de componentes de um sistema que pode ficar indisponível como resultado de uma única falha. Os domínios de falha de exemplo incluem o seguinte:

- Uma máquina física •
Um rack em um datacenter servido por uma única fonte de alimentação • Vários racks em um datacenter que são atendidos por uma única peça de rede equipamento
- Um datacenter que pode ficar indisponível por um corte de cabo de fibra óptica • Um conjunto de datacenters em uma única área geográfica que pode ser afetado por um único desastre natural, como um furacão

Em geral, à medida que a distância entre as réplicas aumenta, também aumenta o tempo de ida e volta entre as réplicas, bem como o tamanho da falha que o sistema poderá tolerar.

Para a maioria dos sistemas de consenso, aumentar o tempo de ida e volta entre as réplicas também aumentará a latência das operações.

A extensão em que a latência importa, bem como a capacidade de sobreviver a uma falha em um domínio específico, depende muito do sistema. Algumas arquiteturas de sistema de consenso não exigem taxa de transferência particularmente alta ou baixa latência: por exemplo, um sistema de consenso que existe para fornecer serviços de associação de grupo e eleição de líder para um serviço altamente disponível provavelmente não está muito carregado e, se a transação de consenso tempo é apenas uma fração do tempo de concessão do líder, seu desempenho não é crítico.

Os sistemas orientados a lotes também são menos afetados pela latência: os tamanhos dos lotes de operação podem ser aumentados para aumentar o rendimento.

Nem sempre faz sentido aumentar continuamente o tamanho do domínio de falha cuja perda o sistema pode suportar. Por exemplo, se todos os clientes que usam um sistema de consenso estão executando dentro de um domínio de falha específico (digamos, a área de Nova York)

e a implantação de um sistema distribuído baseado em consenso em uma área geográfica mais ampla permitiria que ele continuasse atendendo durante interrupções nesse domínio de falha (digamos, furacão Sandy), vale a pena? Provavelmente não, porque os clientes do sistema também ficarão inativos, de modo que o sistema não verá tráfego. O custo extra em termos de latência, taxa de transferência e recursos de computação não traria nenhum benefício.

Você deve levar em consideração a recuperação de desastres ao decidir onde localizar suas réplicas: em um sistema que armazena dados críticos, as réplicas de consenso também são essencialmente cópias online dos dados do sistema. No entanto, quando dados críticos estão em jogo, é importante fazer backup de instantâneos regulares em outro lugar, mesmo no caso de sistemas sólidos baseados em consenso que são implantados em diversos domínios de falha. Existem dois domínios de falha que você nunca pode escapar: o próprio software e erro humano por parte dos administradores do sistema. Bugs no software podem surgir em circunstâncias incomuns e causar perda de dados, enquanto a configuração incorreta do sistema pode ter efeitos semelhantes.

Operadores humanos também podem errar ou realizar sabotagem causando perda de dados.

Ao tomar decisões sobre a localização das réplicas, lembre-se de que a medida mais importante de desempenho é a percepção do cliente: idealmente, o tempo de ida e volta da rede dos clientes às réplicas do sistema de consenso deve ser minimizado. Em uma rede de longa distância, protocolos sem líder como Mencius ou Egalitarian Paxos podem ter uma vantagem de desempenho, principalmente se as restrições de consistência do aplicativo significarem que é possível executar operações somente leitura em qualquer réplica do sistema sem realizar uma operação de consenso.

Capacidade e balanceamento de carga Ao

projetar uma implantação, você deve certificar-se de que haja capacidade suficiente para lidar com a carga. No caso de implantações fragmentadas, você pode ajustar a capacidade ajustando o número de fragmentos. No entanto, para sistemas que podem ler de membros do grupo de consenso que não são líderes, você pode aumentar a capacidade de leitura adicionando mais réplicas.

Adicionar mais réplicas tem um custo: em um algoritmo que usa um líder forte, adicionar réplicas impõe mais carga ao processo líder, enquanto em um protocolo ponto a ponto, adicionar réplicas impõe mais carga a todos os processos. No entanto, se houver ampla capacidade para operações de gravação, mas uma carga de trabalho pesada de leitura estiver sobrecarregando o sistema, adicionar réplicas pode ser a melhor abordagem.

Deve-se observar que adicionar uma réplica em um sistema de quorum majoritário pode diminuir um pouco a disponibilidade do sistema (como mostrado na [Figura 23-10](#)). Uma implantação típica para Zookeeper ou Chubby usa cinco réplicas, portanto, um quorum majoritário requer três réplicas. O sistema continuará progredindo se duas réplicas, ou 40%, não estiverem disponíveis. Com seis réplicas, um quorum requer quatro réplicas: apenas 33% das réplicas podem ficar indisponíveis se o sistema permanecer ativo.

Considerações sobre domínios de falha, portanto, se aplicam ainda mais fortemente quando uma sexta réplica é adicionada: se uma organização tem cinco datacenters e geralmente executa conj

sensus com cinco processos, um em cada datacenter, então a perda de um datacenter ainda deixa uma réplica sobressalente em cada grupo. Se uma sexta réplica for implantada em um dos cinco datacenters, uma interrupção nesse datacenter removerá ambas as réplicas sobressalentes do grupo, reduzindo assim a capacidade em 33%.

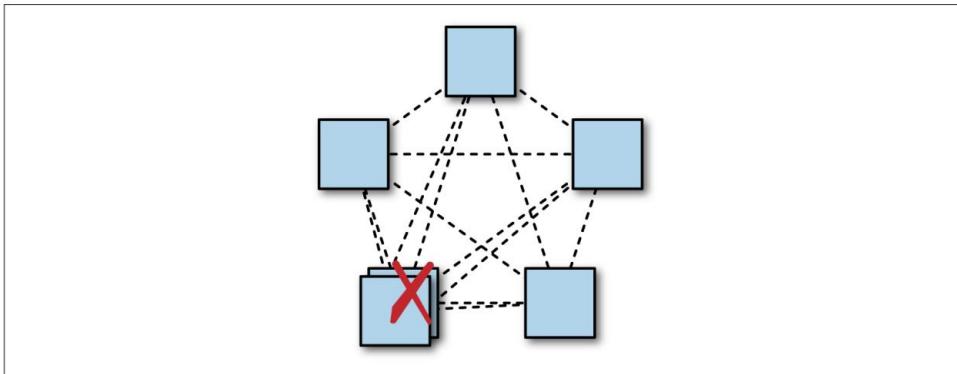


Figura 23-10. Adicionar uma réplica extra em uma região pode reduzir a disponibilidade do sistema. Colocar várias réplicas em um único datacenter pode reduzir a disponibilidade do sistema: aqui, há um quorum sem qualquer redundância restante.

Se os clientes forem densos em uma determinada região geográfica, é melhor localizar réplicas próximas aos clientes. No entanto, decidir onde exatamente localizar as réplicas pode exigir uma reflexão cuidadosa sobre o balanceamento de carga e como um sistema lida com a sobrecarga. Conforme mostrado na Figura 23-11, se um sistema simplesmente roteia as solicitações de leitura do cliente para a réplica mais próxima, um grande pico de carga concentrado em uma região pode sobrecarregar a réplica mais próxima e, em seguida, a próxima réplica mais próxima, e assim por diante. —esta é uma falha em cascata (ver Capítulo 22). Esse tipo de sobrecarga geralmente pode ocorrer como resultado do início de trabalhos em lote, especialmente se vários começarem ao mesmo tempo.

Já vimos o motivo pelo qual muitos sistemas de consenso distribuídos usam um processo líder para melhorar o desempenho. No entanto, é importante entender que as réplicas líderes usarão mais recursos computacionais, principalmente a capacidade da rede de saída. Isso ocorre porque o líder envia mensagens de proposta que incluem os dados propostos, mas as réplicas enviam mensagens menores, geralmente contendo apenas a concordância com um determinado ID de transação de consenso. As organizações que executam sistemas de consenso altamente fragmentados com um número muito grande de processos podem achar necessário garantir que os processos líderes para os diferentes fragmentos sejam equilibrados de forma relativamente uniforme em diferentes datacenters. Isso evita que o sistema como um todo seja limitado na capacidade de rede de saída para apenas um datacenter e aumenta a capacidade geral do sistema.

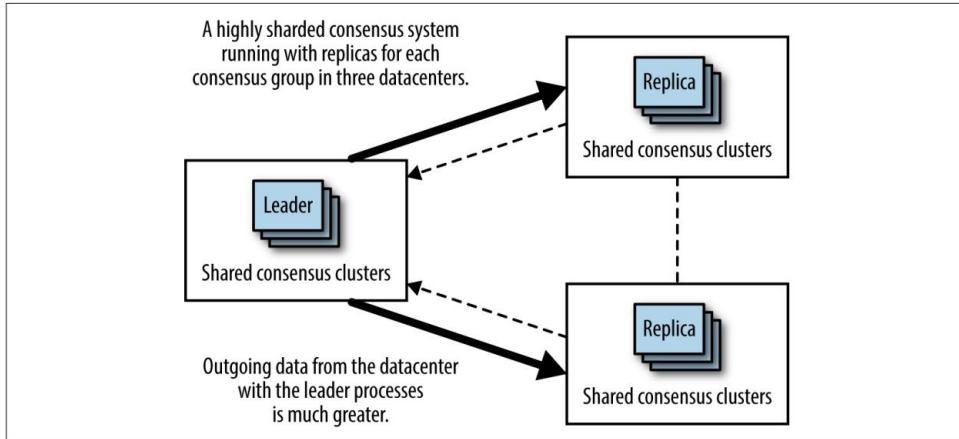


Figura 23-11. Os processos líderes de colocação levam a uma utilização de largura de banda desigual

Outra desvantagem de implantar grupos de consenso em vários datacenters (mostrado na [Figura 23-11](#)) é a mudança muito extrema no sistema que pode ocorrer se o datacenter que hospeda os líderes sofrer uma falha generalizada (energia, falha de equipamento de rede ou corte de fibra). , por exemplo). Conforme mostrado na [Figura 23-12](#), nesse cenário de falha, todos os líderes devem fazer failover para outro datacenter, divididos igualmente ou em massa em um datacenter. Em ambos os casos, o link entre os outros dois datacenters receberá de repente muito mais tráfego de rede desse sistema. Este seria um momento inoportuno para descobrir que a capacidade naquele link é insuficiente.

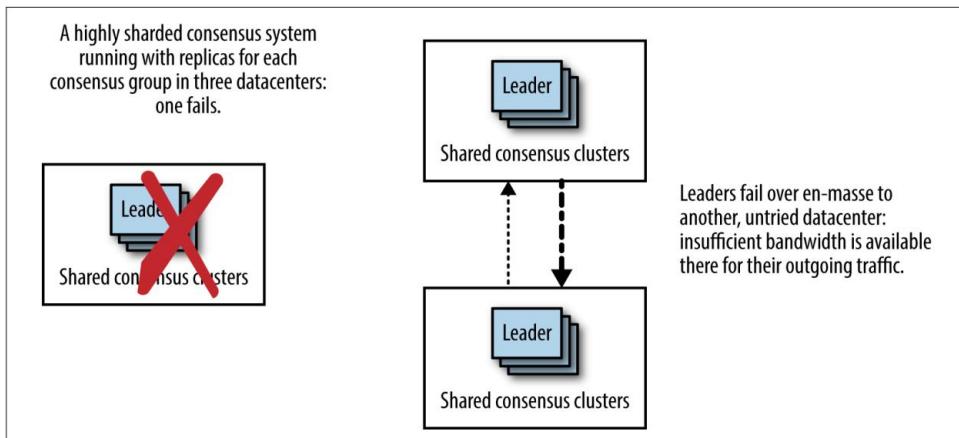


Figura 23-12. Quando os líderes colocalizados fazem failover em massa, os padrões de utilização da rede mudam drasticamente

No entanto, esse tipo de implantação pode facilmente ser um resultado não intencional de processos automáticos no sistema que influenciam a forma como os líderes são escolhidos. Por exemplo:

- Os clientes terão melhor latência para qualquer operação realizada por meio do líder se o líder estiver localizado mais próximo a eles. Um algoritmo que tenta localizar líderes de sites perto da maioria dos clientes pode tirar proveito dessa percepção.
- Um algoritmo pode tentar localizar líderes em máquinas com melhor desempenho.
Uma armadilha dessa abordagem é que, se um dos três datacenters abrigar máquinas mais rápidas, uma quantidade desproporcional de tráfego será enviada para esse datacenter, resultando em mudanças extremas de tráfego caso esse datacenter fique offline. Para evitar esse problema, o algoritmo também deve levar em conta o balanço de distribuição em relação às capacidades da máquina ao selecionar as máquinas.
- Um algoritmo de eleição de líder pode favorecer processos que estão em execução há mais tempo. Os processos de execução mais longa provavelmente estarão correlacionados com a localização se as versões de software forem executadas por datacenter.

Composição de

quorum Ao determinar onde localizar réplicas em um grupo de consenso, é importante considerar o efeito da distribuição geográfica (ou, mais precisamente, as latências de rede entre réplicas) no desempenho do grupo.

Uma abordagem é distribuir as réplicas o mais uniformemente possível, com RTTs semelhantes entre todas as réplicas. Todos os outros fatores sendo iguais (como carga de trabalho, hardware e desempenho de rede), esse arranjo deve levar a um desempenho bastante consistente em todas as regiões, independentemente de onde o líder do grupo esteja localizado (ou para cada membro do grupo de consenso, se um líder sem líder protocolo está em uso).

A geografia pode complicar muito essa abordagem. Isto é particularmente verdadeiro para tráfego intracontinental versus tráfego transpacífico e transatlântico. Considere um sistema que abrange a América do Norte e a Europa: é impossível localizar réplicas equidistantes umas das outras porque sempre haverá um atraso maior para o tráfego transatlântico do que para o tráfego intracontinental. Não importa o que aconteça, as transações de uma região precisarão fazer uma viagem de ida e volta transatlântica para chegar a um consenso.

No entanto, conforme mostrado na [Figura 23-13](#), para tentar distribuir o tráfego da maneira mais uniforme possível, os projetistas de sistemas podem optar por localizar cinco réplicas, com duas réplicas aproximadamente no centro dos EUA, uma na costa leste e duas na Europa. Tal distribuição significaria que, no caso médio, o consenso poderia ser alcançado na América do Norte sem esperar por respostas da Europa, ou que da Europa, o consenso pode ser alcançado trocando mensagens apenas com a réplica da costa leste. A réplica da costa leste atua como uma espécie de eixo, onde dois quóruns possíveis se sobrepõem.



Figura 23-13. Sobreposição de quórums com uma réplica atuando como um link

Conforme mostrado na [Figura 23-14](#), a perda dessa réplica significa que a latência do sistema provavelmente mudará drasticamente: em vez de ser amplamente influenciada pelo RTT central dos EUA para a costa leste ou RTT da UE para a costa leste, a latência será baseada na UE para o centro RTT, que é cerca de 50% superior ao RTT da costa leste da UE. A distância geográfica e o RTT da rede entre o quorum mais próximo aumenta enormemente.



Figura 23-14. A perda da réplica do link leva imediatamente a um RTT mais longo para qualquer quorum

Esse cenário é um ponto fraco importante do quorum de maioria simples quando aplicado a grupos compostos de réplicas com RTTs muito diferentes entre os membros. Nesses casos, uma abordagem de quórum hierárquico pode ser útil. Conforme diagramado na [Figura 23-15](#), nove réplicas podem ser implantadas em três grupos de três. Um quórum pode

ser formado por uma maioria de grupos, e um grupo pode ser incluído no quórum se a maioria dos membros do grupo estiver disponível. Isso significa que uma réplica pode ser perdida no grupo central sem causar um grande impacto no desempenho geral do sistema porque o grupo central ainda pode votar em transações com duas de suas três réplicas.

Há, no entanto, um custo de recursos associado à execução de um número maior de réplicas. Em um sistema altamente fragmentado com uma carga de trabalho de leitura pesada que é amplamente preenchível por réplicas, podemos mitigar esse custo usando menos grupos de consenso. Tal estratégia significa que o número total de processos no sistema pode não mudar.



Figura 23-15. Quóruns hierárquicos podem ser usados para reduzir a dependência da réplica central

Monitoramento de Sistemas de Consenso Distribuídos

Como já vimos, os algoritmos de consenso distribuído estão no centro de muitos dos sistemas críticos do Google ([Ana13], [Bur06], [Cor12], [Shu13]). Todos os sistemas de produção importantes precisam de monitoramento, para detectar interrupções ou problemas e para solucionar problemas. A experiência nos mostrou que existem certos aspectos específicos dos sistemas de consenso distribuído que merecem atenção especial. Estes são:

O número de membros em execução em cada grupo de consenso e o status de cada processo (saudável ou não)

Um processo pode estar em execução, mas incapaz de progredir por algum motivo (por exemplo, relacionado ao hardware).

Réplicas persistentemente

atrasadas Membros saudáveis de um grupo de consenso ainda podem estar em vários estados diferentes.

Um membro do grupo pode estar recuperando o estado de seus pares após a inicialização, ou atrasado em relação ao quórum do grupo, ou pode estar atualizado e participando plenamente, e pode ser o líder.

Se existe ou não um líder

Um sistema baseado em um algoritmo como o Multi-Paxos que usa um papel de líder deve ser monitorado para garantir que exista um líder, pois se o sistema não tiver líder, ele estará totalmente indisponível.

Número de mudanças de líder

Mudanças rápidas de liderança prejudicam o desempenho dos sistemas de consenso que usam um líder estável, portanto, o número de mudanças de líder deve ser monitorado. Os algoritmos de consenso geralmente marcam uma mudança de liderança com um novo termo ou número de visão, portanto, esse número fornece uma métrica útil para monitorar. Um aumento muito rápido nas mudanças de líderes sinaliza que o líder está oscilando, talvez devido a problemas de conectividade de rede. Uma diminuição no número de visualizações pode sinalizar um bug sério.

Número de transação de consenso

Os operadores precisam saber se o sistema de consenso está progredindo ou não. A maioria dos algoritmos de consenso usa um número crescente de transações de consenso para indicar o progresso. Esse número deve aumentar ao longo do tempo se um sistema estiver saudável.

Número de propostas vistas; número de propostas acordadas

Esses números indicam se o sistema está funcionando corretamente ou não.

Taxa de transferência e latência

Embora não sejam específicas para sistemas de consenso distribuídos, essas características de seu sistema de consenso devem ser monitoradas e compreendidas pelos administradores.

Para entender o desempenho do sistema e ajudar a solucionar problemas de desempenho, você também pode monitorar o seguinte:

- Distribuições de latência para aceitação de propostas •

Distribuições de latências de rede observadas entre partes do sistema em diferentes locais de entrada

- A quantidade de tempo que os aceitadores gastam no registro durável •

Bytes gerais aceitos por segundo no sistema

Conclusão

Exploramos a definição do problema de consenso distribuído e apresentamos alguns padrões de arquitetura de sistema para sistemas baseados em consenso distribuído, bem como examinamos as características de desempenho e algumas das preocupações operacionais em torno de sistemas baseados em consenso distribuído.

Evitamos deliberadamente uma discussão aprofundada sobre algoritmos, protocolos ou implementações específicos neste capítulo. Sistemas de coordenação distribuídos e a tecnologia

As tecnologias subjacentes a eles estão evoluindo rapidamente, e essas informações rapidamente se tornariam desatualizadas, ao contrário dos fundamentos discutidos aqui. No entanto, esses fundamentos, juntamente com os artigos mencionados ao longo deste capítulo, permitirão que você use as ferramentas de coordenação distribuídas disponíveis hoje, bem como softwares futuros.

Se você não se lembrar de mais nada deste capítulo, tenha em mente os tipos de problemas que o consenso distribuído pode ser usado para resolver e os tipos de problemas que podem surgir quando métodos ad hoc, como pulsações, são usados em vez de consenso distribuído.

Sempre que você vir a eleição de um líder, estado compartilhado crítico ou bloqueio distribuído, pense no consenso distribuído: qualquer abordagem menor é uma bomba-relógio esperando para explodir em seus sistemas.

CAPÍTULO 24

Agendamento Periódico Distribuído com Cron

Escrito por Štýpán Davidovič¹
Editado por Kavita Giuliani

Este capítulo descreve a implementação do Google de um serviço cron distribuído que atende à grande maioria das equipes internas que precisam de agendamento periódico de trabalhos de computação. Ao longo da existência do cron, aprendemos muitas lições sobre como projetar e implementar o que pode parecer um serviço básico. Aqui, discutimos os problemas que os crons distribuídos enfrentam e descrevemos algumas soluções potenciais.

Cron é um utilitário comum do Unix projetado para lançar periodicamente trabalhos arbitrários em horários ou intervalos definidos pelo usuário. Primeiro analisamos os princípios básicos do cron e suas implementações mais comuns e, em seguida, revisamos como um aplicativo como o cron pode funcionar em um ambiente grande e distribuído para aumentar a confiabilidade do sistema contra falhas de máquina única. Descrevemos um sistema cron distribuído que é implantado em um pequeno número de máquinas, mas pode iniciar tarefas cron em um datacenter inteiro em conjunto com um sistema de agendamento de datacenter como Borg [Ver15].

Cron

Vamos discutir como o cron é normalmente usado, no caso de uma única máquina, antes de começar a executá-lo como um serviço entre datacenters.

Introdução

O Cron foi projetado para que os administradores do sistema e usuários comuns do sistema possam especificar os comandos a serem executados e quando esses comandos são executados. Cron executa vários

¹ Este capítulo foi publicado anteriormente em parte no ACM Queue (março de 2015, vol. 13, número 3).

tipos de trabalhos, incluindo coleta de lixo e análise periódica de dados. O formato de especificação de tempo mais comum é chamado de “crontab”. Este formato suporta intervalos simples (por exemplo, “uma vez por dia ao meio-dia” ou “todas as horas na hora”). Intervalos complexos, como “todo sábado, que também é o 30º dia do mês”, também podem ser configurados.

Cron é geralmente implementado usando um único componente, que é comumente referido como crond. crond é um daemon que carrega a lista de tarefas cron agendadas. Os trabalhos são iniciados de acordo com seus tempos de execução especificados.

Perspectiva de Confiabilidade

Vários aspectos do serviço cron são notáveis do ponto de vista da confiabilidade:

- O domínio de falha do Cron é essencialmente apenas uma máquina. Se a máquina não estiver em execução, nem o agendador cron nem os trabalhos que ele inicia podem ser executados.² Considere um caso distribuído muito simples com duas máquinas, no qual seu agendador cron inicia trabalhos em uma máquina de trabalho diferente (por exemplo, usando SSH). Esse cenário apresenta dois domínios de falha distintos que podem afetar nossa capacidade de iniciar tarefas: a máquina agendadora ou a máquina de destino podem falhar.
- O único estado que precisa persistir nas reinicializações do crond (incluindo reinicializações da máquina) é a própria configuração do crontab. Os lançamentos do cron são disparados e esquecidos, e o crond não faz nenhuma tentativa de rastrear esses lançamentos.
- o anacron é uma exceção notável a isso. o anacron tenta iniciar trabalhos que seriam lançados quando o sistema estava inativo. As tentativas de reinicialização são limitadas a trabalhos executados diariamente ou com menos frequência. Essa funcionalidade é muito útil para executar tarefas de manutenção em estações de trabalho e notebooks e é facilitada por um arquivo que retém o carimbo de data/hora da última inicialização para todas as tarefas cron registradas.

Cron Jobs e Idempotência

Os trabalhos Cron são projetados para realizar trabalhos periódicos, mas além disso, é difícil saber antecipadamente qual função eles têm. A variedade de requisitos que o conjunto diversificado de tarefas cron acarreta obviamente afeta os requisitos de confiabilidade.

Alguns trabalhos cron, como processos de coleta de lixo, são idempotentes. Em caso de mau funcionamento do sistema, é seguro iniciar esses trabalhos várias vezes. Outros cron jobs, como um processo que envia um boletim informativo por e-mail para uma ampla distribuição, não devem ser lançados mais de uma vez.

² O fracasso de trabalhos individuais está além do escopo desta análise.

Para tornar as coisas mais complicadas, a falha na inicialização é aceitável para alguns cron jobs, mas não para outros. Por exemplo, uma tarefa cron de coleta de lixo agendada para ser executada a cada cinco minutos pode pular uma inicialização, mas uma tarefa cron de folha de pagamento agendada para ser executada uma vez por mês não deve ser ignorada.

Essa grande variedade de tarefas cron dificulta o raciocínio sobre os modos de falha: em um sistema como o serviço cron, não há uma resposta única que se adapte a todas as situações. Em geral, preferimos pular lançamentos ao invés de arriscar lançamentos duplos, tanto quanto a infraestrutura permite. Isso ocorre porque a recuperação de um lançamento ignorado é mais sustentável do que a recuperação de um lançamento duplo. Os proprietários de cron jobs podem (e devem!) monitorar seus cron jobs; por exemplo, um proprietário pode ter o estado de exposição do serviço cron para suas tarefas cron gerenciadas ou configurar o monitoramento independente do efeito das tarefas cron. No caso de uma inicialização ignorada, os proprietários do cron job podem executar ações que correspondam adequadamente à natureza do cron job. No entanto, desfazer um lançamento duplo, como o exemplo do boletim informativo mencionado anteriormente, pode ser difícil ou até mesmo impossível. Portanto, preferimos “falhar fechado” para evitar a criação sistêmica de mau estado.

Cron em grande escala

Afastar-se de máquinas únicas para implantações em grande escala requer uma reformulação fundamental de como fazer o cron funcionar bem em tal ambiente.

Antes de apresentar os detalhes da solução Google cron, discutiremos essas diferenças entre implantação em pequena e grande escala e descreveremos quais mudanças de design as implantações em grande escala exigiram.

Infraestrutura estendida

Em suas implementações “regulares”, o cron é limitado a uma única máquina. As implantações de sistema em grande escala estendem nossa solução cron para várias máquinas.

Hospedar seu serviço cron em uma única máquina pode ser catastrófico em termos de confiabilidade. Digamos que esta máquina esteja localizada em um datacenter com exatamente 1.000 máquinas. Uma falha de apenas 1/1000 de suas máquinas disponíveis pode derrubar todo o serviço cron. Por razões óbvias, esta implementação não é aceitável.

Para aumentar a confiabilidade do cron, dissociamos os processos das máquinas. Se você deseja executar um serviço, basta especificar os requisitos do serviço e em qual datacenter ele deve ser executado. O sistema de agendamento do datacenter (que deve ser confiável) determina a máquina ou máquinas nas quais implantar seu serviço, além de lidar com mortes de máquinas. O lançamento de um trabalho em um datacenter se transforma efetivamente no envio de um ou mais RPCs para o agendador do datacenter.

Este processo, no entanto, não é instantâneo. A descoberta de uma máquina inativa envolve tempos limite de verificação de integridade, enquanto o reagendamento do serviço em uma máquina diferente requer tempo para instalar o software e iniciar o novo processo.

Como mover um processo para uma máquina diferente pode significar a perda de qualquer estado local armazenado na máquina antiga (a menos que a migração ao vivo seja empregada) e o tempo de reagendamento pode exceder o menor intervalo de agendamento de um minuto, precisamos de procedimentos para mitigar ambos perda de dados e requisitos de tempo excessivos. Para manter o estado local da máquina antiga, você pode simplesmente persistir o estado em um sistema de arquivos distribuído, como o GFS, e usar esse sistema de arquivos durante a inicialização para identificar as tarefas que falharam ao iniciar devido ao reagendamento. No entanto, essa solução fica aquém das expectativas de pontualidade: se você executar uma tarefa cron a cada cinco minutos, um atraso de um a dois minutos causado pela sobrecarga total de reprogramação do sistema cron é potencialmente inaceitavelmente substancial. Nesse caso, hot spares, que seriam capazes de entrar rapidamente e retomar a operação, podem reduzir significativamente essa janela de tempo.

Requisitos estendidos Os sistemas

de máquina única normalmente apenas colocam todos os processos em execução com isolamento limitado. Embora os contêineres agora sejam comuns, não é necessário ou comum usar contêineres para isolar todos os componentes de um serviço implantado em uma única máquina. Portanto, se o cron fosse implantado em uma única máquina, o crond e todos os cron jobs que ele executa provavelmente não seriam isolados.

A implantação em escala de datacenter geralmente significa implantação em contêineres que impõem isolamento. O isolamento é necessário porque a expectativa básica é que os processos independentes executados no mesmo datacenter não devem impactar negativamente uns aos outros. Para impor essa expectativa, você deve saber antecipadamente a quantidade de recursos que precisa adquirir para qualquer processo que deseja executar - tanto para o sistema cron quanto para os trabalhos que ele inicia. Um trabalho cron pode ser atrasado se o datacenter não tiver recursos disponíveis para atender às demandas do trabalho cron. Os requisitos de recursos, além da demanda do usuário para monitoramento de lançamentos de cron job, significa que precisamos rastrear o estado completo de nossos lançamentos de cron job, desde o lançamento agendado até o término.

A dissociação de lançamentos de processos de máquinas específicas expõe o sistema cron a uma falha parcial de inicialização. A versatilidade das configurações de cron job também significa que o lançamento de um novo cron job em um datacenter pode precisar de vários RPCs, de modo que às vezes encontramos um cenário no qual alguns RPCs foram bem-sucedidos, mas outros não (por exemplo, porque o processo de envio dos RPCs morreu no meio da execução dessas tarefas).

O procedimento de recuperação do cron também deve levar em conta esse cenário.

Em termos de modo de falha, um datacenter é um ecossistema substancialmente mais complexo do que uma única máquina. O serviço cron que começou como um binário relativamente simples em uma única máquina agora tem muitas dependências óbvias e não óbvias quando implantado em uma escala maior. Para um serviço tão básico quanto o cron, queremos garantir que, mesmo que o datacenter sofra uma falha parcial (por exemplo, queda parcial de energia ou problemas com serviços de armazenamento), o serviço ainda possa funcionar. Ao exigir que o datacenter

escalonador localiza réplicas do cron em diversos locais dentro do datacenter, evitamos o cenário em que a falha de uma única unidade de distribuição de energia tire todos os processos do serviço cron.

Pode ser possível implantar um único serviço cron em todo o mundo, mas implantar o cron em um único datacenter tem benefícios: o serviço desfruta de baixa latência e compartilha o destino com o agendador de datacenter, a dependência principal do cron.

Construindo o Cron no Google

Esta seção aborda os problemas que devem ser resolvidos para fornecer uma implantação distribuída em larga escala do cron de forma confiável. Ele também destaca algumas decisões importantes feitas em relação ao cron distribuído no Google.

Rastreando o estado dos cron jobs Conforme

discutido nas seções anteriores, precisamos manter uma certa quantidade de estado sobre os cron jobs e sermos capazes de restaurar essas informações rapidamente em caso de falha. Além disso, a consistência desse estado é primordial. Lembre-se de que muitos cron jobs, como uma execução de folha de pagamento ou envio de um boletim informativo por e-mail, não são idempotentes.

Temos duas opções para rastrear o estado dos cron jobs:

- Armazenar dados externamente em armazenamento distribuído geralmente disponível
- Usar um sistema que armazena um pequeno volume de estado como parte do próprio serviço cron

Ao projetar o cron distribuído, escolhemos a segunda opção. Fizemos essa escolha por vários motivos:

- Sistemas de arquivos distribuídos, como GFS ou HDFS, geralmente atendem ao caso de uso de arquivos muito grandes (por exemplo, a saída de programas de rastreamento da Web), enquanto as informações que precisamos armazenar sobre tarefas cron são muito pequenas. Pequenas gravações em um sistema de arquivos distribuído são muito caras e vêm com alta latência, porque o sistema de arquivos não é otimizado para esses tipos de gravações.
- Os serviços de base para os quais as interrupções têm grande impacto (como o cron) devem ter muito poucas dependências. Mesmo que partes do datacenter desapareçam, o serviço cron deve funcionar por pelo menos algum tempo. Mas esse requisito não significa que o armazenamento tenha que fazer parte do processo cron diretamente (como o armazenamento é tratado é essencialmente um detalhe de implementação). No entanto, o cron deve ser capaz de operar independentemente dos sistemas downstream que atendem a um grande número de usuários internos.

O uso de Paxos

Implantamos várias réplicas do serviço cron e usamos o algoritmo de consenso distribuído Paxos (consulte o [Capítulo 23](#)) para garantir que tenham um estado consistente. Contanto que a maioria dos membros do grupo esteja disponível, o sistema distribuído como um todo pode processar com sucesso novas mudanças de estado, apesar da falha de subconjuntos limitados da infraestrutura.

Conforme mostrado na [Figura 24-1](#), o cron distribuído usa uma única tarefa líder, que é a única réplica que pode modificar o estado compartilhado, bem como a única réplica que pode iniciar tarefas cron. Aproveitamos o fato de que a variante de Paxos que usamos, Fast Paxos [[Lam06](#)], usa uma réplica líder internamente como uma otimização - a réplica líder Fast Paxos também atua como líder de serviço cron.

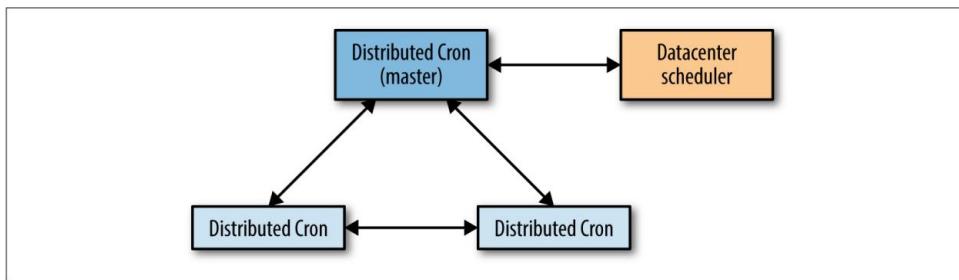


Figura 24-1. As interações entre réplicas cron distribuídas

Se a réplica líder morrer, o mecanismo de verificação de integridade do grupo Paxos descobre esse evento rapidamente (em segundos). Como outro processo cron já está iniciado e disponível, podemos eleger um novo líder. Assim que o novo líder é eleito, seguimos um protocolo de eleição de líder específico para o serviço cron, que é responsável por assumir todo o trabalho inacabado pelo líder anterior. O líder específico para o serviço cron é o mesmo que o líder Paxos, mas o serviço cron precisa tomar medidas adicionais na promoção. O tempo de reação rápido para a reeleição do líder nos permite permanecer dentro de um tempo de failover geralmente tolerável de um minuto.

O estado mais importante que mantemos no Paxos é a informação sobre quais cron jobs são lançados. Informamos de forma síncrona um quórum de réplicas do início e do fim de cada inicialização agendada para cada cron job.

Os papéis do líder e do seguidor

Como acabamos de descrever, nosso uso do Paxos e sua implantação no serviço cron tem duas funções atribuídas: o líder e o seguidor. As seções a seguir descrevem cada função.

O Líder

A réplica líder é a única réplica que inicia ativamente tarefas cron. O líder tem um escalonador interno que, muito parecido com o crond simples descrito no início deste capítulo, mantém a lista de tarefas cron ordenadas pelo horário de lançamento agendado.

A réplica líder aguarda até o horário de inicialização agendado do primeiro trabalho.

Ao atingir o horário de lançamento agendado, a réplica líder anuncia que está prestes a iniciar o lançamento desse cron job específico e calcula o novo horário de lançamento agendado, exatamente como faria uma implementação regular do crond. É claro que, como acontece com o crond normal, uma especificação de lançamento do cron job pode ter mudado desde a última execução, e essa especificação de lançamento também deve ser mantida em sincronia com os seguidores.

Simplesmente identificar o cron job não é suficiente: também devemos identificar exclusivamente o lançamento específico usando a hora de início; caso contrário, pode ocorrer ambiguidade no rastreamento de inicialização do cron job. (Tal ambiguidade é especialmente provável no caso de tarefas cron de alta frequência, como aquelas que são executadas a cada minuto.) Conforme visto na [Figura 24-2](#), essa comunicação é realizada em Paxos.

É importante que a comunicação do Paxos permaneça síncrona e que o lançamento real do cron job não prossiga até receber a confirmação de que o quorum do Paxos recebeu a notificação de lançamento. O serviço cron precisa entender se cada tarefa cron foi iniciada para decidir o próximo curso de ação em caso de failover de líder. Não executar essa tarefa de forma síncrona pode significar que toda a inicialização do cron job ocorre no líder sem informar as réplicas do seguidor.

Em caso de failover, as réplicas do seguidor podem tentar executar a mesma inicialização novamente porque não estão cientes de que a inicialização já ocorreu.

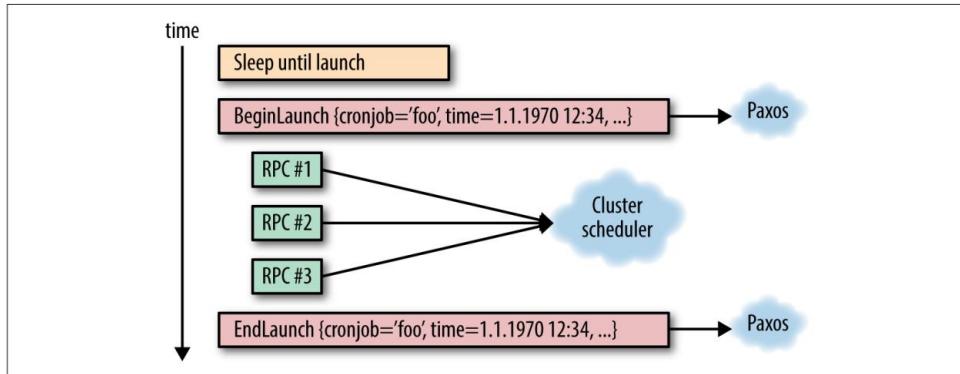


Figura 24-2. Ilustração do progresso de um lançamento de cron job, da perspectiva do líder

A conclusão do lançamento do cron job é anunciada via Paxos para as outras réplicas de forma síncrona. Observe que não importa se a inicialização foi bem-sucedida ou falhou por motivos externos (por exemplo, se o agendador do datacenter não estiver disponível). Aqui, estamos simplesmente acompanhando o fato de que o serviço cron tentou o lançamento em

o horário determinado. Também precisamos ser capazes de resolver falhas do sistema cron no meio desta operação, conforme discutido na seção a seguir.

Outra característica extremamente importante do líder é que assim que ele perde sua liderança por qualquer motivo, ele deve imediatamente parar de interagir com o agendador do datacenter. Manter a liderança deve garantir a exclusão mútua do acesso ao agendador de datacenter. Na ausência dessa condição de exclusão mútua, os líderes antigos e novos podem executar ações conflitantes no agendador do datacenter.

O seguidor

As réplicas do seguidor acompanham o estado do mundo, conforme fornecido pelo líder, para assumir o controle a qualquer momento, se necessário. Todas as mudanças de estado rastreadas pelas réplicas seguidoras são comunicadas via Paxos, a partir da réplica líder. Assim como o líder, os seguidores também mantêm uma lista de todos os cron jobs no sistema, e essa lista deve ser mantida consistente entre as réplicas (através do uso do Paxos).

Ao receber uma notificação sobre uma inicialização iniciada, a réplica do seguidor atualiza seu próximo horário de inicialização agendado local para o cron job fornecido. Essa mudança de estado muito importante (que é executada de forma síncrona) garante que todos os agendamentos de cron job no sistema sejam consistentes. Acompanhamos todos os lançamentos abertos (lançamentos iniciados, mas não concluídos).

Se uma réplica líder morrer ou apresentar mau funcionamento (por exemplo, for particionada de outras réplicas na rede), um seguidor deve ser eleito como novo líder. A eleição deve convergir mais rápido do que um minuto, a fim de evitar o risco de perder ou atrasar injustificadamente um cron job. Uma vez que um líder é eleito, todos os lançamentos abertos (ou seja, falhas parciais) devem ser concluídos. Esse processo pode ser bastante complicado, impondo requisitos adicionais tanto no sistema cron quanto na infraestrutura do datacenter. A seção a seguir discute como resolver falhas parciais desse tipo.

Resolvendo falhas parciais

Conforme mencionado, a interação entre a réplica líder e o agendador do datacenter pode falhar entre o envio de vários RPCs que descrevem uma única inicialização lógica de cron job. Nossos sistemas devem ser capazes de lidar com essa condição.

Lembre-se de que todo lançamento de cron job tem dois pontos de sincronização:

- Quando estamos prestes a realizar o lançamento
- Quando terminarmos o lançamento

Esses dois pontos permitem delimitar o lançamento. Mesmo que o lançamento consista em um único RPC, como sabemos se o RPC foi realmente enviado? Considere o caso em que sabemos que o lançamento programado começou, mas não fomos notificados de sua conclusão antes que a réplica líder morresse.

Para determinar se o RPC foi realmente enviado, uma das seguintes condições deve ser atendida:

- Todas as operações em sistemas externos, que possamos precisar continuar após a reeleição, devem ser idempotentes (ou seja, podemos realizar as operações com segurança novamente)
- Devemos ser capazes de consultar o estado de todas as operações em sistemas externos em para determinar inequivocamente se eles completaram ou não

Cada uma dessas condições impõe restrições significativas e pode ser difícil de implementar, mas ser capaz de atender a pelo menos uma dessas condições é fundamental para a operação precisa de um serviço cron em um ambiente distribuído que pode sofrer uma ou várias falhas parciais. Não lidar com isso adequadamente pode levar a lançamentos perdidos ou lançamentos duplos do mesmo cron job.

A maioria das infraestruturas que iniciam trabalhos lógicos em datacenters (Mesos, por exemplo) fornece nomes para esses trabalhos de datacenter, tornando possível pesquisar o estado dos trabalhos, interrompê-los ou realizar outras manutenções. Uma solução razoável para o problema de idempotência é construir nomes de tarefas antecipadamente (evitando, assim, causar qualquer operação de mutação no agendador de datacenter) e, em seguida, distribuir os nomes para todas as réplicas de seu serviço cron. Se o líder do serviço cron morrer durante o lançamento, o novo líder simplesmente procura o estado de todos os nomes pré-computados e inicia os nomes ausentes.

Observe que, semelhante ao nosso método de identificar lançamentos de cron job individuais por seu nome e hora de lançamento, é importante que os nomes de trabalho construídos no agendador do datacenter incluam o horário de lançamento agendado específico (ou tenham essas informações recuperáveis de outra forma). Em operação regular, o serviço cron deve fazer failover rapidamente em caso de falha do líder, mas um failover rápido nem sempre acontece.

Lembre-se de que rastreamos o horário de lançamento agendado ao manter o estado interno entre as réplicas. Da mesma forma, precisamos desambiguar nossa interação com o agendador do datacenter, também usando o horário de lançamento agendado. Por exemplo, considere um trabalho cron de curta duração, mas executado com frequência. A tarefa cron é iniciada, mas antes que a inicialização seja comunicada a todas as réplicas, o líder trava e ocorre um failover incomumente longo – longo o suficiente para que a tarefa cron termine com sucesso. O novo líder consulta o estado do cron job, observa sua conclusão e tenta iniciar o job novamente. Se o horário de lançamento tivesse sido incluído, o novo líder saberia que o trabalho no agendador de datacenter é o resultado desse lançamento de trabalho cron específico, e esse lançamento duplo não teria acontecido.

A implementação real tem um sistema mais complicado para pesquisa de estado, orientado pelos detalhes de implementação da infraestrutura subjacente. No entanto, a descrição anterior cobre os requisitos independentes de implementação de tais sistemas.

tem. Dependendo da infraestrutura disponível, você também pode precisar considerar a compensação entre arriscar uma inicialização dupla e arriscar pular uma inicialização.

Armazenando o Estado

Usar o Paxos para obter consenso é apenas uma parte do problema de como lidar com o Estado. Paxos é essencialmente um registro contínuo de mudanças de estado, anexado sincronicamente à medida que as mudanças de estado ocorrem. Esta característica do Paxos tem duas implicações:

- A tora precisa ser compactada, para evitar que cresça infinitamente
- A tora em si deve ser armazenada em algum lugar

Para evitar o crescimento infinito do log do Paxos, podemos simplesmente tirar um instantâneo do estado atual, o que significa que podemos reconstruir o estado sem a necessidade de reproduzir todas as entradas do log de mudança de estado que levam ao estado atual. Para dar um exemplo: se nossas alterações de estado armazenadas em logs forem “Incrementar um contador em 1”, depois de mil iterações, teremos mil entradas de log que podem ser facilmente alteradas para um instantâneo de “Definir contador para 1.000”.

No caso de logs perdidos, só perdemos o estado desde o último snapshot. Os instantâneos são, de fato, nosso estado mais crítico - se perdermos nossos instantâneos, essencialmente teremos que começar do zero novamente porque perdemos nosso estado interno. A perda de logs, por outro lado, apenas causa uma perda limitada de estado e envia o sistema cron de volta no tempo para o ponto em que o último instantâneo foi tirado.

Temos duas opções principais para armazenar nossos dados:

- Externamente em um armazenamento distribuído geralmente disponível
- Em um sistema que armazena o pequeno volume de estado como parte do próprio serviço cron

Ao projetar o sistema, combinamos elementos de ambas as opções.

Armazenamos os logs do Paxos no disco local da máquina onde as réplicas do serviço cron são agendadas. Ter três réplicas em operação padrão implica que temos três cópias dos logs. Também armazenamos os instantâneos no disco local. No entanto, como eles são críticos, também fazemos backup deles em um sistema de arquivos distribuído, protegendo assim contra falhas que afetam as três máquinas.

Não armazenamos logs em nosso sistema de arquivos distribuído. Decidimos conscientemente que perder logs, que representam uma pequena quantidade das mudanças de estado mais recentes, é um risco aceitável. Armazenar logs em um sistema de arquivos distribuído pode acarretar uma penalidade de desempenho substancial causada por pequenas gravações frequentes. A perda simultânea de todas as três máquinas é improvável e, se ocorrer uma perda simultânea, restauraremos automaticamente a partir do instantâneo. Com isso, perdemos apenas uma pequena quantidade de logs: aqueles obtidos desde o último instantâneo, que executamos em intervalos configuráveis. Claro que essas trocas

pode ser diferente dependendo dos detalhes da infraestrutura, bem como dos requisitos colocados no sistema cron.

Além dos logs e instantâneos armazenados no disco local e backups de instantâneos no sistema de arquivos distribuído, uma réplica recém-iniciada pode buscar o instantâneo de estado e todos os logs de uma réplica já em execução na rede. Essa capacidade torna a inicialização da réplica independente de qualquer estado na máquina local. Portanto, reprogramar uma réplica para uma máquina diferente na reinicialização (ou morte da máquina) não é essencialmente um problema para a confiabilidade do serviço.

Executando um Cron Grande

Existem outras implicações menores, mas igualmente interessantes, de executar uma implantação de cron grande. Um cron tradicional é pequeno: no máximo, provavelmente contém dezenas de cron jobs. No entanto, se você executar um serviço cron para milhares de máquinas em um datacenter, seu uso aumentará e você poderá ter problemas.

Cuidado com o grande e conhecido problema dos sistemas distribuídos: o rebanho trovejante. Com base na configuração do usuário, o serviço cron pode causar picos substanciais no uso do datacenter. Quando as pessoas pensam em um “trabalho cron diário”, geralmente configuram esse trabalho para ser executado à meia-noite. Essa configuração funciona bem se o cron job for iniciado na mesma máquina, mas e se o cron job puder gerar um MapReduce com milhares de workers? E se 30 equipes diferentes decidirem executar um cron job diário como esse, no mesmo datacenter? Para resolver este problema, introduzimos uma extensão para o formato crontab.

No crontab comum, os usuários especificam o minuto, hora, dia do mês (ou semana) e mês em que o cron job deve ser iniciado ou asterisco para especificar qualquer valor. Executando à meia-noite, diariamente, teria a especificação crontab de "0 0 * * *" (ou seja, minuto zero, hora zero, todos os dias da semana, todos os meses e todos os dias da semana). Também introduzimos o uso do ponto de interrogação, o que significa que qualquer valor é aceitável, e o sistema cron tem a liberdade de escolher o valor.

Os usuários escolhem esse valor fazendo um hash da configuração do cron job em um determinado intervalo de tempo (por exemplo, 0..23 por hora), distribuindo, portanto, esses lançamentos de maneira mais uniforme.

Apesar dessa mudança, a carga causada pelos cron jobs ainda é muito pontiaguda. O gráfico na [Figura 24-3](#) ilustra o número global agregado de lançamentos de cron jobs no Google. Este gráfico destaca os picos frequentes em lançamentos de cron job, que geralmente são causados por cron jobs que precisam ser lançados em um horário específico, por exemplo, devido à dependência temporal de eventos externos.

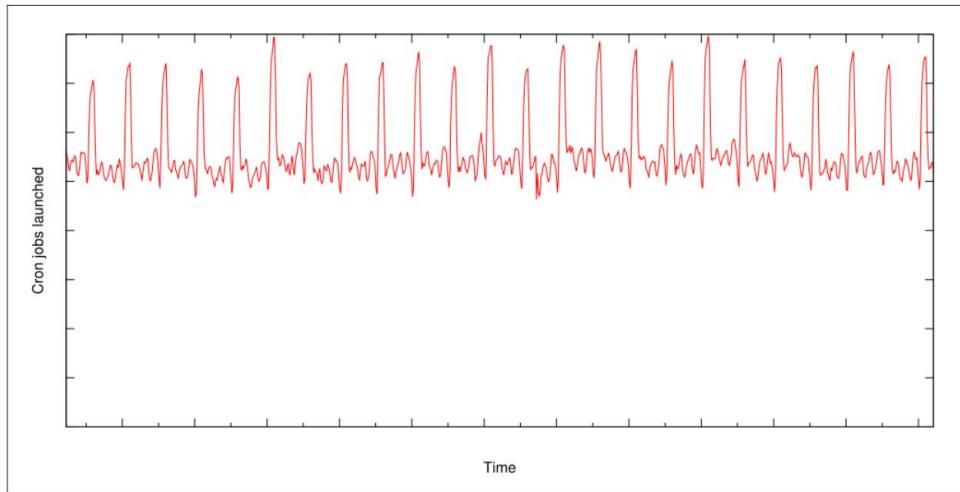


Figura 24-3. O número de cron jobs lançados globalmente

Resumo

Um serviço cron tem sido um recurso fundamental em sistemas UNIX por muitas décadas. A mudança do setor para grandes sistemas distribuídos, nos quais um datacenter pode ser a menor unidade efetiva de hardware, requer mudanças em grandes porções da pilha. Cron não é exceção a esta tendência. Uma análise cuidadosa das propriedades necessárias de um serviço cron e os requisitos de tarefas cron impulsionam o novo design do Google.

Discutimos as novas restrições exigidas por um ambiente de sistema distribuído e um possível projeto do serviço cron baseado na solução do Google. Esta solução requer fortes garantias de consistência no ambiente distribuído. O núcleo da implementação do cron distribuído é, portanto, o Paxos, um algoritmo comum para chegar a um consenso em um ambiente não confiável. O uso do Paxos e a análise correta de novos modos de falha de cron jobs em um ambiente distribuído de grande escala nos permitiu construir um serviço cron robusto que é muito usado no Google.

CAPÍTULO 25**Pipelines de Processamento de Dados**

**Escrito por Dan Dennison
Editado por Tim Harvey**

Este capítulo se concentra nos desafios da vida real de gerenciar pipelines de processamento de dados de profundidade e complexidade. Ele considera o continuum de frequência entre pipelines periódicos que são executados com pouca frequência até pipelines contínuos que nunca param de funcionar e discute as descontinuidades que podem produzir problemas operacionais significativos. Uma nova visão do modelo líder-seguidor é apresentada como uma alternativa mais confiável e de melhor dimensionamento ao pipeline periódico para processamento de Big Data.

Origem do padrão de projeto de pipeline

A abordagem clássica para o processamento de dados é escrever um programa que leia os dados, os transforme de alguma maneira desejada e gere novos dados. Normalmente, o programa é agendado para ser executado sob o controle de um programa de agendamento periódico, como o cron. Esse padrão de design é chamado de pipeline de dados. Os pipelines de dados remontam às rotinas co [Con63], aos arquivos de comunicação DTSS [Bul80], ao canal UNIX [Mcl86] e, posteriormente, aos pipelines ETL,¹ mas esses pipelines ganharam maior atenção com o surgimento de “Big Data,” ou “conjuntos de dados tão grandes e tão complexos que os aplicativos tradicionais de processamento de dados são inadequados.”²

¹ Wikipedia: Extrair, transformar, carregar, http://en.wikipedia.org/wiki/Extract,_transform,_load

² Wikipedia: Big data, http://en.wikipedia.org/wiki/Big_data

Efeito inicial de Big Data no padrão de pipeline simples

Programas que realizam transformações periódicas ou contínuas em Big Data são geralmente chamados de “pipelines simples e monofásicos”.

Dada a escala e a complexidade de processamento inerentes ao Big Data, os programas são normalmente organizados em uma série encadeada, com a saída de um programa tornando-se a entrada para o próximo. Pode haver várias justificativas para esse arranjo, mas ele é normalmente projetado para facilitar o raciocínio sobre o sistema e geralmente não é voltado para a eficiência operacional. Os programas organizados dessa maneira são chamados de pipelines multifásicos, porque cada programa na cadeia atua como uma fase discreta de processamento de dados.

O número de programas encadeados em série é uma medida conhecida como profundidade de um pipeline. Assim, um pipeline raso pode ter apenas um programa com uma medição de profundidade de pipeline correspondente de um, enquanto um pipeline profundo pode ter uma profundidade de pipeline em dezenas ou centenas de programas.

Desafios com o padrão de pipeline periódico

Os pipelines periódicos geralmente são estáveis quando há trabalhadores suficientes para o volume de dados e a demanda de execução está dentro da capacidade computacional. Além disso, instabilidades como gargalos de processamento são evitadas quando o número de trabalhos encadeados e a taxa de transferência relativa entre os trabalhos permanecem uniformes.

Os pipelines periódicos são úteis e práticos, e os executamos regularmente no Google. Eles são escritos com frameworks como MapReduce [Dea04] e Flume [Cha10], entre outros.

No entanto, a experiência coletiva do SRE mostra que o modelo de pipeline periódico é frágil. Descobrimos que quando um pipeline periódico é instalado pela primeira vez com dimensionamento de trabalhadores, periodicidade, técnica de agrupamento e outros parâmetros cuidadosamente ajustados, o desempenho é inicialmente confiável. No entanto, o crescimento orgânico e a mudança inevitavelmente começam a estressar o sistema e surgem problemas. Exemplos de tais problemas incluem tarefas que excedem seu prazo de execução, esgotamento de recursos e blocos de processamento suspensos que envolvem carga operacional correspondente.

Problemas causados pela distribuição desigual do trabalho

O principal avanço do Big Data é a aplicação generalizada de algoritmos “embarracosamente paralelos” [Mol86] para cortar uma grande carga de trabalho em pedaços pequenos o suficiente para caber em máquinas individuais. Às vezes, os blocos exigem uma quantidade desigual de recursos em relação uns aos outros, e raramente é inicialmente óbvio por que determinados blocos exigem quantidades diferentes de recursos. Por exemplo, em uma carga de trabalho partitionada por cliente, os blocos de dados de alguns clientes podem ser muito maiores do que outros.

Como o cliente é o ponto de indivisibilidade, o tempo de execução de ponta a ponta é limitado ao tempo de execução do maior cliente.

O problema do “pedaço suspenso” pode ocorrer quando recursos são atribuídos devido a diferenças entre máquinas em um cluster ou superalocação para um trabalho. Este problema surge devido à dificuldade de algumas operações em tempo real em fluxos, como a classificação de dados “vaporizados”. O padrão de código de usuário típico é esperar que a computação total seja concluída antes de avançar para o próximo estágio do pipeline, geralmente porque a classificação pode estar envolvida, o que requer que todos os dados continuem. Isso pode atrasar significativamente o tempo de conclusão do pipeline, porque a conclusão é bloqueada no desempenho do pior caso, conforme ditado pela metodologia de fragmentação em uso.

Se esse problema for detectado por engenheiros ou infraestrutura de monitoramento de cluster, a resposta pode piorar a situação. Por exemplo, a resposta “sensata” ou “padrão” a um fragmento suspenso é eliminar imediatamente o trabalho e permitir que ele reinicie, pois o bloqueio pode ser o resultado de fatores não determinísticos. No entanto, como as implementações de pipeline por design geralmente não incluem checkpoints, o trabalho em todos os blocos é reiniciado desde o início, desperdiçando tempo, ciclos de CPU e esforço humano investido no ciclo anterior.

Desvantagens de Pipelines Periódicos em Distribuídos Ambientes

Os pipelines periódicos de Big Data são amplamente usados no Google e, portanto, a solução de gerenciamento de cluster do Google inclui um mecanismo de agendamento alternativo para esses pipelines. Esse mecanismo é necessário porque, diferentemente dos pipelines em execução contínua, os pipelines periódicos normalmente são executados como trabalhos em lote de baixa prioridade. Uma designação de prioridade mais baixa funciona bem nesse caso porque o trabalho em lote não é sensível à latência da mesma forma que os serviços da Web voltados para a Internet. Além disso, para controlar o custo maximizando a carga de trabalho da máquina, o Borg (sistema de gerenciamento de cluster do Google, [Ver15]) atribui trabalho em lote às máquinas disponíveis. Essa prioridade pode resultar em latência de inicialização degradada, de modo que os trabalhos de pipeline podem sofrer atrasos de inicialização em aberto.

Os trabalhos invocados por meio desse mecanismo têm várias limitações naturais, resultando em vários comportamentos distintos. Por exemplo, tarefas agendadas nas lacunas deixadas por tarefas de serviço da Web voltadas para usuários podem ser afetadas em termos de disponibilidade de recursos de baixa latência, preço e estabilidade de acesso a recursos. O custo de execução é inversamente proporcional ao atraso de inicialização solicitado e diretamente proporcional aos recursos consumidos. Embora o agendamento em lote possa funcionar sem problemas na prática, o uso excessivo do agendador de lote (Capítulo 24) coloca os trabalhos em risco de preempção (consulte a seção 2.5 de [Ver15]) quando a carga do cluster é alta porque outros usuários estão sem recursos em lote.

À luz das compensações de risco, a execução bem-sucedida de um pipeline periódico bem ajustado é um equilíbrio delicado entre o alto custo dos recursos e o risco de preempções.

Atrasos de até algumas horas podem ser aceitáveis para pipelines que são executados diariamente. No entanto, à medida que a frequência de execução agendada aumenta, o tempo mínimo entre as execuções pode atingir rapidamente o ponto de atraso médio mínimo, colocando um limite inferior na latência que um pipeline periódico pode esperar atingir. Reduzir o intervalo de execução do trabalho abaixo desse limite inferior efetivo simplesmente resulta em comportamento indesejável em vez de aumento do progresso. O modo de falha específico depende da política de agendamento em lote em uso. Por exemplo, cada nova execução pode se acumular no agendador de cluster porque a execução anterior não foi concluída. Pior ainda, a execução atualmente em execução e quase concluída pode ser interrompida quando a próxima execução estiver programada para começar, interrompendo completamente todo o progresso em nome de execuções crescentes.

Observe onde a linha de intervalo ocioso inclinada para baixo cruza o atraso de programação na [Figura 25-1](#). Nesse cenário, diminuir o intervalo de execução muito abaixo de 40 minutos para esse trabalho de aproximadamente 20 minutos resulta em execuções potencialmente sobrepostas com consequências.

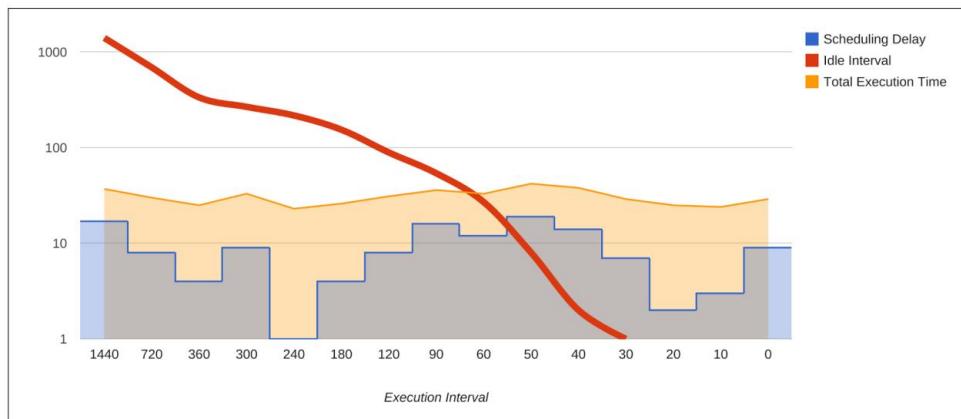


Figura 25-1. Intervalo de execução periódica do pipeline versus tempo ocioso (escala de log)

A solução para este problema é garantir capacidade de servidor suficiente para operação adequada. No entanto, a aquisição de recursos em um ambiente compartilhado e distribuído está sujeita à oferta e demanda. Como esperado, as equipes de desenvolvimento tendem a relutar em passar pelos processos de aquisição de recursos quando os recursos devem ser contribuídos para um pool comum e compartilhados. Para resolver isso, uma distinção entre recursos de programação em lote versus recursos prioritários de produção deve ser feita para racionalizar os custos de aquisição de recursos.

Problemas de monitoramento em pipelines periódicos

Para pipelines com duração de execução suficiente, ter informações em tempo real sobre métricas de desempenho em tempo de execução pode ser tão importante, se não mais importante, do que conhecer as métricas gerais. Isso ocorre porque os dados em tempo real são importantes para fornecer suporte operacional, incluindo resposta a emergências. Na prática, o modelo de monitoramento padrão envolve a coleta de métricas durante a execução do trabalho e o relatório de métricas somente após a conclusão. Se a tarefa falhar durante a execução, nenhuma estatística será fornecida.

Os pipelines contínuos não compartilham esses problemas porque suas tarefas estão em execução constante e sua telemetria é projetada rotineiramente para que as métricas em tempo real estejam disponíveis. As tubulações periódicas não devem ter problemas de monitoramento inerentes, mas observamos uma forte associação.

Problemas de “rebanho trovejante” Além

dos desafios de execução e monitoramento está o problema do “rebanho trovejante” endêmico em sistemas distribuídos, também discutido no [Capítulo 24](#). Dado um pipeline periódico suficientemente grande, para cada ciclo, potencialmente milhares de trabalhadores começam a trabalhar imediatamente. Se houver muitos trabalhadores ou se os trabalhadores forem configurados incorretamente ou invocados por lógica de repetição defeituosa, os servidores nos quais eles são executados ficarão sobrecarregados, assim como os serviços de cluster compartilhados subjacentes, e qualquer infraestrutura de rede que estava sendo usada também ficará sobrecarregada.

Agravando ainda mais esta situação, se a lógica de repetição não for implementada, problemas de correção podem ocorrer quando o trabalho é interrompido após falha e o trabalho não será repetido. Se a lógica de repetição estiver presente, mas for ingênuo ou mal implementada, a repetição em caso de falha pode agravar o problema.

A intervenção humana também pode contribuir para esse cenário. Engenheiros com experiência limitada no gerenciamento de pipelines tendem a amplificar esse problema adicionando mais trabalhadores ao pipeline quando o trabalho não é concluído dentro de um período de tempo desejado.

Independentemente da origem do problema do “rebanho trovejante”, nada é mais difícil para a infraestrutura do cluster e os SREs responsáveis pelos vários serviços de um cluster do que um trabalho de pipeline de 10.000 trabalhadores.

Padrão de Carga Moiré

Às vezes, o problema do rebanho trovejante pode não ser óbvio para detectar isoladamente. Um problema relacionado que chamamos de “padrão de carregamento Moiré” ocorre quando dois ou mais pipelines são executados simultaneamente e suas sequências de execução ocasionalmente se sobreponem, fazendo com que eles consumam simultaneamente um recurso compartilhado comum. Esse problema pode ocorrer mesmo em pipelines contínuos, embora seja menos comum quando a carga chega de forma mais uniforme.

Os padrões de carga moiré são mais aparentes em gráficos de uso de pipeline de recursos compartilhados. Por exemplo, a Figura 25-2 identifica o uso de recursos de três pipelines periódicos. Na Figura 25-3, que é uma versão empilhada dos dados do gráfico anterior, o pico de impacto que causa a dor de plantão ocorre quando a carga agregada se aproxima de 1,2 M.

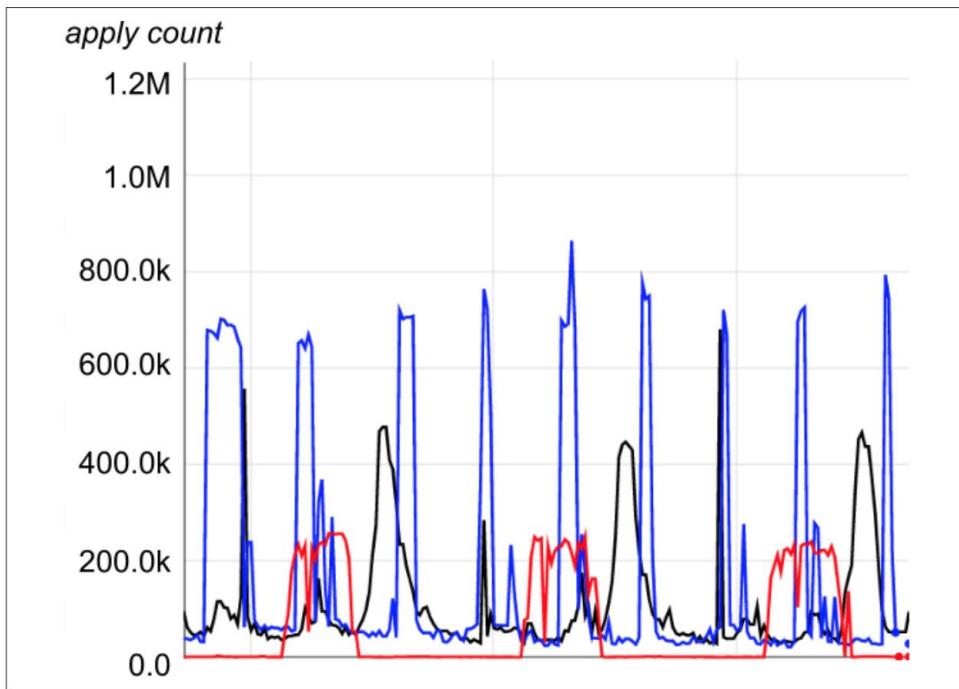


Figura 25-2. Padrão de carga moiré em infraestrutura separada

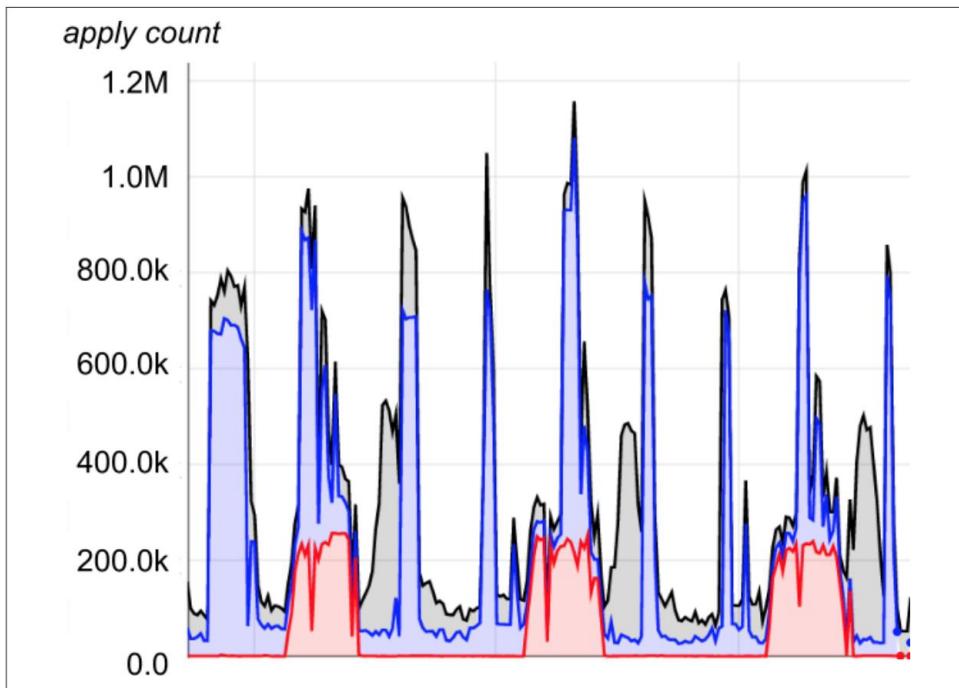


Figura 25-3. Padrão de carga moiré em infraestrutura compartilhada

Introdução ao Google Workflow

Quando um pipeline de lotes inherentemente único é sobre carregado pelas demandas de negócios por resultados continuamente atualizados, a equipe de desenvolvimento do pipeline geralmente considera refatorar o projeto original para satisfazer as demandas atuais ou mudar para um modelo de pipeline contínuo. Infelizmente, as demandas de negócios geralmente ocorrem no momento menos conveniente para refatorar o sistema de pipeline em um sistema de processamento contínuo online. Clientes mais novos e maiores que se deparam com problemas de dimensionamento forçados normalmente também desejam incluir novos recursos e esperam que esses requisitos cumpram prazos imutáveis. Ao antecipar esse desafio, é importante verificar vários detalhes no início do projeto de um sistema envolvendo um pipeline de dados proposto. Certifique-se de definir o escopo da trajetória de crescimento esperada,³ demanda por modificações de design, recursos adicionais esperados e requisitos de latência esperados do negócio.

Diante dessas necessidades, o Google desenvolveu em 2003 um sistema chamado “Workflow” que disponibiliza o processamento contínuo em escala. O fluxo de trabalho usa o líder-seguidor

³ A palestra de Jeff Dean sobre “Conselhos de Engenharia de Software da Construção de Sistemas Distribuídos em Grande Escala” é um excelente recurso: [\[Dea07\]](#).

(trabalhadores) padrão de projeto de sistemas distribuídos [Sha00] e o padrão de projeto de prevalência do sistema.⁴ Essa combinação permite pipelines de dados transacionais de grande escala, garantindo correção com semântica exatamente uma vez.

Fluxo de trabalho como padrão Model-View-Controller

Por causa de como a prevalência do sistema funciona, pode ser útil pensar no Workflow como o equivalente de sistemas distribuídos do padrão model-view-controller conhecido do desenvolvimento da interface do usuário.⁵ Conforme mostrado na Figura 25-4, esse padrão de projeto divide um determinado software aplicativo em três partes interconectadas para separar as representações internas de informações das formas como as informações são apresentadas ou aceitas pelo usuário.⁶

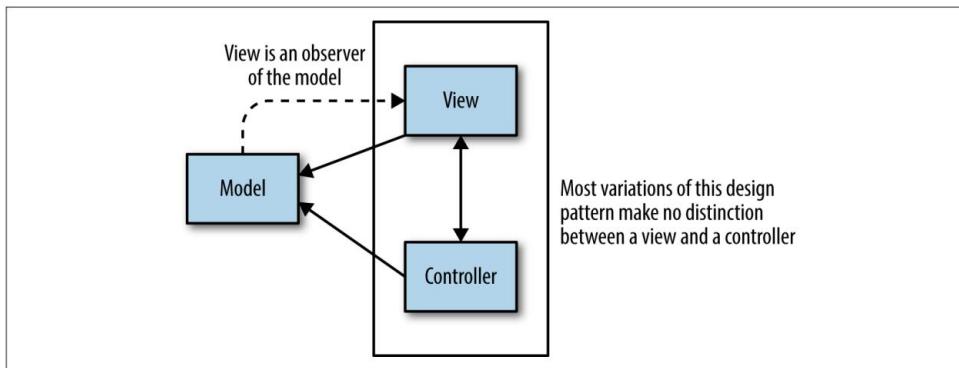


Figura 25-4. O padrão model-view-controller usado no design da interface do usuário

Adaptando esse padrão para Workflow, o modelo é mantido em um servidor chamado “Task Master”. O Mestre de Tarefas usa o padrão de prevalência do sistema para manter todos os estados de trabalho na memória para disponibilidade rápida enquanto registra em diário as mutações de forma síncrona no disco permanente. A visão são os trabalhadores que atualizam continuamente o estado do sistema de forma transacional com o mestre de acordo com sua perspectiva como um subcomponente do pipeline. Embora todos os dados do pipeline possam ser armazenados no Task Master, o melhor desempenho geralmente é alcançado quando apenas os ponteiros para o trabalho são armazenados no Task Master e os dados reais de entrada e saída são armazenados em um sistema de arquivos comum ou outro armazenamento. Apoiando essa analogia, os trabalhadores são completamente apátridas e podem ser descartados a qualquer momento. Um controlador pode ser opcionalmente adicionado como um terceiro componente do sistema para dar suporte eficiente

⁴ Wikipedia: Prevalência do Sistema, http://en.wikipedia.org/wiki/System_Prevalence

⁵ O padrão “model-view-controller” é uma analogia para sistemas distribuídos que foi muito vagamente emprestado de Smalltalk, que foi originalmente usado para descrever a estrutura de design de interfaces gráficas de usuário [Fow08].

⁶ Wikipedia: Model-view-controller, <http://en.wikipedia.org/wiki/Model%20view%20controller>

uma série de atividades auxiliares do sistema que afetam o pipeline, como dimensionamento de tempo de execução do pipeline, captura instantânea, controle de estado do ciclo de trabalho, reversão do estado do pipeline ou até mesmo interdição global para continuidade de negócios. A [Figura 25-5](#) ilustra o padrão de projeto.

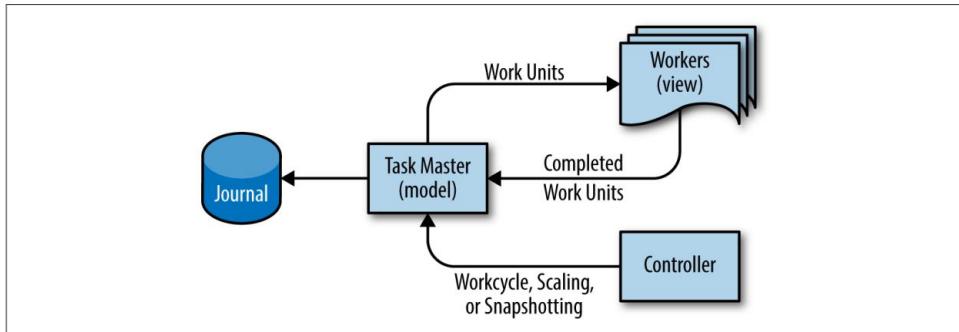


Figura 25-5. O padrão de design model-view-controller adaptado para o Google Workflow

Etapas de Execução em Fluxo de Trabalho

Podemos aumentar a profundidade do pipeline para qualquer nível dentro do Workflow subdividindo o processamento em grupos de tarefas mantidos no Task Master. Cada grupo de tarefas mantém o trabalho correspondente a um estágio de pipeline que pode realizar operações arbitrárias em algum dado. É relativamente simples realizar mapeamento, embaralhamento, classificação, divisão, mesclagem ou qualquer outra operação em qualquer estágio.

Um estágio geralmente tem algum tipo de trabalhador associado a ele. Pode haver várias instâncias simultâneas de um determinado tipo de trabalhador, e os trabalhadores podem ser autoagendados no sentido de que podem procurar diferentes tipos de trabalho e escolher qual tipo executar.

O trabalhador consome unidades de trabalho de um estágio anterior e produz unidades de saída.

A saída pode ser um ponto final ou entrada para algum outro estágio de processamento. Dentro do sistema, é fácil garantir que todo o trabalho seja executado, ou pelo menos refletido no estado permanente, exatamente uma vez.

Garantias de Correção de Trabalho

Não é prático armazenar todos os detalhes do estado do pipeline dentro do Task Master, porque o Task Master é limitado pelo tamanho da RAM. No entanto, uma garantia de correção dupla persiste porque o mestre mantém uma coleção de ponteiros para dados com nomes exclusivos e cada unidade de trabalho tem uma concessão exclusiva. Os trabalhadores adquirem trabalho com um contrato de arrendamento e só podem comprometer o trabalho de tarefas para as quais possuem atualmente um contrato de arrendamento válido.

Para evitar a situação em que um trabalhador órfão pode continuar trabalhando em uma unidade de trabalho, destruindo assim o trabalho do trabalhador atual, cada arquivo de saída aberto por um

trabalhador tem um nome exclusivo. Dessa forma, até mesmo os trabalhadores órfãos podem continuar escrevendo independentemente do mestre até que tentem se comprometer. Ao tentar um compromisso, eles não poderão fazê-lo porque outro trabalhador detém a locação dessa unidade de trabalho. Além disso, os trabalhadores órfãos não podem destruir o trabalho produzido por um trabalhador válido, porque o esquema de nome de arquivo exclusivo garante que cada trabalhador esteja gravando em um arquivo distinto. Dessa forma, a dupla garantia de correção se mantém: os arquivos de saída são sempre únicos e o estado do pipeline está sempre correto em virtude de tarefas com concessões.

Como se não bastasse uma dupla garantia de correção, o Workflow também faz a versão de todas as tarefas. Se a tarefa for atualizada ou a concessão da tarefa for alterada, cada operação produzirá uma nova tarefa exclusiva substituindo a anterior, com um novo ID atribuído à tarefa. Como toda a configuração de pipeline no Workflow é armazenada no mestre de tarefas da mesma forma que as próprias unidades de trabalho, para confirmar o trabalho, um trabalhador deve possuir uma concessão ativa e fazer referência ao número de ID da tarefa da configuração usada para produzir seu resultado. Se a configuração foi alterada enquanto a unidade de trabalho estava em execução, todos os trabalhadores desse tipo não poderão se comprometer apesar de possuírem concessões atuais. Assim, todo o trabalho realizado após uma mudança de configuração é consistente com a nova configuração, ao custo de trabalho sendo jogado fora por trabalhadores infelizes o suficiente para manter os antigos arrendamentos.

Essas medidas fornecem uma garantia tripla de correção: configuração, propriedade do arrendamento e exclusividade do nome do arquivo. No entanto, mesmo isso não é suficiente para todos os casos.

Por exemplo, e se o endereço de rede do Task Master mudou e um Task Master diferente o substituiu no mesmo endereço? E se uma corrupção de memória alterasse o endereço IP ou o número da porta, resultando em outro Mestre de Tarefas do outro lado? Ainda mais comumente, e se alguém (mal) configurou sua configuração do Task Master inserindo um balanceador de carga na frente de um conjunto de Task Masters independentes?

O fluxo de trabalho incorpora um token de servidor, um identificador exclusivo para esse mestre de tarefas específico, nos metadados de cada tarefa para evitar que um mestre de tarefas não autorizado ou configurado incorretamente corrompa o pipeline. Tanto o cliente quanto o servidor verificam o token em cada operação, evitando uma configuração incorreta muito útil na qual todas as operações são executadas sem problemas até que ocorra uma colisão de identificador de tarefa.

Para resumir, as quatro garantias de correção do Workflow são:

- A saída do trabalhador por meio de tarefas de configuração cria barreiras para prever trabalho.
- Todo trabalho comprometido requer um contrato de arrendamento atualmente válido mantido pelo trabalhador.
- Os arquivos de saída são nomeados exclusivamente pelos trabalhadores.
- O cliente e o servidor validam o próprio mestre de tarefas verificando um token de servidor em cada operação.

Neste ponto, pode ocorrer a você que seria mais simples renunciar ao mestre de tarefas especializado e usar o Spanner [Cor12] ou outro banco de dados. No entanto, o Workflow é especial porque cada tarefa é única e imutável. Essas propriedades gêmeas evitam que muitos problemas potencialmente sutis com distribuição de trabalho em larga escala ocorram.

Por exemplo, o aluguel obtido pelo trabalhador faz parte da própria tarefa, exigindo uma tarefa totalmente nova mesmo para alterações de aluguel. Se um banco de dados é usado diretamente e seus logs de transações funcionam como um “diário”, cada leitura deve ser parte de uma transação de longa duração. Esta configuração é certamente possível, mas terrivelmente ineficiente.

Garantindo a Continuidade dos Negócios

Os pipelines de Big Data precisam continuar processando apesar de falhas de todos os tipos, incluindo cortes de fibra, eventos climáticos e falhas de rede elétrica em cascata. Esses tipos de falhas podem desabilitar datacenters inteiros. Além disso, pipelines que não empregam a prevalência do sistema para obter fortes garantias sobre a conclusão do trabalho são frequentemente desabilitados e entram em um estado indefinido. Essa lacuna de arquitetura cria uma estratégia de continuidade de negócios frágil e implica em duplicação em massa dispendiosa de esforços para restaurar pipelines e dados.

O fluxo de trabalho resolve esse problema de forma conclusiva para pipelines de processamento contínuo. Para obter consistência global, o Task Master armazena diários no Spanner, usando-o como um sistema de arquivos globalmente disponível, globalmente consistente, mas de baixo rendimento. Para determinar qual Task Master pode escrever, cada Task Master usa o serviço de bloqueio distribuído chamado Chubby [Bur06] para eleger o escritor, e o resultado é persistido no Spanner. Finalmente, os clientes procuram o mestre de tarefas atual usando serviços de nomenclatura internos.

Como o Spanner não é um sistema de arquivos de alto rendimento, os fluxos de trabalho distribuídos globalmente empregam dois ou mais fluxos de trabalho locais executados em clusters distintos, além de uma noção de tarefas de referência armazenadas no fluxo de trabalho global. À medida que as unidades de trabalho (tarefas) são consumidas por meio de um pipeline, tarefas de referência equivalentes são inseridas no Workflow global pelo binário rotulado “estágio 1” na [Figura 25-6](#). À medida que as tarefas terminam, as tarefas de referência são removidas transacionalmente do Workflow global, conforme descrito no “estágio n” da [Figura 25-6](#). Se as tarefas não puderem ser removidas do Workflow global, o Workflow local será bloqueado até que o Workflow global fique disponível novamente, garantindo a correção transacional.

Para automatizar o failover, um binário auxiliar rotulado como “estágio 1” na [Figura 25-6](#) é executado dentro de cada Workflow local. O fluxo de trabalho local permanece inalterado, conforme descrito pela caixa “fazer trabalho” no diagrama. Esse binário auxiliar atua como um “controlador” no sentido MVC e é responsável por criar tarefas de referência, bem como atualizar uma tarefa de pulsação especial dentro do fluxo de trabalho global. Se a tarefa de pulsação não for atualizada dentro do período de tempo limite, o binário auxiliar do Workflow remoto captura o trabalho em andamento conforme documentado pelas tarefas de referência e o pipeline continua, sem impedimentos por qualquer coisa que o ambiente possa fazer com o trabalho.

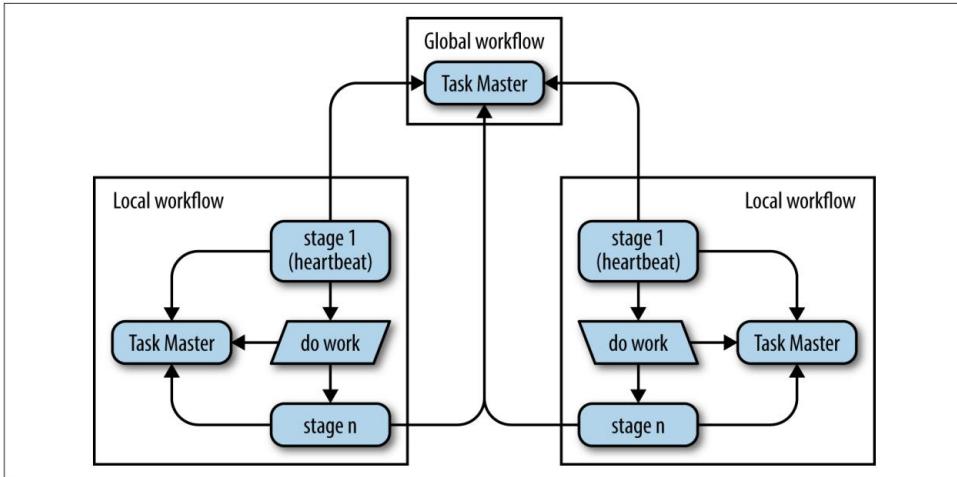


Figura 25-6. Um exemplo de fluxo de dados e processos distribuídos usando pipelines de fluxo de trabalho

Resumo e Observações Finais

As tubulações periódicas são valiosas. No entanto, se um problema de processamento de dados for contínuo ou crescer organicamente para se tornar contínuo, não use um pipeline periódico. Em vez disso, use uma tecnologia com características semelhantes ao Workflow.

Descobrimos que o processamento contínuo de dados com fortes garantias, conforme fornecido pelo Workflow, tem bom desempenho e dimensionamento em infraestrutura de cluster distribuído, produz resultados nos quais os usuários podem confiar e é um sistema estável e confiável para a equipe de Engenharia de Confiabilidade do Site gerenciar e manter.

CAPÍTULO 26

Integridade dos dados: o que você lê é O que você escreveu

Escrito por Raymond Blum e Rhandeev Singh
Editado por Betsy Beyer

O que é "integridade de dados"? Quando os usuários vêm em primeiro lugar, a integridade dos dados é o que os usuários pensam que é.

Podemos dizer que a integridade dos dados é uma medida da acessibilidade e precisão dos armazenamentos de dados necessários para fornecer aos usuários um nível adequado de serviço. Mas esta definição é insuficiente.

Por exemplo, se um bug de interface do usuário no Gmail exibir uma caixa de correio vazia por muito tempo, os usuários podem acreditar que os dados foram perdidos. Assim, mesmo que nenhum dado fosse realmente perdido, o mundo questionaria a capacidade do Google de agir como um administrador responsável de dados, e a viabilidade da computação em nuvem estaria ameaçada. Se o Gmail exibisse uma mensagem de erro ou manutenção por muito tempo enquanto "apenas um pouco de metadados" fosse reparado, a confiança dos usuários do Google também seria corroída.

Quanto tempo é "muito longo" para que os dados fiquem indisponíveis? Conforme demonstrado por um incidente real do Gmail em 2011 [Hic11], quatro dias é muito tempo, talvez "muito tempo". Posteriormente, acreditamos que 24 horas é um bom ponto de partida para estabelecer o limite de "muito longo" para o Google Apps.

Raciocínio semelhante se aplica a aplicativos como Google Fotos, Drive, Cloud Storage e Cloud Datastore, porque os usuários não necessariamente fazem distinção entre esses produtos distintos (raciocínio, "este produto ainda é o Google" ou "Google, Amazon, o que for; isso produto ainda faz parte da nuvem"). Perda de dados, corrupção de dados e indisponibilidade estendida são normalmente indistinguíveis para os usuários. Portanto, a integridade dos dados se aplica a todos os tipos de dados em todos os serviços. Ao considerar a integridade dos dados,

o que importa é que os serviços na nuvem permaneçam acessíveis aos usuários. O acesso do usuário aos dados é especialmente importante.

Requisitos Estritos de Integridade de Dados

Ao considerar as necessidades de confiabilidade de um determinado sistema, pode parecer que as necessidades de uptime (disponibilidade do serviço) são mais rigorosas do que as de integridade dos dados. Por exemplo, os usuários podem achar inaceitável uma hora de inatividade de e-mail, enquanto podem viver mal-humorados com uma janela de tempo de quatro dias para recuperar uma caixa de correio. No entanto, há uma maneira mais apropriada de considerar as demandas de tempo de atividade versus integridade de dados.

Um SLO de 99,99% de tempo de atividade deixa espaço para apenas uma hora de inatividade em um ano inteiro. Esse SLO estabelece um padrão bastante alto, que provavelmente excede as expectativas da maioria dos usuários da Internet e do Enterprise.

Por outro lado, um SLO de 99,99% de bytes bons em um artefato de 2 GB tornaria documentos, executáveis e bancos de dados corrompidos (até 200 KB ilegíveis). Essa quantidade de corrupção é catastrófica na maioria dos casos – resultando em executáveis com opcodes aleatórios e bancos de dados completamente descarregáveis.

Da perspectiva do usuário, então, cada serviço tem requisitos independentes de tempo de atividade e integridade de dados, mesmo que esses requisitos estejam implícitos. O pior momento para discordar dos usuários sobre esses requisitos é após o desaparecimento de seus dados!



Para revisar nossa definição anterior de integridade de dados, podemos dizer que integridade de dados significa que os serviços na nuvem permanecem acessíveis aos usuários. O acesso do usuário aos dados é especialmente importante, portanto, esse acesso deve permanecer em perfeitas condições.

Agora, suponha que um artefato seja corrompido ou perdido exatamente uma vez por ano. Se a perda for irrecuperável, o tempo de atividade do artefato afetado será perdido para aquele ano. O meio mais provável de evitar essa perda é por meio da detecção proativa, juntamente com o reparo rápido.

Em um universo alternativo, suponha que a corrupção foi detectada imediatamente antes que os usuários fossem afetados e que o artefato foi removido, corrigido e devolvido ao serviço em meia hora. Ignorando qualquer outro tempo de inatividade durante esses 30 minutos, esse objeto estaria 99,99% disponível naquele ano.

Surpreendentemente, pelo menos da perspectiva do usuário, nesse cenário, a integridade dos dados ainda é 100% (ou perto de 100%) durante o tempo de vida acessível do objeto. Conforme demonstrado por este exemplo, o segredo para uma integridade de dados superior é a detecção proativa e o reparo e a recuperação rápidos.

Escolhendo uma estratégia para integridade de dados superior

Há muitas estratégias possíveis para detecção, reparo e recuperação rápida de dados perdidos. Todas essas estratégias trocam o tempo de atividade contra a integridade dos dados em relação aos usuários afetados. Algumas estratégias funcionam melhor do que outras, e algumas estratégias requerem investimentos de engenharia mais complexos do que outras. Com tantas opções disponíveis, quais estratégias você deve utilizar? A resposta depende do seu paradigma de computação.

A maioria dos aplicativos de computação em nuvem busca otimizar alguma combinação de tempo de atividade, latência, escala, velocidade e privacidade. Para fornecer uma definição funcional para cada um desses termos:

Tempo de atividade

Também conhecida como disponibilidade, a proporção de tempo que um serviço é utilizável por seus usuários.

Latência

Quão responsivo um serviço aparece para seus usuários.

Escala

O volume de usuários de um serviço e a mistura de cargas de trabalho que o serviço pode manipular antes que a latência sofra ou o serviço desmorone.

Velocidade

A rapidez com que um serviço pode inovar para fornecer aos usuários valor superior a um custo razoável.

Privacidade

Este conceito impõe requisitos complexos. Como simplificação, este capítulo limita seu escopo na discussão de privacidade à exclusão de dados: os dados devem ser destruídos dentro de um prazo razoável após a exclusão dos usuários.

Muitos aplicativos em nuvem evoluem continuamente sobre uma mistura de APIs ACID1 e BASE2 para atender às demandas desses cinco componentes.³ O BASE permite maior disponibilidade

1 Atomicidade, Consistência, Isolamento, Durabilidade; consulte <https://en.wikipedia.org/wiki/ACID>. bancos de dados SQL, como MySQL e PostgreSQL se esforçam para alcançar essas propriedades.

2 basicamente disponível, estado suave, consistência eventual; consulte https://en.wikipedia.org/wiki/Eventual_consistency. Sistemas BASE, como Bigtable e Megastore, também são frequentemente descritos como "NoSQL".

3 Para ler mais sobre APIs ACID e BASE, consulte [Gol14] e [Bai13].

do que o ACID, em troca de uma garantia de consistência distribuída mais suave. Especificamente, o BASE garante apenas que, uma vez que um dado não seja mais atualizado, seu valor eventualmente se tornará consistente entre os locais de armazenamento (potencialmente distribuídos).

O cenário a seguir fornece um exemplo de como as compensações entre tempo de atividade, latência, escala, velocidade e privacidade podem ocorrer.

Quando a velocidade supera outros requisitos, os aplicativos resultantes dependem de uma coleção arbitrária de APIs que são mais familiares para os desenvolvedores específicos que trabalham no aplicativo.

Por exemplo, um aplicativo pode tirar proveito de uma API de armazenamento BLOB⁴ eficiente , como o Blobstore, que negligencia a consistência distribuída em favor do dimensionamento para cargas de trabalho pesadas com alto tempo de atividade, baixa latência e baixo custo. Para compensar:

- O mesmo aplicativo pode confiar pequenas quantidades de metadados autoritativos pertencentes a seus blobs a um serviço baseado em Paxos de maior latência, menos disponível e mais caro, como Megastore [Bak11], [Lam98].
- Certos clientes do aplicativo podem armazenar em cache alguns desses metadados localmente e acessar blobs diretamente, reduzindo a latência ainda mais do ponto de vista dos usuários. • Outro aplicativo pode manter metadados no Bigtable, sacrificando uma forte consistência distribuída porque seus desenvolvedores estão familiarizados com o Bigtable.

Esses aplicativos em nuvem enfrentam uma variedade de desafios de integridade de dados em tempo de execução, como integridade referencial entre datastores (no exemplo anterior, Blobstore, Megastore e caches do lado do cliente). Os caprichos da alta velocidade ditam que mudanças de esquema, migrações de dados, empilhamento de novos recursos sobre recursos antigos, reescritas e pontos de integração em evolução com outros aplicativos conluem para produzir um ambiente repleto de relacionamentos complexos entre vários dados que nenhum único engenheiro totalmente groks.

Para evitar que os dados desse aplicativo se degradem diante dos olhos de seus usuários, é necessário um sistema de verificações e balanços fora de banda dentro e entre seus armazenamentos de dados.

[“Terceira Camada: Detecção Antecipada” na página 356](#) discute esse sistema.

Além disso, se tal aplicativo depende de backups independentes e não coordenados de vários armazenamentos de dados (no exemplo anterior, Blobstore e Megastore), sua capacidade de fazer uso efetivo de dados restaurados durante um esforço de recuperação de dados é complicada pela variedade de relacionamentos entre dados restaurados e ativos. Nosso aplicativo de exemplo teria que classificar e distinguir entre blobs restaurados versus Megastore ao vivo, Megastore restaurado versus blobs ativos, blobs restaurados versus Megastore restaurado e interações com caches do lado do cliente.

4 Objeto Binário Grande; veja https://en.wikipedia.org/wiki/Binary_large_object.

Considerando essas dependências e complicações, quantos recursos devem ser investidos em esforços de integridade de dados e onde?

Backups versus arquivos

Tradicionalmente, as empresas “protegem” os dados contra perdas investindo em estratégias de backup. No entanto, o foco real desses esforços de backup deve ser a recuperação de dados, que distingue backups reais de arquivos. Como às vezes é observado: Ninguém quer realmente fazer backups; o que as pessoas realmente querem são restaurações.

Seu “backup” é realmente um arquivo, em vez de apropriado para uso em recuperação de desastres?



A diferença mais importante entre backups e archives é que os backups podem ser carregados de volta em um aplicativo, enquanto os archives não. Portanto, backups e archives têm casos de uso bastante diferentes.

Arquiva os dados em segurança por longos períodos de tempo para atender às necessidades de auditoria, descoberta e conformidade. A recuperação de dados para esses fins geralmente não precisa ser concluída dentro dos requisitos de tempo de atividade de um serviço. Por exemplo, pode ser necessário reter dados de transações financeiras por sete anos. Para atingir esse objetivo, você pode mover os logs de auditoria acumulados para armazenamento de arquivamento de longo prazo em um local externo uma vez por mês. Recuperar e recuperar os logs durante uma auditoria financeira de um mês pode levar uma semana, e essa janela de tempo de uma semana para recuperação pode ser aceitável para um arquivamento.

Por outro lado, quando ocorre um desastre, os dados devem ser recuperados de backups reais rapidamente, de preferência dentro das necessidades de tempo de atividade de um serviço. Caso contrário, os usuários afetados ficam sem acesso útil ao aplicativo desde o início do problema de integridade dos dados até a conclusão do esforço de recuperação.

Também é importante considerar que, como os dados mais recentes correm risco até que seja feito backup com segurança, pode ser ideal agendar backups reais (em oposição a arquivamentos) para ocorrer diariamente, de hora em hora ou com mais frequência, usando completo e incremental ou contínuo (streaming) abordagens.

Portanto, ao formular uma estratégia de backup, considere a rapidez com que você precisa se recuperar de um problema e quantos dados recentes você pode perder.

Requisitos do ambiente de nuvem em perspectiva

Os ambientes de nuvem apresentam uma combinação única de desafios técnicos:

- Se o ambiente usar uma mistura de backup transacional e não transacional e soluções de restauração, os dados recuperados não estarão necessariamente corretos.
- Se os serviços devem evoluir sem cair para manutenção, versões diferentes da lógica de negócios podem atuar nos dados em paralelo.
- Se os serviços de interação tiverem versões independentes, versões incompatíveis de diferentes serviços podem interagir momentaneamente, aumentando ainda mais a chance de corrupção acidental de dados ou perda de dados.

Além disso, para manter a economia de escala, os provedores de serviços devem fornecer apenas um número limitado de APIs. Essas APIs devem ser simples e fáceis de usar para a grande maioria dos aplicativos, ou poucos clientes as usarão. Ao mesmo tempo, as APIs devem ser robustas o suficiente para entender o seguinte:

- Localidade e cache de dados •

Distribuição de dados local e global • Consistência forte e/ou eventual • Durabilidade, backup e recuperação de dados

Caso contrário, clientes sofisticados não podem migrar aplicativos para a nuvem, e aplicativos simples que se tornam complexos e grandes precisarão de reescritas completas para usar APIs diferentes e mais complexas.

Os problemas surgem quando os recursos de API anteriores são usados em determinadas combinações. Se o provedor de serviços não resolver esses problemas, os aplicativos que se deparam com esses desafios devem identificá-los e resolvê-los de forma independente.

Objetivos do Google SRE na manutenção da integridade dos dados e Disponibilidade

Embora o objetivo da SRE de “manter a integridade de dados persistentes” seja uma boa visão, prosperamos em objetivos concretos com indicadores mensuráveis. O SRE define as principais métricas que usamos para definir expectativas para os recursos de nossos sistemas e processos por meio de testes e para rastrear seu desempenho durante um evento real.

A integridade dos dados é o meio; Disponibilidade de dados é o objetivo

A integridade dos dados refere-se à precisão e consistência dos dados ao longo de sua vida útil.

Os usuários precisam saber que as informações estarão corretas e não mudarão de forma inesperada desde o momento em que são registradas pela primeira vez até a última vez em que são observadas. Mas será que tal garantia é suficiente?

Considere o caso de um provedor de e-mail que sofreu uma interrupção de dados de uma semana [Kinc09].

No espaço de 10 dias, os usuários tiveram que encontrar outros métodos temporários de conduzir seus negócios com a expectativa de que logo retornariam às suas contas de e-mail estabelecidas, identidades e históricos acumulados.

Então, a pior notícia possível chegou: o provedor anunciou que, apesar das expectativas anteriores, o tesouro de e-mails e contatos anteriores na verdade havia desaparecido - evaporado e nunca mais seria visto. Parecia que uma série de contratemplos no gerenciamento da integridade dos dados conspirava para deixar o provedor de serviços sem backups utilizáveis. Usuários furiosos ficaram com suas identidades provisórias ou estabeleceram novas identidades, abandonando seu antigo provedor de e-mail problemático.

Mas espere! Vários dias após a declaração de perda absoluta, o provedor anunciou que as informações pessoais dos usuários poderiam ser recuperadas. Não houve perda de dados; isso foi apenas uma interrupção. Tudo foi bem!

Exceto, nem tudo estava bem. Os dados do usuário foram preservados, mas os dados não eram acessíveis pelas pessoas que precisavam deles por muito tempo.

A moral deste exemplo: do ponto de vista do usuário, a integridade dos dados sem disponibilidade de dados esperada e regular é efetivamente o mesmo que não ter dados.

Fornecendo um sistema de recuperação, em vez de um sistema de backup

Fazer backups é uma tarefa classicamente negligenciada, delegada e adiada da administração do sistema. Os backups não são uma prioridade alta para ninguém — eles consomem tempo e recursos continuamente e não geram nenhum benefício visível imediato. Por esse motivo, a falta de diligência na implementação de uma estratégia de backup geralmente é recebida com um revirar de olhos simpático. Pode-se argumentar que, como a maioria das medidas de proteção contra perigos de baixo risco, tal atitude é pragmática. O problema fundamental com essa estratégia indiferente é que os perigos que ela acarreta podem ser de baixo risco, mas também de alto impacto. Quando os dados do seu serviço estão indisponíveis, sua resposta pode fazer ou quebrar seu serviço, produto e até mesmo sua empresa.

Em vez de se concentrar no trabalho ingrato de fazer um backup, é muito mais útil, para não dizer mais fácil, motivar a participação em fazer backups concentrando-se em uma tarefa com uma recompensa visível: a restauração! Os backups são um imposto, pago de forma contínua pelo serviço municipal de disponibilidade de dados garantida. Em vez de enfatizar a

fiscal, chamar a atenção para o serviço dos fundos fiscais: disponibilização de dados. Não fazemos as equipes "praticar" seus backups, em vez disso:

- As equipes definem objetivos de nível de serviço (SLOs) para disponibilidade de dados em uma variedade de modos de falha.
- Uma equipe pratica e demonstra sua capacidade de atender a esses SLOs.

Tipos de falhas que levam à perda de dados Conforme

ilustrado pela [Figura 26-1](#), em um nível muito alto, existem 24 tipos distintos de falhas quando os 3 fatores podem ocorrer em qualquer combinação. Você deve considerar cada uma dessas falhas potenciais ao projetar um programa de integridade de dados. Os fatores dos modos de falha de integridade dos dados são os seguintes:

Causa raiz

Uma perda irrecuperável de dados pode ser causada por vários fatores: ação do usuário, erro do operador, bugs de aplicativos, defeitos na infraestrutura, hardware defeituoso ou catástrofes no local.

Escopo

Algumas perdas são generalizadas, afetando muitas entidades. Algumas perdas são restritas e direcionadas, excluindo ou corrompendo dados específicos de um pequeno subconjunto de usuários.

Taxa

Algumas perdas de dados são um evento big bang (por exemplo, 1 milhão de linhas são substituídas por apenas 10 linhas em um único minuto), enquanto algumas perdas de dados são crescentes (por exemplo, 10 linhas de dados são excluídas a cada minuto ao longo das semanas).

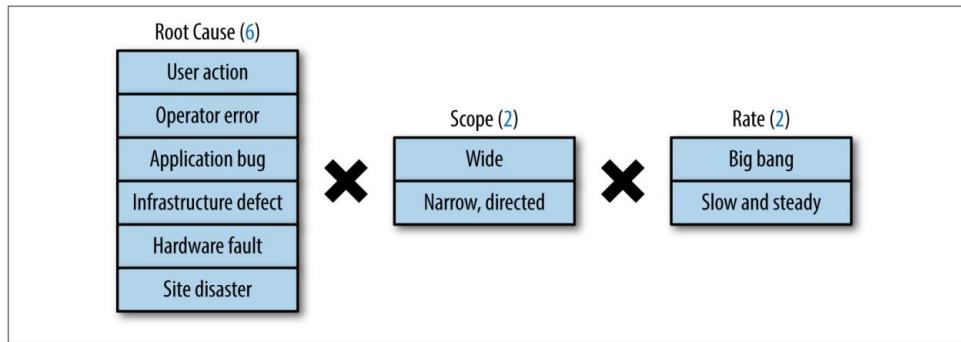


Figura 26-1. Os fatores dos modos de falha de integridade de dados

Um plano de restauração eficaz deve levar em conta qualquer um desses modos de falha que ocorrem em qualquer combinação concebível. O que pode ser uma estratégia perfeitamente eficaz para proteger

contra uma perda de dados causada por um bug de aplicativo rastejante pode não ajudar em nada quando seu datacenter de colocation pegar fogo.

Um estudo de 19 esforços de recuperação de dados no Google descobriu que os cenários de perda de dados visíveis do usuário mais comuns envolviam exclusão de dados ou perda de integridade referencial causada por bugs de software. As variantes mais desafiadoras envolveram corrupção ou exclusão de baixo grau que foi descoberta semanas a meses depois que os bugs foram lançados pela primeira vez no ambiente de produção. Portanto, as proteções que o Google emprega devem ser adequadas para evitar ou se recuperar desses tipos de perda.

Para se recuperar de tais cenários, um aplicativo grande e bem-sucedido precisa recuperar dados de talvez milhões de usuários espalhados por dias, semanas ou meses. O aplicativo também pode precisar recuperar cada artefato afetado em um único ponto no tempo. Esse cenário de recuperação de dados é chamado de “recuperação pontual” fora do Google e “viagem no tempo” dentro do Google.

Uma solução de backup e recuperação que fornece recuperação pontual para um aplicativo em seus datastores ACID e BASE, ao mesmo tempo em que atende a metas rigorosas de tempo de atividade, latência, escalabilidade, velocidade e custo é uma quimera hoje!

Resolver esse problema com seus próprios engenheiros implica sacrificar a velocidade. Muitos projetos se comprometem adotando uma estratégia de backup em camadas sem recuperação pontual. Por exemplo, as APIs abaixo de seu aplicativo podem oferecer suporte a vários mecanismos de recuperação de dados. “Instantâneos” locais caros podem fornecer proteção limitada contra bugs de aplicativos e oferecer funcionalidade de restauração rápida, então você pode reter alguns dias desses “instantâneos” locais, tirados com várias horas de intervalo. Cópias completas e incrementais econômicas a cada dois dias podem ser retidas por mais tempo. A recuperação pontual é um recurso muito bom se uma ou mais dessas estratégias o suportarem.

Considere as opções de recuperação de dados fornecidas pelas APIs de nuvem que você está prestes a usar. Negocie a recuperação pontual contra uma estratégia em camadas, se necessário, mas não recorra a não usar nenhuma delas! Se você puder ter os dois recursos, use os dois recursos. Cada um desses recursos (ou ambos) será valioso em algum momento.

Desafios de manter a integridade dos dados profunda e ampla Ao projetar um programa de integridade de dados, é importante reconhecer que replicação e redundância não são recuperabilidade.

Problemas de dimensionamento: Completos, incrementais e as forças concorrentes de backups e restaurações Uma resposta clássica, mas falha, à pergunta “Você tem um backup?” é “Temos algo ainda melhor do que um backup - replicação!” A replicação oferece muitos benefícios, incluindo a localidade dos dados e proteção contra um desastre específico do local, mas não pode protegê-lo de muitas fontes de perda de dados. Datastores que sincronizam automaticamente multiÿ

plas réplicas garantem que uma linha de banco de dados corrompida ou uma exclusão errônea sejam enviadas para todas as suas cópias, provavelmente antes que você possa isolar o problema.

Para resolver esse problema, você pode fazer cópias não veiculadas de seus dados em algum outro formato, como exportações frequentes de banco de dados para um arquivo nativo. Essa medida adicional adiciona proteção contra os tipos de erros contra os quais a replicação não protege – erros de usuário e bugs de camada de aplicativo – mas não faz nada para proteger contra perdas introduzidas em uma camada inferior. Essa medida também apresenta risco de bugs durante a conversão de dados (em ambas as direções) e durante o armazenamento do arquivo nativo, além de possíveis incompatibilidades na semântica entre os dois formatos. Imagine um ataque de dia zero⁵ em algum nível baixo de sua pilha, como o sistema de arquivos ou driver de dispositivo. Quaisquer cópias que dependam do componente de software comprometido, incluindo as exportações de banco de dados que foram gravadas no mesmo sistema de arquivos que suporta seu banco de dados, são vulneráveis.

Assim, vemos que a diversidade é fundamental: proteger contra uma falha na camada X requer armazenar dados em diversos componentes nessa camada. O isolamento de mídia protege contra falhas de mídia: é improvável que um bug ou ataque em um driver de dispositivo de disco afete as unidades de fita. Se pudéssemos, faríamos cópias de backup de nossos valiosos dados em tabletas de argila.⁶ As forças da atualização dos dados e da conclusão da restauração competem com a proteção abrangente. Quanto mais para baixo na pilha você enviar um instantâneo de seus dados, mais tempo levará para fazer uma cópia, o que significa que a frequência de cópias diminui. No nível do banco de dados, uma transação pode levar alguns segundos para ser replicada. Exportar um instantâneo de banco de dados para o sistema de arquivos abaixo pode levar 40 minutos. Um backup completo do sistema de arquivos subjacente pode levar horas.

Nesse cenário, você pode perder até 40 minutos dos dados mais recentes ao restaurar o instantâneo mais recente. Uma restauração do backup do sistema de arquivos pode resultar em horas de transações ausentes. Além disso, a restauração provavelmente leva tanto tempo quanto o backup, portanto, o carregamento dos dados pode levar horas. Obviamente, você gostaria de ter os dados mais recentes de volta o mais rápido possível, mas, dependendo do tipo de falha, essa cópia mais recente e disponível imediatamente pode não ser uma opção.

Retenção

A retenção — por quanto tempo você mantém cópias de seus dados — é outro fator a ser considerado em seus planos de recuperação de dados.

Embora seja provável que você ou seus clientes percebam rapidamente o esvaziamento repentino de um banco de dados inteiro, pode levar dias para uma perda mais gradual de dados atrair o

5 Veja [https://en.wikipedia.org/wiki/Zero-day_\(computing\)](https://en.wikipedia.org/wiki/Zero-day_(computing)).

6 Tábuas de barro são os mais antigos exemplos conhecidos de escrita. Para uma discussão mais ampla sobre a preservação de dados por um longo transporte, veja [Con96].

atenção da pessoa certa. Restaurar os dados perdidos no último cenário requer instantâneos feitos mais para trás no tempo. Ao chegar até aqui, você provavelmente desejará mesclar os dados restaurados com o estado atual. Isso complica significativamente o processo de restauração.

Como o Google SRE enfrenta os desafios da integridade dos dados

Semelhante à nossa suposição de que os sistemas subjacentes do Google são propensos a falhas, assumimos que qualquer um de nossos mecanismos de proteção também está sujeito às mesmas forças e pode falhar da mesma maneira e nos momentos mais inconvenientes. Manter uma garantia de integridade de dados em larga escala, um desafio ainda mais complicado pela alta taxa de mudança dos sistemas de software envolvidos, requer uma série de práticas complementares, mas desacopladas, cada uma escolhida para oferecer um alto grau de proteção. sozinho.

As 24 combinações de modos de falha de integridade de dados

Dadas as muitas maneiras pelas quais os dados podem ser perdidos (conforme descrito anteriormente), não existe uma bala de prata que proteja contra as muitas combinações de modos de falha. Em vez disso, você precisa de defesa em profundidade. A defesa em profundidade compreende várias camadas, com cada camada sucessiva de defesa conferindo proteção contra cenários de perda de dados cada vez menos comuns. A [Figura 26-2](#) ilustra a jornada de um objeto desde a exclusão temporária até a destruição e as estratégias de recuperação de dados que devem ser empregadas ao longo dessa jornada para garantir a defesa em profundidade.

A primeira camada é a exclusão (ou “exclusão preguiçosa” no caso de ofertas de API do desenvolvedor), que provou ser uma defesa eficaz contra cenários de exclusão inadvertida de dados. A segunda linha de defesa são os backups e seus métodos de recuperação relacionados.

A terceira e última camada é a validação regular de dados, abordada em [“Terceira camada: detecção precoce” na página 356](#). Em todas essas camadas, a presença de replicação é ocasionalmente útil para recuperação de dados em cenários específicos (embora os planos de recuperação de dados não devam confiar na replicação).

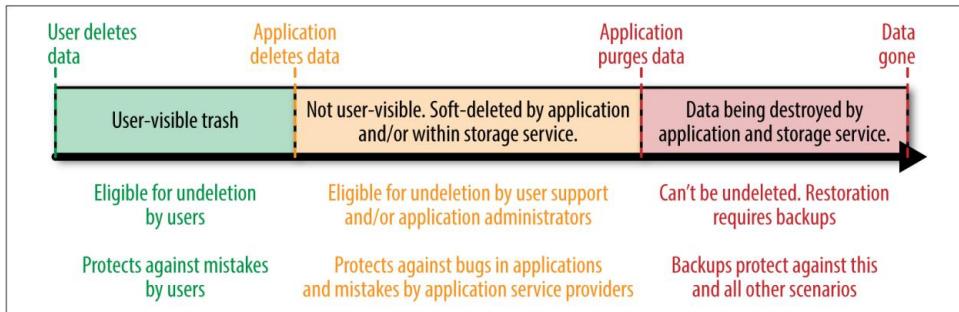


Figura 26-2. A jornada de um objeto da exclusão à destruição

Primeira camada: exclusão suave

Quando a velocidade é alta e a privacidade é importante, os bugs nos aplicativos são responsáveis pela grande maioria dos eventos de perda e corrupção de dados. Na verdade, os bugs de exclusão de dados podem se tornar tão comuns que a capacidade de recuperar dados por um tempo limitado se torna a principal linha de defesa contra a maioria das perdas de dados permanentes e inadvertidas.

Qualquer produto que mantenha a privacidade de seus usuários deve permitir que os usuários exclam subconjuntos selecionados e/ou todos os seus dados. Esses produtos incorrem em uma carga de suporte devido à exclusão acidental. Dar aos usuários a capacidade de recuperar seus dados (por exemplo, por meio de uma pasta de lixo) reduz, mas não elimina completamente, essa carga de suporte, principalmente se o seu serviço também suportar complementos de terceiros que também podem excluir dados.

A exclusão suave pode reduzir drasticamente ou até eliminar completamente essa carga de suporte. A exclusão suave significa que os dados excluídos são imediatamente marcados como tal, tornando-os inutilizáveis por todos, exceto pelos caminhos de código administrativo do aplicativo. Os caminhos de código administrativo podem incluir descoberta legal, recuperação de conta sequestrada, administração corporativa, suporte ao usuário e solução de problemas e seus recursos relacionados. Conduza a exclusão reversível quando um usuário esvaziar sua lixeira e forneça uma ferramenta de suporte ao usuário que permita que administradores autorizados recuperem a exclusão de itens excluídos acidentalmente pelos usuários. O Google implementa essa estratégia para nossos aplicativos de produtividade mais populares; caso contrário, a carga de engenharia de suporte ao usuário seria insustentável.

Você pode estender ainda mais a estratégia de exclusão suave, oferecendo aos usuários a opção de recuperar dados excluídos. Por exemplo, a lixeira do Gmail permite que os usuários acessem mensagens que foram excluídas há menos de 30 dias.

Outra fonte comum de exclusão de dados indesejados ocorre como resultado da conta sequestro. Em cenários de sequestro de conta, um sequestrador geralmente exclui os dados do usuário original antes de usar a conta para spam e outros fins ilegais. Quando você combina a exclusão acidental de usuários com o risco de exclusão de dados por sequestradores, o caso para uma exclusão programática e uma interface de recuperação dentro e/ou abaixo do seu aplicativo fica claro.

A exclusão suave implica que, uma vez que os dados sejam marcados como tal, eles sejam destruídos após um atraso razoável. A duração do atraso depende das políticas e leis aplicáveis de uma organização, dos recursos e custos de armazenamento disponíveis, do preço do produto e do posicionamento no mercado, especialmente nos casos que envolvem muitos dados de curta duração. As opções comuns de atrasos de exclusão suave são 15, 30, 45 ou 60 dias. De acordo com a experiência do Google, a maioria dos problemas de invasão de conta e integridade de dados são relatados ou detectados em 60 dias.

Portanto, o caso de exclusão reversível de dados por mais de 60 dias pode não ser forte.

O Google também descobriu que os casos de exclusão aguda de dados mais devastadores são causados por desenvolvedores de aplicativos que não estão familiarizados com o código existente, mas que trabalham no código relacionado à exclusão, especialmente pipelines de processamento em lote (por exemplo, um pipeline off-line MapReduce ou Hadoop). É vantajoso projetar suas interfaces para atrapalhar os desenvolvedores

não familiarizado com seu código de contornar recursos de exclusão suave com novo código.

Uma maneira eficaz de conseguir isso é implementar ofertas de computação em nuvem que incluem APIs de exclusão suave e recuperação de exclusão integradas, certificando-se de habilitar esse recurso.⁷ Mesmo a melhor armadura é inútil se você não a colocar.

As estratégias de exclusão suave abrangem recursos de exclusão de dados em produtos de consumo como Gmail ou Google Drive, mas e se você oferecer suporte a uma oferta de computação em nuvem?

Supondo que sua oferta de computação em nuvem já ofereça suporte a um recurso programático de exclusão e recuperação de software com padrões razoáveis, os cenários de exclusão acidental de dados restantes se originarão em erros cometidos por seus próprios desenvolvedores internos ou seus clientes desenvolvedores.

Nesses casos, pode ser útil introduzir uma camada adicional de exclusão suave, que chamaremos de “exclusão preguiçosa”. Você pode pensar na exclusão lenta como uma limpeza nos bastidores, controlada pelo sistema de armazenamento (enquanto a exclusão suave é controlada e expressa para o aplicativo ou serviço cliente). Em um cenário de exclusão lenta, os dados excluídos por um aplicativo de nuvem ficam imediatamente inacessíveis ao aplicativo, mas são preservados pelo provedor de serviços de nuvem por até algumas semanas antes da destruição. A exclusão lenta não é aconselhável em todas as estratégias de defesa em profundidade: um longo período de exclusão lenta é caro em sistemas com muitos dados de curta duração e impraticável em sistemas que devem garantir a destruição de dados excluídos dentro de um prazo razoável (ou seja, aqueles que oferecem garantias de privacidade).

Para resumir a primeira camada de defesa em profundidade:

- Uma pasta de lixo que permite aos usuários recuperar dados é a principal defesa contra erro do usuário.
- A exclusão temporária é a defesa primária contra erros do desenvolvedor e a defesa secundária contra erros do usuário. • Nas ofertas do desenvolvedor, a exclusão lenta é a principal defesa contra erros internos do desenvolvedor e a defesa secundária contra erros externos do desenvolvedor.

E o histórico de revisões? Alguns produtos oferecem a capacidade de reverter itens para estados anteriores. Quando esse recurso está disponível para os usuários, é uma forma de lixo. Quando

⁷ Ao ler este conselho, pode-se perguntar: já que você precisa oferecer uma API no topo do armazenamento de dados para implementar a exclusão suave, por que parar na exclusão suave, quando você pode oferecer muitos outros recursos que protegem contra a exclusão acidental de dados pelos usuários? Para pegar um exemplo específico da experiência do Google, considere o Blobstore: em vez de permitir que os clientes excluam dados e metadados do Blob diretamente, as APIs do Blob implementam muitos recursos de segurança, incluindo políticas de backup padrão (réplicas offline), somas de verificação de ponta a ponta, e tempos de vida padrão de tombstone (exclusão suave). Acontece que, em várias ocasiões, a exclusão suave salvou os clientes do Blobstore da perda de dados que poderia ter sido muito, muito pior. Certamente há muitos recursos de proteção contra exclusão que valem a pena ser destacados, mas para empresas com prazos de exclusão de dados necessários, a exclusão suave foi a proteção mais pertinente contra bugs e exclusão acidental no caso dos clientes do Blobstore.

disponível para desenvolvedores, pode ou não substituir a exclusão reversível, dependendo de sua implementação.

No Google, o histórico de revisões provou ser útil na recuperação de determinados cenários de corrupção de dados, mas não na recuperação da maioria dos cenários de perda de dados envolvendo exclusão acidental, programática ou não. Isso ocorre porque algumas implementações de histórico de revisão tratam a exclusão como um caso especial em que os estados anteriores devem ser removidos, em oposição à mutação de um item cujo histórico pode ser retido por um determinado período de tempo. Para fornecer proteção adequada contra exclusão indesejada, aplique os princípios de exclusão lenta e/ou suave também ao histórico de revisões.

Segunda camada: Backups e seus métodos de recuperação relacionados Os backups e a recuperação de dados são a segunda linha de defesa após a exclusão suave. O princípio mais importante nessa camada é que os backups não importam; o que importa é a recuperação. Os fatores que dão suporte à recuperação bem-sucedida devem orientar suas decisões de backup, e não o contrário.

Em outras palavras, os cenários nos quais você deseja que seus backups ajudem na recuperação devem ditar o seguinte:

- Quais métodos de backup e recuperação devem ser usados
- Com que frequência você estabelece pontos de restauração ao fazer back\back completo ou incremental ups dos seus dados
- Onde você armazena os backups
- Por quanto tempo você retém os backups

Quantos dados recentes você pode perder durante um esforço de recuperação? Quanto menos dados você puder perder, mais sério você deve ser sobre uma estratégia de backup incremental. Em um dos casos mais extremos do Google, usamos uma estratégia de backup de streaming quase em tempo real para uma versão mais antiga do Gmail.

Mesmo que o dinheiro não seja uma limitação, backups completos frequentes são caros de outras maneiras. Mais notavelmente, eles impõem uma carga de computação nos armazenamentos de dados ativos de seu serviço enquanto ele atende aos usuários, aproximando seu serviço de seus limites de escalabilidade e desempenho. Para aliviar essa carga, você pode fazer backups completos fora do horário de pico e, em seguida, uma série de backups incrementais quando seu serviço estiver mais ocupado.

Com que rapidez você precisa se recuperar? Quanto mais rápido seus usuários precisarem ser resgatados, mais locais seus backups deverão ser. Muitas vezes, o Google retém recursos caros, mas de restauração rápida

snapshots⁸ por períodos de tempo muito curtos na instância de armazenamento e armazena backups menos recentes em armazenamento distribuído de acesso aleatório no mesmo datacenter (ou próximo) por um tempo um pouco mais longo. Essa estratégia por si só não protegeria contra falhas no nível do site, portanto, esses backups geralmente são transferidos para locais nearline ou off-line por um período mais longo antes de expirarem em favor de backups mais recentes.

Até onde seus backups devem chegar? Sua estratégia de backup se torna mais cara à medida que você recua, enquanto os cenários dos quais você pode esperar para recuperar aumentam (embora esse aumento esteja sujeito a retornos decrescentes).

De acordo com a experiência do Google, bugs de exclusão ou mutação de dados de baixo grau no código do aplicativo exigem o maior alcance no tempo, já que alguns desses bugs foram detectados meses após o início da primeira perda de dados. Esses casos sugerem que você gostaria de poder voltar no tempo o mais longe possível.

Por outro lado, em um ambiente de desenvolvimento de alta velocidade, as alterações no código e no esquema podem tornar os backups mais antigos caros ou impossíveis de usar. Além disso, é um desafio recuperar diferentes subconjuntos de dados para diferentes pontos de restauração, pois isso envolveria vários backups. No entanto, esse é exatamente o tipo de esforço de recuperação exigido por cenários de corrupção ou exclusão de dados de baixo grau.

As estratégias descritas em “Terceira Camada: Detecção Antecipada” na página 356 destinam-se a acelerar a detecção de erros de exclusão ou mutação de dados de baixo grau no código do aplicativo, evitando pelo menos parcialmente a necessidade desse tipo de esforço de recuperação complexo. Ainda assim, como você confere proteção razoável antes de saber quais tipos de problemas detectar? O Google optou por traçar a linha entre 30 e 90 dias de backups para muitos serviços. Onde um serviço se enquadra nessa janela depende de sua tolerância à perda de dados e seus investimentos relativos em detecção precoce.

Para resumir nosso conselho para proteção contra as 24 combinações de modos de falha de integridade de dados: abordar uma ampla variedade de cenários a um custo razoável exige uma estratégia de backup em camadas. A primeira camada compreende muitos backups frequentes e rapidamente restaurados armazenados mais próximos dos datastores ativos, talvez usando tecnologias de armazenamento iguais ou semelhantes às fontes de dados. Isso confere proteção contra a maioria dos cenários envolvendo bugs de software e erros do desenvolvedor. Devido à despesa relativa, os backups são retidos nesta camada por horas a dias de um dígito e podem levar minutos para restaurar.

A segunda camada compreende menos backups retidos por dias de um dígito ou dois dígitos baixos em sistemas de arquivos distribuídos de acesso aleatório locais ao site. Esses backups podem

⁸ “Snapshot” aqui se refere a uma visualização estática somente leitura de uma instância de armazenamento, como snapshots de bancos de dados SQL.

Os instantâneos geralmente são implementados usando semântica de cópia na gravação para eficiência de armazenamento. Eles podem ser caros por dois motivos: primeiro, eles disputam a mesma capacidade de armazenamento que os datastores ativos e, segundo, quanto mais rápido seus dados sofrerem mutações, menos eficiência será obtida com a cópia na gravação.

levar horas para restaurar e conferir proteção adicional contra acidentes que afetam tecnologias de armazenamento específicas em sua pilha de serviço, mas não as tecnologias usadas para conter os backups. Essa camada também protege contra bugs em seu aplicativo que são detectados tarde demais para depender da primeira camada de sua estratégia de backup. Se você estiver introduzindo novas versões do seu código para produção duas vezes por semana, pode fazer sentido manter esses backups por pelo menos uma ou duas semanas antes de excluí-los.

As camadas subsequentes aproveitam o armazenamento nearline, como bibliotecas de fitas dedicadas e armazenamento externo da mídia de backup (por exemplo, fitas ou unidades de disco). Os backups nessas camadas conferem proteção contra problemas no nível do site, como falta de energia do datacenter ou corrupção do sistema de arquivos distribuído devido a um bug.

É caro mover grandes quantidades de dados de e para camadas. Por outro lado, a capacidade de armazenamento nas camadas posteriores não compete com o crescimento das instâncias de armazenamento de produção ao vivo do seu serviço. Como resultado, os backups nessas camadas tendem a ser feitos com menos frequência, mas retidos por mais tempo.

Camada abrangente: replicação Em um

mundo ideal, todas as instâncias de armazenamento, incluindo as instâncias que contêm seus backups, seriam replicadas. Durante um esforço de recuperação de dados, a última coisa que você deseja é descobrir que seus próprios backups perderam os dados necessários ou que o datacenter que contém o backup mais útil está em manutenção.

À medida que o volume de dados aumenta, a replicação de cada instância de armazenamento nem sempre é viável. Nesses casos, faz sentido escalar backups sucessivos em sites diferentes, cada um dos quais pode falhar independentemente, e gravar seus backups usando um método de redundância, como RAID, códigos de eliminação Reed-Solomon ou replicação no estilo GFS.⁹ Ao escolher um sistema de redundância, não confie em um esquema usado com pouca frequência cujos únicos “testes” de eficácia são suas próprias tentativas de recuperação de dados pouco frequentes.

Em vez disso, escolha um esquema popular que seja de uso comum e contínuo por muitos de seus usuários.

1T versus 1E: não “apenas” um backup maior Processos

e práticas aplicados a volumes de dados medidos em T (terabytes) não se adaptam bem a dados medidos em E (exabytes). Validar, copiar e realizar testes de ida e volta em alguns gigabytes de dados estruturados é um problema interessante. No entanto, supondo que você tenha conhecimento suficiente de seu esquema e modelo de transação, este exercício não apresenta nenhum desafio especial. Você normalmente só precisa

⁹ Para obter mais informações sobre replicação no estilo GFS, consulte [Ghe03]. Para obter mais informações sobre os códigos de eliminação Reed-Solomon, consulte https://en.wikipedia.org/wiki/Reed–Solomon_error_correction.

adquira os recursos da máquina para iterar sobre seus dados, execute alguma lógica de validação e delegue armazenamento suficiente para manter algumas cópias de seus dados.

Agora vamos aumentar a aposta: em vez de alguns gigabytes, vamos tentar proteger e validar 700 petabytes de dados estruturados. Assumindo o desempenho ideal do SATA 2.0 de 300 MB/s, uma única tarefa que itere sobre todos os seus dados e execute até mesmo as verificações de validação mais básicas levará 8 décadas. Fazer alguns backups completos, supondo que você tenha a mídia, levará pelo menos o mesmo tempo. O tempo de restauração, com algum pós-processamento, levará ainda mais tempo. Agora estamos olhando para quase um século inteiro para restaurar um backup que tinha até 80 anos quando você iniciou a restauração. Obviamente, tal estratégia precisa ser repensada.

A técnica mais comum e amplamente eficaz usada para fazer backup de grandes quantidades de dados é estabelecer “pontos de confiança” em seus dados – partes de seus dados armazenados que são verificados depois de se tornarem imutáveis, geralmente com o passar do tempo. Assim que soubermos que um determinado perfil de usuário ou transação foi corrigido e não estará sujeito a mais alterações, podemos verificar seu estado interno e fazer cópias adequadas para fins de recuperação. Você pode então fazer backups incrementais que incluem apenas dados que foram modificados ou adicionados desde seu último backup. Essa técnica alinha o tempo de backup com o tempo de processamento da “linha principal”, o que significa que backups incrementais frequentes podem poupar você do trabalho monolítico de verificação e cópia de 80 anos.

No entanto, lembre-se de que nos preocupamos com restaurações, não com backups. Digamos que fizemos um backup completo há três anos e desde então fazemos backups incrementais diários. Uma restauração completa de nossos dados processará em série uma cadeia de mais de 1.000 backups altamente interdependentes. Cada backup independente incorre em risco adicional de falha, sem mencionar a carga logística de agendamento e o custo de tempo de execução desses trabalhos.

Outra maneira de reduzir o tempo de espera de nossos trabalhos de cópia e verificação é distribuir a carga. Se fragmentarmos bem nossos dados, é possível executar N tarefas em paralelo, com cada tarefa responsável por copiar e verificar 1/N dos nossos dados. Fazer isso requer alguma premeditação e planejamento no design do esquema e na implantação física de nossos dados para:

- Equilibre os dados corretamente
- Garanta a independência de cada fragmento •

Evite a contenção entre as tarefas simultâneas dos irmãos

Entre distribuir a carga horizontalmente e restringir o trabalho a fatias verticais dos dados demarcados pelo tempo, podemos reduzir essas oito décadas de wall time em várias ordens de grandeza, tornando nossas restaurações relevantes.

Terceira camada: detecção precoce Os

dados “ruins” não ficam de braços cruzados, eles se propagam. As referências a dados ausentes ou corrompidos são copiadas, os links se espalham e, a cada atualização, a qualidade geral do armazenamento de dados diminui. As transações dependentes subsequentes e as possíveis alterações no formato de dados tornam a restauração de um determinado backup mais difícil à medida que o relógio avança. Quanto mais cedo você souber sobre uma perda de dados, mais fácil e completa será sua recuperação.

Desafios enfrentados pelos desenvolvedores de nuvem

Em ambientes de alta velocidade, os serviços de infraestrutura e aplicativos em nuvem enfrentam muitos desafios de integridade de dados em tempo de execução, como:

- Integridade referencial entre datastores • Mudanças de esquema • Código envelhecido • Migrações de dados sem tempo de inatividade • Pontos de integração em evolução com outros serviços

Sem um esforço consciente de engenharia para rastrear relacionamentos emergentes em seus dados, a qualidade dos dados de um serviço bem-sucedido e crescente diminui com o tempo.

Frequentemente, o desenvolvedor de nuvem iniciante que escolhe uma API de armazenamento consistente distribuído (como Megastore) delega a integridade dos dados do aplicativo ao algoritmo consistente distribuído implementado sob a API (como Paxos; consulte o [Capítulo 23](#)).

O desenvolvedor argumenta que apenas a API selecionada manterá os dados do aplicativo em boa forma. Como resultado, eles unificam todos os dados do aplicativo em uma única solução de armazenamento que garante consistência distribuída, evitando problemas de integridade referencial em troca de desempenho e/ou escala reduzidos.

Embora tais algoritmos sejam infalíveis em teoria, suas implementações são muitas vezes repletas de hacks, otimizações, bugs e suposições educadas. Por exemplo: em teoria, o Paxos ignora nós de computação com falha e pode progredir desde que um quorum de nós em funcionamento seja mantido. Na prática, no entanto, ignorar um nó com falha pode corresponder a tempos limite, tentativas e outras abordagens de tratamento de falhas sob a implementação particular do Paxos [\[Cha07\]](#). Por quanto tempo o Paxos deve tentar entrar em contato com um nó que não responde antes de expirar? Quando uma determinada máquina falha (talvez intermitentemente) de uma certa maneira, com um determinado tempo e em um datacenter específico, o resultado é um comportamento imprevisível. Quanto maior a escala de um aplicativo, mais frequentemente o aplicativo é afetado, sem o conhecimento, por tais inconsistências. Se essa lógica for verdadeira mesmo quando aplicada a implementações do Paxos (como tem sido verdade para o Google), então deve ser mais verdadeira para implementações eventualmente consistentes, como Bigtable (que também se mostrou verdadeira). Os aplicativos afetados não têm como

saiba que 100% de seus dados são bons até verificar: confie nos sistemas de armazenamento, mas verifique!

Para complicar esse problema, para recuperar cenários de corrupção ou exclusão de dados de baixo grau, devemos recuperar diferentes subconjuntos de dados para diferentes pontos de restauração usando backups diferentes, enquanto alterações no código e no esquema podem tornar os backups mais antigos ineficazes em ambientes de alta velocidade.

Validação de dados fora de banda

Para evitar que a qualidade dos dados se degrade diante dos olhos dos usuários e detectar cenários de corrupção de dados de baixo grau ou perda de dados antes que se tornem irrecuperáveis, é necessário um sistema de verificações e balanceamentos fora de banda dentro e entre os armazenamentos de dados de um aplicativo.

Na maioria das vezes, esses pipelines de validação de dados são implementados como coleções de reduções de mapa ou trabalhos do Hadoop. Freqüentemente, esses pipelines são adicionados como uma reflexão tardia a serviços que já são populares e bem-sucedidos. Às vezes, esses pipelines são tentados pela primeira vez quando os serviços atingem os limites de escalabilidade e são reconstruídos do zero.

O Google criou validadores em resposta a cada uma dessas situações.

Desviar alguns desenvolvedores para trabalhar em um pipeline de validação de dados pode diminuir a velocidade da engenharia no curto prazo. No entanto, dedicar recursos de engenharia à validação de dados dá a outros desenvolvedores a coragem de avançar mais rápido no longo prazo, porque os engenheiros sabem que os bugs de corrupção de dados são menos propensos a entrar na produção despercebidos. Semelhante aos efeitos desfrutados quando os testes de unidade são introduzidos no início do ciclo de vida do projeto, um pipeline de validação de dados resulta em uma aceleração geral dos projetos de desenvolvimento de software.

Para citar um exemplo específico: o Gmail possui vários validadores de dados, cada um dos quais detectou problemas reais de integridade de dados na produção. Os desenvolvedores do Gmail se confortam com o conhecimento de que os bugs que introduzem inconsistências nos dados de produção são detectados em 24 horas e estremecem ao pensar em executar seus validadores de dados com menos frequência do que diariamente. Esses validadores, juntamente com uma cultura de teste de unidade e regressão e outras práticas recomendadas, deram aos desenvolvedores do Gmail a coragem de introduzir alterações de código na implementação de armazenamento de produção do Gmail com mais frequência do que uma vez por semana.

A validação de dados fora de banda é difícil de implementar corretamente. Quando alterações muito rigorosas, mesmo simples e apropriadas, fazem com que a validação falhe. Como resultado, os engenheiros abandonam completamente a validação de dados. Se a validação de dados não for suficientemente rigorosa, a corrupção de dados que afeta a experiência do usuário pode passar despercebida. Para encontrar o equilíbrio certo, valide apenas invariantes que causem devastação aos usuários.

Por exemplo, o Google Drive valida periodicamente se o conteúdo do arquivo está alinhado às listagens nas pastas do Drive. Se esses dois elementos não se alinharem, alguns arquivos não terão dados—

um resultado desastroso. Os desenvolvedores de infraestrutura de unidades estavam tão investidos na integridade dos dados que também aprimoraram seus validadores para corrigir automaticamente essas inconsistências.

Essa salvaguarda transformou-se em uma emergência em potencial “arquivos de mão no convés-omigosh-estão desaparecendo!” situação de perda de dados em 2013 em um negócio como de costume, “vamos ir para casa e corrigir a causa raiz na segunda-feira”, situação. Ao transformar emergências em negócios como sempre, os validadores melhoram o moral da engenharia, a qualidade de vida e a previsibilidade.

Os validadores fora de banda podem ser caros em escala. Uma parte significativa do volume de recursos de computação do Gmail oferece suporte a uma coleção de validadores diários. Para aumentar essa despesa, esses validadores também reduzem as taxas de acerto do cache do lado do servidor, reduzindo a capacidade de resposta do lado do servidor experimentada pelos usuários. Para atenuar esse impacto na capacidade de resposta, o Gmail fornece vários botões para limitar a taxa de seus validadores e refatora periodicamente os validadores para reduzir a contenção do disco. Em um desses esforços de refatoração, reduzimos a disputa por fusos de disco em 60% sem reduzir significativamente o escopo dos invariantes cobertos. Enquanto a maioria dos validadores do Gmail são executados diariamente, a carga de trabalho do maior validador é dividida em 10 a 14 fragmentos, com um fragmento validado por dia por motivos de escala.

O Google Compute Storage é outro exemplo dos desafios que a escala implica na validação de dados. Quando seus validadores fora de banda não puderam mais terminar em um dia, os engenheiros do Compute Storage tiveram que criar uma maneira mais eficiente de verificar seus metadados do que o uso de força bruta apenas. Semelhante à sua aplicação na recuperação de dados, uma estratégia em camadas também pode ser útil na validação de dados fora de banda. À medida que um serviço cresce, sacrifique o rigor nos validadores diários. Certifique-se de que os validadores diários continuem a capturar os cenários mais desastrosos em 24 horas, mas continue com uma validação mais rigorosa com frequência reduzida para conter custos e latência.

A solução de problemas de validações com falha pode exigir um esforço significativo. As causas de uma falha intermitente na validação podem desaparecer em minutos, horas ou dias. Portanto, a capacidade de detalhar rapidamente os logs de auditoria de validação é essencial. Os serviços maduros do Google fornecem aos engenheiros de plantão documentação e ferramentas abrangentes para solucionar problemas. Por exemplo, os engenheiros de plantão do Gmail recebem:

- Um conjunto de entradas de manual descrevendo como responder a um alerta de falha de validação • Uma ferramenta de investigação semelhante ao BigQuery
- Um painel de validação de dados

A validação eficaz de dados fora de banda exige o seguinte:

- Gerenciamento de trabalhos de validação • Monitoramento, alertas e painéis • Recursos de limitação de taxa • Ferramentas de solução de problemas

- Manuais de produção • APIs

de validação de dados que facilitam a adição e refatoração de validadores

A maioria das pequenas equipes de engenharia que operam em alta velocidade não pode se dar ao luxo de projetar, construir e manter todos esses sistemas. Se eles são pressionados a fazê-lo, o resultado geralmente é frágil, limitado e desperdiçado, que rapidamente cai em desuso.

Portanto, estruture suas equipes de engenharia de forma que uma equipe de infraestrutura central forneça uma estrutura de validação de dados para várias equipes de engenharia de produto. A equipe de infraestrutura central mantém a estrutura de validação de dados fora de banda, enquanto as equipes de engenharia de produto mantêm a lógica de negócios personalizada no coração do validador para acompanhar a evolução de seus produtos.

Sabendo que a recuperação de dados funcionará Quando

uma lâmpada quebra? Ao apertar o interruptor não acende a luz? Nem sempre - muitas vezes a lâmpada já falhou e você simplesmente percebe a falha ao apertar o interruptor sem resposta. A essa altura, o quarto está escuro e você deu uma

dedo do pé.

Da mesma forma, suas dependências de recuperação (ou seja, principalmente, mas não apenas, seu backup) podem estar em um estado latente quebrado, do qual você não está ciente até tentar recuperar os dados.

Se você descobrir que seu processo de restauração foi interrompido antes de precisar confiar nele, poderá resolver a vulnerabilidade antes de ser vítima dela: você pode fazer outro backup, provisionar recursos adicionais e alterar seu SLO. Mas para realizar essas ações de forma proativa, primeiro você precisa saber que elas são necessárias. Para detectar essas vulnerabilidades:

- Teste continuamente o processo de recuperação como parte de suas operações normais •

Configure alertas que são acionados quando um processo de recuperação não fornece uma indicação de pulsação de seu sucesso

O que pode dar errado no seu processo de recuperação? Qualquer coisa e tudo - e é por isso que o único teste que deve permitir que você durma à noite é um teste completo de ponta a ponta. Que a prova esteja no pudim. Mesmo que você tenha executado uma recuperação bem-sucedida recentemente, partes do processo de recuperação ainda podem ser interrompidas. Se você tirar apenas uma lição deste capítulo, lembre-se de que você só sabe que pode recuperar seu estado recente se realmente fizer isso.

assim.

Se os testes de recuperação forem um evento manual e encenado, o teste se tornará um trabalho árduo indesejável que não é realizado com profundidade ou frequência suficiente para merecer sua confiança. Portanto, automatize esses testes sempre que possível e execute-os continuamente.

Os aspectos do seu plano de recuperação que você deve confirmar são inúmeros:

- Seus backups são válidos e completos ou estão vazios? • Você tem recursos de máquina suficientes para executar todas as tarefas de configuração, restauração e pós-processamento que compõem sua recuperação? • O processo de recuperação é concluído em um tempo razoável? • Você consegue monitorar o estado do seu processo de recuperação à medida que ele avança? • Você está livre de dependências críticas de recursos fora de seu controle, como

como acesso a um cofre de armazenamento de mídia externo que não está disponível 24 horas por dia, 7 dias por semana?

Nossos testes descobriram as falhas acima mencionadas, bem como falhas de muitos outros componentes de uma recuperação de dados bem-sucedida. Se não tivéssemos descoberto essas falhas em testes regulares, ou seja, se encontrássemos as falhas apenas quando precisávamos recuperar os dados do usuário em emergências reais, é bem possível que alguns dos produtos de maior sucesso do Google hoje não tivessem resistido ao teste de tempo.

As falhas são inevitáveis. Se você esperar para descobri-los quando estiver sob pressão, enfrentando uma perda real de dados, estará brincando com fogo. Se o teste forçar as falhas a acontecerem antes que ocorra uma catástrofe real, você pode corrigir os problemas antes que qualquer dano se concretize.

Estudos de caso

A vida imita a arte (ou, neste caso, a ciência) e, como previmos, a vida real nos deu oportunidades infelizes e inevitáveis de testar nossos sistemas e processos de recuperação de dados, sob pressão do mundo real. Duas das mais notáveis e interessantes dessas oportunidades são discutidas aqui.

Gmail—fevereiro de 2011: Restauração do GTape

O primeiro estudo de caso de recuperação que examinaremos foi único em alguns aspectos: o número de falhas que coincidiram para causar a perda de dados e o fato de que foi o maior uso de nossa última linha de defesa, o sistema de backup offline GTape.

Domingo, 27 de fevereiro de 2011, tarde da noite

O pager do sistema de backup do Gmail é acionado, exibindo um número de telefone para participar de uma teleconferência. O evento que há muito temíamos – na verdade, o motivo da existência do sistema de backup – aconteceu: o Gmail perdeu uma quantidade significativa de dados do usuário. Apesar das muitas proteções do sistema, verificações internas e redundâncias, os dados desapareceram do Gmail.

Este foi o primeiro uso em larga escala do GTape, um sistema de backup global para o Gmail, para restaurar dados de clientes ao vivo. Felizmente, não foi a primeira restauração desse tipo, pois situações semelhantes já haviam sido simuladas muitas vezes. Assim, conseguimos:

- Entregar uma estimativa de quanto tempo levaria para restaurar a maioria dos afetados contas de usuário
- Restaure todas as contas dentro de algumas horas de nossa estimativa inicial
- Recupere mais de 99% dos dados antes do tempo estimado de conclusão

A capacidade de formular tal estimativa foi sorte? Não — nosso sucesso foi fruto de planejamento, adesão às melhores práticas, trabalho árduo e cooperação, e ficamos felizes em ver nosso investimento em cada um desses elementos dar tão certo quanto deu. O Google conseguiu restaurar os dados perdidos em tempo hábil, executando um plano elaborado de acordo com as práticas recomendadas de Defesa em Profundidade e Preparação para Emergências.

Quando o Google revelou publicamente que recuperamos esses dados de nosso sistema de backup em fita não divulgado anteriormente [Slo11], a reação do público foi uma mistura de surpresa e diversão. Fita? O Google não tem muitos discos e uma rede rápida para replicar dados tão importantes? É claro que o Google tem esses recursos, mas o princípio de Defesa em Profundidade determina o fornecimento de várias camadas de proteção para se proteger contra a quebra ou comprometimento de qualquer mecanismo de proteção único. O backup de sistemas on-line, como o Gmail, oferece defesa profunda em duas camadas:

- Uma falha dos subsistemas internos de redundância e backup do Gmail
- Uma falha ampla ou vulnerabilidade de dia zero em um driver de dispositivo ou sistema de arquivos que afeta a mídia de armazenamento subjacente (disco)

Essa falha específica resultou do primeiro cenário - embora o Gmail tivesse meios internos de recuperar dados perdidos, essa perda foi além do que os meios internos poderiam recuperar.

Um dos aspectos mais celebrados internamente da recuperação de dados do Gmail foi o grau de cooperação e coordenação suave que compreendia a recuperação. Muitas equipes, algumas completamente não relacionadas ao Gmail ou à recuperação de dados, ajudaram. A recuperação não poderia ter sido tão tranquila sem um plano central para coreografar um esforço hercúleo tão amplamente distribuído; este plano foi o produto de ensaios regulares e ensaios. A devoção do Google à preparação para emergências nos leva a ver essas falhas como inevitáveis. Aceitando essa inevitabilidade, não esperamos ou apostamos em evitar tais desastres, mas antecipamos que eles ocorrerão. Assim, precisamos de um plano para lidar não apenas com as falhas previsíveis, mas também com alguma quantidade de quebra aleatória indiferenciada.

Em suma, sempre soubemos que a adesão às melhores práticas é importante e foi bom ver essa máxima comprovada.

Google Music—março de 2012: detecção de exclusão descontrolada A

segunda falha que examinaremos envolve desafios logísticos exclusivos da escala do armazenamento de dados que está sendo recuperado: onde você armazena mais de 5.000 fitas e como você armazena de forma eficiente (ou mesmo viável) ler tantos dados da mídia offline em um período de tempo razoável?

terça-feira, 6 de março de 2012, meio-dia

Descobrindo o problema

Um usuário do Google Music relata que faixas anteriormente não problemáticas estão sendo ignoradas. A equipe responsável pela interface com os usuários do Google Music notifica os engenheiros do Google Music. O problema é investigado como um possível problema de streaming de mídia.

Em 7 de março, o engenheiro investigador descobre que nos metadados da faixa não reproduzível falta uma referência que deve apontar para os dados de áudio reais. Ele está surpreso. A solução óbvia é localizar os dados de áudio e restabelecer a referência aos dados.

No entanto, a engenharia do Google se orgulha de uma cultura de correção de problemas na raiz, então o engenheiro se aprofunda.

Quando ele encontra a causa do lapso de integridade dos dados, ele quase tem um ataque cardíaco: a referência de áudio foi removida por um pipeline de exclusão de dados que protege a privacidade. Esta parte do Google Music foi projetada para excluir um grande número de faixas de áudio gravadas. Tempo.

Avaliando os danos A

política de privacidade do Google protege os dados pessoais de um usuário. Conforme aplicado especificamente ao Google Music, nossa política de privacidade significa que os arquivos de música e os metadados relevantes são removidos dentro de um prazo razoável após a exclusão dos usuários. À medida que a popularidade do Google Music aumentava, a quantidade de dados crescia rapidamente, de modo que a implementação de exclusão original precisava ser redesenhada em 2012 para ser mais eficiente. Em 6 de fevereiro, o pipeline de exclusão de dados atualizado teve sua primeira execução, para remover metadados relevantes. Nada parecia errado na época, então um segundo estágio do pipeline também foi autorizado a remover os dados de áudio associados.

O pior pesadelo do engenheiro poderia ser verdade? Ele imediatamente soou o alarme, elevando a prioridade do caso de suporte para a classificação mais urgente do Google e relatando o problema ao gerenciamento de engenharia e à Engenharia de confiabilidade do site. Uma pequena equipe de desenvolvedores do Google Music e SREs se reuniu para resolver o problema, e o pipeline ofensivo foi temporariamente desativado para conter a maré de baixas de usuários externos.

Em seguida, verificar manualmente os metadados de milhões a bilhões de arquivos organizados em vários datacenters seria impensável. Então, a equipe preparou um apressado

MapReduce para avaliar os danos e esperou desesperadamente pela conclusão do trabalho.

Eles congelaram quando seus resultados chegaram em 8 de março: o pipeline de exclusão de dados refatorado havia removido aproximadamente 600.000 referências de áudio que não deveriam ter sido removidas, afetando arquivos de áudio para 21.000 usuários. Como o pipeline de diagnóstico apressado fez algumas simplificações, a verdadeira extensão do dano poderia ser pior.

Fazia mais de um mês desde que o pipeline de exclusão de dados com bugs foi executado pela primeira vez, e essa primeira execução removeu centenas de milhares de faixas de áudio que não deveriam ser removidas. Havia alguma esperança de recuperar os dados? Se as faixas não fossem recuperadas, ou não fossem recuperadas com rapidez suficiente, o Google teria que enfrentar a música de seus usuários. Como não notamos essa falha?

Resolvendo o problema

Esforços paralelos de identificação e recuperação de bugs. O primeiro passo para resolver o problema foi identificar o bug real e determinar como e por que o bug aconteceu. Enquanto a causa raiz não for identificada e corrigida, qualquer esforço de recuperação será em vão. Estaríamos sob pressão para reativar o pipeline para respeitar as solicitações de usuários que excluíram faixas de áudio, mas isso prejudicaria usuários inocentes que continuariam a perder músicas compradas em lojas, ou pior, seus próprios arquivos de áudio cuidadosamente gravados. A única maneira de escapar do Catch-22¹⁰ era corrigir o problema em sua raiz e corrigi-lo rapidamente.

No entanto, não havia tempo a perder antes de montar o esforço de recuperação. As próprias faixas de áudio foram copiadas em fita, mas, diferentemente do nosso estudo de caso do Gmail, as fitas de backup criptografadas do Google Music foram transportadas para locais de armazenamento externos, porque essa opção oferecia mais espaço para backups volumosos de dados de áudio dos usuários. Para restaurar a experiência dos usuários afetados rapidamente, a equipe decidiu solucionar a causa raiz enquanto recuperava as fitas de backup externas (uma opção de restauração bastante demorada) em paralelo.

Os engenheiros se dividiram em dois grupos. Os SREs mais experientes trabalharam no esforço de recuperação, enquanto os desenvolvedores analisaram o código de exclusão de dados e tentaram corrigir o erro de perda de dados em sua raiz. Devido ao conhecimento incompleto da raiz do problema, a recuperação teria que ser encenada em várias passagens. O primeiro lote de quase meio milhão de faixas de áudio foi identificado, e a equipe que mantinha o sistema de backup em fita foi notificada sobre o esforço de recuperação de emergência às 16h34, horário do Pacífico, em 8 de março.

A equipe de recuperação tinha um fator trabalhando a seu favor: esse esforço de recuperação ocorreu apenas algumas semanas após o exercício anual de teste de recuperação de desastres da empresa (consulte [Kri12]). A equipe de backup em fita já conhecia os recursos e limitações de seus subsistemas que haviam sido objeto de testes do DiRT e começou a tirar o pó de uma nova ferramenta

10 Veja [http://en.wikipedia.org/wiki/Catch-22_\(logic\)](http://en.wikipedia.org/wiki/Catch-22_(logic)).

eles testaram durante um exercício DiRT. Usando a nova ferramenta, a equipe de recuperação combinada iniciou o esforço meticoloso de mapear centenas de milhares de arquivos de áudio para backups registrados no sistema de backup em fita e, em seguida, mapear os arquivos de backups para fitas reais.

Dessa forma, a equipe determinou que o esforço inicial de recuperação envolveria o recall de mais de 5.000 fitas de backup por caminhão. Depois, os técnicos do datacenter teriam que liberar espaço para as fitas nas bibliotecas de fitas. Seguiria-se um processo longo e complexo de registro das fitas e extração dos dados das fitas, envolvendo soluções alternativas e mitigações no caso de fitas ruins, unidades ruins e interações inesperadas do sistema.

Infelizmente, apenas 436.223 das aproximadamente 600.000 faixas de áudio perdidas foram encontradas em backups de fita, o que significa que cerca de 161.000 outras faixas de áudio foram consumidas antes que pudessem ser copiadas. A equipe de recuperação decidiu descobrir como recuperar as 161.000 faixas perdidas depois de iniciar o processo de recuperação das faixas com backups em fita.

Enquanto isso, a equipe de causa raiz perseguiu e abandonou uma pista falsa: eles inicialmente pensaram que um serviço de armazenamento do qual o Google Music dependia forneceu dados com bugs que enganaram os pipelines de exclusão de dados para remover os dados de áudio errados. Após uma investigação mais detalhada, essa teoria se provou falsa. A equipe de causa raiz coçou a cabeça e continuou sua busca pelo bug indescritível.

Primeira onda de recuperação. Depois que a equipe de recuperação identificou as fitas de backup, a primeira onda de recuperação começou em 8 de março. Solicitar 1,5 petabytes de dados distribuídos entre milhares de fitas do armazenamento externo era uma questão, mas extrair os dados das fitas era outra. A pilha de software de backup em fita personalizada não foi projetada para lidar com uma única operação de restauração de tamanho tão grande, portanto, a recuperação inicial foi dividida em 5.475 tarefas de restauração. Levaria um operador humano digitando em um comando de restauração um minuto mais de três dias para solicitar que muitas restaurações, e qualquer operador humano sem dúvida cometaria muitos erros. Apenas solicitar a restauração do sistema de backup em fita exigia que o SRE desenvolvesse uma solução programática.¹¹

À meia-noite de 9 de março, o Music SRE terminou de solicitar todas as 5.475 restaurações. O sistema de backup em fita começou a fazer sua mágica. Quatro horas depois, ele cuspiu uma lista de 5.337 fitas de backup a serem recuperadas de locais externos. Em mais oito horas, as fitas chegaram a um datacenter em uma série de entregas de caminhões.

¹¹ Na prática, chegar a uma solução programática não foi um obstáculo porque a maioria dos SREs são engenheiros de software experientes, como foi o caso aqui. A expectativa de tal experiência torna os SREs notoriamente difíceis de encontrar e contratar, e a partir deste estudo de caso e de outros pontos de dados, você pode começar a entender por que o SRE contrata engenheiros de software praticantes; veja [Jon15].

Enquanto os caminhões estavam a caminho, os técnicos do datacenter retiraram várias bibliotecas de fitas para manutenção e removeram milhares de fitas para abrir caminho para a operação de recuperação de dados em massa. Em seguida, os técnicos começaram a carregarmeticulosamente as fitas à mão, à medida que milhares de fitas chegavam nas primeiras horas da manhã. Nos exercícios anteriores do DiRT, esse processo manual provou ser centenas de vezes mais rápido para restaurações em massa do que os métodos baseados em robôs fornecidos pelos fornecedores de bibliotecas de fitas. Em três horas, as bibliotecas estavam fazendo backup das fitas e executando milhares de trabalhos de restauração no armazenamento de computação distribuído.

Apesar da experiência da equipe em DiRT, a recuperação massiva de 1,5 petabyte levou mais tempo do que os dois dias estimados. Na manhã de 10 de março, apenas 74% dos 436.223 arquivos de áudio haviam sido transferidos com sucesso de 3.475 fitas de backup recuperadas para o armazenamento do sistema de arquivos distribuído em um cluster de computação próximo. As outras 1.862 fitas de backup foram omitidas do processo de recuperação de fitas por um fornecedor. Além disso, o processo de recuperação foi interrompido por 17 fitas ruins. Antecipando uma falha devido a fitas ruins, uma codificação redundante foi usada para gravar os arquivos de backup. Entregas de caminhões adicionais foram iniciadas para recolher as fitas de redundância, juntamente com as outras 1.862 fitas que haviam sido omitidas pelo primeiro recall externo.

Na manhã de 11 de março, mais de 99,95% da operação de restauração havia sido concluída e o recall de fitas de redundância adicionais para os arquivos restantes estava em andamento.

Embora os dados estivessem seguros em sistemas de arquivos distribuídos, foram necessárias etapas adicionais de recuperação de dados para torná-los acessíveis aos usuários. A equipe do Google Music começou a exercitar essas etapas finais do processo de recuperação de dados em paralelo em uma pequena amostra de arquivos de áudio recuperados para garantir que o processo ainda funcionasse conforme o esperado.

Naquele momento, os pagers de produção do Google Music soaram devido a uma falha de produção não relacionada, mas crítica, que afetou o usuário, uma falha que envolveu totalmente a equipe do Google Music por dois dias. O esforço de recuperação de dados foi retomado em 13 de março, quando todas as 436.223 faixas de áudio foram novamente disponibilizadas aos usuários. Em pouco menos de 7 dias, 1,5 petabytes de dados de áudio foram restabelecidos aos usuários com a ajuda de backups de fita externos; 5 dos 7 dias incluiram o esforço real de recuperação de dados.

Segunda onda de recuperação. Com a primeira onda do processo de recuperação atrás deles, a equipe mudou seu foco para os outros 161.000 arquivos de áudio ausentes que haviam sido excluídos pelo bug antes do backup. A maioria desses arquivos eram faixas promocionais e compradas em lojas, e as cópias originais da loja não foram afetadas pelo bug.

Essas faixas foram rapidamente restabelecidas para que os usuários afetados pudesse desfrutar de suas músicas novamente.

No entanto, uma pequena parte dos 161.000 arquivos de áudio foi carregada pelos próprios usuários. A equipe do Google Music solicitou que seus servidores solicitasse que os clientes do Google Music dos usuários afetados reenviassem arquivos datados de 14 de março em diante.

Esse processo durou mais de uma semana. Assim concluiu o esforço de recuperação completo para o incidente.

Resolvendo a causa raiz

Eventualmente, a equipe do Google Music identificou a falha em seu pipeline de exclusão de dados refatorados. Para entender essa falha, primeiro você precisa de contexto sobre como os sistemas de processamento de dados offline evoluem em grande escala.

Para um serviço grande e complexo que compreende vários subsistemas e serviços de armazenamento, mesmo uma tarefa tão simples quanto a remoção de dados excluídos precisa ser executada em etapas, cada uma envolvendo diferentes datastores.

Para que o processamento de dados seja concluído rapidamente, o processamento é paralelizado para ser executado em dezenas de milhares de máquinas que exercem uma grande carga em vários subsistemas. Essa distribuição pode retardar o serviço para os usuários ou fazer com que o serviço falhe sob a carga pesada.

Para evitar esses cenários indesejáveis, os engenheiros de computação em nuvem geralmente fazem uma cópia de curta duração dos dados no armazenamento secundário, onde o processamento de dados é realizado. A menos que a idade relativa das cópias secundárias de dados seja cuidadosamente coordenada, esta prática introduz condições de corrida.

Por exemplo, dois estágios de um pipeline podem ser projetados para serem executados em estrita sucessão, com três horas de intervalo, de modo que o segundo estágio possa fazer uma suposição simplificadora sobre a exatidão de suas entradas. Sem essa suposição simplificadora, a lógica do segundo estágio pode ser difícil de paralelizar. Mas os estágios podem levar mais tempo para serem concluídos à medida que o volume de dados aumenta. Eventualmente, as suposições do projeto original podem não ser mais válidas para certos dados necessários para o segundo estágio.

A princípio, essa condição de corrida pode ocorrer para uma pequena fração dos dados. Mas à medida que o volume de dados aumenta, uma fração cada vez maior dos dados corre o risco de desencadear uma condição de corrida. Esse cenário é probabilístico — o pipeline funciona corretamente para a grande maioria dos dados e na maior parte do tempo. Quando essas condições de corrida ocorrem em um pipeline de exclusão de dados, os dados errados podem ser excluídos de forma não determinística.

O pipeline de exclusão de dados do Google Music foi projetado com coordenação e grandes margens de erro. Mas quando os estágios upstream do pipeline começaram a exigir mais tempo à medida que o serviço crescia, otimizações de desempenho foram implementadas para que o Google Music pudesse continuar atendendo aos requisitos de privacidade. Como resultado, a probabilidade de uma condição de corrida de exclusão de dados inadvertida nesse pipeline começou a aumentar.

Quando o pipeline foi refatorado, essa probabilidade novamente aumentou significativamente, até um ponto em que as condições de corrida ocorreram com mais regularidade.

Após o esforço de recuperação, o Google Music redesenhou seu pipeline de exclusão de dados para eliminar esse tipo de condição de corrida. Além disso, melhoramos o monitoramento de produção

sistemas de toring e alerta para detectar bugs de exclusão descontrolados em grande escala semelhantes com o objetivo de detectar e corrigir esses problemas antes que os usuários percebam qualquer problema.¹²

Princípios Gerais do SRE Aplicados à Integridade dos Dados

Os princípios gerais do SRE podem ser aplicados às especificidades da integridade dos dados e da computação em nuvem, conforme descrito nesta seção.

Beginner's Mind Serviços

complexos e de grande escala têm bugs inerentes que não podem ser totalmente corrigidos. Nunca pense que você entende o suficiente de um sistema complexo para dizer que ele não falhará de uma certa maneira. Confie, mas verifique e aplique a defesa em profundidade. (Nota: “Mente de principiante” não sugere colocar um novo contratado responsável por esse pipeline de exclusão de dados!)

Confie, mas verifique

Qualquer API da qual você dependa não funcionará perfeitamente o tempo todo. É certo que, independentemente da qualidade da engenharia ou do rigor dos testes, a API terá defeitos.

Verifique a exatidão dos elementos mais críticos de seus dados usando validadores de dados fora de banda, mesmo se a semântica da API sugerir que você não precisa fazer isso. Algoritmos perfeitos podem não ter implementações perfeitas.

A esperança não é uma estratégia

Os componentes do sistema que não são continuamente exercitados falham quando você mais precisa deles. Prove que a recuperação de dados funciona com exercícios regulares ou a recuperação de dados não funcionará. Os humanos não têm disciplina para exercitar continuamente os componentes do sistema, então a automação é sua amiga. No entanto, quando você emprega esses esforços de automação com engenheiros que têm prioridades concorrentes, pode acabar com brechas temporárias.

Defesa em profundidade

Mesmo o sistema mais à prova de balas é suscetível a bugs e erros do operador. Para que os problemas de integridade de dados possam ser corrigidos, os serviços devem detectar esses problemas rapidamente. Toda estratégia eventualmente falha em ambientes em mudança. As melhores estratégias de integridade de dados

¹² Em nossa experiência, os engenheiros de computação em nuvem geralmente relutam em configurar alertas de produção sobre as taxas de exclusão de dados devido à variação natural das taxas de exclusão de dados por usuário com o tempo. No entanto, como a intenção de tal alerta é detectar anomalias de taxa de exclusão globais em vez de locais, seria mais útil alertar quando a taxa de exclusão de dados global, agregada em todos os usuários, cruzar um limite extremo (como 10x o 95º observado percentil), em oposição a alertas de taxa de exclusão por usuário menos úteis.

são multicamadas — várias estratégias que se alternam e abordam uma ampla faixa de cenários juntos a um custo razoável.

Revisitar e reexaminar

O fato de que seus dados “estavam seguros ontem” não vai ajudá-lo amanhã, nem mesmo hoje. Os sistemas e a infraestrutura mudam, e você precisa provar que suas suposições e processos permanecem relevantes diante do progresso. Considere o seguinte.

O serviço de Shakespeare recebeu um pouco de imprensa positiva, e sua base de usuários está aumentando constantemente. Nenhuma atenção real foi dada à integridade dos dados enquanto o serviço estava sendo construído. É claro que não queremos servir bits ruins, mas se o índice Bigtable for perdido, podemos recriá-lo facilmente a partir dos textos originais de Shakespeare e de um MapReduce.

Fazer isso levaria muito pouco tempo, então nunca fizemos backups do índice.

Agora, um novo recurso permite que os usuários façam anotações de texto. De repente, nosso conjunto de dados não pode mais ser recriado facilmente, enquanto os dados do usuário são cada vez mais valiosos para nossos usuários. Portanto, precisamos rever nossas opções de replicação — não estamos apenas replicando para latência e largura de banda, mas também para integridade de dados. Portanto, precisamos criar e testar um procedimento de backup e restauração. Esse procedimento também é testado periodicamente por um exercício DiRT para garantir que possamos restaurar as anotações dos usuários dos backups dentro do tempo definido pelo SLO.

Conclusão

A disponibilidade de dados deve ser a principal preocupação de qualquer sistema centrado em dados. Em vez de se concentrar nos meios para o fim, o Google SRE considera útil emprestar uma página do desenvolvimento orientado a testes, provando que nossos sistemas podem manter a disponibilidade de dados com um tempo de inatividade máximo previsto. Os meios e mecanismos que usamos para atingir esse objetivo final são males necessários. Mantendo os olhos no objetivo, evitamos cair na armadilha em que “A operação foi um sucesso, mas o sistema morreu”.

Reconhecer que nem tudo pode dar errado, mas que tudo dará errado é um passo significativo para a preparação para qualquer emergência real. Uma matriz de todas as combinações possíveis de desastres com planos para lidar com cada um desses desastres permite que você durma profundamente por pelo menos uma noite; manter seus planos de recuperação atualizados e exercitados permite que você durma as outras 364 noites do ano.

À medida que você se recupera melhor de qualquer quebra em um tempo razoável N , encontre maneiras de reduzir esse tempo por meio de uma detecção de perda mais rápida e detalhada, com o objetivo de se aproximar de $N = 0$. Você pode então alternar do planejamento de recuperação para o planejamento de prevenção, com o objetivo de alcançar o santo graal de todos os dados, o tempo todo. Alcançar este objetivo, poderá dormir na praia naquelas merecidas férias.

CAPÍTULO 27

Lançamentos de produtos confiáveis em escala

**Escrito por Rhandeep Singh e Sebastian Kirsch com Vivek Rau
Editado por Betsy Beyer**

Empresas de Internet como o Google são capazes de lançar novos produtos e recursos em iterações muito mais rápidas do que as empresas tradicionais. A função da Site Reliability nesse processo é permitir um ritmo rápido de mudança sem comprometer a estabilidade do site. Criamos uma equipe dedicada de “Engenheiros de Coordenação de Lançamento” para consultar as equipes de engenharia sobre os aspectos técnicos de um lançamento bem sucedido.

A equipe também organizou uma “lista de verificação de lançamento” de perguntas comuns para fazer sobre um lançamento e receitas para resolver problemas comuns. A lista de verificação provou ser uma ferramenta útil para garantir lançamentos confiáveis e reproduutíveis.

Considere um serviço comum do Google – por exemplo, Keyhole, que fornece imagens de satélite para o Google Maps e o Google Earth. Em um dia normal, o Keyhole oferece vários milhares de imagens de satélite por segundo. Mas na véspera de Natal de 2011, recebeu 25 vezes seu tráfego normal de pico – mais de um milhão de solicitações por segundo.

O que causou esse aumento maciço no tráfego?

Papai Noel estava chegando.

Alguns anos atrás, o Google colaborou com o NORAD (Comando de Defesa Aeroespacial da América do Norte) para hospedar um site com tema de Natal que acompanhava o progresso do Papai Noel em todo o mundo, permitindo que os usuários o assistissem entregar presentes em tempo real. Parte da experiência foi um “sobrevoô virtual”, que usou imagens de satélite para rastrear o progresso do Papai Noel em um mundo simulado.

Embora um projeto como o NORAD Tracks Santa possa parecer caprichoso, ele tinha todas as características que definem um lançamento difícil e arriscado: um prazo difícil (o Google não poderia pedir

Papai Noel virá uma semana depois se o site não estiver pronto), muita publicidade, uma audiência de milhões e um aumento de tráfego muito acentuado (todo mundo estaria assistindo o site na véspera de Natal). Nunca subestime o poder de milhões de crianças ansiosas por presentes – esse projeto tinha uma possibilidade muito real de colocar os servidores do Google de joelhos.

A equipe de engenharia de confiabilidade do site do Google trabalhou arduamente para preparar nossa infraestrutura para este lançamento, garantindo que o Papai Noel pudesse entregar todos os seus presentes no prazo sob o olhar atento de um público expectante. A última coisa que queríamos era fazer as crianças chorarem porque não podiam ver o Papai Noel entregar os presentes. Na verdade, apelidamos os vários interruptores de interrupção incorporados à experiência para proteger nossos serviços de "Faça interruptores para crianças chorarem". Antecipar as várias maneiras pelas quais esse lançamento poderia dar errado e a coordenação entre os diferentes grupos de engenharia envolvidos no lançamento coube a uma equipe especial da Engenharia de Confiabilidade do Site: os Engenheiros de Coordenação de Lançamento (LCE).

O lançamento de um novo produto ou recurso é o momento da verdade para todas as empresas – o ponto em que meses ou anos de esforço são apresentados ao mundo. As empresas tradicionais lançam novos produtos a uma taxa bastante baixa. O ciclo de lançamento nas empresas de Internet é marcadamente diferente. Lançamentos e iterações rápidas são muito mais fáceis porque novos recursos podem ser implementados no lado do servidor, em vez de exigir a implementação de software em estações de trabalho de clientes individuais.

O Google define um lançamento como qualquer novo código que introduz uma alteração visível externamente em um aplicativo. Dependendo das características de um lançamento — a combinação de atributos, o tempo, o número de etapas envolvidas e a complexidade — o processo de lançamento pode variar muito. De acordo com essa definição, o Google às vezes realiza até 70 lançamentos por semana.

Essa rápida taxa de mudança fornece a lógica e a oportunidade para criar um processo de lançamento simplificado. Uma empresa que lança um produto apenas a cada três anos não precisa de um processo de lançamento detalhado. Quando ocorrer um novo lançamento, a maioria dos componentes do processo de lançamento desenvolvido anteriormente estará desatualizado. Nem as empresas tradicionais têm a oportunidade de desenhar um processo de lançamento detalhado, pois não acumulam experiência suficiente na realização de lançamentos para gerar um processo robusto e maduro.

Engenharia de Coordenação de Lançamento

Bons engenheiros de software têm muita experiência em codificação e design e entendem muito bem a tecnologia de seus próprios produtos. No entanto, os mesmos engenheiros podem não estar familiarizados com os desafios e armadilhas de lançar um produto para milhões de usuários, minimizando simultaneamente as interrupções e maximizando o desempenho.

O Google abordou os desafios inerentes aos lançamentos criando uma equipe de consultoria dedicada dentro do SRE encarregada do lado técnico do lançamento de um novo produto ou recurso. Formada por engenheiros de software e engenheiros de sistemas, alguns com experiência em outras equipes de SRE, essa equipe é especializada em orientar os desenvolvedores na criação de produtos confiáveis e rápidos que atendam aos padrões do Google de robustez, escalabilidade e confiabilidade. Essa equipe de consultoria, Engenharia de Coordenação de Lançamento (LCE), facilita um processo de lançamento tranquilo de algumas maneiras:

- Auditoria de produtos e serviços para conformidade com os padrões de confiabilidade e práticas recomendadas do Google e fornecimento de ações específicas para melhorar a confiabilidade
- Atuando como uma ligação entre as várias equipes envolvidas em um lançamento • Conduzindo os aspectos técnicos de um lançamento, certificando-se de que as tarefas sejam mantidas impulso
- Atuar como gatekeepers e aprovar lançamentos considerados “seguros” • Educar os desenvolvedores sobre as práticas recomendadas e como integrar-se aos serviços do Google, equipando-os com documentação interna e recursos de treinamento para acelerar o aprendizado

Os membros da equipe de LCE auditam os serviços em vários momentos durante o ciclo de vida do serviço. A maioria das auditorias é realizada antes do lançamento de um novo produto ou serviço. Se uma equipe de desenvolvimento de produto realizar um lançamento sem suporte SRE, a LCE fornecerá o conhecimento de domínio apropriado para garantir um lançamento tranquilo. Mas mesmo os produtos que já têm um forte suporte SRE geralmente se envolvem com a equipe de LCE durante lançamentos críticos. Os desafios que as equipes enfrentam ao lançar um novo produto são substancialmente diferentes da operação diária de um serviço confiável (uma tarefa na qual as equipes de SRE já se destacam), e a equipe de LCE pode aproveitar a experiência de centenas de lançamentos. A equipe de LCE também facilita as auditorias de serviço quando novos serviços se envolvem com o SRE.

A função do engenheiro de coordenação de lançamento Nossa

equipe de engenharia de coordenação de lançamento é composta por engenheiros de coordenação de lançamento (LCEs), que são contratados diretamente para essa função ou são SREs com experiência prática na execução de serviços do Google. Os LCEs são submetidos aos mesmos requisitos técnicos que qualquer outro SRE, e também devem ter fortes habilidades de comunicação e liderança – um LCE reúne partes diferentes para trabalhar em direção a um objetivo comum, media conflitos ocasionais e orienta, treina , e educa colegas engenheiros.

Uma equipe dedicada a coordenar lançamentos oferece as seguintes vantagens:

Ampla experiência Como

uma verdadeira equipe multiproduto, os membros são ativos em quase todas as áreas de produtos do Google. Amplo conhecimento de vários produtos e relacionamentos com muitas equipes em toda a empresa tornam os LCEs excelentes veículos para a transferência de conhecimento.

Perspectiva multifuncional

Os LCEs têm uma visão holística do lançamento, o que lhes permite coordenar equipes diferentes em SRE, desenvolvimento e gerenciamento de produtos. Essa abordagem holística é particularmente importante para lançamentos complicados que podem abranger mais de meia dúzia de equipes em vários fusos horários.

Objetividade

Como um consultor apartidário, um LCE desempenha um papel de equilíbrio e mediação entre as partes interessadas, incluindo SRE, desenvolvedores de produtos, gerentes de produtos e marketing.

Como o Engenheiro de Coordenação de Lançamento é uma função do SRE, os LCEs são incentivados a priorizar a confiabilidade sobre outras preocupações. Uma empresa que não compartilha as metas de confiabilidade do Google, mas compartilha sua rápida taxa de mudança, pode escolher um incentivo diferente estrutura.

Configurando um processo de lançamento

O Google aprimorou seu processo de lançamento ao longo de um período de mais de 10 anos. Ao longo do tempo, identificamos uma série de critérios que caracterizam um bom processo de lançamento:

Leve

Fácil para os desenvolvedores

Robusto

Captura erros óbvios

Minucioso

Aborda detalhes importantes de forma consistente e reproduzível

Escalável

Acomoda um grande número de lançamentos simples e lançamentos menos complexos

Adaptável

Funciona bem para tipos comuns de lançamentos (por exemplo, adicionar um novo idioma de interface do usuário a um produto) e novos tipos de lançamentos (por exemplo, o lançamento inicial do navegador Chrome ou Google Fiber)

Como você pode ver, alguns desses requisitos estão em conflito óbvio. Por exemplo, é difícil projetar um processo que seja simultaneamente leve e completo. Equilibrar esses critérios entre si requer trabalho contínuo. O Google empregou com sucesso algumas táticas para nos ajudar a atingir esses critérios:

Simplicidade

Acerte no básico. Não planeje para todas as eventualidades.

Uma abordagem de alto toque

Engenheiros experientes personalizam o processo para se adequar a cada lançamento.

Caminhos comuns rápidos

Identifique classes de lançamentos que sempre seguem um padrão comum (como o lançamento de um produto em um novo país) e forneça um processo de lançamento simplificado para essa classe.

A experiência demonstrou que os engenheiros tendem a evitar processos que consideram muito onerosos ou que agregam valor insuficiente, especialmente quando uma equipe já está no modo de crise e o processo de lançamento é visto como apenas mais um item bloqueando seu lançamento. Por esse motivo, a LCE deve otimizar a experiência de lançamento continuamente para encontrar o equilíbrio certo entre custo e benefício.

A lista de verificação de lançamento

As listas de verificação são usadas para reduzir falhas e garantir consistência e integridade em uma variedade de disciplinas. Exemplos comuns incluem listas de verificação de pré-voo de aviação e listas de verificação cirúrgicas [Gaw09]. Da mesma forma, a LCE emprega uma lista de verificação de lançamento para qualificação de lançamento. A lista de verificação (Apêndice E) ajuda um LCE a avaliar o lançamento e fornece à equipe de lançamento itens de ação e indicadores para mais informações. Aqui estão alguns exemplos de itens que uma lista de verificação pode incluir:

- **Pergunta:** Você precisa de um novo nome de domínio?
 - **Item de ação:** Coordene com o marketing do nome de domínio desejado e solicite o registro do domínio.
Aqui está um link para o formulário de marketing.
- **Pergunta:** Você está armazenando dados persistentes?
 - **Item de ação:** certifique-se de implementar backups. Aqui estão as instruções para implementar backups. •
- **Pergunta:** Um usuário pode abusar do seu serviço?
 - **Item de ação:** Implementar limitação de taxa e cotas. Use o seguinte compartilhado serviço.

Na prática, há um número quase infinito de perguntas a serem feitas sobre qualquer sistema, e é fácil que a lista de verificação cresça para um tamanho incontrolável. A manutenção de uma gestão

O fardo para os desenvolvedores requer uma curadoria cuidadosa da lista de verificação. Em um esforço para conter seu crescimento, a certa altura, adicionar novas perguntas à lista de verificação de lançamento do Google exigia a aprovação de um vice-presidente. A LCE agora usa as seguintes diretrizes:

- A importância de cada pergunta deve ser comprovada, idealmente por um lançamento anterior desastre.
- Cada instrução deve ser concreta, prática e razoável para os desenvolvedores concluir.

A lista de verificação precisa de atenção contínua para permanecer relevante e atualizada: as recomendações mudam ao longo do tempo, os sistemas internos são substituídos por sistemas diferentes e as áreas de preocupação de lançamentos anteriores tornam-se obsoletas devido a novas políticas e processos. Os LCEs organizam a lista de verificação continuamente e fazem pequenas atualizações quando os membros da equipe percebem itens que precisam ser modificados. Uma ou duas vezes por ano, um membro da equipe revisa toda a lista de verificação para identificar itens obsoletos e, em seguida, trabalha com proprietários de serviços e especialistas no assunto para modernizar seções da lista de verificação.

Impulsionando a convergência e a simplificação

Em uma grande organização, os engenheiros podem não estar cientes da infraestrutura disponível para tarefas comuns (como limitação de taxa). Na falta de orientação adequada, eles provavelmente irão implementar novamente as soluções existentes. Convergir em um conjunto de bibliotecas de infraestrutura comuns evita esse cenário e oferece benefícios óbvios para a empresa: reduz o esforço duplicado, torna o conhecimento mais facilmente transferível entre os serviços e resulta em um nível mais alto de engenharia e qualidade de serviço devido à concentração dada à infraestrutura.

Quase todos os grupos do Google participam de um processo de lançamento comum, o que torna a lista de verificação de lançamento um veículo para impulsionar a convergência em infraestrutura comum.

Em vez de implementar uma solução personalizada, a LCE pode recomendar a infraestrutura existente como blocos de construção – infraestrutura que já é reforçada por anos de experiência e que pode ajudar a mitigar os riscos de capacidade, desempenho ou escalabilidade.

Os exemplos incluem infraestrutura comum para limitação de taxa ou cotas de usuários, envio de novos dados para servidores ou lançamento de novas versões de um binário. Esse tipo de padronização ajudou a simplificar radicalmente a lista de verificação de lançamento: por exemplo, longas seções da lista de verificação que tratam dos requisitos de limitação de taxa podem ser substituídas por uma única linha que declara: “Implementar limitação de taxa usando o sistema X”.

Devido à ampla experiência em todos os produtos do Google, os LCEs também estão em uma posição única para identificar oportunidades de simplificação. Enquanto trabalham em um lançamento, eles testemunham os obstáculos em primeira mão: quais partes de um lançamento estão causando mais dificuldade, quais etapas levam uma quantidade de tempo desproporcional, quais problemas são resolvidos independentemente repetidamente de maneiras semelhantes, onde as faltas de infraestrutura comum, ou onde existe duplicação na infraestrutura comum.

Os LCEs têm várias maneiras de otimizar a experiência de lançamento e atuar como defensores das equipes de lançamento. Por exemplo, LCEs podem trabalhar com os proprietários de um processo de aprovação particularmente árduo para simplificar seus critérios e implementar aprovações automáticas para casos comuns. Os LCEs também podem escalar os pontos problemáticos para os proprietários da infraestrutura comum e criar um diálogo com os clientes. Aproveitando a experiência adquirida ao longo de vários lançamentos anteriores, os LCEs podem dedicar mais atenção às preocupações e sugestões individuais.

Lançando o Inesperado Quando um

projeto entra em um novo espaço de produto ou vertical, um LCE pode precisar criar uma lista de verificação apropriada do zero. Fazer isso geralmente envolve sintetizar a experiência de especialistas relevantes no domínio. Ao elaborar uma nova lista de verificação, pode ser útil estruturar a lista de verificação em torno de temas amplos, como confiabilidade, modos de falha e processos.

Por exemplo, antes de lançar o Android, o Google raramente lidava com dispositivos de consumo em massa com lógica do lado do cliente que não controlávamos diretamente. Embora possamos corrigir um bug no Gmail com mais ou menos facilidade em horas ou dias, enviando novas versões do JavaScript para navegadores, essas correções não são uma opção para dispositivos móveis. Portanto, os LCEs que trabalham em lançamentos móveis envolveram especialistas em domínio móvel para determinar quais seções das listas de verificação existentes se aplicavam ou não e onde eram necessárias novas perguntas da lista de verificação. Nessas conversas, é importante manter a intenção de cada pergunta em mente para evitar a aplicação descuidada de uma pergunta concreta ou item de ação que não seja relevante para o design do produto exclusivo que está sendo lançado. Um LCE que enfrenta um lançamento incomum deve retornar aos primeiros princípios abstratos de como executar um lançamento seguro e, em seguida, se especializar para tornar a lista de verificação concreta e útil para os desenvolvedores.

Desenvolvendo uma lista de verificação de lançamento

Uma lista de verificação é fundamental para o lançamento de novos serviços e produtos com confiabilidade reproduzível. Nossa lista de verificação de lançamento cresceu ao longo do tempo e foi periodicamente selecionada por membros da equipe de Engenharia de Coordenação de Lançamento. Os detalhes de uma lista de verificação de lançamento serão diferentes para cada empresa, porque as especificidades devem ser adaptadas aos serviços internos e à infraestrutura de uma empresa. Nas seções a seguir, extraímos vários temas das listas de verificação de LCE do Google e fornecemos exemplos de como esses temas podem ser desenvolvidos.

Arquitetura e dependências Uma revisão de

arquitetura permite determinar se o serviço está usando a infraestrutura compartilhada corretamente e identifica os proprietários da infraestrutura compartilhada como partes interessadas adicionais no lançamento. O Google tem um grande número de serviços internos que são usados como blocos de construção para novos produtos. Durante os estágios posteriores do planejamento de capacidade

(consulte [\[Hix15a\]](#)), a lista de dependências identificadas nesta seção da lista de verificação pode ser usada para garantir que cada dependência seja provisionada corretamente.

Exemplos de perguntas da lista de verificação

- Qual é o seu fluxo de solicitação do usuário para o front-end e para o back-end? •

Existem diferentes tipos de solicitações com diferentes requisitos de latência?

Exemplo de itens de ação

- Isolte as solicitações voltadas ao usuário das solicitações não direcionadas ao usuário. • Valide as suposições de volume de solicitações. Uma visualização de página pode se transformar em muitas solicitações de.

Integração Os

serviços de muitas empresas são executados em um ecossistema interno que inclui diretrizes sobre como configurar máquinas, configurar novos serviços, configurar monitoramento, integrar com平衡amento de carga, configurar endereços DNS e assim por diante. Esses ecossistemas internos geralmente crescem com o tempo e geralmente têm suas próprias idiossincrasias e armadilhas para navegar.

Assim, esta seção da lista de verificação varia muito de empresa para empresa.

Exemplo de itens de ação

- Configure um novo nome DNS para seu serviço. • Configure平衡adores de carga para falar com seu serviço.
- Configure o monitoramento para seu novo serviço.

Planejamento de capacidade

Novos recursos podem apresentar um aumento temporário no uso no lançamento que desaparece em alguns dias. O tipo de carga de trabalho ou combinação de tráfego de um pico de lançamento pode ser substancialmente diferente do estado estável, gerando resultados de teste de carga. O interesse público é notoriamente difícil de prever, e alguns produtos do Google tiveram que acomodar picos de lançamento até 15 vezes maiores do que o estimado inicialmente. O lançamento inicial em uma região ou país de cada vez ajuda a desenvolver a confiança para lidar com lançamentos maiores.

A capacidade interage com redundância e disponibilidade. Por exemplo, se você precisar de três implantações replicadas para atender 100% do seu tráfego no pico, precisará manter quatro ou cinco implantações, uma ou duas delas redundantes, para proteger os usuários contra manutenção e mau funcionamento inesperado. Recursos de datacenter e rede

muitas vezes têm um longo prazo de entrega e precisam ser solicitados com antecedência suficiente para sua empresa obtê-los.

Exemplos de perguntas da lista de verificação

- Este lançamento está vinculado a um comunicado de imprensa, anúncio, postagem de blog ou outra forma de promoção?
- Quanto tráfego e taxa de crescimento você espera durante e após o lançamento? • Você obteve todos os recursos de computação necessários para dar suporte ao seu tráfego?

Modos de falha

Uma análise sistemática dos possíveis modos de falha de um novo serviço garante alta confiabilidade desde o início. Nesta parte da lista de verificação, examine cada componente e dependência e identifique o impacto de sua falha. O serviço pode lidar com falhas de máquinas individuais? Interrupções do datacenter? Falhas de rede? Como lidamos com dados de entrada ruins? Estamos preparados para a possibilidade de um ataque de negação de serviço (DoS)?

O serviço pode continuar atendendo no modo degradado se uma de suas dependências falhar?

Como lidamos com a indisponibilidade de uma dependência na inicialização do serviço?

Durante o tempo de execução?

Exemplos de perguntas da lista de verificação

- Você tem pontos únicos de falha em seu projeto? • Como você reduz a indisponibilidade de suas dependências?

Exemplo de itens de ação

- Implemente prazos de solicitação para evitar a falta de recursos para longo prazo solicitações de.
- Implemente a redução de carga para rejeitar novas solicitações antecipadamente em situações de sobrecarga.

Comportamento do cliente

Em um site tradicional, raramente há necessidade de levar em consideração o comportamento abusivo de usuários legítimos. Quando cada solicitação é acionada por uma ação do usuário, como um clique em um link, as taxas de solicitação são limitadas pela rapidez com que os usuários podem clicar. Para dobrar a carga, o número de usuários teria que dobrar.

Esse axioma não é mais válido quando consideramos clientes que iniciam ações sem entrada do usuário - por exemplo, um aplicativo de telefone celular que sincroniza periodicamente seus dados no

nuvem ou um site que é atualizado periodicamente. Em qualquer um desses cenários, o comportamento abusivo do cliente pode facilmente ameaçar a estabilidade de um serviço. (Há também o tópico de proteger um serviço contra tráfego abusivo, como scrapers e ataques de negação de serviço, o que é diferente de projetar um comportamento seguro para clientes primários.)

Exemplo de pergunta da lista de verificação

- Você tem a funcionalidade de auto-save/auto-complete/heartbeat?

Exemplo de itens de ação

- Certifique-se de que seu cliente recue exponencialmente em caso de falha.
- Certifique-se de fazer jitter em solicitações automáticas.

Processos e Automação

O Google incentiva os engenheiros a usar ferramentas padrão para automatizar processos comuns. No entanto, a automação nunca é perfeita e todo serviço tem processos que precisam ser executados por um ser humano: criar uma nova versão, mover o serviço para um data center diferente, restaurar dados de backups e assim por diante. Por motivos de confiabilidade, nos esforçamos para minimizar pontos únicos de falha, que incluem humanos.

Esses processos restantes devem ser documentados antes do lançamento para garantir que as informações sejam traduzidas da mente de um engenheiro para o papel enquanto ainda estão frescas e que estejam disponíveis em caso de emergência. Os processos devem ser documentados de forma que qualquer membro da equipe possa executar um determinado processo em caso de emergência.

Exemplo de pergunta da lista de verificação

- Existem processos manuais necessários para manter o serviço funcionando?

Exemplo de itens de ação

- Documente todos os processos manuais.
- Documente o processo para mover seu serviço para um novo datacenter. •

Automatize o processo de construção e lançamento de uma nova versão.

Processo de desenvolvimento

O Google é um grande usuário de controle de versão e quase todos os processos de desenvolvimento estão profundamente integrados ao sistema de controle de versão. Muitas de nossas práticas recomendadas

giram em torno de como usar o sistema de controle de versão de forma eficaz. Por exemplo, realizamos a maior parte do desenvolvimento na ramificação principal, mas as versões são criadas em ramificações separadas por versão. Essa configuração facilita a correção de bugs em uma versão sem extrair alterações não relacionadas da linha principal.

O Google também usa o controle de versão para outros fins, como armazenar arquivos de configuração. Muitas das vantagens do controle de versão — rastreamento de histórico, atribuição de alterações a indivíduos e revisões de código — também se aplicam a arquivos de configuração. Em alguns casos, também propagamos as alterações do sistema de controle de versão para os servidores ativos automaticamente, de modo que um engenheiro só precisa enviar uma alteração para ativá-la.

Exemplo de itens de ação

- Verifique todo o código e arquivos de configuração no sistema de controle de versão.
- Corte cada versão em uma nova ramificação de versão.

Dependências externas

Às vezes,

um lançamento depende de fatores além do controle da empresa. Identificar esses fatores permite mitigar a imprevisibilidade que eles acarretam. Por exemplo, a dependência pode ser uma biblioteca de códigos mantida por terceiros ou um serviço ou dados fornecidos por outra empresa. Quando ocorre uma interrupção do fornecedor, um bug, um erro sistemático, um problema de segurança ou um limite de escalabilidade inesperado, o planejamento prévio permitirá que você evite ou reduza os danos aos seus usuários. No histórico de lançamentos do Google, usamos filtragem e/ou reescrita de proxies, pipelines de transcodificação de dados e caches para mitigar alguns desses riscos.

Exemplos de perguntas da lista de verificação

- Quais códigos, dados, serviços ou eventos de terceiros fazem o serviço ou o lançamento depender de?
- Algum parceiro depende do seu serviço? Em caso afirmativo, eles precisam ser notificados seu lançamento?
- O que acontece se você ou o fornecedor não cumprirem um prazo de lançamento rígido?

Planejamento de Rollout

Em grandes sistemas distribuídos, poucos eventos acontecem instantaneamente. Por motivos de confiabilidade, esse imediatismo geralmente não é ideal. Uma inicialização complicada pode exigir a ativação de recursos individuais em vários subsistemas diferentes, e cada uma dessas alterações de configuração pode levar horas para ser concluída. Ter uma configuração de trabalho

em uma instância de teste não garante que a mesma configuração possa ser implementada na instância ativa. Às vezes, uma dança complicada ou uma funcionalidade especial é necessária para fazer com que todos os componentes sejam iniciados de forma limpa e na ordem correta.

Requisitos externos de equipes como marketing e relações públicas podem adicionar complicações adicionais. Por exemplo, uma equipe pode precisar que um recurso esteja disponível a tempo para a palestra em uma conferência, mas precisa manter o recurso invisível antes da palestra.

As medidas de contingência são outra parte do planejamento de implantação. E se você não conseguir habilitar o recurso a tempo da palestra? Às vezes, essas medidas de contingência são tão simples quanto preparar um conjunto de slides de backup que diz: "Lançaremos esse recurso nos próximos dias" em vez de "Lançamos esse recurso".

Exemplo de itens de ação

- Configure um plano de lançamento que identifique as ações a serem tomadas para lançar o serviço. Identificar quem é responsável por cada item.
- Identifique o risco nas etapas individuais de lançamento e implemente medidas de contingência.

Técnicas selecionadas para lançamentos confiáveis

Conforme descrito em outras partes deste livro, o Google desenvolveu várias técnicas para executar sistemas confiáveis ao longo dos anos. Algumas dessas técnicas são particularmente adequadas para lançar produtos com segurança. Eles também oferecem vantagens durante a operação regular do serviço, mas é particularmente importante acertá-los durante a fase de lançamento.

Distribuições graduais e graduais Um ditado

da administração do sistema é “nunca mude um sistema em execução”. Qualquer mudança representa risco, e o risco deve ser minimizado para garantir a confiabilidade de um sistema. O que é verdade para qualquer sistema pequeno é duplamente verdade para sistemas altamente replicados e distribuídos globalmente, como os executados pelo Google.

Muito poucos lançamentos no Google são do tipo “push-button”, em que lançamos um novo produto em um momento específico para o mundo inteiro usar. Com o tempo, o Google desenvolveu vários padrões que nos permitem lançar produtos e recursos gradualmente e, assim, minimizar os riscos; consulte o [Apêndice B](#).

Quase todas as atualizações dos serviços do Google ocorrem gradualmente, de acordo com um processo definido, com etapas de verificação apropriadas intercaladas. Um novo servidor pode ser instalado em algumas máquinas em um datacenter e observado por um período de tempo definido. Se tudo estiver bem, o servidor será instalado em todas as máquinas em um datacenter, observado novamente e instalado em todas as máquinas globalmente. Os primeiros estágios de um lançamento geralmente são

chamados de “canários” – uma alusão aos canários carregados por mineiros em uma mina de carvão para detectar gases perigosos. Nossos servidores canários detectam efeitos perigosos do comportamento do novo software sob tráfego real de usuários.

O teste canário é um conceito incorporado em muitas das ferramentas internas do Google usadas para fazer alterações automatizadas, bem como para sistemas que alteram arquivos de configuração. As ferramentas que gerenciam a instalação de um novo software normalmente observam o servidor recém-iniciado por um tempo, certificando-se de que o servidor não falhe ou se comporte mal.

Se a alteração não passar no período de validação, ela será revertida automaticamente.

O conceito de lançamentos graduais se aplica até mesmo a softwares que não são executados nos servidores do Google. Novas versões de um aplicativo Android podem ser lançadas de maneira gradual, na qual a versão atualizada é oferecida a um subconjunto das instalações para atualização. A porcentagem de instâncias atualizadas aumenta gradualmente ao longo do tempo até atingir 100%.

Esse tipo de distribuição é particularmente útil se a nova versão resultar em tráfego adicional para os servidores de back-end nos datacenters do Google. Dessa forma, podemos observar o efeito em nossos servidores à medida que lançamos gradualmente a nova versão e detectamos problemas antecipadamente.

O sistema de convite é outro tipo de lançamento gradual. Frequentemente, em vez de permitir inscrições gratuitas para um novo serviço, apenas um número limitado de usuários pode se inscrever por dia. As inscrições com taxa limitada geralmente são associadas a um sistema de convites, no qual um usuário pode enviar um número limitado de convites para amigos.

Estruturas de sinalização de

recursos O Google geralmente aumenta os testes de pré-lançamento com estratégias que reduzem o risco de interrupção. Um mecanismo para implementar mudanças lentamente, permitindo a observação do comportamento total do sistema sob cargas de trabalho reais, pode pagar seu investimento de engenharia em confiabilidade, velocidade de engenharia e tempo de lançamento no mercado. Esses mecanismos se mostraram particularmente úteis nos casos em que ambientes de teste realistas são impraticáveis ou para lançamentos particularmente complexos para os quais os efeitos podem ser difíceis de prever.

Além disso, nem todas as mudanças são iguais. Às vezes, você simplesmente quer verificar se um pequeno ajuste na interface do usuário melhora a experiência de seus usuários. Essas pequenas mudanças não devem envolver milhares de linhas de código ou um processo de inicialização pesado. Você pode querer testar centenas dessas mudanças em paralelo.

Finalmente, às vezes você quer descobrir se uma pequena amostra de usuários gosta de usar um protótipo inicial de um novo recurso difícil de implementar. Você não quer gastar meses de esforço de engenharia para fortalecer um novo recurso para atender a milhões de usuários, apenas para descobrir que o recurso é um fracasso.

Para acomodar os cenários anteriores, vários produtos do Google criaram estruturas de sinalização de recursos. Algumas dessas estruturas foram projetadas para lançar novos recursos gradualmente de 0% a 100% dos usuários. Sempre que um produto introduzia um framework desse tipo, o próprio framework era endurecido o máximo possível para que a maior parte de sua aplicação

não precisariam de qualquer envolvimento da LCE. Esses frameworks geralmente atendem aos seguintes requisitos:

- Implemente muitas mudanças em paralelo, cada uma para alguns servidores, usuários, entidades ou datacenters
- Aumente gradualmente para um grupo maior, mas limitado de usuários, geralmente entre 1 e 10 por cento
- Tráfego direto através de diferentes servidores dependendo de usuários, sessões, objetos, e/ou locais
- Trate automaticamente a falha dos novos caminhos de código por design, sem afetar usuários
- Reverta independentemente cada uma dessas alterações imediatamente no caso de erros graves ou efeitos colaterais
- Meça até que ponto cada mudança melhora a experiência do usuário

Os frameworks de sinalizadores de recursos do Google se dividem em duas classes gerais:

- Aqueles que facilitam principalmente melhorias na interface do usuário •
Aqueles que suportam mudanças arbitrárias no lado do servidor e na lógica de negócios

A estrutura de sinalizador de recurso mais simples para alterações na interface do usuário em um serviço sem estado é um reescritor de carga útil HTTP em servidores de aplicativos front-end, limitado a um subconjunto de cookies ou outro atributo de solicitação/resposta HTTP semelhante. Um mecanismo de configuração pode especificar um identificador associado aos novos caminhos de código e ao escopo da mudança (por exemplo, intervalo de modificação de hash de cookie), listas brancas e listas negras.

Os serviços com estado tendem a limitar os sinalizadores de recursos a um subconjunto de identificadores de usuário logados exclusivos ou às entidades de produto reais acessadas, como o ID de documentos, planilhas ou objetos de armazenamento. Em vez de reescrever cargas HTTP, os serviços com estado são mais propensos a fazer proxy ou redirecionar solicitações para diferentes servidores, dependendo da alteração, conferindo a capacidade de testar lógica de negócios aprimorada e novos recursos mais complexos.

Lidando com o comportamento abusivo do cliente O

exemplo mais simples de comportamento abusivo do cliente é um julgamento errado das taxas de atualização. Um novo cliente que sincroniza a cada 60 segundos, em oposição a cada 600 segundos, causa 10 vezes a carga no serviço. O comportamento de repetição tem várias armadilhas que afetam as solicitações iniciadas pelo usuário, bem como as solicitações iniciadas pelo cliente. Veja o exemplo de um serviço que está sobrecarregado e, portanto, está falhando em algumas solicitações: se os clientes tentarem novamente as solicitações com falha, eles adicionarão carga a um serviço já sobrecarregado, resultando em mais tentativas e ainda mais solicitações. Em vez disso, os clientes precisam reduzir a frequência de tentativas, geralmente adicionando atrasos exponencialmente crescentes entre as tentativas, além de considerar cuidadosamente

os tipos de erros que justificam uma nova tentativa. Por exemplo, um erro de rede geralmente garante uma nova tentativa, mas um erro HTTP 4xx (que indica um erro no lado do cliente) geralmente não.

A sincronização intencional ou inadvertida de solicitações automatizadas em um rebanho estrondoso (muito semelhante às descritas nos Capítulos 24 e 25) é outro exemplo comum de comportamento abusivo do cliente. Um desenvolvedor de aplicativo de telefone pode decidir que 2 da manhã é um bom momento para baixar atualizações, porque o usuário provavelmente está dormindo e não será incomodado pelo download. No entanto, esse design resulta em uma enxurrada de solicitações ao servidor de download às 2 da manhã todas as noites e quase nenhuma solicitação em nenhum outro momento. Em vez disso, cada cliente deve escolher o horário para esse tipo de solicitação aleatoriamente.

A aleatoriedade também precisa ser injetada em outros processos periódicos. Voltando às tentativas mencionadas anteriormente: tomemos o exemplo de um cliente que envia uma solicitação e, quando encontra uma falha, tenta novamente após 1 segundo, depois 2 segundos, depois 4 segundos e assim por diante. Sem aleatoriedade, um breve pico de solicitação que leva a um aumento da taxa de erro pode se repetir devido a novas tentativas após 1 segundo, depois 2 segundos e depois 4 segundos. Para equilibrar esses eventos sincronizados, cada atraso precisa ser ajustado (isto é, ajustado por um valor aleatório).

A capacidade de controlar o comportamento de um cliente do lado do servidor provou ser uma ferramenta importante no passado. Para um aplicativo em um dispositivo, esse controle pode significar instruir o cliente a verificar periodicamente com o servidor e baixar um arquivo de configuração. O arquivo pode habilitar ou desabilitar determinados recursos ou definir parâmetros, como a frequência com que o cliente é sincronizado ou com que frequência ele tenta novamente.

A configuração do cliente pode até habilitar uma funcionalidade totalmente nova voltada para o usuário. Ao hospedar o código que dá suporte à nova funcionalidade no aplicativo cliente antes de ativar esse recurso, reduzimos bastante o risco associado a um lançamento. O lançamento de uma nova versão se torna muito mais fácil se não precisarmos manter faixas de lançamento paralelas para uma versão com a nova funcionalidade versus sem a funcionalidade. Isso é particularmente verdadeiro se não estivermos lidando com uma única peça de nova funcionalidade, mas com um conjunto de recursos independentes que podem ser lançados em horários diferentes, o que exigiria a manutenção de uma explosão combinatória de diferentes versões.

Ter esse tipo de funcionalidade inativa também facilita o aborto de lançamentos quando os efeitos adversos são descobertos durante um lançamento. Nesses casos, podemos simplesmente desativar o recurso, iterar e lançar uma versão atualizada do aplicativo. Sem esse tipo de configuração de cliente, teríamos que fornecer uma nova versão do aplicativo sem o recurso e atualizar o aplicativo nos telefones de todos os usuários.

Comportamento de sobrecarga e testes de carga

As situações de sobrecarga são um modo de falha particularmente complexo e, portanto, merecem atenção adicional. O sucesso descontrolado geralmente é a causa mais bem-vinda da sobrecarga

quando um novo serviço é iniciado, mas existem inúmeras outras causas, incluindo falhas de balanceamento de carga, interrupções de máquina, comportamento sincronizado do cliente e ataques externos.

Um modelo ingênuo assume que o uso da CPU em uma máquina que fornece um serviço específico é dimensionado linearmente com a carga (por exemplo, número de solicitações ou quantidade de dados processados) e, uma vez esgotada a CPU disponível, o processamento simplesmente se torna mais lento.

Infelizmente, os serviços raramente se comportam dessa forma ideal no mundo real. Muitos serviços são muito mais lentos quando não são carregados, geralmente devido ao efeito de vários tipos de caches, como caches de CPU, caches JIT e caches de dados específicos de serviço.

À medida que a carga aumenta, geralmente há uma janela na qual o uso da CPU e a carga no serviço correspondem linearmente e os tempos de resposta permanecem praticamente constantes.

Em algum momento, muitos serviços atingem um ponto de não linearidade à medida que se aproximam da sobrecarga. Nos casos mais benignos, os tempos de resposta simplesmente começam a aumentar, resultando em uma experiência do usuário degradada, mas não necessariamente causando uma interrupção (embora uma dependência lenta possa causar erros visíveis ao usuário na pilha, devido a prazos RPC excedidos). Nos casos mais drásticos, um serviço trava completamente em resposta à sobrecarga.

Para citar um exemplo específico de comportamento de sobrecarga: um serviço registrou informações de depuração em resposta a erros de backend. Descobriu-se que registrar informações de depuração era mais caro do que lidar com a resposta de back-end em um caso normal. Portanto, à medida que o serviço ficou sobrecarregado e expirou as respostas de back-end dentro de sua própria pilha RPC, o serviço gastou ainda mais tempo de CPU registrando essas respostas, expirando mais solicitações nesse meio tempo até que o serviço parou completamente. Em serviços executados na Java Virtual Machine (JVM), um efeito semelhante de travamento é às vezes chamado de “GC (garbage collection) thrashing”. Nesse cenário, o gerenciamento de memória interna da máquina virtual é executado em ciclos cada vez mais próximos, tentando liberar memória até que a maior parte do tempo de CPU seja consumida pelo gerenciamento de memória.

Infelizmente, é muito difícil prever a partir dos primeiros princípios como um serviço reagirá à sobrecarga. Portanto, os testes de carga são uma ferramenta inestimável, tanto por motivos de confiabilidade quanto por planejamento de capacidade, e o teste de carga é necessário para a maioria dos lançamentos.

Desenvolvimento do LCE

Nos anos de formação do Google, o tamanho da equipe de engenharia dobrou a cada ano por vários anos consecutivos, fragmentando o departamento de engenharia em muitas equipes pequenas trabalhando em muitos novos produtos e recursos experimentais. Em tal clima, engenheiros novatos correm o risco de repetir os erros de seus antecessores, especialmente quando se trata de lançar novos recursos e produtos com sucesso.

Para mitigar a repetição de tais erros, capturando as lições aprendidas em lançamentos anteriores, um pequeno grupo de engenheiros experientes, chamados de “Engenheiros de Lançamento”, volý

desvinculado para atuar como uma equipe de consultoria. Os Engenheiros de Lançamento desenvolveram checklists para lançamentos de novos produtos, abrangendo tópicos como:

- Quando consultar o departamento jurídico • Como selecionar nomes de domínio
- Como registrar novos domínios sem configurar incorretamente o DNS • Armadilhas comuns de projeto de engenharia e implantação de produção

As “Revisões de Lançamento”, como as sessões de consultoria dos Engenheiros de Lançamento passaram a ser chamadas, tornaram-se uma prática comum dias ou semanas antes do lançamento de muitos novos produtos.

Em dois anos, os requisitos de implantação do produto na lista de verificação de lançamento ficaram longos e complexos. Combinado com a crescente complexidade do ambiente de implantação do Google, tornou-se cada vez mais desafiador para os engenheiros de produto manterem-se atualizados sobre como fazer alterações com segurança. Ao mesmo tempo, a organização SRE estava crescendo rapidamente, e SREs inexperientes eram às vezes excessivamente cautelosos e avessos a mudanças. O Google corria o risco de que as negociações resultantes entre essas duas partes reduzissem a velocidade dos lançamentos de produtos/recursos.

Para mitigar esse cenário do ponto de vista da engenharia, a SRE contratou uma pequena equipe de LCEs em tempo integral em 2004. Eles foram responsáveis por acelerar os lançamentos de novos produtos e recursos e, ao mesmo tempo, aplicar a experiência em SRE para garantir que o Google enviasse produtos confiáveis com alta disponibilidade e baixa latência.

Os LCEs eram responsáveis por garantir que os lançamentos fossem executados rapidamente sem que os serviços caíssem e que, se um lançamento falhasse, não derrubaria outros produtos.

As LCEs também eram responsáveis por manter as partes interessadas informadas sobre a natureza e a probabilidade de tais falhas sempre que os cantos eram cortados, a fim de acelerar o tempo de colocação no mercado. Suas sessões de consultoria foram formalizadas como Avaliações de Produção.

Evolução da Lista de Verificação LCE

À medida que o ambiente do Google se tornou mais complexo, o mesmo aconteceu com a lista de verificação de engenharia de coordenação de lançamentos (consulte o [Apêndice E](#)) e o volume de lançamentos. Em 3,5 anos, um LCE executou 350 lançamentos por meio da Lista de Verificação de LCE. Como a equipe teve uma média de cinco engenheiros durante esse período, isso se traduz em um rendimento de lançamento do Google de mais de 1.500 lançamentos em 3,5 anos!

Embora cada pergunta da Lista de Verificação LCE seja simples, há muita complexidade embutida no que motivou a pergunta e nas implicações de sua resposta. Para entender completamente esse grau de complexidade, uma nova contratação de LCE requer cerca de seis meses de treinamento.

À medida que o volume de lançamentos crescia, acompanhando a duplicação anual da equipe de engenharia do Google, os LCEs buscavam maneiras de otimizar suas análises. LCEs identificaram gato

egories de lançamentos de baixo risco que eram altamente improváveis de enfrentar ou causar contratemplos. Por exemplo, um lançamento de recurso que não envolvesse novos executáveis de servidor e um aumento de tráfego abaixo de 10% seria considerado de baixo risco. Esses lançamentos foram confrontados com uma lista de verificação quase trivial, enquanto os lançamentos de maior risco passaram por toda a gama de verificações e balanços. Em 2008, 30% das revisões foram consideradas de baixo risco.

Simultaneamente, o ambiente do Google estava aumentando, removendo restrições em muitos lançamentos. Por exemplo, a aquisição do YouTube forçou o Google a construir sua rede e utilizar a largura de banda de forma mais eficiente. Isso significava que muitos produtos menores “cairiam dentro das brechas”, evitando processos complexos de planejamento e provisionamento de capacidade de rede, acelerando assim seus lançamentos. O Google também começou a construir datacenters muito grandes capazes de hospedar vários serviços dependentes sob o mesmo teto. Esse desenvolvimento simplificou o lançamento de novos produtos que precisavam de grandes quantidades de capacidade em vários serviços preexistentes dos quais dependiam.

Problemas que a LCE não resolveu

Embora as LCEs tentassem reduzir ao mínimo a burocracia das revisões, tais esforços foram insuficientes. Em 2009, as dificuldades de lançar um pequeno novo serviço no Google se tornaram uma lenda. Os serviços que cresceram em maior escala enfrentaram seu próprio conjunto de problemas que a Coordenação de Lançamento não conseguiu resolver.

Mudanças de

escalabilidade Quando os produtos são bem sucedidos muito além de quaisquer estimativas iniciais, e seu uso aumenta em mais de duas ordens de magnitude, manter o ritmo de sua carga exige muitas mudanças de projeto. Essas mudanças de escalabilidade, combinadas com adições contínuas de recursos, geralmente tornam o produto mais complexo, frágil e difícil de operar. Em algum momento, a arquitetura original do produto se torna incontrolável e o produto precisa ser completamente rearquitetado. Rearquitetar o produto e, em seguida, migrar todos os usuários da arquitetura antiga para a nova requer um grande investimento de tempo e recursos de desenvolvedores e SREs, diminuindo a taxa de desenvolvimento de novos recursos durante esse período.

Carga operacional crescente

Ao executar um serviço após o lançamento, a carga operacional, a quantidade de engenharia manual e repetitiva necessária para manter um sistema funcionando, tende a crescer com o tempo, a menos que esforços sejam feitos para controlar essa carga. O ruído das notificações automatizadas, a complexidade dos procedimentos de implantação e a sobrecarga do trabalho de manutenção manual tendem a aumentar com o tempo e consumir quantidades cada vez maiores de largura de banda do proprietário do serviço, deixando a equipe menos tempo para o desenvolvimento de recursos. A SRE tem uma meta anunciada internamente de manter o trabalho operacional abaixo de um máximo de

50%; veja o [Capítulo 5](#). Ficar abaixo desse máximo requer rastreamento constante de fontes de trabalho operacional, bem como esforço direcionado para remover essas fontes.

Rotação de infraestrutura

Se a infraestrutura subjacente (como sistemas para gerenciamento de cluster, armazenamento, monitoramento, balanceamento de carga e transferência de dados) estiver mudando devido ao desenvolvimento ativo das equipes de infraestrutura, os proprietários dos serviços executados na infraestrutura devem investir grandes quantidades de trabalho para simplesmente manter com as mudanças de infraestrutura. À medida que os recursos de infraestrutura dos quais os serviços dependem são preteridos e substituídos por novos recursos, os proprietários de serviços devem modificar continuamente suas configurações e reconstruir seus executáveis, consequentemente “executando rápido apenas para permanecer no mesmo lugar”. A solução para esse cenário é promulgar algum tipo de política de redução de rotatividade que proíba os engenheiros de infraestrutura de liberar recursos incompatíveis com versões anteriores até que eles também automatizem a migração de seus clientes para o novo recurso. A criação de ferramentas de migração automatizadas para acompanhar os novos recursos minimiza o trabalho imposto aos proprietários de serviços para acompanhar a rotatividade da infraestrutura.

Resolver esses problemas requer esforços de toda a empresa que estão muito além do escopo da LCE: uma combinação de melhores APIs e estruturas de plataforma (consulte o [Capítulo 32](#)), automação contínua de criação e teste e melhor padronização e automação nos serviços de produção do Google.

Conclusão

Empresas em rápido crescimento com alta taxa de mudança em produtos e serviços podem se beneficiar do equivalente a uma função de Engenharia de Coordenação de Lançamento.

Essa equipe é especialmente valiosa se uma empresa planeja duplicar seus desenvolvedores de produtos a cada um ou dois anos, se deve escalar seus serviços para centenas de milhões de usuários e se a confiabilidade, apesar de uma alta taxa de mudança, for importante para seus usuários.

A equipe LCE foi a solução do Google para o problema de alcançar a segurança sem impedir a mudança. Este capítulo apresentou algumas das experiências acumuladas por nossa função exclusiva de LCE durante um período de 10 anos exatamente nessas circunstâncias. Esperamos que nossa abordagem ajude a inspirar outras pessoas que enfrentam desafios semelhantes em suas respectivas organizações.

PARTE IV

Gestão

Nossa seleção final de tópicos abrange o trabalho em equipe e o trabalho em equipe. Nenhum SRE é uma ilha, e existem algumas maneiras distintas de trabalharmos.

Qualquer organização que pretenda levar a sério a administração de um braço SRE eficaz precisa considerar o treinamento. Ensinar os SREs a pensar em um ambiente complicado e em rápida mudança com um programa de treinamento bem pensado e bem executado tem a promessa de incutir as melhores práticas nas primeiras semanas ou meses de um novo contratado que, de outra forma, levariam meses ou anos para acumular. Discutimos estratégias para fazer exatamente isso no [Capítulo 28, Acelerando SREs para On-Call e Além.](#)

Como qualquer pessoa no mundo das operações sabe, a responsabilidade por qualquer serviço significativo vem com muitas interrupções: produção ficando em mau estado, pessoas solicitando atualizações para seu binário favorito, uma longa fila de solicitações de consulta... gerenciar interrupções em condições turbulentas é necessário habilidade, como discutiremos no [Capítulo 29, Lidando com Interrupções.](#)

Se as condições turbulentas persistirem por tempo suficiente, uma equipe SRE precisa começar a se recuperar da sobrecarga operacional. Temos apenas o plano de voo para você no [Capítulo 30, Incorporando um SRE para Recuperar da Sobrecarga Operacional.](#)

Escrevemos no [Capítulo 31, Comunicação e Colaboração no SRE](#), sobre os diferentes papéis dentro do SRE; comunicação entre equipes, entre locais e entre continentes; realização de reuniões de produção; e estudos de caso de como a SRE tem colaborado bem.

Por fim, o Capítulo 32, O Modelo de Engajamento do SRE em Evolução, examina uma pedra angular da operação do SRE: a revisão de prontidão de produção (PRR), uma etapa crucial na integração de um novo serviço. Discutimos como conduzir PRRs e como ir além desse modelo bem-sucedido, mas também limitado.

Leitura adicional do Google SRE

Construir sistemas confiáveis requer uma combinação cuidadosamente calibrada de habilidades, que vão desde o desenvolvimento de software até as disciplinas de engenharia e análise de sistemas menos conhecidas. Escrevemos sobre as últimas disciplinas em “The Systems Engineering Side of Site Reliability Engineering” [Hix15b].

Contratar bem os SREs é fundamental para ter uma organização de confiabilidade de alto funcionamento, conforme explorado em “Contratando Engenheiros de Confiabilidade do Local” [Jon15]. As práticas de contratação do Google foram detalhadas em textos como Work Rules! [Boc15],¹ mas a contratação de SREs tem suas próprias particularidades. Mesmo para os padrões gerais do Google, os candidatos SRE são difíceis de encontrar e ainda mais difíceis de entrevistar de forma eficaz.

¹ Escrito por Laszlo Bock, vice-presidente sênior de operações de pessoas do Google.

CAPÍTULO 28

Acelerando SREs para plantão e além

Como posso amarrar um jetpack aos meus
novatos enquanto mantenho os SREs seniores atualizados?

Escrito por Andrew Widdowson
Editado por Shylaja Nukala

Você contratou seu(s) próximo(s) SRE(s), e agora?

Você contratou novos funcionários para sua organização e eles estão começando como Engenheiros de Confiabilidade do Site. Agora você tem que treiná-los no trabalho. O investimento inicial na educação e orientação técnica dos novos SREs irá transformá-los em melhores engenheiros. Esse treinamento os acelerará a um estado de proficiência mais rápido, ao mesmo tempo em que tornará seu conjunto de habilidades mais robusto e equilibrado.

Equipes de SRE bem-sucedidas são construídas com base na confiança — para manter um serviço consistente e global, você precisa confiar que seus colegas de plantão sabem como seu sistema funciona,¹ podem diagnosticar comportamentos atípicos do sistema, sentir-se à vontade para pedir ajuda e pode reagir sob pressão para salvar o dia. É essencial, então, mas não suficiente, pensar na educação SRE através das lentes de “O que um novato precisa aprender para ficar de plantão?” Dado os requisitos em relação à confiança, você também precisa fazer perguntas como:

- Como meus atendentes existentes podem avaliar a prontidão do novato para o plantão? •
- Como podemos aproveitar o entusiasmo e a curiosidade de nossos novos contratados para garantir que os SREs existentes se beneficiem disso?

¹ E não funciona!

- Quais atividades posso comprometer nossa equipe para que beneficiem a educação de todos, mas que todos gostem?

Os alunos têm uma ampla gama de preferências de aprendizagem. Reconhecendo que você contratará pessoas que tenham uma mistura dessas preferências, seria míope atender apenas a um estilo em detrimento dos outros. Assim, não existe um estilo de educação que funcione melhor para treinar novos SREs, e certamente não existe uma fórmula mágica que funcione para todas as equipes de SRE. A Tabela 28-1 lista as práticas de treinamento recomendadas (e seus antipadrões correspondentes) que são bem conhecidas do SRE no Google. Essas práticas representam uma ampla gama de opções disponíveis para tornar sua equipe bem informada sobre os conceitos de SRE, tanto agora quanto continuamente.

Tabela 28-1. Práticas de educação SRE

Padrões recomendados	
Projetar experiências de aprendizagem concretas e sequenciais para os alunos seguirem	os alunos com trabalho braçal (por exemplo, alerta/triagem de ingressos) para treiná-los; "julgamento por re"
Incentivando a engenharia reversa, o pensamento estatístico e o trabalho a partir de princípios fundamentais	Treinamento estritamente por meio de procedimentos do operador, listas de verificação e manuais Tratar as interrupções como segredos a serem enterrados
Celebrando a análise do fracasso sugerindo post-mortems para os alunos lerem	para evitar culpa
Criando quebras contidas, mas realistas, para os alunos usarem monitoramento e ferramentas reais	Ter a primeira chance de x algo só ocorrer depois que um aluno já estiver de plantão
Interpretando desastres teóricos como um grupo, para misturar as abordagens de solução de problemas de uma equipe	Criar especialistas na equipe cujas técnicas e conhecimentos são compartimentados
Permitir que os alunos acompanhem sua rotação de plantão mais cedo, comparando notas com o chamador	Levar os alunos a serem os principais de plantão antes que eles alcancem uma compreensão holística de seu serviço
Emparelhar os alunos com SREs especializados para revisar seções específicas do plano de treinamento de plantão	Tratar planos de treinamento de plantão como estáticos e intocáveis, exceto por especialistas no assunto
Esculpir trabalhos de projeto não triviais para os alunos realizarem, permitindo que eles obtenham propriedade parcial da pilha	Conceder todos os novos trabalhos de projeto aos SREs mais seniores, deixando os SREs juniores para pegar os restos

O restante deste capítulo apresenta os principais temas que consideramos eficazes na aceleração de SREs para plantão e além. Esses conceitos podem ser visualizados em um projeto para SREs bootstrap ([Figura 28-1](#)).

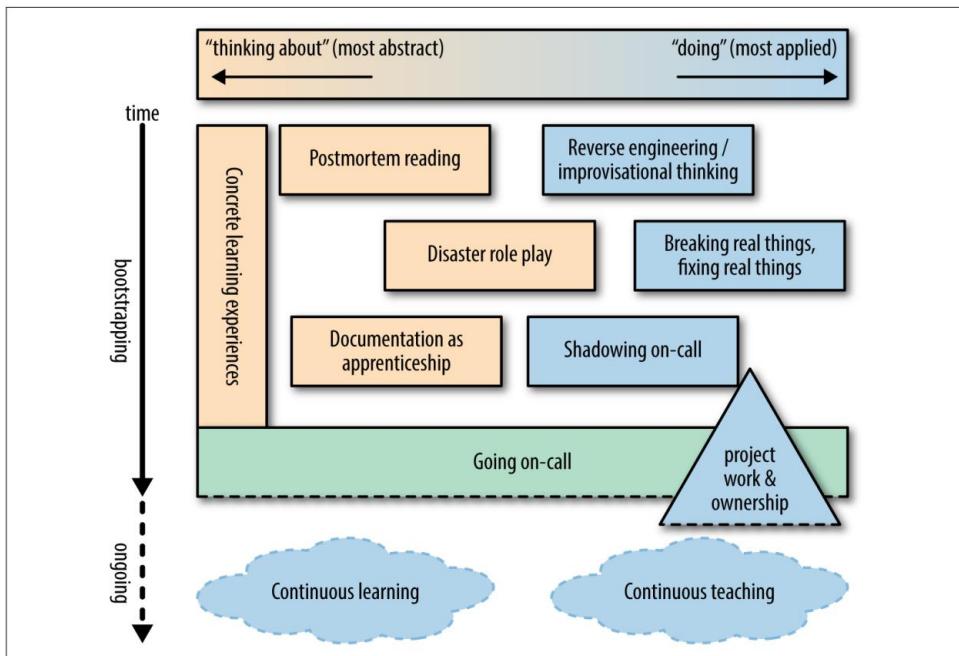


Figura 28-1. Um plano para inicializar um SRE para plantão e além

Esta ilustração captura as melhores práticas que as equipes de SRE podem escolher para ajudar a recrutar novos membros, mantendo os talentos seniores atualizados. Das muitas ferramentas aqui, você pode escolher as atividades que melhor se adequam à sua equipe.

A ilustração tem dois eixos:

- O eixo x representa o espectro entre diferentes tipos de trabalho, desde resumo às atividades aplicadas.
- O eixo y representa o tempo. Lendo de cima para baixo, os novos SREs têm muito pouco conhecimento sobre os sistemas e serviços pelos quais serão responsáveis, então post-mortems detalhando como esses sistemas falharam no passado são um bom ponto de partida. Os novos SREs também podem tentar fazer engenharia reversa de sistemas a partir dos fundamentos, já que estão começando do zero. Uma vez que eles entendam mais sobre seus sistemas e tenham feito algum trabalho prático, os SREs estão prontos para acompanhar o plantão e começar a consertar a documentação incompleta ou desatualizada.

Dicas para interpretar esta ilustração:

- Ficar de plantão é um marco na carreira de um novo SRE, após o qual o aprendizado se torna muito mais nebuloso, indefinido e autodirigido – daí as linhas tracejadas em torno das atividades que acontecem durante ou após o SRE ficar de plantão.

- A forma triangular do trabalho do projeto e propriedade indica que o trabalho do projeto começa pequeno e se desenvolve ao longo do tempo, tornando-se mais complexo e provavelmente continuando bem depois de ficar de plantão.
- Algumas dessas atividades e práticas são muito abstratas/passivas, e algumas são muito aplicadas/ativas. Algumas atividades são misturas de ambos. É bom ter uma variedade de modalidades de aprendizagem para se adequar a diferentes estilos de aprendizagem. • Para efeito máximo, as atividades e práticas de treinamento devem ser ritmadas adequadamente: algumas são apropriadas para serem realizadas imediatamente, algumas devem acontecer logo antes de um SRE entrar oficialmente em serviço e algumas devem ser contínuas e contínuas mesmo por SREs experientes. As experiências concretas de aprendizagem devem acontecer durante todo o tempo que antecede o SRE de plantão.

Experiências Iniciais de Aprendizagem: O Caso da Estrutura Sobre Caos

Conforme discutido em outra parte deste livro, as equipes de SRE realizam uma mistura natural de trabalho proativo² e reativo³. Deve ser uma meta forte de toda equipe de SRE conter e reduzir o trabalho reativo por meio de ampla proatividade, e a abordagem que você adota para integrar seu(s) novato(s) não deve ser exceção. Considere o seguinte processo de integração muito comum, mas infelizmente abaixo do ideal:

John é o mais novo membro da equipe FooServer SRE. Os SREs seniores dessa equipe são encarregados de muito trabalho pesado, como responder a tickets, lidar com alertas e realizar implementações binárias tediosas. No primeiro dia de trabalho de John, ele recebe todos os novos tíquetes recebidos. Ele é informado de que pode pedir a qualquer membro da equipe SRE que o ajude a obter os antecedentes necessários para decifrar um ticket. “Claro, haverá muito aprendizado inicial que você terá que fazer”, diz o gerente de John, “mas eventualmente você ficará muito mais rápido nesses tickets. Um dia, ele vai clicar e você saberá muito sobre todas as ferramentas que usamos, os procedimentos que seguimos e os sistemas que mantemos.” Um membro sênior da equipe comenta: “Estamos jogando você no fundo da piscina aqui”.

Esse método de “prova de fogo” de orientar os novatos geralmente nasce do ambiente atual de uma equipe; Equipes de SRE reativas e orientadas a operações “treinam” seus membros mais novos, fazendo-os... bem, reagir! Uma e outra vez. Se você tiver sorte, os engenheiros que já são bons em lidar com a ambiguidade vão rastejar para fora do buraco em que você os colocou. Mas as chances são de que essa estratégia tenha alienado vários engenheiros capazes. Embora tal abordagem possa eventualmente produzir ótimos funcionários de operações, seus resultados ficarão aquém da marca. A abordagem de teste por fogo também pressupõe que muitos ou a maioria dos aspectos de uma equipe podem ser ensinados estritamente fazendo, em vez de raciocinar. Se o conjunto

² Exemplos de trabalho proativo de SRE incluem automação de software, consultoria de design e coordenação de lançamento.

³ Exemplos de trabalho reativo do SRE incluem depuração, solução de problemas e tratamento de escalões de plantão.

de trabalho que se encontra em uma fila de tickets fornecerá treinamento adequado para esse trabalho, então esta não é uma posição SRE.

Os alunos do SRE terão perguntas como as seguintes:

- No que estou trabalhando? •

Quanto progresso eu fiz? • Quando essas

atividades acumularão experiência suficiente para eu ficar de plantão?

Sair de uma empresa ou universidade anterior e mudar de função (de engenheiro de software tradicional ou administrador de sistemas tradicional) para essa função nebulosa de engenheiro de confiabilidade de site geralmente é suficiente para derrubar a confiança dos alunos várias vezes. Para personalidades mais introspectivas (especialmente em relação às questões nº 2 e nº 3), as incertezas decorrentes de respostas nebulosas ou pouco claras podem levar a problemas de desenvolvimento ou retenção mais lentos. Em vez disso, considere as abordagens descritas nas seções a seguir. Essas sugestões são tão concretas quanto qualquer ticket ou alerta, mas também são sequenciais e, portanto, muito mais gratificantes.

Caminhos de aprendizado que são cumulativos e ordenados Coloque

alguma ordem de aprendizado em seu(s) sistema(s) para que seus novos SREs vejam um caminho antes deles. Qualquer tipo de treinamento é melhor do que bilhetes aleatórios e interrupções, mas faça um esforço consciente para combinar a combinação certa de teoria e aplicação: conceitos abstratos que se repetirão várias vezes na jornada de um novato devem ser antecipados em sua educação, enquanto o aluno deve também receberem experiência prática o mais rápido possível.

Aprender sobre sua(s) pilha(s) e subsistema(s) requer um ponto de partida. Considere se faz mais sentido agrupar os treinamentos por semelhança de propósito ou por ordem de execução de caso normal. Por exemplo, se sua equipe for responsável por uma pilha de veiculação voltada para o usuário em tempo real, considere uma ordem de currículo como esta:

1) Como uma consulta entra no sistema

Fundamentos de rede e datacenter, balanceamento de carga de front-end, proxies, etc.

2) Serviço de frontend

Frontend(s) de aplicativo, registro de consultas, SLO(s) de experiência do usuário, etc.

3) Serviços intermediários

Caches, balanceamento de carga de back-end

4) Infraestrutura

Back-ends, infraestrutura e recursos de computação

5) Juntando tudo

Técnicas de depuração, procedimentos de escalação e cenários de emergência

Como você escolhe apresentar as oportunidades de aprendizado (bate-papos informais em quadro branco, palestras formais ou exercícios práticos de descoberta) depende de você e dos SREs que o ajudam a estruturar, projetar e fornecer treinamento. A equipe de SRE da Pesquisa Google estrutura esse aprendizado por meio de um documento chamado "lista de verificação de aprendizado de plantão". Uma seção simplificada de uma lista de verificação de aprendizado de plantão pode ter a seguinte aparência:

O Servidor de Mistura de Resultados ("Mixer")	
Frontend por: Servidor frontend	Saiba antes de seguir em frente:
Backends chamados: Servidor de recuperação de resultados, Servidor de geolocalização, banco de dados de personalização	• Quais clusters têm o Mixer implantado • Como reverte uma versão do Mixer • Quais back-ends
Especialistas em SRE: Sally W, Dave K, Jen P	do Mixer são considerados "caminho crítico"
Contatos do desenvolvedor: Jim T, results-team@	e porque
Leia e entenda os seguintes documentos:	Perguntas de compreensão:
<ul style="list-style-type: none"> • Visão geral da mixagem de resultados: seção "Execução da consulta" • Visão geral da mixagem de resultados: seção "Produção" • Manual: Como implantar um novo servidor de mixagem de resultados <p>Uma análise de desempenho do Mixer</p>	<ul style="list-style-type: none"> • P: Como o cronograma de lançamento muda se ocorrer um feriado da empresa no dia normal de compilação do lançamento? • P: Como você pode fazer um push ruim do conjunto de dados de geolocalização?

Observe que a seção anterior não codifica diretamente procedimentos, etapas de diagnóstico ou manuais; em vez disso, é um artigo relativamente à prova de futuro focando estritamente na enumeração de contatos de especialistas, destacando os recursos de documentação mais úteis, estabelecendo o conhecimento básico que você deve reunir e internalizar e fazendo perguntas de sondagem que só podem ser respondidas quando esse conhecimento básico tiver sido concluído. foi absorvido. Ele também fornece resultados concretos, para que o aluno saiba que tipos de conhecimento e habilidades ele terá adquirido ao concluir esta seção da lista de verificação de aprendizado.

É uma boa ideia que todas as partes interessadas tenham uma noção de quanta informação o trainee está retendo. Embora esse mecanismo de feedback talvez não precise ser tão formal quanto um questionário, é uma boa prática ter pedaços completos de lição de casa que façam perguntas sobre como seu(s) serviço(s) funciona(m). As respostas satisfatórias, verificadas pelo mentor do aluno, são um sinal de que a aprendizagem deve continuar para a fase seguinte. As perguntas sobre o funcionamento interno do seu serviço podem ser semelhantes às seguintes:

- Quais back-ends desse servidor são considerados "no caminho crítico" e por quê? • Quais aspectos deste servidor podem ser simplificados ou automatizados? • Onde você acha que está o primeiro gargalo nessa arquitetura? Se esse gargalo fosse saturado, que medidas você poderia tomar para aliviá-lo?

Dependendo de como as permissões de acesso estão configuradas para seu serviço, você também pode considerar a implementação de um modelo de acesso em camadas. A primeira camada de acesso permitiria ao aluno acesso somente leitura ao funcionamento interno dos componentes, e uma camada posterior permitiria que eles alterassem o estado de produção. Completando seções do

A lista de verificação de aprendizagem de plantão satisfatoriamente daria ao aluno um acesso progressivamente mais profundo ao sistema. A equipe do Search SRE chama esses níveis alcançados de “powerups”⁴ no caminho para o plantão, pois os trainees são eventualmente adicionados ao nível mais alto dos sistemas Acesso.

Trabalho de projeto direcionado, não trabalho braçal Os

SREs são solucionadores de problemas, então dê a eles um problema sério para resolver! Ao começar, ter até mesmo um pequeno senso de propriedade no serviço da equipe pode fazer maravilhas pelo aprendizado. Ao contrário, essa propriedade também pode fazer grandes avanços para a construção da confiança entre os colegas seniores, porque eles abordarão seu colega júnior para aprender sobre o(s) novo(s) componente(s) ou processo(s). As primeiras oportunidades de propriedade são padrão no Google em geral: todos os engenheiros recebem um projeto inicial destinado a fornecer um tour pela infraestrutura suficiente para permitir que eles façam uma pequena, mas útil contribuição antecipada. Ter o novo SRE dividindo o tempo entre o aprendizado e o trabalho do projeto também lhes dará um senso de propósito e produtividade, o que não aconteceria se eles gastossem tempo apenas no aprendizado ou no trabalho do projeto. Vários padrões de projetos iniciais que parecem funcionar bem incluem:

- Fazer uma alteração trivial de recurso visível ao usuário em uma pilha de serviço e, posteriormente, conduzir o lançamento do recurso até a produção. Compreender tanto a cadeia de ferramentas de desenvolvimento quanto o processo de liberação binária estimula a empatia pelos desenvolvedores.
- Adicionando monitoramento ao seu serviço onde existem atualmente pontos cegos. O novato terá que raciocinar com a lógica de monitoramento, enquanto reconcilia sua compreensão de um sistema com como ele realmente (mal) se comporta. • Automatizar um ponto problemático que não é doloroso o suficiente para já ter sido automatizado, proporcionando ao novo SRE uma apreciação do valor que os SREs atribuem ao remover o trabalho pesado de nossas operações diárias.

Criando engenharia reversa estelar e improvisação Pensadores

Podemos propor um conjunto de diretrizes de como treinar novos SREs, mas em que devemos treiná-los? O material de treinamento dependerá das tecnologias usadas no trabalho, mas a questão mais importante é: que tipo de engenheiros estamos tentando criar? Na escala e complexidade em que os SREs operam, eles não podem se dar ao luxo de simplesmente

4 Um aceno para os videogames do passado.

ser administradores de sistema tradicionais focados em operações. Além de ter uma mentalidade de engenharia em larga escala, os SREs devem apresentar as seguintes características:

- No decorrer de seus trabalhos, eles encontrarão sistemas que nunca viram antes, portanto, precisam ter fortes habilidades de engenharia reversa.
- Em escala, haverá anomalias difíceis de detectar, portanto, elas precisarão da capacidade pensar estatisticamente, em vez de proceduralmente, para desvendar problemas.
- Quando os procedimentos operacionais padrão falham, eles precisam ser capazes de improvisar completamente.

Vamos examinar melhor esses atributos, para que possamos entender como equipar nossos SREs para essas habilidades e comportamentos.

Engenheiros reversos: descobrindo como as coisas funcionam

Os engenheiros estão curiosos sobre como funcionam os sistemas que nunca viram antes — ou, mais provavelmente, como funcionam as versões atuais dos sistemas que eles conheciam muito bem. Ao ter uma compreensão básica de como os sistemas funcionam em sua empresa, juntamente com a disposição de se aprofundar nas ferramentas de depuração, limites de RPC e logs de seus binários para descobrir seus fluxos, os SREs se tornarão mais eficientes em identificar problemas inesperados em arquiteturas de sistema inesperadas. Ensine seus SREs sobre as superfícies de diagnóstico e depuração de seus aplicativos e faça com que eles pratiquem inferências a partir das informações que essas superfícies revelam, para que tal comportamento se torne reflexivo ao lidar com interrupções futuras.

Pensadores Estatísticos e Comparativos: Administradores do Científico Método Sob Pressão

Você pode pensar na abordagem de um SRE para resposta a incidentes para sistemas de grande escala como navegar por uma enorme árvore de decisão que se desenvolve na frente deles. Na janela de tempo limitada oferecida pelas demandas de resposta a incidentes, o SRE pode realizar algumas ações entre centenas com o objetivo de mitigar a interrupção, seja no curto ou no longo prazo. Porque o tempo é muitas vezes da maior importância, o SRE tem que podar eficaz e eficientemente esta árvore de decisão. A capacidade de fazer isso é parcialmente adquirida através da experiência, que só vem com o tempo e a exposição a uma variedade de sistemas de produção. Essa experiência deve ser conjugada com a construção cuidadosa de hipóteses que, quando comprovadas ou refutadas, estreitam ainda mais esse espaço de decisão. Dito de outra forma, rastrear falhas no sistema geralmente é semelhante a jogar um jogo de “qual dessas coisas não é como a outra?” onde “coisas” podem incluir versão do kernel, arquitetura da CPU, versão(es) binária(s) em sua pilha, mix de tráfego regional ou uma centena de outros fatores. Arquitetonicamente, é responsabilidade da equipe garantir que todos esses fatores possam ser controlados e analisados e comparados individualmente. No entanto,

também devemos treinar nossos SREs mais novos para se tornarem bons analistas e comparadores desde os primeiros momentos no trabalho.

Artistas de improvisação: quando o inesperado acontece Você

tenta uma correção para a quebra, mas não funciona. O(s) desenvolvedor(es) por trás do sistema com falha não são encontrados em lugar algum. O que você faz agora? Você improvisa! Aprender várias ferramentas que podem resolver partes do seu problema permite que você pratique a defesa em profundidade em seus próprios comportamentos de resolução de problemas. Ser muito processual diante de uma interrupção, esquecendo assim suas habilidades analíticas, pode ser a diferença entre ficar preso e encontrar a causa raiz. Um caso de solução de problemas atulado pode ser agravado ainda mais quando um SRE traz muitas suposições não testadas sobre a causa de uma interrupção em sua tomada de decisão. Demonstrar que existem muitas armadilhas analíticas nas quais os SREs podem cair, que exigem “diminuir o zoom” e adotar uma abordagem diferente para a resolução, é uma lição valiosa para os SREs aprenderem desde o início.

Dados esses três atributos aspiracionais de SREs de alto desempenho, quais cursos e experiências podemos oferecer aos novos SREs para enviá-los no caminho certo? Você precisa criar seu próprio conteúdo de curso que incorpore esses atributos, além dos outros atributos específicos da sua cultura SRE. Vamos considerar uma classe que acreditamos atingir todos os pontos mencionados acima.

Unindo isso: engenharia reversa de um serviço de produção

“Quando chegou a hora de aprender [parte da pilha do Google Maps], [um novo SRE] perguntou se, em vez de passivamente ter alguém explicando o serviço, ela poderia fazer isso sozinha – aprendendo tudo por meio de técnicas de classe de engenharia reversa e tendo fazendo com que o resto de nós a corrija/preencha os espaços em branco para o que ela perdeu ou errou. O resultado? Bem, provavelmente foi mais correto e útil do que teria sido se eu tivesse dado a palestra, e estou de plantão para isso há mais de 5 anos!”

—Paul Cowan, engenheiro de confiabilidade do site do Google

Uma classe popular que oferecemos no Google é chamada de “Engenharia Reversa de um Serviço de Produção (sem a ajuda de seus proprietários)”. O cenário do problema apresentado parece simples à primeira vista. Toda a equipe do Google Notícias — SRE, engenheiros de software, gerenciamento de produtos e assim por diante — fez uma viagem da empresa: um cruzeiro pelo Triângulo das Bermudas. Não temos notícias da equipe há 30 dias, então nossos alunos são a recém-nomeada Equipe de SRE do Google Notícias. Eles precisam descobrir como a pilha de serviço funciona de ponta a ponta para comandá-la e mantê-la funcionando.

Depois de receber esse cenário, os alunos são conduzidos por exercícios interativos e orientados a propósitos, nos quais traçam o caminho de entrada da consulta do navegador da Web pela infraestrutura do Google. Em cada etapa do processo, enfatizamos que é importante aprender várias maneiras de descobrir a conectividade entre os servidores de produção, para que as conexões não sejam perdidas. No meio da aula, desafiamos

os alunos para encontrar outro ponto final para o tráfego de entrada, demonstrando que nossa suposição inicial era muito restrita. Em seguida, desafiamos nossos alunos a encontrar outras maneiras de entrar na pilha. Exploramos a natureza altamente instrumentada de nossos binários de produção, que relatam sua conectividade RPC, bem como nosso monitoramento disponível de caixa branca e caixa preta, para determinar qual(is) caminho(s) as consultas dos usuários seguem.⁵ Ao longo do forma, construímos um diagrama de sistema e também discutimos componentes que são infraestrutura compartilhada que nossos alunos provavelmente verão novamente no futuro.

No final da aula, os alunos são encarregados de uma tarefa. Cada aluno volta para sua equipe de origem e pede a um SRE sênior para ajudá-los a selecionar uma pilha ou fatia de uma pilha para a qual estarão de plantão. Usando as habilidades aprendidas nas aulas, o aluno então desenha os diagramas por conta própria e apresenta suas descobertas ao SRE sênior.

Sem dúvida, o aluno perderá alguns detalhes sutis, o que fará uma boa discussão. Também é provável que o SRE sênior também aprenda algo com o exercício, expondo desvios em sua compreensão anterior do sistema em constante mudança.

Devido à rápida mudança dos sistemas de produção, é importante que sua equipe aceite qualquer chance de se familiarizar novamente com um sistema, inclusive aprendendo com os membros mais novos, em vez dos mais antigos.

Cinco práticas para aspirantes a visitantes

Estar de plantão não é o objetivo mais importante de qualquer SRE, mas as responsabilidades da engenharia de produção geralmente envolvem algum tipo de cobertura de notificação urgente. Alguém que é capaz de atender de forma responsável é alguém que entende o sistema em que trabalha com uma profundidade e amplitude razoáveis. Portanto, usaremos "capaz de atender" como um proxy útil para "sabe o suficiente e pode descobrir o resto".

Uma fome de fracasso: lendo e compartilhando postmortems

"Aqueles que não conseguem se lembrar do passado estão condenados a repeti-lo."

—George Santayana, filósofo e ensaísta

Postmortems (veja o [Capítulo 15](#)) são uma parte importante da melhoria contínua.

Eles são uma maneira livre de culpa de chegar às muitas causas-raiz de uma interrupção significativa ou visível. Ao escrever uma autópsia, tenha em mente que seu público mais agradecido pode ser um engenheiro que ainda não foi contratado. Sem edição radical, mudanças sutis podem ser feitas em nossos melhores postmortems para torná-los "ensináveis" postmortem. tem.

⁵ Essa abordagem "sigo o RPC" também funciona bem para sistemas de lote/pipeline; comece com a operação que inicia o sistema. Para sistemas em lote, essa operação pode ser a chegada de dados que precisam ser processados, uma transação que precisa ser validada ou muitos outros eventos.

Mesmo as melhores autópsias não são úteis se ficarem no fundo de um arquivo virtual. Segue-se então que sua equipe deve coletar e organizar post-mortems valiosos para servir como recursos educacionais para futuros novatos. Alguns post-mortems são mecânicos, mas “post-mortems ensináveis” que fornecem insights sobre falhas estruturais ou novas de sistemas de grande escala são tão bons quanto ouro para novos alunos.

A propriedade dos postmortems não se limita apenas à autoria. É um motivo de orgulho para muitas equipes terem sobrevivido e documentado suas maiores interrupções. Colete seus melhores postmortems e disponibilize-os com destaque para seus novatos - além de partes interessadas de equipes relacionadas e/ou integrantes - para ler. Peça às equipes relacionadas que publiquem seus melhores postmortems onde você possa acessá-los.

Algumas equipes de SRE no Google administram “clubes de leitura postmortem”, onde postmortems fascinantes e perspicazes são divulgados, pré-lidos e depois discutidos. O(s) autor(es) original(is) da autópsia pode(m) ser o(s) convidado(s) de honra da reunião. Outras equipes organizam reuniões de “contos de fracasso” onde o(s) autor(es) postmortem apresentam-se semiformalmente, relatando a interrupção e conduzindo efetivamente a discussão.

Leituras regulares ou apresentações sobre interrupções, incluindo condições de disparo e etapas de mitigação, fazem maravilhas para construir um novo mapa mental do SRE e compreender a produção e a resposta de plantão. Postmortems também são um excelente combustível para futuros cenários abstratos de desastres.

Dramatização de Desastres

“Uma vez por semana, temos uma reunião em que uma vítima é escolhida para estar na frente do grupo e um cenário – muitas vezes real, tirado dos anais da história do Google – é lançado contra ela. A vítima, que eu considero um participante de um game show, diz ao apresentador do game show o que ele faria ou faria perguntas para entender ou resolver o problema, e o apresentador diz à vítima o que acontece com cada ação ou observação. É como SRE Zork. Você está em um labirinto de consoles de monitoramento sinuosos, todos iguais. Você deve evitar que usuários inocentes caiam no Abismo da Latência Excessiva de Consulta, salvar os datacenters do Quase Certo Meltdown e nos poupar de todo o constrangimento da Exibição Errônea do Google Doodle.”

—Robert Kennedy, ex-Engenheiro de Confiabilidade do Site da Pesquisa Google e Healthcare.gov⁶

Quando você tem um grupo de SREs com níveis de experiência muito diferentes, o que você pode fazer para reuní-los e permitir que aprendam uns com os outros? Como você impressiona a cultura SRE e a natureza de solução de problemas de sua equipe em um novato, ao mesmo tempo em que mantém veteranos grisalhos informados sobre novas mudanças e recursos em sua pilha? As equipes de SRE do Google lidam com esses desafios por meio de uma tradição consagrada pelo tempo de dramatização regular de desastres. Entre outros nomes, este exercício é comumente

6 Ver “Vida nas Trincheiras de Healthcare.gov”.

conhecido como “Roda do Infortúnio” ou “Andar na Prancha”. A sensação de perigo humorístico que esses títulos conferem ao exercício o torna menos intimidador para SREs recém-contratados.

Na melhor das hipóteses, esses exercícios se tornam um ritual semanal em que cada membro do grupo aprende alguma coisa. A fórmula é direta e tem alguma semelhança com um RPG de mesa (Role Playing Game): o “mestre do jogo” (GM) escolhe dois membros da equipe para serem os principais e os secundários de plantão; esses dois SREs se juntam ao GM na frente da sala. Uma página recebida é anunciada e a equipe de plantão responde com o que faria para mitigar e investigar a interrupção.

O GM preparou cuidadosamente um cenário que está prestes a se desenrolar. Esse cenário pode ser baseado em uma interrupção anterior para a qual os membros mais novos da equipe não estavam por perto ou que os membros mais antigos da equipe esqueceram. Ou talvez o cenário seja uma incursão em uma quebra hipotética de um recurso novo ou prestes a ser lançado na pilha, deixando todos os membros da sala igualmente despreparados para lidar com a situação. Melhor ainda, um colega de trabalho pode encontrar uma nova e inovadora ruptura na produção, e o cenário de hoje expande essa nova ameaça.

Nos próximos 30 a 60 minutos, os chamadores primários e secundários tentam identificar a causa raiz do problema. O GM felizmente fornece contexto adicional à medida que o problema se desenvola, talvez informando aos que ligam (e seu público) como os gráficos em seu painel de monitoramento podem parecer durante a interrupção. Se o incidente exigir escalada fora do time da casa, o GM finge ser um membro daquele outro time para os propósitos do cenário. Nenhum cenário virtual será perfeito, então, às vezes, o GM pode ter que orientar os participantes de volta ao caminho, redirecionando os visitantes para longe de pistas falsas, introduzindo urgência e clareza ao adicionar outros estímulos,⁷ ou fazendo perguntas urgentes e pontuais.⁸

Quando seu RPG de desastre for bem-sucedido, todos terão aprendido alguma coisa: talvez uma nova ferramenta ou truque, uma perspectiva diferente de como resolver um problema ou (especialmente gratificante para novos membros da equipe) uma validação de que você poderia ter resolvido o problema desta semana se você foi escolhido. Com alguma sorte, este exercício inspirará os companheiros de equipe a aguardar ansiosamente a aventura da próxima semana ou a pedir para se tornar o mestre do jogo na próxima semana.

Quebre coisas reais, conserte coisas reais Um

novato pode aprender muito sobre SRE lendo documentação, post-mortems e participando de treinamentos. A dramatização de desastres pode ajudar a colocar a mente de um novato no jogo. No entanto, a experiência derivada da experiência prática de quebrar e/ou consertar

7 Por exemplo: “Você está sendo chamado por outra equipe que traz mais informações. Aqui está o que eles dizer...”

8 Por exemplo: “Estamos perdendo dinheiro rapidamente! Como você poderia parar o sangramento a curto prazo?”

sistemas de produção reais é ainda melhor. Haverá bastante tempo para experiência prática quando um novato estiver de plantão, mas esse aprendizado deve acontecer antes que um novo SRE chegue a esse ponto. Portanto, forneça essas experiências práticas muito mais cedo para desenvolver as respostas reflexivas do aluno para usar as ferramentas e o monitoramento de sua empresa para abordar uma interrupção em desenvolvimento.

O realismo é primordial nessas interações. Idealmente, sua equipe tem uma pilha multihomed e provisionada de forma que você tenha pelo menos uma instância que possa desviar do tráfego ao vivo e emprestar temporariamente para um exercício de aprendizado. Alternativamente, você pode ter uma instância menor, mas ainda cheia de recursos, de teste ou de controle de qualidade de sua pilha que pode ser emprestada por um curto período de tempo. Se possível, sujeite a pilha a uma carga sintética que se aproxime do tráfego real do usuário/cliente, além do consumo de recursos, se possível.

As oportunidades de aprendizado de um sistema de produção real sob carga sintética são abundantes. SREs seniores terão enfrentado todos os tipos de problemas: configurações incorretas, vazamentos de memória, regressões de desempenho, falhas de consultas, gargalos de armazenamento e assim por diante. Nesse ambiente realista, mas relativamente livre de riscos, os inspetores podem manipular o conjunto de tarefas de maneira a alterar o comportamento da pilha, forçando novos SREs a encontrar diferenças, determinar fatores contribuintes e, finalmente, reparar sistemas para restaurar o comportamento apropriado.

Como alternativa à sobrecarga de pedir a um SRE sênior para planejar cuidadosamente um tipo específico de quebra que o(s) novo(s) SRE(s) deve(m) reparar, você também pode trabalhar na direção oposta com um exercício que também pode aumentar a participação de toda a equipe: trabalhe a partir de uma boa configuração conhecida e prejudique lentamente a pilha em gargalos selecionados, observando os esforços de upstream e downstream por meio de seu monitoramento. Este exercício é valorizado pela equipe do Google Search SRE, cuja versão deste exercício é chamada "Vamos queimar um cluster de pesquisa no chão!" O exercício prossegue da seguinte forma:

1. Em grupo, discutimos quais características de desempenho observáveis podem mudam à medida que enfraquecemos a pilha.
2. Antes de infligir o dano planejado, pesquisamos os participantes para suas suposições e raciocínio sobre suas previsões sobre como o sistema reagirá.
3. Validamos suposições e justificamos o raciocínio por trás dos comportamentos que vemos.

Este exercício, que realizamos trimestralmente, elimina novos bugs que corrigimos avidamente, porque nossos sistemas nem sempre se degradam tão graciosamente quanto esperávamos.

Documentação como Aprendizagem

Muitas equipes de SRE mantêm uma “lista de verificação de aprendizado de plantão”, que é uma lista organizada de leitura e compreensão das tecnologias e conceitos relevantes para o(s) sistema(s)

eles mantêm. Essa lista deve ser internalizada por um aluno antes que ele seja elegível para servir como sombra de plantão. Reserve um momento para revisitar o exemplo de lista de verificação de aprendizado de plantão na [Tabela 28-1](#). A lista de verificação de aprendizado serve a propósitos diferentes para pessoas diferentes:

- **Para o aluno:**

- Este documento ajuda a estabelecer os limites do sistema que sua equipe suporta.
- Ao estudar esta lista, o aluno ganha uma noção de quais sistemas são mais importantes e por quê. Quando eles entendem as informações contidas nele, eles podem passar para outros tópicos que precisam aprender, em vez de se deter em aprender detalhes esotéricos que podem ser aprendidos ao longo do tempo.

- **Para mentores e gerentes:** O progresso do aluno na lista de verificação de aprendizado pode ser observado. A lista de verificação responde a perguntas como: — Em quais seções você está trabalhando hoje?

- Quais seções são as mais confusas?

- **A todos os membros da equipe:** O doc torna-se um contrato social pelo qual (ao domínio) o aluno ingressa nas fileiras do plantão. A lista de verificação de aprendizagem define o padrão que todos os membros da equipe devem aspirar e defender.

Em um ambiente em rápida mudança, a documentação pode ficar desatualizada rapidamente.

Documentação desatualizada é um problema menor para SREs seniores que já estão atualizados, porque eles mantêm o estado do mundo e suas mudanças em suas próprias cabeças. Os SREs iniciantes precisam muito mais de documentação atualizada, mas podem não se sentir capacitados ou conhecedores o suficiente para fazer alterações. Quando projetada com a quantidade certa de estrutura, a documentação de plantão pode se tornar um corpo de trabalho adaptável que aproveita o entusiasmo dos novatos e o conhecimento sênior para manter todos atualizados.

No Search SRE, antecipamos a chegada de novos membros da equipe revisando nossa lista de verificação de aprendizado de plantão e classificando suas seções de acordo com a atualização. À medida que o novo membro da equipe chega, indicamos a eles a lista de verificação geral de aprendizado, mas também os encarregamos de revisar uma ou duas das seções mais desatualizadas. Como você pode ver na [Tabela 28-1](#), rotulamos o SRE sênior e os contatos do desenvolvedor para cada tecnologia. Incentivamos o aluno a fazer uma conexão precoce com esses especialistas no assunto, para que possam aprender diretamente o funcionamento interno da tecnologia selecionada. Mais tarde, à medida que se familiarizarem com o escopo e o tom da lista de verificação de aprendizado, espera-se que contribuam com uma seção revisada da lista de verificação de aprendizado, que deve ser revisada por um ou mais SREs seniores listados como especialistas.

Plantão de sombra cedo e frequentemente

Em última análise, nenhuma quantidade de exercícios hipotéticos de desastres ou outros mecanismos de treinamento prepararão totalmente um SRE para ficar de plantão. No final das contas, lidar com interrupções reais sempre será mais benéfico do ponto de vista do aprendizado do que se envolver com hipóteses. No entanto, é injusto fazer com que os novatos esperem até sua primeira página real para ter a chance de aprender e reter conhecimento.

Depois que o aluno tiver feito todos os fundamentos do sistema (preenchendo, por exemplo, uma lista de verificação de aprendizado de plantão), considere configurar seu sistema de alerta para copiar as páginas recebidas para o novato, a princípio apenas durante o horário comercial.

Confie na curiosidade deles para liderar o caminho. Esses turnos de plantão “sombra” são uma ótima maneira de um mentor obter visibilidade do progresso de um aluno e de um aluno obter visibilidade das responsabilidades de estar de plantão. Ao providenciar para que o novato acompanhe vários membros de sua equipe, a equipe ficará cada vez mais confortável com o pensamento dessa pessoa entrar na rotação de plantão. Instilar confiança dessa maneira é um método eficaz de construir confiança, permitindo que os membros mais seniores se desapeguem quando não estiverem de plantão, ajudando assim a evitar o esgotamento da equipe.

Quando uma página chega, o novo SRE não é o chamador designado, uma condição que elimina qualquer pressão de tempo para o aluno. Eles agora têm um assento na primeira fila para a interrupção enquanto ela se desenrola, e não depois que o problema é resolvido. Pode ser que o aluno e o principal responsável pela chamada compartilhem uma sessão de terminal ou se sentem próximos um do outro para comparar notas prontamente. Em um momento de conveniência mútua após o término da interrupção, o chamador pode revisar o raciocínio e os processos seguidos em benefício do aluno. Este exercício aumentará a retenção da sombra do chamador sobre o que realmente ocorreu.



Caso ocorra uma interrupção para a qual escrever uma autópsia seja benéfico, o chamador deve incluir o novato como coautor.

Não despeje a redação apenas no aluno, porque pode ser mal entendido que as autópsias são de alguma forma um trabalho pesado para ser passado para os mais jovens. Seria um erro criar tal impressão.

Algumas equipes também incluirão uma etapa final: fazer com que o chamador experiente “inverta” o aluno. O novato se tornará o principal de plantão e será o dono de todas as escalasções recebidas, mas o experiente de plantão se esconderá nas sombras, diagnosticando independentemente a situação sem modificar nenhum estado. O SRE experiente estará disponível para fornecer suporte ativo, ajuda, validação e dicas conforme necessário.

De plantão e além: Ritos de passagem e prática Educação continuada

À medida que a compreensão aumenta, o aluno chegará a um ponto em sua carreira em que será capaz de raciocinar confortavelmente pela maior parte da pilha e poderá improvisar seu caminho pelo resto. Neste ponto, eles devem ir de plantão para o seu serviço. Algumas equipes criam uma espécie de exame final que testa seus alunos uma última vez antes de conceder-lhes poderes e responsabilidades de plantão. Outros novos SREs apresentarão o preenchimento da lista de verificação de aprendizado de plantão como prova de que estão prontos. Independentemente de como você abre esse marco, ficar de plantão é um rito de passagem e deve ser celebrado em equipe.

A aprendizagem pára quando um aluno se junta às fileiras de plantão? Claro que não! Para permanecer vigilante como SREs, sua equipe sempre precisará estar ativa e ciente das mudanças que estão por vir. Enquanto sua atenção está em outro lugar, partes de sua pilha podem ser reprojetadas e estendidas, deixando o conhecimento operacional de sua equipe como histórico na melhor das hipóteses.

Configure uma série de aprendizado regular para toda a sua equipe, onde as visões gerais das mudanças novas e futuras em sua pilha são apresentadas como apresentações pelos SREs que estão conduzindo as mudanças, que podem apresentar juntamente com os desenvolvedores conforme necessário. Se puder, grave as apresentações para poder construir uma biblioteca de treinamento para futuros alunos.

Com alguma prática, você ganhará muito envolvimento oportuno tanto dos SREs dentro de sua equipe quanto dos desenvolvedores que trabalham em estreita colaboração com sua equipe, ao mesmo tempo em que mantém a mente de todos atualizada sobre o futuro. Existem outros locais para envolvimento educacional também: considere ter SREs dando palestras para seus colegas desenvolvedores. Quanto melhor seus colegas de desenvolvimento entenderem seu trabalho e os desafios que sua equipe enfrenta, mais fácil será tomar decisões totalmente informadas em projetos posteriores.

Considerações finais

Um investimento inicial em treinamento SRE vale a pena, tanto para os alunos ansiosos para entender seu ambiente de produção quanto para as equipes gratas por receber os alunos nas fileiras de plantão. Por meio do uso de práticas aplicáveis descritas neste capítulo, você criará SREs completos mais rapidamente, enquanto aprimora as habilidades da equipe perpetuamente. Como você aplica essas práticas depende de você, mas a cobrança é clara: como SRE, você precisa dimensionar seus humanos mais rápido do que dimensionar suas máquinas. Boa sorte para você e suas equipes na criação de uma cultura de aprendizagem e ensino!

CAPÍTULO 29

Lidando com interrupções

**Escrito por Dave O'Connor
Editado por Diane Bates**

A “carga operacional”, quando aplicada a sistemas complexos, é o trabalho que deve ser feito para manter o sistema em um estado funcional. Por exemplo, se você possui um carro, você (ou alguém que você paga) sempre acaba fazendo manutenção nele, colocando gasolina nele ou fazendo outras manutenções regulares para mantê-lo desempenhando sua função.

Qualquer sistema complexo é tão imperfeito quanto seus criadores. Ao gerenciar a carga operacional criada por esses sistemas, lembre-se de que seus criadores também são máquinas imperfeitas.

A carga operacional, quando aplicada ao gerenciamento de sistemas complexos, assume muitas formas, algumas mais óbvias que outras. A terminologia pode mudar, mas a carga operacional se enquadra em três categorias gerais: páginas, tickets e atividades operacionais contínuas.

As páginas dizem respeito a alertas de produção e suas consequências, e são acionadas em resposta a emergências de produção. Às vezes podem ser monótonos e recorrentes, exigindo pouca reflexão. Eles também podem ser envolventes e envolver pensamento tático em profundidade.

As páginas sempre têm um tempo de resposta esperado (SLO), que às vezes é medido em minutos.

Os tickets dizem respeito a solicitações de clientes que exigem que você execute uma ação. Assim como as páginas, os ingressos podem ser simples e chatos, ou exigir uma reflexão real. Um ticket simples pode solicitar uma revisão de código para uma configuração que a equipe possui. Um ticket mais complexo pode envolver uma solicitação especial ou incomum de ajuda com um projeto ou plano de capacidade. Os tickets também podem ter um SLO, mas o tempo de resposta é mais provavelmente medido em horas, dias ou semanas.

As responsabilidades operacionais contínuas (também conhecidas como “chutar a lata no caminho” e “labutar”; consulte o [Capítulo 5](#)) incluem atividades como código de propriedade da equipe ou lançamentos de sinalizadores, ou respostas a perguntas pontuais e urgentes de clientes. Embora possam não ter um SLO definido, essas tarefas podem interromper você.

Alguns tipos de carga operacional são facilmente previstos ou planejados, mas grande parte da carga não é planejada ou pode interromper alguém em um momento não específico, exigindo que essa pessoa determine se o problema pode esperar.

Gerenciando a carga operacional

O Google tem vários métodos de gerenciamento de cada tipo de carga operacional no nível da equipe.

As páginas são mais comumente gerenciadas por um engenheiro de plantão primário dedicado. Esta é uma única pessoa que responde às páginas e gerencia os incidentes ou interrupções resultantes.

O engenheiro de plantão principal também pode gerenciar comunicações de suporte ao usuário, encaminhamento para desenvolvedores de produtos e assim por diante. Para minimizar a interrupção que uma página causa a uma equipe e evitar o efeito espectador, os turnos de plantão do Google são gerenciados por um único engenheiro. O engenheiro de plantão pode encaminhar as páginas para outro membro da equipe se um problema não for bem compreendido.

Normalmente, um engenheiro de plantão secundário atua como um backup para o primário. As funções do engenheiro secundário variam. Em algumas rotações, o único dever do secundário é entrar em contato com o primário se as páginas falharem. Neste caso, o secundário pode estar em outra equipe. O engenheiro secundário pode ou não considerar-se em interrupções, dependendo das funções.

Os tickets são gerenciados de algumas maneiras diferentes, dependendo da equipe SRE: um engenheiro de plantão principal pode trabalhar em tickets enquanto estiver de plantão, um engenheiro secundário pode trabalhar em tickets enquanto estiver de plantão ou uma equipe pode ter um ticket dedicado pessoa que não está de plantão. Os tíquetes podem ser distribuídos automaticamente de forma aleatória entre os membros da equipe, ou espera-se que os membros da equipe atendam tíquetes ad hoc.

As responsabilidades operacionais contínuas também são gerenciadas de várias maneiras. Às vezes, o engenheiro de plantão faz o trabalho (empurrar, virar a bandeira, etc.). Alternativamente, cada responsabilidade pode ser atribuída a membros da equipe ad hoc, ou um engenheiro de plantão pode assumir uma responsabilidade duradoura (ou seja, uma distribuição ou tíquete de várias semanas) que dure além de sua semana de turno.

Fatores para determinar como as interrupções são tratadas

Para dar um passo atrás na mecânica de como a carga operacional é gerenciada, há uma série de métricas que determinam como cada uma dessas interrupções é tratada. Algumas equipes de SRE do Google consideraram as seguintes métricas úteis para decidir como gerenciar interrupções:

- SLO de interrupção ou tempo de resposta esperado •

O número de interrupções que geralmente estão em atraso

- A gravidade das interrupções • A frequência das interrupções • O número de pessoas disponíveis para lidar com um certo tipo de interrupção (por exemplo, algumas equipes exigem uma certa quantidade de trabalho de bilhete antes de ir de plantão)

Você pode notar que todas essas métricas são adequadas para atender ao menor tempo de resposta possível, sem considerar mais custos humanos. Tentar fazer um balanço do custo humano e de produtividade é difícil.

Máquinas imperfeitas

Os humanos são máquinas imperfeitas. Eles ficam entediados, têm processadores (e às vezes interfaces de usuário) que não são muito bem compreendidos e não são muito eficientes. Reconhecer o elemento humano como “trabalhando como pretendido” e tentar contornar ou melhorar a forma como os humanos trabalham poderia preencher um espaço muito maior do que o fornecido aqui; no momento, algumas idéias básicas podem ser úteis para determinar como as interrupções devem funcionar.

Estado de Fluxo Cognitivo O

conceito de estado de fluxo¹ é amplamente aceito e pode ser empiricamente reconhecido por praticamente todos que trabalham em Engenharia de Software, Sysadmin, SRE ou na maioria das outras disciplinas que exigem períodos de concentração focados. Estar na “zona” pode aumentar a produtividade, mas também pode aumentar a criatividade artística e científica. Alcançar esse estado encoraja as pessoas a realmente dominar e melhorar a tarefa ou projeto em que estão trabalhando. Ser interrompido pode chutá-lo para fora desse estado, se a interrupção for suficientemente perturbadora. Você deseja maximizar a quantidade de tempo gasto neste Estado.

O fluxo cognitivo também pode ser aplicado a atividades menos criativas onde o nível de habilidade exigido é menor e os elementos essenciais do fluxo ainda são cumpridos (objetivos claros, feedback imediato, senso de controle e distorção de tempo associada); exemplos incluem trabalho doméstico ou condução.

Você pode entrar na zona trabalhando em problemas de baixa habilidade e baixa dificuldade, como jogar um videogame repetitivo. Você pode facilmente chegar lá resolvendo problemas de alta habilidade e alta dificuldade, como aqueles que um engenheiro pode enfrentar. Os métodos para chegar a um estado de fluxo cognitivo diferem, mas o resultado é essencialmente o mesmo.

1 Veja Wikipedia: Flow (psicologia), [http://en.wikipedia.org/wiki/Flow_\(psicologia\)](http://en.wikipedia.org/wiki/Flow_(psicologia)).

Estado de fluxo cognitivo: Criativo e

engajado Esta é a zona: alguém trabalha em um problema por um tempo, está ciente e confortável com os parâmetros do problema e sente que pode corrigi-lo ou resolvê-lo. A pessoa trabalha intensamente no problema, perdendo a noção do tempo e ignorando as interrupções o máximo possível. Maximizar a quantidade de tempo que uma pessoa pode passar nesse estado é muito desejável – ela produzirá resultados criativos e fará um bom trabalho por volume. Eles serão mais felizes no trabalho que estão fazendo.

Infelizmente, muitas pessoas em funções do tipo SRE gastam muito do seu tempo tentando e falhando em entrar nesse modo e ficando frustradas quando não conseguem, ou nunca tentando chegar a esse modo, em vez disso, definhando no estado interrompido.

Estado de fluxo cognitivo:

Angry Birds As pessoas gostam de realizar tarefas que sabem fazer. Na verdade, executar essas tarefas é um dos caminhos mais claros para o fluxo cognitivo. Alguns SREs estão de plantão quando atingem um estado de fluxo cognitivo. Pode ser muito gratificante perseguir as causas dos problemas, trabalhar com outras pessoas e melhorar a saúde geral do sistema de maneira tão tangível. Por outro lado, para a maioria dos engenheiros de plantão estressados, o estresse é causado pelo volume do pager ou porque eles estão tratando o plantão como uma interrupção. Eles estão tentando codificar ou trabalhar em projetos ao mesmo tempo em que estão de plantão ou em interrupções em tempo integral. Esses engenheiros existem em um estado de interrupção constante, ou interruptibilidade. Este ambiente de trabalho é extremamente estressante.

Por outro lado, quando uma pessoa está se concentrando em tempo integral nas interrupções, as interrupções deixam de ser interrupções. Em um nível muito visceral, fazer melhorias incrementais no sistema, eliminar tíquetes e corrigir problemas e interrupções se torna um conjunto claro de metas, limites e feedback claro: você fecha X bugs ou para de ser chamado. Tudo o que resta são distrações. Quando você está fazendo interrupções, seus projetos são uma distração.

Embora as interrupções possam ser um uso satisfatório do tempo no curto prazo, em um ambiente misto de projeto/de plantão, as pessoas ficam mais felizes com o equilíbrio entre esses dois tipos de trabalho. O equilíbrio ideal varia de engenheiro para engenheiro. É importante estar ciente de que alguns engenheiros podem não saber realmente qual é o equilíbrio que melhor os motiva (ou podem pensar que sabem, mas você pode discordar).

Faça uma coisa bem

Você pode estar se perguntando sobre as implicações práticas do que você leu até agora.

As sugestões a seguir, baseadas no que funcionou para várias equipes de SRE que gerenciei no Google, são principalmente para o benefício de gerentes de equipe ou influenciadores. Este documento é agnóstico aos hábitos pessoais - as pessoas são livres para gerenciar seu próprio tempo como acharem melhor. A concentração aqui está em direcionar a estrutura de como a própria equipe gerencia as interrupções, para que as pessoas não sejam preparadas para o fracasso por causa da função da equipe ou estrutura.

Distração

As maneiras pelas quais um engenheiro pode ser distraído e, portanto, impedido de atingir um estado de fluxo cognitivo são inúmeras. Por exemplo: considere um SRE aleatório chamado Fred. Fred chega ao trabalho na segunda-feira de manhã. Fred não está de plantão ou sendo interrompido hoje, então Fred claramente gostaria de trabalhar em seus projetos. Ele pega um café, coloca seus fones de ouvido “não perturbe” e se senta em sua mesa. Zona de tempo, certo?

Exceto, a qualquer momento, qualquer uma das seguintes coisas pode acontecer:

- A equipe de Fred usa um sistema de tíquetes automatizado para atribuir tíquetes aleatoriamente ao equipe. Um bilhete é atribuído a ele, com vencimento hoje.
- O colega de Fred está de plantão e recebe uma página sobre um componente que Fred está especialista, e o interrompe para perguntar sobre isso.
- Um usuário do serviço de Fred aumenta a prioridade de um ticket que lhe foi atribuído desde a semana passada, quando estava de plantão.
- Um lançamento de sinalizador que está sendo implementado ao longo de 3 a 4 semanas e é atribuído a Fred dá errado, forçando Fred a largar tudo para examinar o lançamento, reverter a mudança e assim por diante.
- Um usuário do serviço de Fred entra em contato com Fred para fazer uma pergunta, porque Fred é tão cap útil.
- E assim por diante.

O resultado final é que, embora Fred tenha o dia inteiro livre para trabalhar em projetos, ele permanece extremamente distraído. Algumas dessas distrações ele pode gerenciar fechando e-mails, desligando mensagens instantâneas ou tomando outras medidas semelhantes. Algumas distrações são causadas por políticas ou por suposições sobre interrupções e responsabilidades contínuas.

Você pode afirmar que algum nível de distração é inevitável e intencional. Essa suposição está correta: as pessoas se apegam a bugs para os quais são o contato principal e também acumulam outras responsabilidades e obrigações. No entanto, existem maneiras de uma equipe gerenciar a resposta de interrupção para que mais pessoas (em média) possam entrar no trabalho pela manhã e se sentirem imperturbáveis.

Polarizando o

tempo Para limitar sua distração, você deve tentar minimizar as trocas de contexto.

Algumas interrupções são inevitáveis. No entanto, ver um engenheiro como uma unidade de trabalho que pode ser interrompida, cujas trocas de contexto são livres, não é ideal se você deseja que as pessoas sejam felizes e produtivas. Atribua um custo às trocas de contexto. Uma interrupção de 20 minutos durante o trabalho em um projeto envolve duas trocas de contexto; realisticamente, essa interrupção resulta na perda de algumas horas de trabalho realmente produtivo. Para evitar a ocorrência constante

rências de perda de produtividade, procure um tempo polarizado entre os estilos de trabalho, com cada período de trabalho durando o maior tempo possível. Idealmente, esse período de tempo é de uma semana, mas um dia ou até meio dia pode ser mais prático. Esta estratégia enquadra-se também no conceito complementar de make time [Gra09].

Polarizar o tempo significa que, quando uma pessoa chega ao trabalho todos os dias, ela deve saber se está apenas fazendo um trabalho de projeto ou apenas interrompendo. Polarizar seu tempo dessa maneira significa que eles se concentram por períodos mais longos na tarefa em questão.

Eles não ficam estressados porque estão sendo amarrados a tarefas que os afastam do trabalho que deveriam estar fazendo.

Sério, diga-me o que fazer Se o modelo

geral apresentado neste capítulo não funcionar para você, aqui estão algumas sugestões específicas de componentes que você pode implementar aos poucos.

Sugestões gerais

Para qualquer classe de interrupção, se o volume de interrupções for muito alto para uma pessoa, adicione outra pessoa. Esse conceito obviamente se aplica a tickets, mas pode se aplicar potencialmente a páginas também – o plantão pode começar a transferir coisas para suas páginas secundárias ou fazer downgrade para tickets.

De plantão

O engenheiro de plantão primário deve se concentrar apenas no trabalho de plantão. Se o pager estiver quieto para o seu serviço, tíquetes ou outros trabalhos baseados em interrupções que podem ser abandonados rapidamente devem fazer parte das tarefas de plantão. Quando um engenheiro está de plantão por uma semana, essa semana deve ser baixada no que diz respeito ao trabalho do projeto. Se um projeto é importante demais para ser deixado de lado por uma semana, essa pessoa não deve estar de plantão. Escalar para designar outra pessoa para o turno de plantão. Nunca se deve esperar que uma pessoa esteja de plantão e também progride em projetos (ou qualquer outra coisa com um alto custo de mudança de contexto).

Os deveres secundários dependem de quão onerosos são esses deveres. Se a função do secundário é fazer backup do primário no caso de uma falha, então talvez você possa assumir com segurança que o secundário também pode realizar o trabalho do projeto. Se alguém que não seja o secundário for designado para lidar com tickets, considere mesclar as funções. Se espera-se que o secundário realmente ajude o primário no caso de alto volume de pager, ele também deve interromper o trabalho.

(Além disso: você nunca fica sem trabalho de limpeza. Sua contagem de tickets pode estar em zero, mas sempre há documentação que precisa ser atualizada, configurações que precisam de limpeza etc. Seus futuros engenheiros de plantão agradecerão, e isso significa que é menos provável que interrompam você durante seu precioso tempo de trabalho).

Ingressos

Se você atualmente atribui tiquetes aleatoriamente às vítimas de sua equipe, pare. Fazer isso é extremamente desrespeitoso com o tempo de sua equipe e funciona completamente contra o princípio de não ser o mais interrompível possível.

Os tiquetes devem ser uma função de tempo integral, por um período de tempo que seja gerenciável para uma pessoa. Se você estiver na posição nada invejável de ter mais tiquetes do que podem ser fechados pelos engenheiros de plantão primário e secundário combinados, estruture sua rotação de tiquetes para ter duas pessoas lidando com tiquetes a qualquer momento. Não espalhe a carga por toda a equipe. As pessoas não são máquinas, e você está apenas causando mudanças de contexto que afetam o valioso tempo de fluxo.

Responsabilidades

contínuas Tanto quanto possível, defina funções que permitam que qualquer pessoa da equipe assuma o papel. Se houver um procedimento bem definido para executar e verificar pushes ou flag flips, então não há razão para uma pessoa ter que pastorear essa mudança por toda a sua vida, mesmo depois de deixar de estar de plantão ou em interrupções. Defina uma função de gerente de push que possa fazer malabarismos com push durante seu tempo de plantão ou em interrupções. Formalize o processo de transferência — é um preço pequeno a pagar por um tempo ininterrupto para as pessoas que não estão de plantão.

Seja interrompido ou não

seja Às vezes, quando uma pessoa não está em interrupções, a equipe recebe uma interrupção que a pessoa está qualificada para lidar. Embora idealmente esse cenário nunca deva acontecer, às vezes acontece. Você deve trabalhar para tornar essas ocorrências raras.

Às vezes, as pessoas trabalham em tiquetes quando não são designadas para lidar com tiquetes porque é uma maneira fácil de parecer ocupada. Tal comportamento não é útil. Isso significa que a pessoa é menos eficaz do que deveria ser. Eles distorcem os números em termos de quão gerenciável é a carga de tickets. Se uma pessoa for designada para os tickets, mas duas ou três outras pessoas também tentarem a fila de tickets, você ainda poderá ter uma fila de tickets incontrolável, mesmo que não perceba.

Reduzindo interrupções

A carga de interrupção de sua equipe pode ser incontrolável se exigir que muitos membros da equipe executem interrupções simultaneamente em um determinado momento. Há uma série de técnicas que você pode usar para reduzir a carga geral de tickets.

Na verdade, analise os

tiquetes Muitas rotações de tiquetes ou de plantão funcionam como uma luva. Isso é especialmente verdadeiro para rotações em equipes maiores. Se você estiver apenas em interrupções a cada dois meses,

é fácil enfrentar o desafio,² dar um suspiro de alívio e depois retornar às suas obrigações normais. Seu sucessor faz o mesmo, e as causas-raiz dos tickets nunca são investigadas. Em vez de avançar, sua equipe fica atolada por uma sucessão de pessoas se irritando com os mesmos problemas.

Deve haver uma entrega de bilhetes, bem como para o trabalho de plantão. Um processo de handoff mantém o estado compartilhado entre os manipuladores de tickets à medida que a responsabilidade muda. Mesmo alguma introspecção básica nas causas das interrupções pode fornecer boas soluções para reduzir a taxa geral. Muitas equipes realizam transferências de plantão e revisões de página. Pouquíssimas equipes fazem o mesmo com os ingressos.

Sua equipe deve realizar uma limpeza regular de tickets e páginas, na qual você examina classes de interrupções para ver se consegue identificar uma causa raiz. Se você acha que a causa raiz pode ser corrigida em um período de tempo razoável, silencie as interrupções até que se espere que a causa raiz seja corrigida. Fazer isso fornece alívio para a pessoa que está lidando com interrupções e cria uma aplicação de prazo útil para a pessoa que está corrigindo a raiz causa.

Respeite a si mesmo e a seus clientes Essa

máxima se aplica mais às interrupções do usuário do que às interrupções automatizadas, embora os princípios se apliquem a ambos os cenários. Se os tíquetes forem particularmente irritantes ou onerosos para resolver, você pode usar a política de forma eficaz para mitigar a carga.

Lembrar:

- Sua equipe define o nível de serviço prestado pelo seu serviço. • Não há problema em empurrar parte do esforço para seus clientes.

Se sua equipe for responsável por lidar com tíquetes ou interrupções para clientes, muitas vezes você poderá usar a política para tornar sua carga de trabalho mais gerenciável. Uma correção de política pode ser temporária ou permanente, dependendo do que faz sentido. Essa correção deve encontrar um bom equilíbrio entre respeito pelo cliente e respeito por si mesmo. A política pode ser uma ferramenta tão poderosa quanto o código.

Por exemplo, se você oferece suporte a uma ferramenta particularmente esquisita que não tem muitos recursos de desenvolvedor e um pequeno número de clientes carentes a usa, considere outras opções. Pense no valor do tempo que você gasta fazendo interrupções para este sistema e se você está gastando esse tempo com sabedoria. Em algum momento, se você não conseguir a atenção necessária para corrigir a causa raiz dos problemas que causam interrupções, talvez o componente que você está dando suporte não seja tão importante. Você deve considerar devolver o pager, desaprovando-o, substituindo-o ou outra estratégia nesse sentido que possa fazer sentido.

2 Consulte http://en.wikipedia.org/wiki/Running_the_gauntlet.

Se houver etapas específicas para uma interrupção que consomem tempo ou são complicadas, mas não exigem que seus privilégios sejam realizadas, considere usar a política para enviar a solicitação de volta ao solicitante. Por exemplo, se as pessoas precisarem doar recursos de computação, prepare uma alteração de código ou configuração ou alguma etapa semelhante e, em seguida, instrua o cliente a executar essa etapa e enviá-la para sua revisão. Lembre-se de que, se o cliente deseja que determinada tarefa seja realizada, ele deve estar preparado para despender algum esforço para conseguir o que deseja.

Uma ressalva para as soluções anteriores é que você precisa encontrar um equilíbrio entre o respeito pelo cliente e por si mesmo. Seu princípio orientador na construção de uma estratégia para lidar com as solicitações dos clientes é que a solicitação seja significativa, racional e forneça todas as informações e todo o trabalho necessário para atender à solicitação. Em troca, sua resposta deve ser útil e oportuna.

CAPÍTULO 30

Incorporando um SRE para recuperar Sobrecarga operacional

Escrito por Randall Bosetti
Editado por Diane Bates

É política padrão para as equipes de SRE do Google dividir uniformemente seu tempo entre projetos e trabalho de operações reativas. Na prática, esse equilíbrio pode ser perturbado por meses a fio por um aumento no volume diário de bilhetes. Uma quantidade onerosa de trabalho operacional é especialmente perigosa porque a equipe de SRE pode ficar esgotada ou ser incapaz de progredir no trabalho do projeto. Quando uma equipe deve alocar uma quantidade desproporcional de tempo para resolver tíquetes ao custo de gastar tempo melhorando o serviço, a escalabilidade e a confiabilidade sofrem.

Uma maneira de aliviar esse fardo é transferir temporariamente um SRE para a equipe sobrecarregada. Uma vez integrado em uma equipe, o SRE se concentra em melhorar as práticas da equipe em vez de simplesmente ajudar a equipe a esvaziar a fila de tickets. O SRE observa a rotina diária da equipe e faz recomendações para melhorar suas práticas. Esta consulta dá à equipe uma nova perspectiva sobre suas rotinas que os membros da equipe não podem fornecer por si mesmos.

Ao usar esta abordagem, não é necessário transferir mais de um engenheiro. Dois SREs não necessariamente produzem melhores resultados e podem causar problemas se a equipe reagir defensivamente.

Se você está iniciando sua primeira equipe de SRE, a abordagem descrita neste capítulo o ajudará a evitar se transformar em uma equipe de operação focada exclusivamente na rotação de tickets. Se você decidir incorporar a si mesmo ou a um de seus relatórios em uma equipe, reserve um tempo para revisar as práticas e a filosofia de SRE na [introdução de Ben Treynor Sloss](#) e no material sobre monitoramento no [Capítulo 6](#).

As seções a seguir fornecem orientação ao SRE que será incorporado a uma equipe.

Fase 1: conheça o serviço e obtenha o contexto

Seu trabalho enquanto integrado à equipe é articular por que processos e hábitos contribuem ou prejudicam a escalabilidade do serviço. Lembre à equipe que mais tickets não devem exigir mais SREs: o objetivo do modelo SRE é apenas introduzir mais humanos à medida que mais complexidade é adicionada ao sistema. Em vez disso, tente chamar a atenção para como hábitos de trabalho saudáveis reduzem o tempo gasto em tickets. Fazer isso é tão importante quanto apontar oportunidades perdidas de automação ou simplificação do serviço.

Modo de operações versus escala não linear

termo modo de operações refere-se a um determinado método de manter um serviço em execução. Vários itens de trabalho aumentam com o tamanho do serviço. Por exemplo, um serviço precisa de uma maneira de aumentar o número de máquinas virtuais (VMs) configuradas à medida que cresce. Uma equipe no modo de operações responde com um número maior de administradores gerenciando essas VMs. O SRE, em vez disso, concentra-se em escrever software ou eliminar preocupações de escalabilidade para que o número de pessoas necessárias para executar um serviço não aumente em função da carga no serviço.

As equipes que entram no modo de operações podem estar convencidas de que a escala não importa para elas ("meu serviço é pequeno"). Acompanhe uma sessão de plantão para determinar se a avaliação é verdadeira, porque o elemento de escala afeta sua estratégia.

Se o serviço principal for importante para o negócio, mas na verdade for pequeno (com poucos recursos ou baixa complexidade), coloque mais foco nas maneiras pelas quais a abordagem atual da equipe os impede de melhorar a confiabilidade do serviço. Lembre-se de que seu trabalho é fazer o serviço funcionar, não proteger a equipe de desenvolvimento de alertas.

Por outro lado, se o serviço está apenas começando, concentre-se em maneiras de preparar a equipe para um crescimento explosivo. Um serviço de 100 solicitações/segundo pode se transformar em um serviço de 10 mil solicitações/segundo em um ano.

Identifique as maiores fontes de estresse As equipes

de SRE às vezes caem no modo operacional porque se concentram em como lidar rapidamente com emergências, em vez de como reduzir o número de emergências. Um padrão para o modo de operações geralmente acontece em resposta a uma pressão esmagadora, real ou imaginária. Depois de aprender o suficiente sobre o serviço para fazer perguntas difíceis sobre seu design e implantação, passe algum tempo priorizando várias interrupções de serviço de acordo com seu impacto nos níveis de estresse da equipe. Tenha em mente que, devido à perspectiva e ao histórico da equipe, alguns problemas ou interrupções muito pequenas podem produzir uma quantidade excessiva de estresse.

Identifique o Kindling

Depois de identificar os maiores problemas existentes de uma equipe, passe para as emergências esperando para acontecer. Às vezes, emergências iminentes vêm na forma de um novo subsistema que não foi projetado para ser autogerenciado. Outras fontes incluem:

Lacunas de

conhecimento Em grandes equipes, as pessoas podem se especializar demais sem consequências imediatas.

Quando uma pessoa se especializa, ela corre o risco de não ter o amplo conhecimento necessário para realizar o suporte de plantão ou permitir que os membros da equipe ignorem as partes críticas do sistema que possuem.

Serviços desenvolvidos pelo SRE que estão aumentando discretamente em importância

Esses serviços geralmente não recebem a mesma atenção cuidadosa que os lançamentos de novos recursos porque são menores em escala e implicitamente endossados por pelo menos um SRE.

Forte dependência da “próxima grande coisa”

As pessoas podem ignorar os problemas por meses a fio porque acreditam que a nova solução que está no horizonte evita correções temporárias.

Alertas comuns que não são diagnosticados pela equipe de desenvolvimento ou pelos

SREs Esses alertas são frequentemente classificados como transitórios, mas ainda distraem seus colegas de equipe para corrigir problemas reais. Investigue esses alertas completamente ou corrija as regras de alerta.

Qualquer serviço que seja objeto de reclamações de seus clientes e careça de um SLI/SLO/SLA

Consulte o [Capítulo 4](#) para obter uma discussão sobre SLIs, SLOs e SLAs.

Qualquer serviço com um plano de capacidade que seja efetivamente “Adicione mais servidores: nossos servidores estavam ficando sem memória ontem à noite”

Os planos de capacidade devem ser suficientemente voltados para o futuro. Se o modelo do seu sistema prevê que os servidores precisam de 2 GB, um teste de carga que passa no curto prazo (revelando 1,99 GB em sua última execução) não significa necessariamente que a capacidade do seu sistema está em condições adequadas.

Postmortems que têm apenas itens de ação para reverter as alterações específicas que causaram uma interrupção. Por exemplo, “Alterar o tempo limite de streaming de volta para 60 segundos”, em vez de “Descobrir por que às vezes leva 60 segundos para buscar o primeiro megabyte do nosso vídeo promocional.”

Qualquer componente de serviço crítico para o qual os SREs existentes respondam a perguntas dizendo: “Não sabemos nada sobre isso; os devs são os donos”

Para fornecer suporte de plantão aceitável para um componente, você deve pelo menos saber as consequências quando ele quebrar e a urgência necessária para corrigir problemas.

Fase 2: Contexto de Compartilhamento

Depois de definir o escopo da dinâmica e dos pontos problemáticos da equipe, estabeleça as bases para a melhoria por meio de práticas recomendadas, como autópsias, identificando fontes de trabalho e como melhor abordá-las.

Escreva um bom post mortem para a equipe

Postmortems oferecem muitas informações sobre o raciocínio coletivo de uma equipe. Postmortems conduzidos por equipes insalubres são muitas vezes ineficazes. Alguns membros da equipe podem considerar as autópsias punitivas ou até mesmo inúteis. Embora você possa ficar tentado a revisar os arquivos post-mortem e deixar comentários para melhorias, isso não ajuda a equipe. Em vez disso, o exercício pode colocar a equipe na defensiva.

Em vez de tentar corrigir erros anteriores, tome posse do próximo postmortem. Haverá uma interrupção enquanto você estiver incorporado. Se você não for a pessoa de plantão, junte-se ao SRE de plantão para escrever uma autópsia excelente e sem culpa. Este documento é uma oportunidade de demonstrar como uma mudança em direção ao modelo SRE beneficia a equipe, tornando as correções de bugs mais permanentes. Correções de bugs mais permanentes reduzem o impacto das interrupções no tempo dos membros da equipe.

Conforme mencionado, você pode encontrar respostas como "Por que eu?" Essa resposta é especialmente provável quando uma equipe acredita que o processo pós-morte é uma retaliação. Essa atitude vem da adesão à Teoria da Maçã Ruim: o sistema está funcionando bem e, se nos livrarmos de todas as maçãs podres e seus erros, o sistema continuará funcionando bem. A Teoria da Maçã Ruim é comprovadamente falsa, como mostram evidências [Dek14] de várias disciplinas, incluindo segurança aérea. Você deve apontar essa falsidade. A frase mais eficaz para uma autópsia é dizer: "Erros são inevitáveis em qualquer sistema com múltiplas interações sutis. Você estava de plantão e confio em você para tomar as decisões certas com as informações certas. Eu gostaria que você escrevesse o que estava pensando em cada momento, para que possamos descobrir onde o sistema o enganou e onde as demandas cognitivas foram muito altas."

Classifique os incêndios de acordo com o tipo

Existem dois tipos de incêndios neste modelo simplificado por conveniência:

- Alguns incêndios não deveriam existir. Eles causam o que é comumente chamado de trabalho ou labuta (ver [Capítulo 5](#)).
- Outros incêndios que causam estresse e/ou digitação furiosa são, na verdade, parte do trabalho. Em ambos os casos, a equipe precisa construir ferramentas para controlar a queima.

Classifique os fogos da equipe em labuta e não labuta. Quando terminar, apresente a lista à equipe e explique claramente por que cada incêndio é um trabalho que deve ser automatizado ou uma sobrecarga aceitável para a execução do serviço.

Fase 3: Impulsionando a Mudança

A saúde da equipe é um processo. Como tal, não é algo que você possa resolver com esforço heróico. Para garantir que a equipe possa se autorregular, você pode ajudá-los a construir um bom modelo mental para um engajamento SRE ideal.



Os seres humanos são muito bons em homeostase, então concentre-se em criar (ou restaurar) as condições iniciais corretas e ensinar o pequeno conjunto de princípios necessários para fazer escolhas saudáveis.

Comece com o básico

As equipes que lutam com a distinção entre o SRE e o modelo de operações tradicional geralmente são incapazes de articular por que certos aspectos do código, dos processos ou da cultura da equipe as incomodam. Em vez de tentar abordar cada uma dessas questões ponto a ponto, trabalhe a partir dos princípios descritos nos Capítulos 1 e 6.

Seu primeiro objetivo para a equipe deve ser escrever um objetivo de nível de serviço (SLO), se ainda não existir. O SLO é importante porque fornece uma medida quantitativa do impacto das interrupções, além da importância de uma mudança de processo. Um SLO é provavelmente a alavanca mais importante para mover uma equipe do trabalho de operações reativas para um foco SRE saudável e de longo prazo. Se este acordo estiver faltando, nenhum outro conselho neste capítulo será útil. Se você estiver em uma equipe sem SLOs, leia primeiro o [Capítulo 4](#), depois coloque os líderes técnicos e o gerenciamento em uma sala e comece a arbitrar.

Obtenha ajuda para limpar o Kindling

Você pode ter um forte desejo de simplesmente corrigir os problemas que identificar. Por favor, resista ao desejo de corrigir esses problemas você mesmo, porque isso reforça a ideia de que “fazer mudanças é para outras pessoas”. Em vez disso, siga os seguintes passos:

1. Encontre um trabalho útil que possa ser realizado por um membro da equipe.
2. Explicar claramente como este trabalho aborda um problema da autópsia de forma permanente.
Mesmo equipes saudáveis podem produzir itens de ação míopes.
3. Servir como revisor para as mudanças de código e revisões de documentos.
4. Repita para duas ou três edições.

Ao identificar um problema adicional, coloque-o em um relatório de bug ou em um documento para a equipe consultar. Fazer isso serve ao duplo propósito de distribuir informações e encorajar os membros da equipe a escrever documentos (que muitas vezes são as primeiras vítimas da pressão do prazo). Sempre explique seu raciocínio e enfatize que uma boa documentação garante que a equipe não repita erros antigos em um contexto ligeiramente novo.

Explique seu raciocínio À medida

que a equipe recupera seu impulso e comprehende os fundamentos das mudanças sugeridas, avance para enfrentar as decisões cotidianas que originalmente levaram à sobrecarga de operações. Prepare-se para este empreendimento ser desafiado. Se você tiver sorte, o desafio será do tipo "Explique por quê. Agora mesmo. No meio da reunião semanal de produção."

Se você não tiver sorte, ninguém exige uma explicação. Evite esse problema inteiramente explicando todas as suas decisões, quer alguém solicite ou não uma explicação. Consulte as noções básicas que ressaltam suas sugestões. Fazer isso ajuda a construir o modelo mental da equipe. Depois que você sair, a equipe deverá ser capaz de prever qual seria seu comentário sobre um design ou lista de alterações. Se você não explicar seu raciocínio, ou o fizer mal, existe o risco de que a equipe simplesmente emule esse comportamento indiferente, então seja explícito.

Exemplos de uma explicação completa da sua decisão:

- "Não estou adiando a versão mais recente porque os testes são ruins. estou empurrando de volta porque o orçamento de erro que definimos para lançamentos está esgotado."
- "Os lançamentos precisam ser seguros contra reversão porque nosso SLO é rígido. O cumprimento desse SLO exige que o tempo médio de recuperação seja pequeno, portanto, o diagnóstico aprofundado antes de uma reversão não é realista."

Exemplos de uma explicação insuficiente da sua decisão:

- "Eu não acho que ter cada servidor gerando sua configuração de roteamento é seguro, porque nós não pode ver."

Esta decisão provavelmente está correta, mas o raciocínio é pobre (ou mal explicado).

A equipe não pode ler sua mente, então eles provavelmente podem imitar o raciocínio ruim observado. Em vez disso, tente "[...] não é seguro porque um bug nesse código pode causar uma falha correlacionada em todo o serviço e o código adicional é uma fonte de bugs que podem retardar uma reversão".

- "A automação deve desistir se encontrar uma implantação conflitante."

Como no exemplo anterior, esta explicação provavelmente está correta, mas insuficiente.

Em vez disso, tente "[...] porque estamos fazendo a suposição simplificadora de que todas as mudanças

passar pela automação, e algo claramente violou essa regra. Se isso acontecer com frequência, devemos identificar e remover fontes de mudança desorganizada".

Faça perguntas principais As

perguntas principais não são perguntas carregadas. Ao conversar com a equipe SRE, tente fazer perguntas de uma forma que encoraje as pessoas a pensar sobre os princípios básicos. É particularmente valioso para você modelar esse comportamento porque, por definição, uma equipe no modo de operações rejeita esse tipo de raciocínio de seus próprios constituintes. Depois de passar algum tempo explicando seu raciocínio para várias questões de política, essa prática reforça o entendimento da equipe sobre a filosofia SRE.

Exemplos de perguntas principais:

- "Vejo que o alerta TaskFailures é acionado com frequência, mas os engenheiros de plantão geralmente não fazem nada para responder ao alerta. Como isso afeta o SLO?" • "Este procedimento de ativação parece bastante complicado. Você sabe por que existem tantos arquivos de configuração para atualizar ao criar uma nova instância do serviço?"

Contra-exemplos de perguntas principais:

- "O que há com todos esses lançamentos antigos e parados?" • "Por que o Frobnitzer faz tantas coisas?"

Conclusão

Seguir os princípios descritos neste capítulo fornece a uma equipe SRE o seguinte:

- Uma perspectiva técnica, possivelmente quantitativa, sobre por que eles deveriam mudar. • Um forte exemplo de como é a mudança. • Uma explicação lógica para grande parte da "sabedoria popular" usada pelo SRE. • Os princípios básicos necessários para abordar novas situações de maneira escalável.

Sua tarefa final é escrever um relatório pós-ação. Este relatório deve reiterar sua perspectiva, exemplos e explicações. Ele também deve fornecer alguns itens de ação para a equipe garantir que eles exercitem o que você ensinou. Você pode organizar o relatório como um postvitam,¹ explicando as decisões críticas em cada etapa que levaram ao sucesso.

¹ Em contraste com uma autópsia.

A maior parte do engajamento agora está completa. Uma vez que sua atribuição incorporada seja concluída, você deve permanecer disponível para revisões de design e código. Fique de olho na equipe nos próximos meses para confirmar que eles estão melhorando lentamente seus processos de planejamento de capacidade, resposta a emergências e distribuição.

CAPÍTULO 31

Comunicação e Colaboração em SRE

Escrito por Niall Murphy com Alex Rodriguez, Carl Crous, Dario Freni, Dylan Curley, Lorenzo Blanco e Todd Underwood
Editado por Betsy Beyer

A posição organizacional da SRE no Google é interessante e afeta a forma como nos comunicamos e colaboramos.

Para começar, há uma tremenda diversidade no que o SRE faz e como o fazemos. Temos equipes de infraestrutura, equipes de serviço e equipes horizontais de produtos. Temos relacionamentos com equipes de desenvolvimento de produtos que variam de equipes muitas vezes do nosso tamanho até equipes aproximadamente do mesmo tamanho que suas contrapartes e situações em que somos a equipe de desenvolvimento de produtos. As equipes de SRE são compostas por pessoas com habilidades de engenharia de sistemas ou arquitetura (consulte [\[Hix15b\]](#)), habilidades de engenharia de software, habilidades de gerenciamento de projetos, instintos de liderança, experiência em todos os tipos de indústrias (consulte o [Capítulo 33](#)) e assim por diante. Não temos apenas um modelo e encontramos uma variedade de configurações que funcionam; essa flexibilidade se adapta à nossa natureza pragmática.

Também é verdade que o SRE não é uma organização de comando e controle. Geralmente, devemos fidelidade a pelo menos dois mestres: para equipes de SRE de serviço ou infraestrutura, trabalhamos em estreita colaboração com as equipes de desenvolvimento de produto correspondentes que trabalham nesses serviços ou nessa infraestrutura; obviamente também trabalhamos no contexto do SRE em geral. A relação de serviço é muito forte, pois somos responsáveis pelo desempenho desses sistemas, mas apesar dessa relação, nossas linhas de reporte reais são através do SRE como um todo. Hoje, gastamos mais tempo apoiando nossos serviços individuais do que em trabalho de produção cruzada, mas nossa cultura e nossos valores compartilhados produzem abordagens fortemente homogêneas para os problemas. Isso ocorre por design.¹

¹ E, como todos sabemos, a cultura sempre vence a estratégia: [\[Mer11\]](#).

Os dois fatos anteriores conduziram a organização SRE em certas direções quando se trata de duas dimensões cruciais de como nossas equipes operam – comunicação e colaboração. O fluxo de dados seria uma metáfora computacional adequada para nossas comunicações: assim como os dados devem fluir em torno da produção, os dados também precisam fluir em torno de uma equipe de SRE – dados sobre projetos, o estado dos serviços, produção e o estado dos indivíduos . Para a máxima eficácia de uma equipe, os dados devem fluir de maneira confiável de uma parte interessada para outra. Uma maneira de pensar nesse fluxo é pensar na interface que uma equipe de SRE deve apresentar a outras equipes, como uma API. Assim como uma API, um bom design é crucial para uma operação eficaz e, se a API estiver errada, pode ser doloroso corrigi-la mais tarde.

A metáfora da API como contrato também é relevante para a colaboração, tanto entre as equipes de SRE quanto entre as equipes de SRE e de desenvolvimento de produtos – todos precisam progredir em um ambiente de mudanças implacáveis. Nessa medida, nossa colaboração se parece muito com a colaboração em qualquer outra empresa em movimento rápido. A diferença é a combinação de habilidades de engenharia de software, experiência em engenharia de sistemas e a sabedoria da experiência de produção que o SRE traz para essa colaboração. Os melhores desenhos e as melhores implementações resultam da preocupação conjunta da produção e do produto serem atendidos em um clima de respeito mútuo. Esta é a promessa do SRE: uma organização carregada de confiabilidade, com as mesmas habilidades que as equipes de desenvolvimento de produtos, melhorará as coisas de forma mensurável. Nossa experiência sugere que simplesmente ter alguém responsável pela confiabilidade, sem também ter o conjunto completo de habilidades, não é suficiente.

Comunicações: Reuniões de Produção

Embora a literatura sobre a realização de reuniões eficazes seja abundante [Kra08], é difícil encontrar alguém que tenha a sorte de ter apenas reuniões úteis e eficazes. Isso é igualmente verdadeiro para o SRE.

No entanto, há um tipo de reunião que temos que é mais útil do que a média, chamada reunião de produção. As reuniões de produção são um tipo especial de reunião em que uma equipe de SRE articula cuidadosamente para si mesma – e para seus convidados – o estado do(s) serviço(s) sob sua responsabilidade, de modo a aumentar a conscientização geral entre todos que se preocupam e melhorar a operação do(s) serviço(s). Em geral, essas reuniões são orientadas para o serviço; eles não são diretamente sobre as atualizações de status dos indivíduos. O objetivo é que todos saiam da reunião com uma ideia do que está acontecendo – a mesma ideia.

O outro objetivo principal das reuniões de produção é melhorar nossos serviços, trazendo a sabedoria da produção para os nossos serviços. Isso significa que falamos em detalhes sobre o desempenho operacional do serviço e relacionamos esse desempenho operacional ao design, configuração ou implementação e fazemos recomendações sobre como corrigir os problemas. Conectar o desempenho do serviço com as decisões de design em uma reunião regular é um ciclo de feedback imensamente poderoso.

Nossas reuniões de produção costumam acontecer semanalmente; dada a antipatia da SRE por reuniões inúteis, essa frequência parece ser quase certa: tempo para permitir que material relevante suficiente se acumule para fazer a reunião valer a pena, embora não seja tão frequente que as pessoas encontrem desculpas para não comparecer. Eles geralmente duram algo entre 30 e 60 minutos. Menos e você provavelmente está cortando algo desnecessariamente curto, ou provavelmente deveria estar aumentando seu portfólio de serviços. Mais e você provavelmente está ficando atolado nos detalhes, ou você tem muito o que falar e deve dividir a equipe ou o conjunto de serviços.

Assim como qualquer outra reunião, a reunião de produção deve ter um presidente. Muitas equipes de SRE alternam a presidência entre vários membros da equipe, o que tem a vantagem de fazer com que todos sintam que têm interesse no serviço e alguma propriedade nocional das questões. É verdade que nem todos têm níveis iguais de habilidade de presidência, mas o valor da propriedade do grupo é tão grande que a troca de sub-otimização temporária vale a pena. Além disso, esta é uma chance de inculcar habilidades de presidência, que são muito úteis no tipo de situações de coordenação de incidentes comumente enfrentadas pelo SRE.

Nos casos em que duas equipes SRE estão se reunindo por vídeo e uma das equipes é muito maior que a outra, notamos uma dinâmica interessante em jogo. Recomendamos colocar sua cadeira no lado menor da chamada por padrão. O lado maior naturalmente tende a se acalmar e alguns dos efeitos negativos de tamanhos de equipes desequilibrados (agravados pelos atrasos inerentes à videoconferência) melhorarão.² Não temos ideia se essa técnica tem alguma base científica, mas tende a trabalhar.

Agenda

Existem muitas formas de realizar uma reunião de produção, atestando a diversidade do que a SRE cuida e como fazemos. Nessa medida, não é apropriado ser prescritivo sobre como conduzir uma dessas reuniões. No entanto, uma agenda padrão (consulte o [Apêndice F](#) para obter um exemplo) pode se parecer com o seguinte:

Próximas mudanças de produção

As reuniões de acompanhamento de mudanças são bem conhecidas em todo o setor e, de fato, reuniões inteiras geralmente são dedicadas a interromper a mudança. No entanto, em nosso ambiente de produção, geralmente adotamos como padrão a ativação da alteração, o que requer o rastreamento do conjunto útil de propriedades dessa alteração: hora de início, duração, efeito esperado e assim por diante. Esta é a visibilidade do horizonte de curto prazo.

² A equipe maior geralmente tende a falar involuntariamente sobre a equipe menor, é mais difícil controlar conversas paralelas que distraem, etc.

Métricas

Uma das principais maneiras pelas quais conduzimos uma discussão orientada a serviços é falando sobre as principais métricas dos sistemas em questão; veja o [Capítulo 4](#). Mesmo que os sistemas não falhem dramaticamente naquela semana, é muito comum estar em uma posição em que você está vendo um aumento gradual (ou acentuado!) de carga ao longo do ano. Manter o controle de como seus valores de latência, valores de utilização de CPU, etc., mudam ao longo do tempo é incrivelmente valioso para desenvolver uma percepção do envelope de desempenho de um sistema.

Algumas equipes rastreiam o uso e a eficiência de recursos, o que também é um indicador útil de mudanças de sistema mais lentas e talvez mais insidiosas.

Interrupções

Este item trata de problemas de tamanho aproximado pós-morte e é uma oportunidade indispensável para aprender. Uma boa análise post-mortem, conforme discutido no [Capítulo 15](#), deve sempre fazer com que os sucos fluam.

Eventos de

paginação São páginas do seu sistema de monitoramento, relacionadas a problemas que podem ser post-mortem, mas geralmente não são. De qualquer forma, enquanto a parte Interrupções examina a imagem maior de uma interrupção, esta seção examina a visão tática: a lista de páginas, quem foi chamado, o que aconteceu então e assim por diante. Há duas perguntas implícitas para esta seção: esse alerta deveria ter paginado da maneira que o fez, e deveria ter paginado? Se a resposta para a última pergunta for não, remova essas páginas não acionáveis.

Eventos sem paginação

Este bucket contém três itens:

- Um problema que provavelmente deveria ter paginado, mas não o fez. Nesses casos, você provavelmente deve corrigir o monitoramento para que esses eventos açãoem uma página.
Muitas vezes, você encontra o problema enquanto tenta corrigir outra coisa ou está relacionado a uma métrica que você está rastreando, mas para a qual você não recebeu um alerta. • Um problema que não é paginável, mas requer atenção, como corrupção de dados de baixo impacto ou lentidão em alguma dimensão do sistema que não seja voltada para o usuário.
O rastreamento do trabalho operacional reativo também é apropriado aqui.
- Um problema que não é paginável e não requer atenção. Esses alertas devem ser removidos, pois criam um ruído extra que distrai os engenheiros de problemas que merecem atenção.

Itens de ação anterior

As discussões detalhadas anteriores geralmente levam a ações que o SRE precisa tomar — consertar isso, monitorar aquilo, desenvolver um subsistema para fazer o outro. Acompanhe essas melhorias da mesma forma que seriam monitoradas em qualquer outra reunião: atribua itens de ação às pessoas e acompanhe seu progresso. É uma boa ideia ter um item de agenda explícito que atue como um catchall, se nada mais. A entrega consistente também é um maravilhoso construtor de credibilidade e confiança.

Não importa como essa entrega seja feita, apenas que seja feita.

Comparecimento

A presença é obrigatória para todos os membros da equipe SRE em causa. Isso é particularmente verdadeiro se sua equipe estiver espalhada por vários países e/ou fusos horários, porque essa é sua principal oportunidade de interagir como um grupo.

As principais partes interessadas também devem participar desta reunião. Quaisquer equipes de desenvolvimento de produtos de parceiros que você possa ter também devem participar. Algumas equipes do SRE fragmentam sua reunião para que os assuntos somente do SRE sejam mantidos no primeiro semestre; essa prática é boa, desde que todos, como dito anteriormente, saiam com a mesma ideia do que está acontecendo. De tempos em tempos, representantes de outras equipes de SRE podem aparecer, principalmente se houver algum problema maior entre equipes para discutir, mas, em geral, a equipe de SRE em questão e outras equipes importantes devem comparecer. Se seu relacionamento é tal que você não pode convidar seus parceiros de desenvolvimento de produtos, você precisa corrigir esse relacionamento: talvez o primeiro passo seja convidar um representante dessa equipe ou encontrar um intermediário confiável para comunicação de proxy ou modelar interações saudáveis. Há muitas razões pelas quais as equipes não se dão bem, e muitos escritos sobre como resolver esse problema: essas informações também são aplicáveis às equipes de SRE, mas é importante que o objetivo final de ter um ciclo de feedback das operações seja cumprido, ou perde-se grande parte do valor de ter uma equipe SRE.

Ocasionalmente, você terá muitas equipes ou participantes ocupados, mas cruciais para convidar.

Existem várias técnicas que você pode usar para lidar com essas situações:

- Os serviços menos ativos podem ser atendidos por um único representante da equipe de desenvolvimento de produtos, ou ter apenas o compromisso da equipe de desenvolvimento de produtos em ler e comentar as atas de agenda.
- Se a equipe de desenvolvimento de produção for muito grande, indique um subconjunto de representantes sentativos.
- Os participantes ocupados, mas cruciais, podem fornecer feedback e/ou orientar antecipadamente os indivíduos, ou usar a técnica de agenda pré-preenchida (descrita a seguir).

A maioria das estratégias de reunião que discutimos são de bom senso, com um toque de serviço. Uma maneira única de tornar as reuniões mais eficientes e inclusivas é usar os recursos colaborativos em tempo real do Google Docs. Muitas equipes de SRE têm esse documento, com um endereço bem conhecido que qualquer pessoa em engenharia pode acessar.

Ter um documento assim possibilita duas ótimas práticas:

- Pré-preencher a agenda com ideias, comentários e informações “de baixo para cima”. • Preparar a agenda em paralelo e com antecedência é realmente eficiente.

Use totalmente os recursos de colaboração de várias pessoas habilitados pelo produto. Não há nada como ver uma cadeira de reunião digitando em uma frase, e então ver outra pessoa fornecer um link para o material de origem entre colchetes depois de terminar de digitar, e então ver outra pessoa arrumar a ortografia e gramática na frase original . Essa colaboração faz as coisas mais rapidamente e faz com que mais pessoas sintam que possuem uma fatia do que a equipe faz.

Colaboração no SRE

Obviamente, o Google é uma organização multinacional. Por causa do componente de resposta de emergência e rotação de pager de nossa função, temos boas razões comerciais para ser uma organização distribuída, separada por pelo menos alguns fusos horários. O impacto prático dessa distribuição é que temos definições muito fluidas para “equipe” em comparação, por exemplo, com a equipe média de desenvolvimento de produtos. Temos equipes locais, a equipe no site, a equipe multicontinental, equipes virtuais de vários tamanhos e coerência, e tudo mais. Isso cria uma mistura alegremente caótica de responsabilidades, habilidades e oportunidades. Pode-se esperar que grande parte da mesma dinâmica pertença a qualquer empresa suficientemente grande (embora possam ser particularmente intensas para empresas de tecnologia). Dado que a maioria das colaborações locais não enfrenta nenhum obstáculo específico, o caso interessante em termos de colaboração é entre equipes, entre sites, entre uma equipe virtual e similares.

Esse padrão de distribuição também informa como as equipes de SRE tendem a ser organizadas. Como nossa razão de ser é agregar valor através do domínio técnico, e o domínio técnico tende a ser difícil, tentamos encontrar uma maneira de ter domínio sobre algum subconjunto de sistemas ou infraestruturas relacionado, a fim de diminuir a carga cognitiva. A especialização é uma forma de atingir este objetivo; ou seja, a equipe X trabalha apenas no produto Y. A especialização é boa, porque leva a maiores chances de melhorar o domínio técnico, mas também é ruim, porque leva à siloização e ao desconhecimento do quadro mais amplo. Tentamos ter uma carta de equipe clara para definir o que uma equipe irá – e mais importante, não irá – apoiar, mas nem sempre conseguimos.

Composição da equipe

Temos uma ampla gama de conjuntos de habilidades em SRE, desde engenharia de sistemas até engenharia de software e organização e gerenciamento. A única coisa que podemos dizer sobre a colaboração é que suas chances de uma colaboração bem-sucedida – e, na verdade, praticamente qualquer outra coisa – são aprimoradas com mais diversidade em sua equipe. Há muitas evidências sugerindo que equipes diversas são simplesmente equipes melhores [Nel14]. Dirigir uma equipe diversificada implica atenção especial à comunicação, preconceitos cognitivos e assim por diante, que não podemos abordar em detalhes aqui.

Formalmente, as equipes de SRE têm as funções de “líder técnico” (TL), “gerente” (SRM) e “gerente de projeto” (também conhecido como PM, TPM, PgM). Algumas pessoas operam melhor quando essas funções têm responsabilidades bem definidas: o principal benefício disso é que elas podem tomar decisões dentro do escopo com rapidez e segurança. Outros operam melhor em um ambiente mais fluido, com responsabilidades variáveis dependendo da negociação dinâmica. Em geral, quanto mais fluida a equipe, mais desenvolvida em termos de capacidades dos indivíduos e mais capaz a equipe é de se adaptar a novas situações – mas ao custo de ter que se comunicar cada vez mais frequentemente, porque menos antecedentes podem ser assumidos.

Independentemente de quão bem esses papéis sejam definidos, em um nível básico, o líder técnico é responsável pela direção técnica da equipe e pode liderar de várias maneiras – desde comentar cuidadosamente o código de todos, até manter a direção trimestral pré-sensações, para a construção de consenso na equipe. No Google, os TLs podem fazer quase todo o trabalho de um gerente, porque nossos gerentes são altamente técnicos, mas o gerente tem duas responsabilidades especiais que um TL não tem: a função de gerenciamento de desempenho e ser um general catchall para tudo o que não é tratado por outra pessoa.

Grandes TLs, SRMs e TPMs têm um conjunto completo de habilidades e podem alegremente se dedicar à organização de um projeto, comentar um documento de design ou escrever código conforme necessário.

Técnicas para trabalhar de forma eficaz

Existem várias maneiras de fazer engenharia eficaz em SRE.

Em geral, projetos singleton falham, a menos que a pessoa seja particularmente talentosa ou o problema seja simples. Para realizar algo significativo, você precisa de várias pessoas. Portanto, você também precisa de boas habilidades de colaboração. Novamente, muito material foi escrito sobre este tópico, e grande parte dessa literatura é aplicável ao SRE.

Em geral, um bom trabalho de SRE exige excelentes habilidades de comunicação quando você está trabalhando fora dos limites de sua equipe puramente local. Para colaborações fora do prédio, trabalhar efetivamente entre fusos horários implica uma ótima comunicação escrita ou muitas viagens para fornecer a experiência pessoal que é adiável, mas

em última análise, necessário para um relacionamento de alta qualidade. Mesmo que você seja um grande escritor, com o tempo você se torna apenas um endereço de e-mail até que você reapareça em carne e osso novamente.

Estudo de Caso de Colaboração no SRE: Viceroy

Um exemplo de colaboração bem-sucedida entre SREs é um projeto chamado Viceroy, que é uma estrutura e serviço de painel de monitoramento. A atual arquitetura organizacional do SRE pode resultar em equipes produzindo várias cópias ligeiramente diferentes do mesmo trabalho; por várias razões, as estruturas do painel de monitoramento foram um terreno particularmente fértil para a duplicação de trabalho.³

Os incentivos que levaram ao sério problema de lixo de muitas estruturas de monitoramento abandonadas e latentes eram bem simples: cada equipe foi recompensada por desenvolver sua própria solução, trabalhar fora dos limites da equipe era difícil e a infraestrutura que tendia a ser desde que todo o SRE estivesse tipicamente mais próximo de um kit de ferramentas do que de um produto. Esse ambiente encorajou engenheiros individuais a usar o kit de ferramentas para fazer outro naufrágio em chamas, em vez de resolver o problema para o maior número possível de pessoas (um esforço que, portanto, levaria muito mais tempo).

A Vinda do Vice -Rei O vice- rei foi

diferente. Tudo começou em 2012, quando várias equipes estavam considerando como migrar para o Monarch, o novo sistema de monitoramento do Google. O SRE é profundamente conservador em relação aos sistemas de monitoramento, então o Monarch, ironicamente, levou mais tempo para obter tração dentro do SRE do que nas equipes não SRE. Mas ninguém poderia argumentar que nosso sistema de monitoramento legado, Borgmon (veja o Capítulo 10), não tinha espaço para melhorias. Por exemplo, nossos consoles eram complicados porque usavam um sistema de modelagem HTML personalizado que era especial, cheio de casos extremos e difíceis de testar. Naquela época, o Monarch havia amadurecido o suficiente para ser aceito em princípio como o substituto do sistema legado e, portanto, estava sendo adotado por mais e mais equipes do Google, mas ainda tínhamos problemas com os consoles.

Aqueles de nós que tentaram usar o Monarch para nossos serviços logo descobriram que ele ficou aquém do suporte do console por dois motivos principais:

- Os consoles eram fáceis de configurar para um serviço pequeno, mas não se adaptavam bem a serviços com consoles complexos.
- Eles também não suportavam o sistema de monitoramento legado, tornando a transição para o Monarch muito difícil.

³ Nesse caso em particular, o caminho para o inferno foi de fato pavimentado com JavaScript.

Como não existia nenhuma alternativa viável para implantar o Monarch dessa maneira na época, vários projetos específicos de equipe foram lançados. Como havia poucas soluções de desenvolvimento coordenadas ou mesmo rastreamento entre grupos na época (um problema que já foi corrigido), acabamos duplicando os esforços mais uma vez. Várias equipes do Spanner, Ads Frontend e uma variedade de outros serviços desenvolveram seus próprios esforços (um exemplo notável foi chamado de Consoles++) ao longo de 12 a 18 meses e, eventualmente, a sanidade prevaleceu quando os engenheiros de todas essas equipes acordaram e descobriu os respectivos esforços uns dos outros. Eles decidiram fazer a coisa sensata e unir forças para criar uma solução geral para toda a SRE. Assim, o projeto Viceroy nasceu em meados de 2012.

No início de 2013, o Viceroy começou a atrair o interesse de equipes que ainda não tinham saído do sistema legado, mas que queriam colocar o pé na água.

Obviamente, as equipes com projetos de monitoramento existentes maiores tiveram menos incentivos para migrar para o novo sistema: era difícil para essas equipes racionalizar o abandono do baixo custo de manutenção de sua solução existente que basicamente funcionava bem, para algo relativamente novo e não comprovado que exigiria muito de esforço para fazer o trabalho. A grande diversidade de requisitos contribuiu para a relutância dessas equipes, embora todos os projetos de console de monitoramento compartilhassem dois requisitos principais, notadamente:

- Suporte a painéis de curadoria complexos •
Suporte ao Monarch e ao sistema de monitoramento legado

Cada projeto também tinha seu próprio conjunto de requisitos técnicos, que dependiam da preferência ou experiência do autor. Por exemplo:

- Várias fontes de dados fora dos principais sistemas de monitoramento
- Definição de consoles usando configuração versus layout HTML explícito • Sem JavaScript versus adoção total de JavaScript com AJAX • Uso exclusivo de conteúdo estático, para que os consoles possam ser armazenados em cache no navegador

Embora alguns desses requisitos fossem mais rígidos do que outros, no geral eles dificultavam os esforços de fusão. De fato, embora a equipe do Consoles++ estivesse interessada em ver como seu projeto se comparava ao Viceroy, seu exame inicial no primeiro semestre de 2013 determinou que as diferenças fundamentais entre os dois projetos eram significativas o suficiente para impedir a integração. A maior dificuldade era que o Viceroy por design não usava muito JavaScript, enquanto o Consoles++ era escrito principalmente em Java Script. Havia um vislumbre de esperança, no entanto, em que os dois sistemas tinham várias semelhanças subjacentes:

- Eles usaram sintaxes semelhantes para renderização de modelo HTML. •

Eles compartilharam várias metas de longo prazo, que nenhuma das equipes ainda havia começado a abordar. Por exemplo, ambos os sistemas queriam armazenar dados de monitoramento em cache e dar suporte a um pipeline offline para produzir periodicamente dados que o console pudesse usar, mas era computacionalmente caro para produzir sob demanda.

Acabamos estacionando a discussão do console unificado por um tempo. No entanto, no final de 2013, tanto o Consoles++ quanto o Viceroy haviam se desenvolvido significativamente. Suas diferenças técnicas diminuíram, porque o Viceroy começou a usar JavaScript para renderizar seus gráficos de monitoramento. As duas equipes se conheceram e descobriram que a integração era muito mais fácil, agora que a integração se resumia a servir os dados do Console++ fora do servidor Viceroy. Os primeiros protótipos integrados foram concluídos no início de 2014 e provaram que os sistemas podem funcionar bem juntos. As duas equipes se sentiram à vontade para se comprometer com um esforço conjunto naquele momento e, como o Viceroy já havia estabelecido sua marca como uma solução de monitoramento comum, o projeto combinado manteve o nome Viceroy.

O desenvolvimento da funcionalidade completa levou alguns trimestres, mas no final de 2014, o sistema combinado estava completo.

A união de forças trouxe enormes benefícios:

- Viceroy recebeu uma série de fontes de dados e clientes JavaScript para acessá-los. • A compilação JavaScript foi reescrita para suportar módulos separados que podem ser incluídos seletivamente. Isso é essencial para dimensionar o sistema para qualquer número de equipes com seu próprio código JavaScript.
- Consoles++ se beneficiaram das muitas melhorias feitas ativamente no Viceøy, como a adição de seu cache e pipeline de dados em segundo plano.
- No geral, a velocidade de desenvolvimento em uma solução foi muito maior que a soma de toda a velocidade de desenvolvimento dos projetos duplicados.

Em última análise, a visão de futuro comum foi o fator chave na combinação dos projetos.

Ambas as equipes encontraram valor em expandir sua equipe de desenvolvimento e se beneficiaram das contribuições uma da outra. O ímpeto foi tal que, no final de 2014, o Viceroy foi oficialmente declarado a solução geral de monitorização para todo o SRE. Talvez caracteristicamente para o Google, essa declaração não exigia que as equipes adotassem o Viceroy: em vez disso, recomendava que as equipes usassem o Viceroy em vez de escrever outro console de monitoramento.

Desafios Embora

tenha sido um sucesso, o Viceroy teve dificuldades, e muitas delas surgiram devido à natureza do projeto em vários locais.

Uma vez que a equipe estendida do vice-rei foi estabelecida, a coordenação inicial entre os membros remotos da equipe provou ser difícil. Ao conhecer pessoas pela primeira vez, dicas sutis na escrita e na fala podem ser mal interpretadas, porque os estilos de comunicação variam substancialmente de pessoa para pessoa. No início do projeto, os membros da equipe que não estavam localizados em Mountain View também perderam as discussões improvisadas sobre bebedouros que muitas vezes aconteciam pouco antes e depois das reuniões (embora a comunicação tenha melhorado consideravelmente desde então).

Enquanto a equipe principal do vice-rei permaneceu bastante consistente, a equipe estendida de colaboradores foi bastante dinâmica. Os colaboradores tinham outras responsabilidades que mudaram ao longo do tempo e, portanto, muitos puderam dedicar entre um e três meses ao projeto. Assim, o pool de colaboradores do desenvolvedor, que era inherentemente maior do que a equipe principal do Viceroy, foi caracterizado por uma quantidade significativa de rotatividade.

A adição de novas pessoas ao projeto exigiu treinar cada colaborador sobre o design geral e a estrutura do sistema, o que levou algum tempo. Por outro lado, quando um SRE contribuía para a funcionalidade central do Viceroy e depois voltava para sua própria equipe, ele era um especialista local no sistema. Essa disseminação imprevista de especialistas locais do Viceroy levou a mais uso e adoção.

À medida que as pessoas entravam e saíam da equipe, descobrimos que as contribuições casuais eram úteis e dispendiosas. O custo principal era a diluição da propriedade: uma vez que os recursos eram entregues e a pessoa saía, os recursos ficavam sem suporte com o tempo e geralmente eram descartados.

Além disso, o escopo do projeto Viceroy cresceu ao longo do tempo. Ele tinha metas ambiciosas no lançamento, mas o escopo inicial era limitado. À medida que o escopo crescia, no entanto, lutamos para entregar os principais recursos no prazo e tivemos que melhorar o gerenciamento de projetos e definir uma direção mais clara para garantir que o projeto continuasse nos trilhos.

Finalmente, a equipe do Viceroy achou difícil possuir completamente um componente que tivesse contribuições significativas (determinantes) de sites distribuídos. Mesmo com a melhor vontade do mundo, as pessoas geralmente optam pelo caminho de menor resistência e discutem questões ou tomam decisões localmente sem envolver os proprietários remotos, o que pode levar a conflitos.

Recomendações

Você só deve desenvolver projetos cross-site quando for necessário, mas muitas vezes há boas razões para isso. O custo de trabalhar entre sites é a latência mais alta para ações e a necessidade de mais comunicação; o benefício é - se você acertar a mecânica - um rendimento muito maior. O projeto de um único site também pode ser prejudicado por ninguém fora desse site saber o que você está fazendo, então há custos para ambas as abordagens.

Colaboradores motivados são valiosos, mas nem todas as contribuições são igualmente valiosas. Certifique-se de que os contribuidores do projeto estejam realmente comprometidos e não estejam apenas participando

algum objetivo nebuloso de auto-realização (querer ganhar um entalhe em seu cinto anexando seu nome a um projeto brilhante; querer codificar em um novo projeto empolgante sem se comprometer a manter esse projeto). Contribuintes com um objetivo específico a atingir geralmente estarão mais motivados e manterão melhor suas contribuições.

À medida que os projetos se desenvolvem, eles geralmente crescem, e você nem sempre tem a sorte de ter pessoas em sua equipe local para contribuir com o projeto. Portanto, pense cuidadosamente sobre a estrutura do projeto. Os líderes de projeto são importantes: eles fornecem visão de longo prazo para o projeto e garantem que todo o trabalho esteja alinhado com essa visão e seja priorizado corretamente. Você também precisa ter uma maneira acordada de tomar decisões e deve otimizar especificamente para tomar mais decisões localmente se houver um alto nível de concordância e confiança.

A estratégia padrão de “dividir para conquistar” se aplica a projetos cross-site; você reduz os custos de comunicação principalmente dividindo o projeto em tantos componentes de tamanho razoável quanto possível e tentando garantir que cada componente possa ser atribuído a um pequeno grupo, de preferência dentro de um site. Divilde esses componentes entre as subequipes do projeto e estabeleça entregas e prazos claros. (Tente não deixar que a lei de Conway distorça muito profundamente a forma natural do software.)⁴ Um objetivo para uma equipe de projeto funciona melhor quando é orientado a fornecer alguma funcionalidade ou resolver algum problema. Essa abordagem garante que os indivíduos que trabalham em um componente saibam o que se espera deles e que seu trabalho só será concluído quando esse componente estiver totalmente integrado e usado no projeto principal.

Obviamente, as melhores práticas usuais de engenharia se aplicam a projetos colaborativos: cada componente deve ter documentos de projeto e revisões com a equipe. Dessa forma, todos na equipe têm a oportunidade de acompanhar as mudanças, além da chance de influenciar e melhorar os projetos. Anotar as coisas é uma das principais técnicas que você tem para compensar a distância física e/ou lógica – use-a.

Os padrões são importantes. As diretrizes de estilo de codificação são um bom começo, mas geralmente são bastante táticas e, portanto, apenas um ponto de partida para estabelecer as normas da equipe. Toda vez que houver um debate sobre qual escolha fazer em um problema, discuta-o totalmente com a equipe, mas com um limite de tempo estrito. Em seguida, escolha uma solução, documente-a e siga em frente. Se você não pode concordar, você precisa escolher algum árbitro que todos respeitem e, novamente, apenas seguir em frente. Com o tempo, você criará uma coleção dessas práticas recomendadas, que ajudarão novas pessoas a se atualizarem.

Em última análise, não há substituto para a interação pessoal, embora parte da interação face a face possa ser adiada pelo bom uso de VC e boa comunicação escrita. Se puder, peça aos líderes do projeto que conheçam o restante da equipe em per

4 Ou seja, o software tem a mesma estrutura que a estrutura de comunicação da organização que produz o software—consulte https://en.wikipedia.org/wiki/Conway%27s_law.

filho. Se o tempo e o orçamento permitirem, organize uma reunião de equipe para que todos os membros da equipe possam interagir pessoalmente. Uma cúpula também oferece uma ótima oportunidade para discutir projetos e objetivos. Para situações em que a neutralidade é importante, é vantajoso realizar reuniões de equipe em um local neutro para que nenhum local individual tenha a “vantagem em casa”.

Finalmente, use o estilo de gerenciamento de projetos que se adequa ao projeto em seu estado atual. Mesmo projetos com metas ambiciosas começarão pequenos, então a sobrecarga deve ser correspondentemente baixa. À medida que o projeto cresce, é apropriado adaptar e mudar a forma como o projeto é gerenciado. Dado o crescimento suficiente, o gerenciamento completo do projeto será necessário.

Colaboração fora do SRE

Como sugerimos, e o [Capítulo 32](#) discute, a colaboração entre a organização de desenvolvimento de produto e o SRE é realmente melhor quando ocorre no início da fase de projeto, idealmente antes de qualquer linha de código ser confirmada. Os SREs estão em melhor posição para fazer recomendações sobre arquitetura e comportamento de software que podem ser bastante difíceis (se não impossíveis) de adaptar. Ter essa voz presente na sala quando um novo sistema está sendo projetado é melhor para todos. De um modo geral, usamos o processo de Objetivos e Resultados-Chave (OKR) [\[Kla12\]](#) para acompanhar esse trabalho. Para algumas equipes de serviço, essa colaboração é a base do que eles fazem – rastrear novos projetos, fazer recomendações, ajudar a implementá-los e conduzi-los até a produção.

Estudo de caso: migração do DFP para F1

Grandes projetos de migração de serviços existentes são bastante comuns no Google. Exemplos típicos incluem portar componentes de serviço para uma nova tecnologia ou atualizar componentes para suportar um novo formato de dados. Com a recente introdução de tecnologias de banco de dados que podem escalar para um nível global, como Spanner [\[Cor12\]](#) e F1 [\[Shu13\]](#), o Google realizou vários projetos de migração em grande escala envolvendo bancos de dados. Um desses projetos foi a migração do banco de dados principal do DoubleClick for Publishers (DFP)⁵ de MySQL para F1. Em particular, alguns dos autores deste capítulo foram responsáveis por uma parte do sistema servidor (mostrado na [Figura 31-1](#)) que extrai e processa continuamente dados do banco de dados, a fim de gerar um conjunto de arquivos indexados que são então carregados e servido em todo o mundo. Esse sistema foi distribuído em vários datacenters e usou cerca de 1.000 CPUs e 8 TB de RAM para indexar 100 TB de dados todos os dias.

⁵ O DoubleClick for Publishers é uma ferramenta para os editores gerenciarem anúncios veiculados em seus sites e aplicativos.

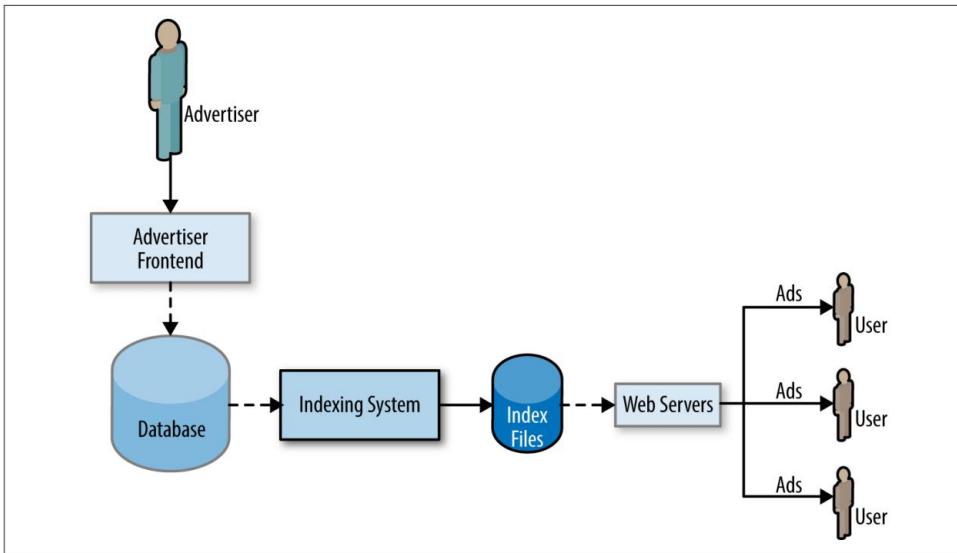


Figura 31-1. Um sistema genérico de veiculação de anúncios

A migração não foi trivial: além de migrar para uma nova tecnologia, o esquema do banco de dados foi significativamente refatorado e simplificado graças à capacidade de F1 de armazenar e indexar dados de buffer de protocolo em colunas de tabela. O objetivo era migrar o sistema de processamento para que pudesse produzir uma saída perfeitamente idêntica ao sistema existente. Isso nos permitiu deixar o sistema de atendimento intocado e realizar, do ponto de vista do usuário, uma migração perfeita. Como uma restrição adicional, o produto exigia que concluíssemos uma migração ao vivo sem qualquer interrupção do serviço para nossos usuários a qualquer momento. Para isso, a equipe de desenvolvimento de produtos e a equipe do SRE começaram a trabalhar em estreita colaboração, desde o início, para desenvolver o novo serviço de indexação.

Como seus principais desenvolvedores, as equipes de desenvolvimento de produto geralmente estão mais familiarizadas com a lógica de negócios (BL) do software e também estão em contato mais próximo com os gerentes de produto e o componente real de “necessidade de negócios” dos produtos. Por outro lado, as equipes de SRE geralmente têm mais experiência em relação aos componentes de infraestrutura do software (por exemplo, bibliotecas para conversar com sistemas de armazenamento distribuído ou bancos de dados), porque os SREs geralmente reutilizam os mesmos blocos de construção em diferentes serviços, aprendendo as muitas advertências e nuances que permitem que o software seja executado de forma escalável e confiável ao longo do tempo.

Desde o início do projeto de migração, o desenvolvimento do produto e o SRE sabiam que teriam que colaborar ainda mais de perto, realizando reuniões semanais para sincronizar o progresso do projeto. Neste caso em particular, as mudanças de BL foram parcialmente dependentes de mudanças de infraestrutura. Por esta razão o projeto começou com o desenho da nova infraestrutura; os SREs, que possuíam amplo conhecimento sobre o domínio da

extraí e processar dados em escala, impulsionou o projeto das mudanças de infraestrutura. Isso envolveu projetar como extraí as várias tabelas de F1, como filtrar e unir os dados, como extraí apenas os dados que foram alterados (em oposição a todo o banco de dados), como sustentar a perda de algumas das máquinas sem afetar o serviço, como garantir que o uso de recursos cresça linearmente com a quantidade de dados extraídos, o planejamento de capacidade e muitos outros aspectos semelhantes. A nova infraestrutura proposta era semelhante a outros serviços que já extraíam e processavam dados da F1. Assim, pudemos ter certeza da solidez da solução e reutilizar partes do monitoramento e ferramental.

Antes de prosseguir com o desenvolvimento desta nova infraestrutura, dois SREs produziram um documento de projeto detalhado. Em seguida, as equipes de desenvolvimento de produto e SRE revisaram minuciosamente o documento, ajustando a solução para lidar com alguns casos extremos e, eventualmente, concordaram com um plano de design. Tal plano identificava claramente que tipo de mudanças a nova infraestrutura traria para o BL. Por exemplo, projetamos a nova infraestrutura para extraí apenas dados alterados, em vez de extraí repetidamente todo o banco de dados; o BL teve que levar em conta essa nova abordagem. No início, definimos as novas interfaces entre infraestrutura e BL, e isso permitiu que a equipe de desenvolvimento de produto trabalhasse de forma independente nas mudanças de BL. Da mesma forma, a equipe de desenvolvimento de produtos manteve a SRE informada sobre as mudanças no BL. Onde eles interagiram (por exemplo, mudanças de BL dependentes de infraestrutura), essa estrutura de coordenação nos permitiu saber que as mudanças estavam acontecendo e tratá-las de forma rápida e correta.

Em fases posteriores do projeto, os SREs começaram a implantar o novo serviço em um ambiente de teste que se assemelhava ao ambiente de produção finalizado do projeto. Esta etapa foi essencial para medir o comportamento esperado do serviço – em particular, desempenho e utilização de recursos – enquanto o desenvolvimento do BL ainda estava em andamento.

A equipe de desenvolvimento do produto utilizou este ambiente de teste para realizar a validação do novo serviço: o índice dos anúncios produzidos pelo serviço antigo (executando em produção) tinha que corresponder perfeitamente ao índice produzido pelo novo serviço (executando no ambiente de teste). Como suspeito, o processo de validação destacou discrepâncias entre os serviços antigos e novos (devido a alguns casos extremos no novo formato de dados), que a equipe de desenvolvimento de produto conseguiu resolver iterativamente: para cada anúncio eles depuraram a causa do erro. a diferença e corrigiu o BL que produziu o resultado ruim. Nesse meio tempo, a equipe do SRE começou a preparar o ambiente de produção: alocando os recursos necessários em um datacenter diferente, configurando processos e regras de monitoramento e treinando os engenheiros designados para o atendimento.

A equipe do SRE também configurou um processo básico de liberação que incluía validação, uma tarefa normalmente realizada pela equipe de desenvolvimento do produto ou pelos engenheiros de liberação, mas neste caso específico foi realizada pelos SREs para acelerar a migração.

Quando o serviço estava pronto, os SREs preparam um plano de implantação em colaboração com a equipe de desenvolvimento do produto e lançaram o novo serviço. O lançamento foi muito bem-sucedido e ocorreu sem problemas, sem nenhum impacto visível para o usuário.

Conclusão

Dada a natureza globalmente distribuída das equipes de SRE, a comunicação eficaz sempre foi uma alta prioridade no SRE. Este capítulo discutiu as ferramentas e técnicas que as equipes de SRE usam para manter relacionamentos eficazes entre sua equipe e com suas várias equipes parceiras.

A colaboração entre equipes de SRE tem seus desafios, mas potencialmente grandes recompensas, incluindo abordagens comuns para plataformas de solução de problemas, permitindo que nos concentremos na solução de problemas mais difíceis.

CAPÍTULO 32**O modelo de engajamento SRE em evolução**

**Escrito por Acácio Cruz e Ashish Bhamhani
Editado por Betsy Beyer e Tim Harvey**

Engajamento do SRE: o que, como e por quê

Discutimos na maior parte deste livro o que acontece quando o SRE já está encarregado de um serviço. Poucos serviços iniciam seu ciclo de vida desfrutando de suporte SRE, portanto, é preciso haver um processo para avaliar um serviço, certificando-se de que ele merece suporte SRE, negociando como melhorar quaisquer déficits que impeçam o suporte SRE e, de fato, instituindo suporte SRE. Chamamos esse processo de integração. Se você estiver em um ambiente em que está cercado por muitos serviços existentes em vários estados de perfeição, sua equipe de SRE provavelmente estará executando uma fila priorizada de integrações por um bom tempo até que a equipe termine de assumir o valor mais alto alvos.

Embora isso seja muito comum e uma maneira completamente razoável de lidar com um ambiente de fato consumado, na verdade existem pelo menos duas maneiras melhores de trazer a sabedoria da produção e o suporte SRE para serviços antigos e novos.

No primeiro caso, assim como na engenharia de software – onde quanto mais cedo o bug for encontrado, mais barato será a correção – quanto mais cedo uma consulta da equipe de SRE acontecer, melhor será o serviço e mais rápido ele sentirá o benefício. Quando o SRE é engajado durante os estágios iniciais do projeto, o tempo de integração é reduzido e o serviço é mais confiável “fora do portão”, geralmente porque não precisamos gastar tempo desfazendo o projeto ou a implementação abaixo do ideal.

Outra maneira, talvez a melhor, é curto-circuitar o processo pelo qual sistemas especialmente criados com muitas variações individuais acabam “chegando” à porta da SRE. Fornecer o desenvolvimento de produtos com uma plataforma de infraestrutura validada por SRE, sobre a qual eles podem construir seus sistemas. Esta plataforma terá o duplo benefício de ser confiável e escalável. Isso evita certas classes de problemas de carga cognitiva

completamente, e ao abordar práticas comuns de infraestrutura, permite que as equipes de desenvolvimento de produtos se concentrem na inovação na camada de aplicativo, onde ela pertence principalmente.

Nas seções a seguir, passaremos algum tempo examinando cada um desses modelos, começando pelo “clássico”, o modelo orientado por PRR.

O Modelo PRR

A etapa inicial mais típica do envolvimento do SRE é a Revisão de Prontidão de Produção (PRR), um processo que identifica as necessidades de confiabilidade de um serviço com base em seus detalhes específicos. Por meio de um PRR, os SREs buscam aplicar o que aprenderam e vivenciaram para garantir a confiabilidade de um serviço operando em produção. Um PRR é considerado um pré-requisito para uma equipe de SRE aceitar a responsabilidade de gerenciar os aspectos de produção de um serviço.

A [Figura 32-1](#) ilustra o ciclo de vida de um serviço típico. A Revisão de Prontidão de Produção pode ser iniciada em qualquer ponto do ciclo de vida do serviço, mas os estágios nos quais o envolvimento do SRE é aplicado se expandiram ao longo do tempo. Este capítulo descreve o Modelo PRR Simples e, em seguida, discute como sua modificação no Modelo de Engajamento Estendido e a estrutura de Estruturas e Plataforma SRE permitiu que o SRE dimensionasse seu processo de engajamento e impacto.

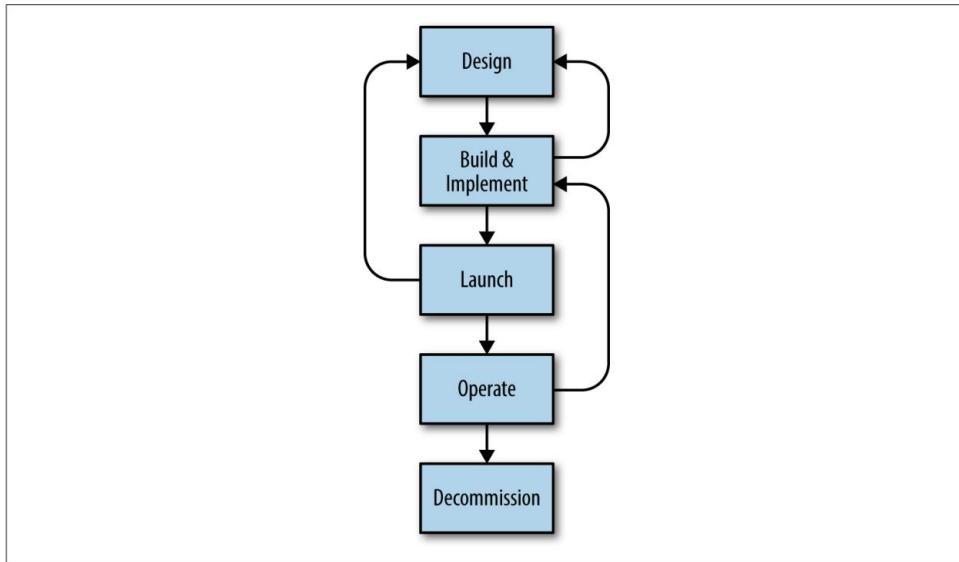


Figura 32-1. Um ciclo de vida de serviço típico

O Modelo de Engajamento SRE

A SRE busca a responsabilidade de produção para serviços importantes para os quais pode dar contribuições concretas para a confiabilidade. O SRE está preocupado com vários aspectos de um serviço, que são coletivamente chamados de produção. Esses aspectos incluem o seguinte:

- Arquitetura do sistema e dependências entre serviços •
- Instrumentação, métricas e monitoramento • Resposta a emergências • Planejamento de capacidade • Gerenciamento de mudanças • Desempenho: disponibilidade, latência e eficiência

Quando os SREs se envolvem com um serviço, buscamos melhorá-lo em todos esses eixos, o que facilita o gerenciamento da produção do serviço.

Suporte alternativo Nem

todos os serviços do Google recebem um envolvimento próximo do SRE. Alguns fatores estão em jogo aqui:

- Muitos serviços não precisam de alta confiabilidade e disponibilidade, então o suporte pode ser provido por outros meios.
- Por design, o número de equipes de desenvolvimento que solicitam suporte SRE excede a largura de banda disponível das equipes SRE (consulte o [Capítulo 1](#)).

Quando o SRE não pode fornecer suporte completo, ele oferece outras opções para fazer melhorias na produção, como documentação e consulta.

Documentação

Guias de desenvolvimento estão disponíveis para tecnologias internas e clientes de sistemas amplamente utilizados. O Guia de produção do Google documenta as práticas recomendadas de produção para serviços, conforme determinado pelas experiências do SRE e das equipes de desenvolvimento. Os desenvolvedores podem implementar as soluções e recomendações em tal documentação para melhorar seus serviços.

Consulta

Os desenvolvedores também podem procurar consultoria SRE para discutir serviços específicos ou áreas problemáticas. A equipe de Engenharia de Coordenação de Lançamento (LCE) (consulte o [Capítulo 27](#)) passa a maior parte do tempo consultando as equipes de desenvolvimento. As equipes de SRE que não são especificamente dedicadas a lançar consultas também se envolvem em consultas com desenvolvimento equipes.

Quando um novo serviço ou um novo recurso é implementado, os desenvolvedores geralmente consultam o SRE para obter conselhos sobre a preparação para a fase de lançamento. A consulta de lançamento geralmente envolve um ou dois SREs que passam algumas horas estudando o projeto e a implementação em alto nível. Os consultores do SRE então se reúnem com a equipe de desenvolvimento para fornecer conselhos sobre áreas de risco que precisam de atenção e discutir padrões ou soluções bem conhecidos que podem ser incorporados para melhorar o serviço em produção.

Alguns desses conselhos podem vir do Guia de Produção mencionado anteriormente.

As sessões de consulta são necessariamente amplas porque não é possível obter uma compreensão profunda de um determinado sistema no tempo limitado disponível. Para algumas equipes de desenvolvimento, a consulta não é suficiente:

- Serviços que cresceram em ordens de magnitude desde o seu lançamento, que agora exigem mais tempo para entender do que é viável por meio de documentação e consulta.
- Serviços nos quais muitos outros serviços passaram a depender posteriormente, que agora hospedam significativamente mais tráfego de muitos clientes diferentes.

Esses tipos de serviços podem ter crescido a ponto de começarem a encontrar dificuldades significativas na produção e, ao mesmo tempo, se tornarem importantes para os usuários. Nesses casos, o envolvimento de SRE de longo prazo se torna necessário para garantir que eles sejam mantidos adequadamente em produção à medida que crescem.

Revisões de prontidão de produção: modelo PRR simples

Quando uma equipe de desenvolvimento solicita que o SRE assuma o gerenciamento de produção de um serviço, o SRE avalia a importância do serviço e a disponibilidade das equipes de SRE. Se o serviço merecer suporte do SRE, e a equipe do SRE e a organização de desenvolvimento concordarem com os níveis de pessoal para facilitar esse suporte, o SRE inicia uma Revisão de Prontidão de Produção com a equipe de desenvolvimento.

Os objetivos da Revisão de Prontidão de Produção são os seguintes:

- Verifique se um serviço atende aos padrões aceitos de configuração de produção e prontidão operacional e se os proprietários do serviço estão preparados para trabalhar com o SRE e aproveitar a experiência do SRE.

- Melhore a confiabilidade do serviço em produção e minimize o número e a gravidade dos incidentes que podem ser esperados. Um PRR visa todos os aspectos da produção com os quais o SRE se preocupa.

Depois que melhorias suficientes são feitas e o serviço é considerado pronto para suporte SRE, uma equipe SRE assume suas responsabilidades de produção.

Isso nos leva ao próprio processo de Revisão de Prontidão de Produção. Existem três modelos de engajamento diferentes, mas relacionados (Modelo PRR Simples, Modelo de Engajamento Antecipado e Estruturas e Plataforma SRE), que serão discutidos por sua vez.

Primeiramente, descreveremos o Modelo PRR Simples, que geralmente é direcionado a um serviço já lançado e será assumido por uma equipe de SRE. Um PRR segue várias fases, como um ciclo de vida de desenvolvimento, embora possa prosseguir independentemente em paralelo com o ciclo de vida de desenvolvimento.

Engajamento A

Liderança do SRE primeiro decide qual equipe do SRE é a mais adequada para assumir o serviço.

Normalmente, um a três SREs são selecionados ou autonomeados para conduzir o processo de PRR.

Esse pequeno grupo inicia a discussão com a equipe de desenvolvimento. A discussão abrange assuntos como:

- Estabelecimento de um SLO/SLA para o serviço •
- Planejamento para mudanças de projeto potencialmente disruptivas necessárias para melhorar a confiabilidade
- Planejamento e cronogramas de treinamento

O objetivo é chegar a um acordo comum sobre o processo, objetivos finais e resultados que são necessários para que a equipe de SRE se envolva com a equipe de desenvolvimento e seu serviço.

Análise

Análise é o primeiro grande segmento de trabalho. Durante esta fase, os revisores do SRE aprendem sobre o serviço e começam a analisá-lo quanto a falhas de produção. Vizam aferir a maturidade do serviço ao longo dos vários eixos de preocupação do SRE. Eles também examinam o design e a implementação do serviço para verificar se ele segue as melhores práticas de produção. Normalmente, a equipe SRE estabelece e mantém uma lista de verificação de PRR explicitamente para a fase de Análise. A lista de verificação é específica para o serviço e geralmente é baseada em conhecimento de domínio, experiência com sistemas relacionados ou similares e melhores práticas do Guia de Produção. A equipe do SRE também pode consultar outras equipes que tenham mais experiência com determinados componentes ou dependências do serviço.

Alguns exemplos de itens da lista de verificação incluem:

- As atualizações no serviço afetam uma porcentagem excessivamente grande do sistema de uma vez só?
- O serviço se conecta à instância de serviço apropriada de suas dependências? Por exemplo, as solicitações do usuário final a um serviço não devem depender de um sistema projetado para um caso de uso de processamento em lote.
- O serviço solicita uma qualidade de serviço de rede suficientemente alta quando faláy para um serviço remoto crítico?
- O serviço relata erros aos sistemas de registro central para análise? Ele relata todas as condições excepcionais que resultam em respostas degradadas ou falhas para os usuários finais?
- Todas as falhas de solicitação visíveis ao usuário estão bem instrumentadas e monitoradas, com alertas adequados configurados?

A lista de verificação também pode incluir padrões operacionais e melhores práticas seguidas por uma equipe específica de SRE. Por exemplo, uma configuração de serviço perfeitamente funcional que não segue o “padrão ouro” de uma equipe de SRE pode ser refatorada para funcionar melhor com ferramentas de SRE para gerenciar configurações de forma escalável. Os SREs também analisam os incidentes recentes e post-mortems do serviço, bem como as tarefas de acompanhamento dos incidentes. Essa avaliação mede as demandas de atendimento emergencial para o serviço e a disponibilidade de controles operacionais bem estabelecidos.

Melhorias e Refatoração A fase de Análise leva à

identificação de melhorias recomendadas para o serviço. Esta próxima fase ocorre da seguinte forma:

1. As melhorias são priorizadas com base na importância para a confiabilidade do serviço.
2. As prioridades são discutidas e negociadas com a equipe de desenvolvimento, e um plano de execução é acordado.
3. As equipes de desenvolvimento de produto e SRE participam e auxiliam umas às outras na refatoração de partes do serviço ou na implementação de recursos adicionais.

Essa fase geralmente varia mais em duração e quantidade de esforço. Quanto tempo e esforço essa fase envolverá depende da disponibilidade de tempo de engenharia para refatoração, da maturidade e complexidade do serviço no início da revisão e de inúmeros outros fatores.

Formação

A responsabilidade pela gestão de um serviço em produção é geralmente assumida por toda uma equipa SRE. Para garantir que a equipa esteja preparada, os revisores do SRE que lideraram o PRR se apropriam do treinamento da equipa, que inclui a documentação necessária para dar suporte ao serviço. Normalmente com a ajuda e participação da equipa de desenvolvimento, esses engenheiros organizam uma série de sessões de treinamento e exercícios. A instrução pode incluir:

- Visão geral do projeto
- Aprofundamento em vários fluxos de solicitação no sistema
- Uma descrição da configuração de produção • Exercícios práticos para vários aspectos das operações do sistema

Ao final do treinamento, a equipa do SRE deve estar preparada para gerenciar o atendimento. Vídeo.

Onboarding A

fase de Treinamento desbloqueia a integração do serviço pela equipa SRE. Envolve uma transferência progressiva de responsabilidades e propriedade de vários aspectos de produção do serviço, incluindo partes das operações, o processo de gerenciamento de mudanças, direitos de acesso e assim por diante. A equipa SRE continua a focar-se nas várias áreas de produção mencionadas anteriormente. Para completar a transição, a equipa de desenvolvimento deve estar disponível para fazer backup e aconselhar a equipa SRE por um período de tempo enquanto ela se acomoda no gerenciamento da produção para o serviço. Essa relação torna-se a base para o trabalho contínuo entre as equipes.

Melhoria contínua Os serviços

ativos mudam continuamente em resposta a novas demandas e condições, incluindo solicitações de usuários para novos recursos, dependências em evolução do sistema e atualizações de tecnologia, além de outros fatores. A equipa de SRE deve manter os padrões de confiabilidade do serviço diante dessas mudanças, impulsionando a melhoria contínua. A equipa SRE responsável naturalmente aprende mais sobre o serviço no decorrer da operação do serviço, revisando novas mudanças, respondendo a incidentes e, especialmente, ao realizar análises post-mortem/causa raiz. Essa expertise é compartilhada com a equipa de desenvolvimento como sugestões e propostas de mudanças no serviço sempre que novos recursos, componentes e dependências podem ser adicionados ao serviço. As lições do gerenciamento do serviço também contribuem para as melhores práticas, que estão documentadas no Guia de Produção e em outros lugares.

Envolvendo-se com Shakespeare

Inicialmente, os desenvolvedores do serviço Shakespeare eram responsáveis pelo produto, incluindo o transporte do pager para resposta a emergências. No entanto, com a crescente utilização do serviço e o crescimento da receita advinda do serviço, o suporte SRE tornou-se desejável. O produto já foi lançado, então a SRE realizou uma Revisão de Prontidão de Produção. Uma das coisas que eles descobriram foi que os painéis não estavam cobrindo completamente algumas das métricas definidas no SLO, então isso precisava ser corrigido. Depois que todos os problemas que foram arquivados foram corrigidos, a SRE assumiu o pager do serviço, embora dois desenvolvedores também estivessem na rotação de plantão. Os desenvolvedores estão participando da reunião semanal de plantão discutindo os problemas da semana passada e como lidar com manutenções em larga escala futuras ou desligamentos de clusters. Além disso, planos futuros para o serviço agora são discutidos com os SREs para garantir que novos lançamentos ocorram perfeitamente (embora a lei de Murphy esteja sempre procurando oportunidades para estragar isso).

Evoluindo o Modelo Simples de PRR: Engajamento Antecipado

Até agora, discutimos a Revisão de Prontidão de Produção como ela é usada no Modelo PRR Simples, que é limitado a serviços que já entraram na fase de Lançamento.

Existem várias limitações e custos associados a este modelo. Por exemplo:

- A comunicação adicional entre as equipes pode aumentar alguma sobrecarga de processo para a equipe de desenvolvimento e sobrecarga cognitiva para os revisores de SRE. • Os revisores de SRE certos devem estar disponíveis e capazes de gerenciar seu tempo e prioridades em relação aos seus compromissos existentes.
- O trabalho feito pelos SREs deve ser altamente visível e suficientemente revisado pela equipe de desenvolvimento para garantir o compartilhamento efetivo do conhecimento. Os SREs devem trabalhar essencialmente como parte da equipe de desenvolvimento, e não como uma unidade externa.

No entanto, as principais limitações do Modelo PRR decorrem do fato de que o serviço é lançado e servido em escala, e o envolvimento do SRE começa muito tarde no ciclo de vida do desenvolvimento. Se o PRR ocorresse mais cedo no ciclo de vida do serviço, a oportunidade do SRE para remediar problemas potenciais no serviço aumentaria consideravelmente. Como resultado, o sucesso do envolvimento do SRE e o sucesso futuro do próprio serviço provavelmente melhorariam. As desvantagens resultantes podem representar um desafio significativo para o sucesso do envolvimento do SRE e o sucesso futuro do próprio serviço.

Candidatos ao Early Engagement O Early

Engagement Model introduz o SRE mais cedo no ciclo de vida de desenvolvimento para obter vantagens adicionais significativas. A aplicação do Early Engagement Model requer a identificação da importância e/ou valor comercial de um serviço no início do ciclo de vida do desenvolvimento e a determinação se o serviço terá escala ou complexidade suficiente para se beneficiar da experiência do SRE. Os serviços aplicáveis geralmente têm as seguintes características:

- O serviço implementa novas funcionalidades significativas e fará parte de um sistema existente já gerido pelo SRE.
- O serviço é uma reescrita significativa ou alternativa a um sistema existente, visando os mesmos casos de uso.
- A equipe de desenvolvimento procurou o conselho do SRE ou abordou o SRE para aquisição lançar.

O Early Engagement Model essencialmente mergulha os SREs no processo de desenvolvimento. O foco da SRE continua o mesmo, embora os meios para alcançar um melhor serviço de produção sejam diferentes. A SRE participa do Design e das fases posteriores, eventualmente assumindo o serviço a qualquer momento durante ou após a fase de Construção. Este modelo é baseado na colaboração ativa entre as equipes de desenvolvimento e SRE.

Benefícios do Modelo de Engajamento Antecipado

Embora o Modelo de Engajamento Antecipado envolva certos riscos e desafios discutidos anteriormente, a experiência e a colaboração adicionais de SRE durante todo o ciclo de vida do produto criam benefícios significativos em comparação com um engajamento iniciado posteriormente no ciclo de vida do serviço.

Fase de

projeto A colaboração do SRE durante a fase de projeto pode evitar que vários problemas ou incidentes ocorram posteriormente na produção. Embora as decisões de projeto possam ser revertidas ou retificadas posteriormente no ciclo de vida do desenvolvimento, essas mudanças têm um alto custo em termos de esforço e complexidade. Os melhores incidentes de produção são aqueles que nunca acontecem!

Ocasionalmente, trocas difíceis levam à seleção de um projeto menos do que ideal. A participação na fase de Design significa que os SREs estão cientes dos trade-offs e fazem parte da decisão de escolher uma opção menos do que ideal. O envolvimento precoce do SRE visa minimizar futuras disputas sobre escolhas de design, uma vez que o serviço esteja em produção.

Construção e

implementação A fase de construção aborda aspectos de produção, como instrumentação e métricas, controles operacionais e de emergência, uso de recursos e eficiência. Durante esta fase, o SRE pode influenciar e melhorar a implementação recomendando bibliotecas e componentes existentes específicos ou ajudando a construir certos controles no sistema. A participação do SRE nesta fase ajuda a facilitar as operações no futuro e permite que o SRE ganhe experiência operacional antes do lançamento.

Lançar

O SRE também pode ajudar a implementar padrões e controles de inicialização amplamente usados. Por exemplo, o SRE pode ajudar a implementar uma configuração de “lançamento escuro”, na qual parte do tráfego de usuários existentes é enviada para o novo serviço, além de ser enviada para o serviço de produção ao vivo. As respostas do novo serviço são “escuras”, pois são jogadas fora e não são exibidas aos usuários. Práticas como lançamentos obscuros permitem que a equipe obtenha insights operacionais, resolva problemas sem afetar os usuários existentes e reduza o risco de encontrar problemas após o lançamento. Um lançamento suave é imensamente útil para manter a carga operacional baixa e manter o ímpeto de desenvolvimento após o lançamento. Interrupções em torno do lançamento podem facilmente resultar em mudanças de emergência no código-fonte e na produção, e interromper o trabalho da equipe de desenvolvimento em recursos futuros.

Pós-lançamento

Ter um sistema estável no momento do lançamento geralmente leva a menos prioridades conflitantes para a equipe de desenvolvimento em termos de escolha entre melhorar a confiabilidade do serviço ou adicionar novos recursos. Nas fases posteriores do serviço, as lições das fases anteriores podem informar melhor a refatoração ou o redesenho.

Com um envolvimento estendido, a equipe SRE pode estar pronta para assumir o novo serviço muito mais cedo do que é possível com o Modelo Simple PRR. O envolvimento mais longo e mais próximo entre o SRE e as equipes de desenvolvimento também cria um relacionamento colaborativo que pode ser sustentado a longo prazo. Um relacionamento positivo entre as equipes promove um sentimento mútuo de solidariedade e ajuda a SRE a estabelecer a propriedade da responsabilidade de produção.

Desengatando-se de um

serviço Às vezes, um serviço não garante um gerenciamento completo da equipe do SRE – essa determinação pode ser feita após o lançamento, ou o SRE pode se envolver com um serviço, mas nunca o assumir oficialmente. Este é um resultado positivo, pois o serviço foi projetado para ser confiável e de baixa manutenção, podendo, portanto, permanecer com a equipe de desenvolvimento.

Também é possível que o SRE se envolva antecipadamente com um serviço que não atenda aos níveis de uso projetados. Nesses casos, o esforço de SRE gasto é simplesmente parte do risco geral do negócio que acompanha os novos projetos e um pequeno custo em relação ao sucesso dos projetos que atendem à escala esperada. A equipe de SRE pode ser reatribuída e as lições aprendidas podem ser incorporadas ao processo de contratação.

Desenvolvimento de Serviços em Evolução: Estruturas e SRE Plataforma

O Early Engagement Model avançou na evolução do engajamento SRE além do Simple PRR Model, que se aplicava apenas a serviços que já haviam sido lançados. No entanto, ainda havia progresso a ser feito no dimensionamento do envolvimento do SRE para o próximo nível, projetando para confiabilidade.

Lições aprendidas

Ao longo do tempo, o modelo de engajamento SRE descrito até agora produziu vários padrões distintos:

- A integração de cada serviço exigia dois ou três SREs e normalmente durava dois ou três trimestres. Os prazos de entrega para um PRR foram relativamente altos (quartos de distância). O nível de esforço exigido foi proporcional ao número de serviços sob revisão e foi limitado pelo número insuficiente de SREs disponíveis para realizar PRRs. Essas condições levaram à serialização de takeovers de serviço e priorização de serviço estrita.
- Devido às diferentes práticas de software nos serviços, cada recurso de produção foi implementado de forma diferente. Para atender aos padrões orientados por PRR, os recursos geralmente precisavam ser reimplementados especificamente para cada serviço ou, na melhor das hipóteses, uma vez para cada pequeno subconjunto de código de compartilhamento de serviços. Essas reimplementações foram um desperdício de esforço de engenharia. Um exemplo canônico é a implementação de estruturas de log funcionalmente semelhantes repetidamente na mesma linguagem porque diferentes serviços não implementaram a mesma estrutura de codificação.
- Uma análise de problemas e interrupções de serviço comuns revelou certos padrões, mas não havia como replicar facilmente correções e melhorias nos serviços. Exemplos típicos incluem situações de sobrecarga de serviço e hot-spotting de dados. • As contribuições da engenharia de software do SRE geralmente eram locais para o serviço. Assim, construir soluções genéricas para serem reutilizadas era difícil. Como consequência, não havia uma maneira fácil de implementar novas lições aprendidas por equipes individuais de SRE e práticas recomendadas em todos os serviços que já haviam sido integrados.

Fatores externos que afetam o SRE

Os fatores externos tradicionalmente pressionam a organização do SRE e seus recursos de várias maneiras.

O Google está cada vez mais seguindo a tendência do setor de migrar para microsserviços.¹ Como resultado, o número de solicitações de suporte SRE e a cardinalidade de serviços para suporte aumentaram. Como cada serviço tem um custo operacional fixo básico, mesmo serviços simples exigem mais pessoal. Os microsserviços também implicam em uma expectativa de lead time menor para implantação, o que não era possível com o modelo PRR anterior (que tinha lead time de meses).

Contratar SREs experientes e qualificados é difícil e caro. Apesar do enorme esforço da organização de recrutamento, nunca há SREs suficientes para apoiar todos os serviços que precisam de sua experiência. Uma vez que os SREs são contratados, seu treinamento também é um processo mais demorado do que o típico para engenheiros de desenvolvimento.

Por fim, a organização do SRE é responsável por atender às necessidades do grande e crescente número de equipes de desenvolvimento que ainda não desfrutam de suporte direto do SRE.

Este mandato exige estender o modelo de suporte SRE muito além do conceito original e do modelo de engajamento.

Em direção a uma solução estrutural: frameworks

Para responder eficazmente a estas condições, tornou-se necessário desenvolver um modelo que permitisse os seguintes princípios:

Práticas recomendadas

codificadas A capacidade de comprometer o que funciona bem na produção com o código, para que os serviços possam simplesmente usar esse código e se tornarem “prontos para produção” por design.

Soluções reutilizáveis

Implementações comuns e facilmente compartilháveis de técnicas usadas para mitigar problemas de escalabilidade e confiabilidade.

Uma plataforma de produção comum com uma superfície de controle comum

Conjuntos uniformes de interfaces para instalações de produção, conjuntos uniformes de controles operacionais e monitoramento, registro e configuração uniformes para todos os serviços.

Automação mais fácil e sistemas mais inteligentes

Uma superfície de controle comum que permite automação e sistemas inteligentes em um nível que antes não era possível. Por exemplo, os SREs podem receber prontamente uma visão única de

¹ Consulte a página da Wikipedia sobre microsserviços em <http://en.wikipedia.org/wiki/Microservices>.

informações relevantes para uma interrupção, em vez de coletar e analisar manualmente principalmente dados brutos de fontes diferentes (logs, dados de monitoramento e assim por diante).

Com base nesses princípios, foi criado um conjunto de estruturas de serviço e plataforma suportadas por SRE, uma para cada ambiente que suportamos (Java, C++, Go). Os serviços criados usando essas estruturas compartilham implementações projetadas para funcionar com a plataforma suportada pelo SRE e são mantidas pelo SRE e pelas equipes de desenvolvimento.

A principal mudança trazida pelos frameworks foi permitir que as equipes de desenvolvimento de produtos projetassem aplicativos usando a solução de framework que foi criada e abençoada pelo SRE, em vez de adaptar o aplicativo às especificações do SRE após o fato ou adaptar mais SREs para dar suporte a um serviço, que era marcadamente diferente de outros serviços do Google.

Um aplicativo normalmente comprehende alguma lógica de negócios, que por sua vez depende de vários componentes de infraestrutura. As preocupações de produção do SRE estão amplamente focadas nas partes relacionadas à infraestrutura de um serviço. As estruturas de serviço implementam o código de infraestrutura de forma padronizada e abordam várias preocupações de produção.

Cada preocupação é encapsulada em um ou mais módulos de estrutura, cada um dos quais fornece uma solução coesa para um domínio de problema ou dependência de infraestrutura. Os módulos de estrutura abordam as várias preocupações do SRE enumeradas anteriormente, como:

- Instrumentação e métricas
- Registro de solicitações
- Sistemas de controle envolvendo gerenciamento de tráfego e carga

A SRE constrói módulos de framework para implementar soluções canônicas para a área de produção em questão. Como resultado, as equipes de desenvolvimento podem focar na lógica de negócios, pois o framework já cuida do uso correto da infraestrutura.

Um framework é essencialmente uma implementação prescritiva para usar um conjunto de componentes de software e uma maneira canônica de combinar esses componentes. A estrutura também pode expor recursos que controlam vários componentes de maneira coesa. Por exemplo, uma estrutura pode fornecer o seguinte:

- Lógica de negócios organizada como componentes semânticos bem definidos que podem ser referenciados usando termos padrão
- Dimensões padrão para instrumentação de monitoramento • Um formato padrão para logs de depuração de solicitação • Um formato de configuração padrão para gerenciamento de rejeição de carga • Capacidade de um único servidor e determinação de “sobrecarga” que podem usar uma medida semanticamente consistente para feedback a vários sistemas de controle

As estruturas fornecem vários ganhos iniciais em consistência e eficiência. Eles liberam os desenvolvedores da necessidade de colar e configurar componentes individuais de uma maneira específica de serviço ad hoc, de maneiras ligeiramente incompatíveis, que precisam ser revisadas manualmente pelos SREs. Eles conduzem uma única solução reutilizável para problemas de produção em todos os serviços, o que significa que os usuários da estrutura acabam com a mesma implementação comum e diferenças mínimas de configuração.

O Google oferece suporte a várias linguagens importantes para o desenvolvimento de aplicativos, e os frameworks são implementados em todas essas linguagens. Embora diferentes implementações da estrutura (digamos, em C++ versus Java) não possam compartilhar código, o objetivo é expor a mesma API, comportamento, configuração e controles para funcionalidades idênticas. Portanto, as equipes de desenvolvimento podem escolher a plataforma de linguagem que atenda às suas necessidades e experiência, enquanto os SREs ainda podem esperar o mesmo comportamento familiar na produção e ferramentas padrão para gerenciar o serviço.

Novos benefícios de serviço e gerenciamento

A abordagem estrutural, baseada em estruturas de serviço e uma plataforma de produção e superfície de controle comuns, forneceu uma série de novos benefícios.

Reduz significativamente a sobrecarga operacional

Uma plataforma de produção construída sobre frameworks com convenções mais fortes reduziu significativamente a sobrecarga operacional, pelos seguintes motivos:

- Ele suporta testes de conformidade fortes para estrutura de codificação, dependências, testes, guias de estilo de codificação e assim por diante. Essa funcionalidade também melhora a privacidade dos dados do usuário, testes e conformidade de segurança.
- Possui implantação, monitoramento e automação de serviços integrados para todos os serviços.
- Facilita o gerenciamento de um grande número de serviços, especialmente microsserviços, que estão crescendo em número. • Permite uma implantação muito mais rápida: uma ideia pode se transformar em SRE totalmente implantado

nível de qualidade de produção em questão de dias!

Supporte universal por design

O crescimento constante no número de serviços no Google significa que a maioria desses serviços não pode garantir o envolvimento do SRE nem ser mantida pelos SREs. Independentemente disso, os serviços que não recebem suporte total ao SRE podem ser criados para usar os recursos de produção desenvolvidos e mantidos por SREs. Essa prática efetivamente quebra a barreira de pessoal do SRE. A ativação de ferramentas e padrões de produção compatíveis com SRE para todas as equipes melhora a qualidade geral do serviço no Google. Além disso, todos os serviços que

implementados com frameworks se beneficiam automaticamente das melhorias feitas ao longo do tempo nos módulos de frameworks.

Engajamentos mais rápidos e menores

A abordagem dos frameworks resulta em uma execução mais rápida do PRR porque podemos contar com:

- Recursos de serviço integrados como parte da implementação da estrutura •

Integração de serviço mais rápida (geralmente realizada por um único SRE durante um trimestre)

- Menos carga cognitiva para as equipes de SRE que gerenciam serviços criados usando estruturas

Essas propriedades permitem que as equipes de SRE reduzam o esforço de avaliação e qualificação para integração de serviços, mantendo um alto nível de qualidade de produção de serviços.

Um novo modelo de engajamento baseado em responsabilidade

compartilhada O modelo original de engajamento do SRE apresentava apenas duas opções: suporte total ao SRE ou aproximadamente nenhum engajamento do SRE.²

Uma plataforma de produção com uma estrutura de serviço comum, convenções e infraestrutura de software possibilitou que uma equipe de SRE fornecesse suporte para a infraestrutura de “plataforma”, enquanto as equipes de desenvolvimento fornecem suporte de plantão para problemas funcionais com o serviço, ou seja, para bugs no código do aplicativo. Sob esse modelo, os SREs assumem a responsabilidade pelo desenvolvimento e manutenção de grandes partes da infraestrutura de software de serviço, particularmente sistemas de controle como rejeição de carga, sobrecarga, automação, gerenciamento de tráfego, registro e monitoramento.

Esse modelo representa um afastamento significativo da forma como o gerenciamento de serviços foi originalmente concebido de duas maneiras principais: envolve um novo modelo de relacionamento para a interação entre o SRE e as equipes de desenvolvimento e um novo modelo de pessoal para o gerenciamento de serviços suportado pelo SRE.³

2 Ocionalmente, houve compromissos de consultoria por equipes de SRE com alguns serviços não integrados, mas as consultas foram uma abordagem de melhor esforço e limitadas em número e escopo.

3 O novo modelo de gestão de serviços muda o modelo de pessoal do SRE de duas maneiras: (1) porque muita tecnologia de serviço é comum, reduz o número de SREs necessários por serviço; (2) permite a criação de plataformas de produção com separação de preocupações entre o suporte à plataforma de produção (feito por SREs) e o suporte à lógica de negócios específica do serviço, que permanece com a equipe de desenvolvimento. Essas equipes de plataformas são formadas com base na necessidade de manter a plataforma, e não na contagem de serviços, e podem ser compartilhadas entre os produtos.

Conclusão

A confiabilidade do serviço pode ser aprimorada por meio do engajamento do SRE, em um processo que inclui revisão sistemática e aprimoramento de seus aspectos de produção. A abordagem sistemática inicial do Google SRE, a Simple Production Readiness Review, avançou na padronização do modelo de engajamento do SRE, mas só era aplicável a serviços que já haviam entrado na fase de lançamento.

Com o tempo, a SRE ampliou e aprimorou esse modelo. O Early Engagement Model envolveu o SRE no início do ciclo de vida de desenvolvimento para “projetar para confiabilidade”.

À medida que a demanda por experiência em SRE continuou a crescer, a necessidade de um modelo de engajamento mais escalável tornou-se cada vez mais evidente. Frameworks para serviços de produção foram desenvolvidos para atender a essa demanda: padrões de código baseados nas melhores práticas de produção foram padronizados e encapsulados em frameworks, de modo que o uso de frameworks tornou-se uma forma recomendada, consistente e relativamente simples de construir serviços prontos para produção.

Todos os três modelos de engajamento descritos ainda são praticados no Google. No entanto, a adoção de estruturas está se tornando uma influência proeminente na criação de serviços prontos para produção no Google, além de expandir profundamente a contribuição do SRE, reduzindo a sobrecarga de gerenciamento de serviços e melhorando a qualidade do serviço de linha de base em toda a organização.

PARTE V

Conclusões

Tendo coberto muito terreno em termos de como o SRE funciona no Google e como os princípios e práticas que desenvolvemos podem ser aplicados a outras organizações em nosso campo, agora parece apropriado voltar nossa visão para o [Capítulo 33, Lições aprendidas de outros setores](#), para examinar como as práticas da SRE se comparam a outras indústrias onde a confiabilidade é extremamente importante.

Finalmente, o vice-presidente de engenharia de confiabilidade de sites do Google, Benjamin Lutch, escreve sobre a evolução do SRE ao longo de sua carreira em [sua conclusão](#), examinando o SRE através das lentes de algumas observações sobre a indústria da aviação.

CAPÍTULO 33

Lições aprendidas de outras indústrias

**Escrito por Jennifer Peto
Editado por Betsy Beyer**

Um mergulho profundo na cultura e nas práticas de SRE no Google naturalmente leva à questão de como outros setores gerenciam seus negócios para confiabilidade. A compilação deste livro no Google SRE criou uma oportunidade de conversar com vários engenheiros do Google sobre suas experiências de trabalho anteriores em vários outros campos de alta confiabilidade para abordar as seguintes questões comparativas:

- Os princípios usados na Engenharia de Confiabilidade do Site também são importantes fora do Google, ou outros setores abordam os requisitos de alta confiabilidade de maneiras marcadamente diferentes?
- Se outras indústrias também aderirem aos princípios SRE, como os princípios se manifestam?
- Quais são as semelhanças e diferenças na implementação desses princípios?
- Que fatores determinam semelhanças e diferenças na implementação? • O que o Google e o setor de tecnologia podem aprender com essas comparações?

Vários princípios fundamentais para a Engenharia de Confiabilidade do Site no Google são discutidos ao longo deste texto. Para simplificar nossa comparação das melhores práticas em outros setores, destilamos esses conceitos em quatro temas principais:

- Testes de Preparação e Desastres
- Cultura Postmortem

- Automação e Redução de Custos Operacionais • Tomada de Decisões Estruturadas e Racionais

Este capítulo apresenta os setores que traçamos o perfil e os veteranos do setor que entrevistamos. Definimos os principais temas de SRE, discutimos como esses temas são implementados no Google e damos exemplos de como esses princípios se revelam em outros setores para fins comparativos. Concluímos com alguns insights e discussões sobre os padrões e antipadrões que descobrimos.

Conheça nossos veteranos da indústria

Peter Dahl é engenheiro-chefe do Google. Anteriormente, ele trabalhou como empreiteiro de defesa em vários sistemas de alta confiabilidade, incluindo muitos GPS de veículos aéreos e com rodas e sistemas de orientação inercial. As consequências de um lapso de confiabilidade em tais sistemas incluem mau funcionamento ou perda do veículo e as consequências financeiras associadas a essa falha.

Mike Doherty é engenheiro de confiabilidade de sites do Google. Ele trabalhou como salva-vidas e treinador de salva-vidas por uma década no Canadá. A confiabilidade é absolutamente essencial por natureza neste campo, porque vidas estão em risco todos os dias.

Erik Gross é atualmente engenheiro de software no Google. Antes de ingressar na empresa, ele passou sete anos projetando algoritmos e códigos para os lasers e sistemas usados para realizar cirurgia refrativa ocular (por exemplo, LASIK). Este é um campo de alto risco e alta confiabilidade, no qual muitas lições relevantes para a confiabilidade em face das regulamentações governamentais e risco humano foram aprendidas à medida que a tecnologia recebeu a aprovação da FDA, melhorou gradualmente e finalmente se tornou onipresente.

Gus Hartmann e Kevin Greer têm experiência no setor de telecomunicações, incluindo a manutenção do sistema de resposta a emergências E911.¹ Atualmente, Kevin é engenheiro de software da equipe do Google Chrome e Gus é engenheiro de sistemas da equipe de engenharia corporativa do Google. As expectativas dos usuários do setor de telecomunicações exigem alta confiabilidade. As implicações de um lapso de serviço vão desde a inconveniência do usuário devido a uma interrupção do sistema até fatalidades se o E911 cair.

Ron Heiby é Gerente de Programa Técnico para Engenharia de Confiabilidade de Site no Google. Ron tem experiência em desenvolvimento para telefones celulares, dispositivos médicos e indústria automotiva. Em alguns casos, ele trabalhou em componentes de interface dessas indústrias (por exemplo, em um dispositivo para permitir que leituras de ECG² em ambulâncias fossem transmitidas pela rede de telefonia digital sem fio). Nessas indústrias, o impacto

1 E911 (911 aprimorado): linha de resposta a emergências nos EUA que aproveita os dados de localização.

2 Leituras de eletrocardiograma: <https://en.wikipedia.org/wiki/Electrocardiography>.

de um problema de confiabilidade pode variar de danos aos negócios incorridos por recalls de equipamentos a impactos indiretos na vida e na saúde (por exemplo, pessoas que não recebem a atenção médica de que precisam se o ECG não puder se comunicar com o hospital).

Adrian Hilton é engenheiro de coordenação de lançamento do Google. Anteriormente, ele trabalhou em aeronaves militares do Reino Unido e dos EUA, aviônicos navais e sistemas de gerenciamento de lojas de aeronaves e sistemas de sinalização ferroviária do Reino Unido. A confiabilidade é fundamental neste espaço porque o impacto dos incidentes varia de perda multimilionária de equipamentos a lesões e fatalidades.

Eddie Kennedy é gerente de projetos da equipe de experiência global do cliente do Google e engenheiro mecânico por formação. Eddie passou seis anos trabalhando como engenheiro de processo Six Sigma Black Belt em uma fábrica que fabrica diamantes sintéticos. Esta indústria é caracterizada por um foco incansável na segurança, porque as exigências extremas de temperatura e pressão do processo representam um alto nível de perigo para os trabalhadores diariamente.

John Li é atualmente engenheiro de confiabilidade do site no Google. John trabalhou anteriormente como administrador de sistemas e desenvolvedor de software em uma empresa comercial proprietária no setor financeiro. As questões de confiabilidade no setor financeiro são levadas muito a sério porque podem levar a sérias consequências fiscais.

Dan Sheridan é engenheiro de confiabilidade do site no Google. Antes de ingressar na empresa, trabalhou como consultor de segurança na indústria nuclear civil no Reino Unido. A confiabilidade é importante na indústria nuclear porque um incidente pode ter sérias repercussões: interrupções podem gerar milhões por dia em perda de receita, enquanto os riscos para os trabalhadores e para a comunidade são ainda mais terríveis, ditando tolerância zero para falhas. A infraestrutura nuclear é projetada com uma série de dispositivos à prova de falhas que interrompem as operações antes que um incidente de qualquer magnitude seja atingido.

Jeff Stevenson é atualmente gerente de operações de hardware do Google. Ele tem experiência anterior como engenheiro nuclear na Marinha dos EUA em um submarino. As apostas de confiabilidade na Marinha nuclear são altas - os problemas que surgem no caso de incidentes variam de equipamentos danificados a impactos ambientais de longa data e possíveis perdas de vidas.

Matthew Toia é um Site Reliability Manager focado em sistemas de armazenamento. Antes do Google, ele trabalhou no desenvolvimento de software e implantação de sistemas de software de controle de tráfego aéreo. Os efeitos de incidentes neste setor variam de inconvenientes para passageiros e companhias aéreas (por exemplo, voos atrasados, aviões desviados) até a perda potencial de vidas em caso de acidente. A defesa em profundidade é uma estratégia fundamental para evitar falhas catastróficas.

Agora que você conheceu nossos especialistas e adquiriu uma compreensão de alto nível de por que a confiabilidade é importante em seus respectivos campos anteriores, vamos nos aprofundar nos quatro temas principais de confiabilidade.

Testes de Preparação e Desastres

"Esperança não é uma estratégia." Este grito de guerra da equipe SRE do Google resume o que queremos dizer com preparação e teste de desastres. A cultura SRE está sempre vigilante e constantemente questionando: O que poderia dar errado? Que ação podemos tomar para resolver esses problemas antes que eles levem a uma interrupção ou perda de dados? Nossos exercícios anuais de Testes de Desastres e Recuperação (DiRT) procuram abordar essas questões de frente [Kri12]. Nos exercícios DiRT, os SREs levam os sistemas de produção ao limite e infligem interrupções reais para:

- Garantir que os sistemas reajam da maneira que achamos que reagirão
- Determinar fraquezas inesperadas
- Descobrir maneiras de tornar os sistemas mais robustos para evitar o descontrole

interrupções lideradas

Várias estratégias para testar a prontidão para desastres e garantir a preparação em outros setores surgiram de nossas conversas. As estratégias incluíam o seguinte:

- Foco organizacional implacável na segurança • Atenção aos detalhes
- Capacidade de giro
- Simulações e simulações ao vivo
- Treinamento e certificação • Foco obsessivo na coleta e design detalhados de requisitos • Defesa em profundidade

Foco organizacional implacável na segurança

Este princípio é particularmente importante em um contexto de engenharia industrial. De acordo com Eddie Kennedy, que trabalhava em uma fábrica onde os trabalhadores enfrentavam riscos de segurança, "toda reunião de gerenciamento começava com uma discussão sobre segurança". A indústria manufatureira se prepara para o inesperado estabelecendo processos altamente definidos que são seguidos rigorosamente em todos os níveis da organização. É fundamental que todos os funcionários levem a segurança a sério e que os trabalhadores se sintam autorizados a falar se e quando algo parecer errado. No caso de energia nuclear, aeronaves militares e indústrias de sinalização ferroviária, os padrões de segurança para software são bem detalhados (por exemplo, UK Defense Standard 00-56, IEC 61508, IEC513, US DO-178B/C e DO-254) e

os níveis de confiabilidade para tais sistemas são claramente identificados (por exemplo, Nível de Integridade de Segurança (SIL) 1–4),³ com o objetivo de especificar abordagens aceitáveis para entregar um produto.

Atenção aos detalhes

Do tempo que passou na Marinha dos EUA, Jeff Stevenson lembra de uma aguda consciência de como a falta de diligência na execução de pequenas tarefas (por exemplo, manutenção de óleo lubrificante) poderia levar a grandes falhas de submarinos. Um pequeno descuido ou erro pode ter grandes efeitos. Os sistemas são altamente interconectados, portanto, um acidente em uma área pode afetar vários componentes relacionados. A Marinha nuclear se concentra na manutenção de rotina para garantir que pequenos problemas não se tornem uma bola de neve.

A utilização do Swing

Capacity System na indústria de telecomunicações pode ser altamente imprevisível. A capacidade absoluta pode ser prejudicada por eventos imprevisíveis, como desastres naturais, bem como eventos grandes e previsíveis, como as Olimpíadas. De acordo com Gus Hartmann, a indústria lida com esses incidentes implantando a capacidade de oscilação na forma de um SOW (switch on wheels), um escritório de telecomunicações móvel. Esse excesso de capacidade pode ser implementado em uma emergência ou em antecipação a um evento conhecido que provavelmente sobrecarregará o sistema.

Os problemas de capacidade também se tornam inesperados em questões não relacionadas à capacidade absoluta. Por exemplo, quando o número de telefone privado de uma celebridade vazou em 2005 e milhares de fãs tentaram ligar para ela simultaneamente, o sistema de telecomunicações exibiu sintomas semelhantes a um DDoS ou erro de roteamento maciço.

Simulações e simulações ao vivo

Os testes de recuperação de desastres do Google têm muito em comum com as simulações e simulações ao vivo que são o foco principal de muitos dos setores estabelecidos que pesquisamos. As consequências potenciais de uma interrupção do sistema determinam se o uso de uma simulação ou uma simulação ao vivo é apropriado. Por exemplo, Matthew Toia ressalta que a indústria da aviação não pode realizar um teste ao vivo “em produção” sem colocar em risco equipamentos e passageiros. Em vez disso, eles empregam simuladores extremamente realistas com feeds de dados ao vivo, nos quais as salas de controle e os equipamentos são modelados nos mínimos detalhes para garantir uma experiência realista sem colocar pessoas reais em risco. Gus Hartmann relata que a indústria de telecomunicações normalmente se concentra em simulações ao vivo centradas na sobrevivência de furacões e outras emergências climáticas. Tal modelagem levou à criação de instalações à prova de intempéries com geradores dentro do edifício capazes de durar mais que um

tempestade.

3 https://en.wikipedia.org/wiki/Safety_integrity_level

A Marinha nuclear dos EUA usa uma mistura de exercícios de pensamento “e se” e exercícios ao vivo. De acordo com Jeff Stevenson, os treinos ao vivo envolvem “realmente quebrar coisas reais, mas com parâmetros de controle. Os treinos ao vivo são realizados religiosamente, todas as semanas, dois a três dias por semana.” Para a Marinha nuclear, exercícios de pensamento são úteis, mas não suficientes para se preparar para incidentes reais. As respostas devem ser praticadas para que não sejam esquecidas.

De acordo com Mike Doherty, os salva-vidas enfrentam exercícios de teste de desastres mais parecidos com uma experiência de “cliente misterioso”. Normalmente, um gerente de instalação trabalha com uma criança ou um salva-vidas incógnito em treinamento para encenar um afogamento simulado. Esses cenários são conduzidos para serem o mais realistas possível para que os salva-vidas não sejam capazes de diferenciar entre emergências reais e encenadas.

Treinamento e Certificação Nossas

entrevistas sugerem que o treinamento e a certificação são particularmente importantes quando vidas estão em jogo. Por exemplo, Mike Doherty descreveu como os salva-vidas completam uma rigorosa certificação de treinamento, além de um processo periódico de recertificação. Os cursos incluem componentes de condicionamento físico (por exemplo, um salva-vidas deve ser capaz de segurar alguém mais pesado do que ele com os ombros fora da água), componentes técnicos como primeiros socorros e RCP e elementos operacionais (por exemplo, se um salva-vidas entrar na água, como outros membros da equipe respondem?). Cada instalação também tem treinamento específico para o local, porque o salva-vidas em uma piscina é muito diferente do salva-vidas em uma praia à beira do lago ou no oceano.

Foco na coleta e projeto detalhados de requisitos Alguns dos engenheiros que entrevistamos discutiram a importância da coleta detalhada de requisitos e documentos de projeto. Essa prática foi particularmente importante ao trabalhar com dispositivos médicos. Em muitos desses casos, o uso real ou a manutenção do equipamento não é da responsabilidade dos projetistas de produtos. Assim, os requisitos de uso e manutenção devem ser obtidos de outras fontes.

Por exemplo, de acordo com Erik Gross, as máquinas de cirurgia ocular a laser são projetadas para serem o mais infalíveis possível. Assim, é particularmente importante solicitar os requisitos dos cirurgiões que realmente utilizam essas máquinas e dos técnicos responsáveis pela sua manutenção. Em outro exemplo, o ex-contratado de defesa Peter Dahl descreveu uma cultura de projeto muito detalhada na qual a criação de um novo sistema de defesa geralmente exigia um ano inteiro de projeto, seguido por apenas três semanas de escrita do código para atualizar o projeto. Ambos os exemplos são marcadamente diferentes da cultura de lançamento e iteração do Google, que promove uma taxa de mudança muito mais rápida com um risco calculado. Outras indústrias (por exemplo, a indústria médica e militar, conforme discutido anteriormente) têm pressões, apetites de risco e requisitos muito diferentes, e seus processos são muito informados por essas circunstâncias.

Defesa em profundidade e amplitude Na

indústria de energia nuclear, a defesa em profundidade é um elemento chave para a preparação [AIEA12]. Os reatores nucleares apresentam redundância em todos os sistemas e implementam uma metodologia de projeto que exige sistemas de fallback atrás dos sistemas primários em caso de falha. O sistema foi projetado com várias camadas de proteção, incluindo uma barreira física final para a liberação de radioatividade ao redor da própria planta. A defesa em profundidade é particularmente importante na indústria nuclear devido à tolerância zero para falhas e incidentes.

Cultura post mortem

Ação corretiva e preventiva (CAPA)⁴ é um conceito bem conhecido para melhorar a confiabilidade que se concentra na investigação sistemática das causas-raiz de problemas ou riscos identificados, a fim de evitar a recorrência. Este princípio é incorporado pela forte cultura da SRE de autópsias irrepreensíveis. Quando algo dá errado (e dada a escala, complexidade e rápida taxa de mudança no Google, algo inevitavelmente dará errado), é importante avaliar todos os itens a seguir:

- O que aconteceu •
- A eficácia da resposta • O que faríamos
- diferente da próxima vez • Que ações serão
- tomadas para garantir que um determinado incidente não aconteça

novamente

Este exercício é realizado sem apontar o dedo para qualquer indivíduo. Em vez de atribuir a culpa, é muito mais importante descobrir o que deu errado e como, como organização, nos mobilizaremos para garantir que isso não aconteça novamente. Pensar em quem pode ter causado a interrupção é contraproducente. Postmortems são realizados após os incidentes e publicados nas equipes de SRE para que todos possam se beneficiar das lições aprendidas.

Nossas entrevistas revelaram que muitas indústrias realizam uma versão da autópsia (embora muitas não usem esse apelido específico, por razões óbvias). A motivação por trás desses exercícios parece ser o principal diferencial entre as práticas da indústria.

Muitas indústrias são fortemente regulamentadas e são responsabilizadas por autoridades governamentais específicas quando algo dá errado. Essa regulamentação é especialmente arraigada quando os riscos de fracasso são altos (por exemplo, vidas estão em jogo). Agente governamental relevante

4 https://en.wikipedia.org/wiki/Corrective_and_preventive_action

incluem a FCC (telecomunicações), FAA (aviação), OSHA (indústrias industriais e químicas), FDA (dispositivos médicos) e as várias Autoridades Nacionais Competentes na UE . regulamentado.

As considerações de segurança são outro fator motivador por trás das autópsias. Nas indústrias de manufatura e química, o risco de ferimentos ou morte está sempre presente devido à natureza das condições necessárias para produzir o produto final (alta temperatura, pressão, toxicidade e corrosividade, para citar alguns). Por exemplo, a Alcoa apresenta uma cultura de segurança digna de nota. O ex-CEO Paul O'Neill exigiu que a equipe o notificasse dentro de 24 horas de qualquer lesão que perdesse um dia de trabalho. Ele até distribuiu seu número de telefone residencial para os trabalhadores no chão de fábrica para que eles pudessem alertá-lo pessoalmente sobre questões de segurança.⁶

As apostas são tão altas nas indústrias de manufatura e química que mesmo “quase acidentes” – quando um determinado evento poderia ter causado sérios danos, mas não causou – são cuidadosamente examinados. Esses cenários funcionam como um tipo de postmortem preventivo. De acordo com VM Brasseur em uma palestra proferida no YAPC NA 2015, “Existem vários quase acidentes em quase todos os desastres e crises de negócios, e normalmente eles são ignorados no momento em que ocorrem. Erro latente, mais uma condição de habilitação, equivale a coisas que não funcionam exatamente como você planejou” [Bra15]. Quase acidentes são efetivamente desastres esperando para acontecer. Por exemplo, cenários em que um trabalhador não segue o procedimento operacional padrão, um funcionário sai do caminho no último segundo para evitar um respingo ou um derramamento na escada não é limpo, todos representam quase acidentes e oportunidades para aprender e melhorar. Da próxima vez, o funcionário e a empresa podem não ter tanta sorte. O CHIRP do Reino Unido (Programa de Relatórios Confidenciais para Aviação e Marítima) busca aumentar a conscientização sobre tais incidentes em toda a indústria, fornecendo um ponto central de relatórios onde o pessoal da aviação e marítimo pode relatar quase acidentes confidencialmente. Relatórios e análises desses quase acidentes são publicados em boletins periódicos.

Salva-vidas tem uma cultura profundamente enraizada de análise pós-incidente e planejamento de ações. Mike Doherty brinca: “Se os pés de um salva-vidas entrarem na água, haverá papelada!” Uma descrição detalhada é necessária após qualquer incidente na piscina ou na praia. No caso de incidentes graves, a equipe examina coletivamente o incidente de ponta a ponta, discutindo o que deu certo e o que deu errado. As mudanças operacionais são então feitas com base nessas descobertas, e o treinamento geralmente é programado para ajudar as pessoas a criar confiança em sua capacidade de lidar com um incidente semelhante no futuro. Em casos de incidentes particularmente chocantes ou traumáticos, um conselheiro é trazido ao local para ajudar a equipe a lidar com as consequências psicológicas. Os salva-vidas podem ter sido bem

5 https://en.wikipedia.org/wiki/Competent_authority

6 <http://ehstoday.com/safety/nsc-2013-oneill-exemplifies-safety-leadership>.

preparados para o que aconteceu na prática, mas podem sentir que não fizeram um trabalho adequado. Semelhante ao Google, o salva-vidas adota uma cultura de análise de incidentes sem culpa. Os incidentes são caóticos e muitos fatores contribuem para qualquer incidente. Neste campo, não é útil colocar a culpa em um único indivíduo.

Automatizando o trabalho repetitivo e operacional A sobrecarga

Em sua essência, os engenheiros de confiabilidade do site do Google são engenheiros de software com baixa tolerância para trabalho reativo repetitivo. Está fortemente enraizado em nossa cultura evitar a repetição de uma operação que não agrupa valor a um serviço. Se uma tarefa pode ser automatizada, por que você executaria um sistema em trabalho repetitivo de baixo valor? A automação reduz a sobrecarga operacional e libera tempo para que nossos engenheiros avaliem e melhorem proativamente os serviços que oferecem suporte.

Os setores que pesquisamos eram variados em termos de se, como e por que adotaram a automação. Certas indústrias confiavam mais nos humanos do que nas máquinas. Durante o mandato de nosso veterano da indústria, a Marinha nuclear dos EUA evitou a automação em favor de uma série de intertravamentos e procedimentos administrativos. Por exemplo, de acordo com Jeff Stevenson, a operação de uma válvula exigia um operador, um supervisor e um membro da tripulação ao telefone com o oficial de serviço de engenharia encarregado de monitorar a resposta à ação tomada. Essas operações eram muito manuais devido à preocupação de que um sistema automatizado não detectasse um problema que um humano definitivamente notaria.

As operações em um submarino são governadas por uma cadeia de decisão humana confiável – uma série de pessoas, em vez de um indivíduo. A Marinha nuclear também estava preocupada com o fato de a automação e os computadores se moverem tão rapidamente que são capazes de cometer um erro grande e irreparável. Quando você está lidando com reatores nucleares, uma abordagem metódica lenta e constante é mais importante do que realizar uma tarefa rapidamente.

De acordo com John Li, a indústria de negociação proprietária tornou-se cada vez mais cautelosa em sua aplicação de automação nos últimos anos. A experiência mostrou que a automação configurada incorretamente pode causar danos significativos e incorrer em grandes perdas financeiras em um período muito curto de tempo. Por exemplo, em 2012, o Knight Capital Group encontrou uma “falha de software” que levou a uma perda de US\$ 440 milhões em apenas algumas horas . trader tentando manipular o mercado com meios automatizados. Embora o mercado tenha se recuperado rapidamente, o Flash Crash resultou em uma

7 Veja “FATOS, Seção B” para a discussão do software Knight e Power Peg em [Sec13].

perda na magnitude de trilhões de dólares em apenas 30 minutos. executar tarefas muito rapidamente, e a velocidade pode ser negativa se essas tarefas forem configuradas incorretamente.

⁸ Os computadores podem

Em contraste, algumas empresas adotam a automação precisamente porque os computadores agem mais rapidamente do que as pessoas. De acordo com Eddie Kennedy, a eficiência e a economia monetária são fundamentais na indústria manufatureira, e a automação fornece meios para realizar tarefas de forma mais eficiente e econômica. Além disso, a automação é geralmente mais confiável e repetível do que o trabalho realizado manualmente por humanos, o que significa que produz padrões de qualidade mais altos e tolerâncias mais rígidas. Dan Sheridan discutiu a automação como implantada na indústria nuclear do Reino Unido. Aqui, uma regra geral determina que, se uma planta precisar responder a uma determinada situação em menos de 30 minutos, essa resposta deve ser automatizada.

Na experiência de Matt Toia, a indústria da aviação aplica a automação de forma seletiva. Por exemplo, o failover operacional é executado automaticamente, mas quando se trata de outras tarefas, o setor confia na automação apenas quando ela é verificada por um ser humano.

Embora a indústria empregue uma boa quantidade de monitoramento automático, as implementações reais do sistema de controle de tráfego aéreo devem ser inspecionadas manualmente por humanos.

De acordo com Erik Gross, a automação tem sido bastante eficaz na redução do erro do usuário na cirurgia ocular a laser. Antes da cirurgia LASIK ser realizada, o médico mede o paciente usando um teste de refração ocular. Originalmente, o médico digitava os números e apertava um botão, e o laser funcionava corrigindo a visão do paciente.

No entanto, erros de entrada de dados podem ser um grande problema. Este processo também implicou a possibilidade de misturar os dados do paciente ou números confusos para o olho esquerdo e direito.

A automação agora diminui muito a chance de os humanos cometerem um erro que afete a visão de alguém. Uma verificação de sanidade computadorizada de dados inseridos manualmente foi a primeira grande melhoria automatizada: se um operador humano insere medições fora de um intervalo esperado, a automação sinaliza prontamente e com destaque esse caso como incomum.

Outras melhorias automatizadas seguiram este desenvolvimento: agora a íris é fotografada durante o teste de refração ocular preliminar. Na hora de realizar a cirurgia, a íris do paciente corresponde automaticamente à íris da foto, eliminando assim a possibilidade de misturar os dados do paciente. Quando esta solução automatizada foi implementada, uma classe inteira de erros médicos desapareceu.

⁸ "Os reguladores culpam o algoritmo do computador pelo 'flash crash' do mercado de ações", Computerworld, <http://www.computerworld.com/article/2516076/financial-it/regulators-blame-computer-algorithm-for-stock-market—flash-crash-.html>.

Tomada de Decisões Estruturadas e Racionais

No Google em geral e na Engenharia de confiabilidade do site em particular, os dados são essenciais. A equipa aspira a tomar decisões de forma estruturada e racional, garantindo que:

- A base para a decisão é acordada antecipadamente, em vez de justificada ex post facto
- As entradas para a decisão são claras •

Quaisquer suposições são explicitamente

declaradas • Decisões baseadas em dados vencem decisões baseadas em sentimentos, palpites ou opiniões
ção do funcionário mais antigo da sala

O Google SRE opera sob a suposição básica de que todos na equipe:

- Tem em mente os melhores interesses dos usuários de um serviço
- Pode descobrir como proceder com base nos dados disponíveis

As decisões devem ser informadas em vez de prescritivas, e são feitas sem deferência a opiniões pessoais – mesmo a da pessoa mais sênior na sala, que Eric Schmidt e Jonathan Rosenberg chamam de “HiPPO”, para “Opinião da pessoa mais bem paga”. ” [Sch14].

A tomada de decisão em diferentes indústrias varia muito. Aprendemos que algumas indústrias usam uma abordagem de se não está quebrado, não conserte... nunca. As indústrias que apresentam sistemas cujo projeto exigiu muita reflexão e esforço são frequentemente caracterizadas por uma relutância em mudar a tecnologia subjacente. Por exemplo, a indústria de telecomunicações ainda usa switches de longa distância que foram implementados na década de 1980. Por que eles confiam na tecnologia desenvolvida há algumas décadas? Esses switches “são praticamente à prova de balas e extremamente redundantes”, de acordo com Gus Hartmann. Conforme relatado por Dan Sheridan, a indústria nuclear também é lenta para mudar. Todas as decisões são sustentadas pelo pensamento: se funciona agora, não mude.

Muitos setores se concentram fortemente em manuais e procedimentos, em vez de na solução de problemas em aberto. Cada cenário humanamente concebível é capturado em uma lista de verificação ou no “fichário”. Quando algo dá errado, esse recurso é a fonte oficial de como reagir. Essa abordagem prescritiva funciona para indústrias que evoluem e se desenvolvem de forma relativamente lenta, porque os cenários do que pode dar errado não estão em constante evolução devido a atualizações ou alterações do sistema. Essa abordagem também é comum em indústrias em que o nível de habilidade dos trabalhadores pode ser limitado, e a melhor maneira de garantir que as pessoas respondam adequadamente em uma emergência é fornecer um conjunto simples e claro de instruções.

Outros setores também adotam uma abordagem clara e orientada por dados para a tomada de decisões. Na experiência de Eddie Kennedy, os ambientes de pesquisa e fabricação são caracterizados por uma cultura de experimentação rigorosa que depende muito da formulação e teste de hipóteses. Essas indústrias realizam regularmente experimentos controlados para garantir que uma determinada mudança produza o resultado esperado em um nível estatisticamente significativo e que nada inesperado ocorra. As alterações só são implementadas quando os dados gerados pelo experimento apoiam a decisão.

Finalmente, alguns setores, como o trading proprietário, dividem a tomada de decisões para gerenciar melhor o risco. De acordo com John Li, essa indústria conta com uma equipe de fiscalização separada dos comerciantes para garantir que riscos indevidos não sejam assumidos em busca de lucro. A equipe de fiscalização é responsável por monitorar os eventos no pregão e interromper a negociação se os eventos saírem do controle. Se ocorrer uma anormalidade no sistema, a primeira resposta da equipe de fiscalização é desligar o sistema. Conforme colocado por John Li, “Se não estamos negociando, não estamos perdendo dinheiro. Também não estamos ganhando dinheiro, mas pelo menos não estamos perdendo dinheiro.” Somente a equipe de fiscalização pode trazer o sistema de volta, apesar de quão doloroso um atraso possa parecer para os traders que estão perdendo uma oportunidade potencialmente lucrativa.

Conclusões

Muitos dos princípios que são fundamentais para a Engenharia de Confiabilidade do Site no Google são evidentes em uma ampla variedade de setores. As lições já aprendidas por indústrias bem estabelecidas provavelmente inspiraram algumas das práticas em uso no Google hoje.

Uma das principais conclusões de nossa pesquisa intersetorial foi que, em muitas partes de seu negócio de software, o Google tem um apetite maior por velocidade do que os players na maioria dos outros setores. A capacidade de se mover ou mudar rapidamente deve ser ponderada em relação às diferentes implicações de uma falha. Nas indústrias nuclear, de aviação ou médica, por exemplo, as pessoas podem ser feridas ou até mesmo morrer no caso de uma interrupção ou falha. Quando as apostas são altas, uma abordagem conservadora para alcançar alta confiabilidade é garantida.

No Google, andamos constantemente na corda bamba entre as expectativas dos usuários por alta confiabilidade versus um foco nítido em mudanças rápidas e inovação. Embora o Google leve a sério a confiabilidade, devemos adaptar nossas abordagens à nossa alta taxa de mudança. Conforme discutido nos capítulos anteriores, muitos de nossos negócios de software, como o Search, tomam decisões conscientes sobre o quão confiável “suficientemente confiável” realmente é.

O Google tem essa flexibilidade na maioria de nossos produtos e serviços de software, que operam em um ambiente no qual vidas não correm risco direto se algo der errado.

Portanto, podemos usar ferramentas como orçamentos de erro ([“Motivação para orçamentos de erro” na página 33](#)) como um meio de “financiar” uma cultura de inovação e assunção de riscos calculados. Em essência, o Google adaptou princípios de confiabilidade conhecidos que foram, em muitos casos, desenvolvidos e aprimorados em outros setores para criar sua própria cultura de confiabilidade exclusiva, que aborda uma equação complicada que equilibra escala, complexidade e velocidade com alta confiabilidade.

CAPÍTULO 34**Conclusão**

Escrito por Benjamin Lutch¹
Editado por Betsy Beyer

Li este livro com enorme orgulho. Desde que comecei a trabalhar na Excite no início dos anos 90, onde meu grupo era uma espécie de grupo SRE neandertal apelidado de “Operações de Software”, passei minha carreira atrapalhando o processo de construção de sistemas. À luz de minhas experiências ao longo dos anos na indústria de tecnologia, é incrível ver como a ideia de SRE se enraizou no Google e evoluiu tão rapidamente. O SRE cresceu de algumas centenas de engenheiros quando entrei para o Google em 2006 para mais de 1.000 pessoas hoje, espalhados por uma dúzia de sites e executando o que considero a infraestrutura de computação mais interessante do planeta.

Então, o que permitiu que a organização SRE do Google evoluísse na última década para manter essa enorme infraestrutura de maneira inteligente, eficiente e escalável? Eu acho que a chave para o sucesso esmagador da SRE é a natureza dos princípios pelos quais ela opera.

As equipes SRE são construídas para que nossos engenheiros dividam seu tempo entre dois tipos de trabalho igualmente importantes. Os turnos de plantão da equipe dos SREs, que envolvem colocar as mãos em torno dos sistemas, observar onde e como esses sistemas quebram e entender os desafios, como melhor dimensioná-los. Mas também temos tempo para refletir e decidir o que construir para tornar esses sistemas mais fáceis de gerenciar. Em essência, temos o prazer de desempenhar os papéis de piloto e engenheiro/designer. Nossas experiências executando uma infraestrutura de computação massiva são codificadas em código real e, em seguida, empacotadas como um produto discreto.

¹ Vice-presidente, Engenharia de confiabilidade do site, da Google, Inc.

Essas soluções podem ser facilmente usadas por outras equipes de SRE e, em última análise, por qualquer pessoa no Google (ou mesmo fora do Google... pense no Google Cloud!) que queira usar ou aprimorar a experiência que acumulamos e os sistemas que criamos.

Quando você aborda a construção de uma equipe ou de um sistema, idealmente sua base deve ser um conjunto de regras ou axiomas que sejam gerais o suficiente para serem imediatamente úteis, mas que permanecerão relevantes no futuro. Muito do que Ben Treynor Sloss delineou na introdução deste livro representa exatamente isso: um conjunto de responsabilidades flexíveis, principalmente à prova de futuro, que permanecem 10 anos após terem sido concebidos, apesar das mudanças e do crescimento que a infraestrutura do Google e a equipe de SRE sofreram .

À medida que o SRE cresceu, notamos algumas dinâmicas diferentes em jogo. A primeira é a natureza consistente das principais responsabilidades e preocupações do SRE ao longo do tempo: nossos sistemas podem ser 1.000 vezes maiores ou mais rápidos, mas, em última análise, eles ainda precisam permanecer confiáveis, flexíveis, fáceis de gerenciar em caso de emergência, bem monitorados e capacidade planejada. Ao mesmo tempo, as atividades típicas realizadas pelo SRE evoluem por necessidade à medida que os serviços do Google e as competências do SRE amadurecem. Por exemplo, o que antes era um objetivo de “construir um painel para 20 máquinas” pode agora ser “automatizar a descoberta, a construção de painéis e alertas sobre uma frota de dezenas de milhares de máquinas”.

Para aqueles que não estiveram nas trincheiras do SRE na última década, uma analogia entre como o SRE pensa sobre sistemas complexos e como a indústria aeronáutica abordou o voo de avião é útil para conceituar como o SRE evoluiu e amadureceu ao longo do tempo. Embora os riscos de fracasso entre os dois setores sejam muito diferentes, certas semelhanças essenciais são verdadeiras.

Imagine que você quisesse voar entre duas cidades há cem anos. Seu avião provavelmente tinha um único motor (dois, se você tivesse sorte), algumas malas de carga e um piloto.

O piloto também ocupou o papel de mecânico e, possivelmente, atuou adicionalmente como carregador e descarregador de carga. O cockpit tinha espaço para o piloto e, se você tivesse sorte, um co-piloto/navegador. Seu pequeno avião saltaria de uma pista com bom tempo e, se tudo corresse bem, você subiria lentamente pelos céus e eventualmente poussaria em outra cidade, talvez a algumas centenas de quilômetros de distância. A falha de qualquer um dos sistemas do avião foi catastrófica, e não era inédito para um piloto ter que sair da cabine para realizar reparos em voo! Os sistemas que alimentavam o cockpit eram essenciais, simples e frágeis, e provavelmente não eram redundantes.

Avance cem anos para um enorme 747 parado na pista. Centenas de passageiros estão carregando em ambos os andares, enquanto toneladas de carga estão sendo carregadas simultaneamente no porão abaixo. O avião está repleto de sistemas confiáveis e redundantes. É um modelo de segurança e confiabilidade; na verdade, você está mais seguro no ar do que no solo em um carro. Seu avião decolará de uma linha pontilhada em uma pista em um continente e poussará facilmente em uma linha pontilhada em outra pista a 6.000 milhas de distância, exatamente

cronograma - dentro de minutos de seu tempo de pouso previsto. Mas dê uma olhada no cockpit e o que você encontra? Apenas dois pilotos novamente!

Como todos os outros elementos da experiência de voo – segurança, capacidade, velocidade e confiabilidade – cresceram tão bem, enquanto ainda há apenas dois pilotos? A resposta a essa pergunta é um grande paralelo de como o Google aborda os sistemas enormes e fantasticamente complexos que o SRE executa. As interfaces para os sistemas operacionais do avião são bem pensadas e acessíveis o suficiente para que aprender a pilotá-los em condições normais não seja uma tarefa intransponível. No entanto, essas interfaces também oferecem flexibilidade suficiente e as pessoas que as operam são suficientemente treinadas para que as respostas às emergências sejam robustas e rápidas. O cockpit foi projetado por pessoas que entendem de sistemas complexos e como apresentá-los aos humanos de uma forma consumível e escalável. Os sistemas subjacentes ao cockpit têm todas as mesmas propriedades discutidas neste livro: disponibilidade, otimização de desempenho, gerenciamento de mudanças, monitoramento e alerta, planejamento de capacidade e resposta a emergências.

Em última análise, o objetivo da SRE é seguir um curso semelhante. Uma equipe de SRE deve ser o mais compacta possível e operar em um alto nível de abstração, contando com muitos sistemas de backup como à prova de falhas e APIs inteligentes para se comunicar com os sistemas. Ao mesmo tempo, a equipe de SRE também deve ter um conhecimento abrangente dos sistemas – como eles operam, como falham e como responder a falhas – que vem da operação diária.

APÊNDICE A

Tabela de Disponibilidade

A disponibilidade geralmente é calculada com base em quanto tempo um serviço ficou indisponível algum período. Assumindo que não há tempo de inatividade planejado, a [Tabela A-1](#) indica quanta inatividade tempo é permitido para atingir um determinado nível de disponibilidade.

Tabela A-1. Tabela de disponibilidade

Nível de disponibilidade	Janela de indisponibilidade permitida						
	por ano	por trimestre	por mês	por semana	por dia	9 dias 4,5 dias	
90%	36,5 dias	21,6 horas		3 dias	16,8 horas	2,4 horas	6 minutos
95%	18,25 dias			1,5 dias	8,4 horas	1,2 horas	3 minutos
99%	3,65 dias			7,2 horas	1,68 horas	14,4 minutos	36 segundos
99,5%	1,83 dias	10,8 horas		3,6 horas	50,4 minutos	7,20 minutos	18 segundos
99,9%	8,76 horas	2,16 horas		43,2 minutos	10,1 minutos	1,44 minutos	3,6 segundos
99,95%	4,38 horas	1,08 horas		21,6 minutos	5,04 minutos	43,2 segundos	1,8 segundos
99,99%		52,6 minutos	12,96 minutos	4,32 minutos	60,5 segundos	8,64 segundos	0,36 segundos
99,999%		5,26 minutos	1,30 minutos	25,9 segundos	6,05 segundos	0,87 segundos	0,04 segundos

Usar uma métrica de indisponibilidade agregada (ou seja, "X% de todas as operações com falha") é mais útil do que focar na duração da interrupção para serviços que podem estar parcialmente disponíveis—por exemplo, devido a várias réplicas, das quais apenas algumas estão indisponíveis—e para serviços cuja carga varia ao longo de um dia ou semana, em vez de permanecendo constante.

Consulte as Equações 3-1 e 3-2 no [Capítulo 3](#) para cálculos.

APÊNDICE B

Uma coleção de práticas recomendadas para Serviços de Produção

Escrito por Ben Treynor Sloss
Editado por Betsy Beyer

Falhar com sensatez

Higienize e valide as entradas de configuração e responda a entradas implausíveis, continuando a operar no estado anterior e alertando para o recebimento de entradas incorretas.

A entrada incorreta geralmente se enquadra em uma dessas categorias:

Dados incorretos

Valide a sintaxe e, se possível, a semântica. Observe dados vazios e dados parciais ou truncados (por exemplo, alerte se a configuração for N% menor que a versão anterior).

Dados atrasados

Isso pode invalidar os dados atuais devido a tempos limite. Alerta bem antes que os dados expirem.

Falhar de uma maneira que preserve a função, possivelmente à custa de ser excessivamente permissivo ou excessivamente simplista. Descobrimos que geralmente é mais seguro que os sistemas continuem funcionando com sua configuração anterior e aguardem a aprovação de um humano antes de usar os novos dados, talvez inválidos.

Exemplos

Em 2005, o sistema global de balanceamento de carga e latência de DNS do Google recebeu um arquivo de entrada DNS vazio como resultado de permissões de arquivo. Ele aceitou esse arquivo vazio e veiculou NXDOMAIN por seis minutos para todas as propriedades do Google. Em resposta, o sistema agora realiza várias verificações de integridade em novas configurações, incluindo a confirmação da presença de IPs virtuais para google.com, e continuará atendendo as entradas DNS anteriores até receber um novo arquivo que passe nas verificações de entrada.

Em 2009, dados incorretos (mas válidos) levaram o Google a marcar toda a Web como contendo malware [May09]. Um arquivo de configuração contendo a lista de URLs suspeitos foi substituído por um único caractere de barra (/), que correspondia a todos os URLs. Verifica alterações drásticas no tamanho do arquivo e verifica se a configuração está correspondendo a sites que provavelmente não contêm malware, o que impediria que isso chegasse à produção.

Lançamentos progressivos

Os lançamentos não emergenciais devem prosseguir em etapas. As alterações de configuração e binária apresentam riscos, e você reduz esse risco aplicando a alteração a pequenas frações de tráfego e capacidade de uma só vez. O tamanho do seu serviço ou lançamento, bem como seu perfil de risco, informarão os percentuais de capacidade de produção para os quais o lançamento é direcionado e o intervalo de tempo adequado entre as etapas. Também é uma boa ideia realizar diferentes estágios em diferentes geografias, a fim de detectar problemas relacionados a ciclos de tráfego diurnos e diferenças de mix de tráfego geográfico.

Os lançamentos devem ser supervisionados. Para garantir que nada inesperado ocorra durante a implantação, ela deve ser monitorada pelo engenheiro que executa a fase de implantação ou, de preferência, por um sistema de monitoramento comprovadamente confiável. Se um comportamento inesperado for detectado, reverta primeiro e diagnostique depois para minimizar o tempo médio de recuperação.

De ne SLOs como um usuário

Meça a disponibilidade e o desempenho em termos que importam para um usuário final. Veja o [Capítulo 4](#) para mais discussão.

Exemplo

A medição das taxas de erro e latência no cliente Gmail, em vez de no servidor, resultou em uma redução substancial em nossa avaliação da disponibilidade do Gmail e provocou alterações no cliente e no código do servidor Gmail. O resultado foi que o Gmail passou de cerca de 99,0% disponível para mais de 99,9% disponível em poucos anos.

Orçamentos de erro

Equilibrar confiabilidade e ritmo de inovação com orçamentos de erro (veja “[Motivação para Orçamentos de Erro](#)” na página 33), que definem o nível aceitável de falha de um serviço, ao longo de um determinado período; muitas vezes usamos um mês. Um orçamento é simplesmente 1 menos o SLO de um serviço; por exemplo, um serviço com uma meta de disponibilidade de 99,99% tem um “orçamento” de 0,01% para indisponibilidade. Desde que o serviço não tenha gasto seu orçamento de erros do mês por meio da taxa de erros em segundo plano mais qualquer tempo de inatividade, a equipe de desenvolvimento está livre (dentro do razoável) para lançar novos recursos, atualizações e assim por diante.

Se o orçamento de erro for gasto, o serviço congelará as alterações (exceto segurança urgente e correções de bugs abordando qualquer causa do aumento de erros) até que o serviço recupere espaço no orçamento ou o mês seja redefinido. Para serviços maduros com um SLO superior a 99,99%, uma redefinição de orçamento trimestral em vez de mensal é apropriada, pois o tempo de inatividade permitido é pequeno.

Os orçamentos errados eliminam a tensão estrutural que poderia se desenvolver entre o SRE e as equipes de desenvolvimento de produtos, fornecendo a eles um mecanismo comum baseado em dados para avaliar o risco de lançamento. Eles também dão às equipes de desenvolvimento de produto e SRE um objetivo comum de desenvolver práticas e tecnologia que permitem inovação mais rápida e mais lançamentos sem “estourar o orçamento”.

Monitoramento

O monitoramento pode ter apenas três tipos de saída:

Páginas

Um humano deve fazer algo agora

Ingressos

Um humano deve fazer algo dentro de alguns dias

Exploração madeireira

Ninguém precisa ver essa saída imediatamente, mas ela está disponível para análise posterior, se necessário

Se for importante o suficiente para incomodar um humano, deve exigir ação imediata (ou seja, página) ou ser tratado como um bug e inserido em seu sistema de rastreamento de bugs. Colocar alertas em e-mail e esperar que alguém leia todos eles e perceba os importantes é o equivalente moral de enviá-los para /dev/null: eles eventualmente serão ignorados. A história demonstra que essa estratégia é um incômodo atraente porque pode funcionar por um tempo, mas depende da eterna vigilância humana, e a interrupção inevitável é, portanto, mais grave quando ocorre.

Pós-morte

As autópsias (veja o [Capítulo 15](#)) devem ser isentas de culpa e focar no processo e na tecnologia, não nas pessoas. Suponha que as pessoas envolvidas em um incidente sejam inteligentes, bem intencionadas e estejam fazendo as melhores escolhas possíveis, de acordo com as informações disponíveis no momento. Segue-se que não podemos "consertar" as pessoas, mas sim consertar seu ambiente: por exemplo, melhorar o design do sistema para evitar classes inteiras de problemas, tornar as informações apropriadas facilmente disponíveis e validar automaticamente as decisões operacionais para dificultar a colocação sistemas em estados perigosos.

Planejamento de capacidade

Provisão para lidar com uma interrupção simultânea planejada e não planejada, sem tornar a experiência do usuário inaceitável; isso resulta em uma configuração "N + 2", onde o tráfego de pico pode ser tratado por N instâncias (possivelmente em modo degradado), enquanto as 2 maiores instâncias estão indisponíveis:

- Valide as previsões de demanda anteriores em relação à realidade até que elas correspondam de forma consistente. A divergência implica previsão instável, provisionamento inefficiente e risco de déficit de capacidade.
- Use o teste de carga em vez da tradição para estabelecer a relação recurso-capacidade: um cluster de máquinas X poderia lidar com consultas Y por segundo há três meses, mas ainda pode fazê-lo devido às alterações no sistema?
- Não confunda a carga do primeiro dia com a carga de estado estacionário. Os lançamentos geralmente atraem mais tráfego, enquanto também são o momento em que você deseja dar o melhor de si no produto. Consulte [o Capítulo 27](#) e [o Apêndice E](#).

Sobrecargas e falhas

Os serviços devem produzir resultados razoáveis, mas abaixo do ideal, se sobrecarregados. Por exemplo, a Pesquisa do Google pesquisará uma fração menor do índice e interromperá a veiculação de recursos como o Instant para continuar a fornecer resultados de pesquisa na Web de boa qualidade quando sobrecarregados. O Search SRE testa os clusters de pesquisa na web além de sua capacidade nominal para garantir que eles tenham um desempenho aceitável quando sobrecarregados com tráfego.

Para momentos em que a carga é alta o suficiente para que mesmo as respostas degradadas sejam muito caras para todas as consultas, pratique o corte de carga gracioso, usando enfileiramento bem comportado e tempos limite dinâmicos; consulte o [Capítulo 21](#). Outras técnicas incluem responder a solicitações após um atraso significativo ("tarpitting") e escolher um subconjunto consistente de clientes para receber erros, preservando uma boa experiência do usuário para o restante.

As novas tentativas podem amplificar as baixas taxas de erro em níveis mais altos de tráfego, levando a falhas em cascata (consulte o [Capítulo 22](#)). Responda a falhas em cascata eliminando uma fração do tráfego (incluindo novas tentativas!) upstream do sistema quando a carga total exceder a capacidade total.

Cada cliente que faz um RPC deve implementar backoff exponencial (com jitter) para novas tentativas, para amortecer a amplificação de erros. Os clientes móveis são especialmente problemáticos porque podem haver milhões deles e atualizar seu código para corrigir o comportamento leva um tempo significativo - possivelmente semanas - e exige que os usuários instalem atualizações.

Equipes SRE

As equipes de SRE não devem gastar mais de 50% de seu tempo em trabalho operacional (consulte o [Capítulo 5](#)); estouro operacional deve ser direcionado para a equipe de desenvolvimento do produto. Muitos serviços também incluem os desenvolvedores de produtos na rotação de plantão e no manuseio de tickets, mesmo que não haja estouro no momento. Isso fornece incentivos para projetar sistemas que minimizem ou eliminem a labuta operacional, além de garantir que os desenvolvedores de produtos estejam em contato com o lado operacional do serviço. Uma reunião regular de produção entre os SREs e a equipe de desenvolvimento (consulte o [Capítulo 31](#)) também é útil.

Constatamos que pelo menos oito pessoas precisam fazer parte da equipe de plantão, a fim de evitar fadiga e permitir uma contratação sustentável e baixa rotatividade. Preferencialmente, aqueles de plantão devem estar em duas localizações geográficas bem separadas (por exemplo, Califórnia e Irlanda) para proporcionar uma melhor qualidade de vida, evitando páginas noturnas; neste caso, seis pessoas em cada local é o tamanho mínimo da equipe.

Não espere lidar com mais de dois eventos por turno de plantão (por exemplo, a cada 12 horas): leva tempo para responder e corrigir interrupções, iniciar a autópsia e arquivar os bugs resultantes. Eventos mais frequentes podem degradar a qualidade da resposta e sugerem que

algo está errado com (pelo menos um dos) design do sistema, sensibilidade de monitoramento e resposta a bugs post-mortem.

Ironicamente, se você implementar essas práticas recomendadas, a equipe de SRE pode acabar perdendo a prática de responder a incidentes devido à sua infrequência, fazendo uma longa interrupção de uma curta. Pratique o tratamento de interrupções hipotéticas (consulte “[Desastre Role Playing](#)” na [página 401](#)) rotineiramente e melhore sua documentação de tratamento de incidentes no processo.

APÊNDICE C

Exemplo de Documento de Estado do Incidente

Sobrecarga do Shakespeare Sonnet++: 2015-10-21 Informações

de gerenciamento de incidentes: <http://incident-management-cheat-sheet> (As comunicações

levam a manter o resumo atualizado.)

Resumo: Falha no serviço de pesquisa Shakespeare em cascata devido ao soneto recém-descoberto que não está no índice de pesquisa.

Status: ativo, incidente nº 465

Posto(s) de Comando: #shakespeare no IRC

Hierarquia de Comandos (todos os respondentes)

- Atual Comandante do Incidente: jennifer
 - Líder de operações: docbrown
 - Líder de planejamento: jennifer
 - Líder de comunicação: jennifer
- Próximo Comandante do Incidente: a ser determinado

(Atualize pelo menos a cada quatro horas e na transferência da função de líder de comunicação.)

Status detalhado (última atualização em 2015-10-21 15:28 UTC por jennifer)

Critério de saída:

- Novo soneto adicionado ao corpus de pesquisa de Shakespeare **TODO** • Dentro de disponibilidade (99,99%) e latência (99%ile < 100 ms) SLOs por mais de 30 minutos
- FAÇAM**

Lista de TODO e bugs arquivados:

- Execute o trabalho MapReduce para reindexar o corpus de Shakespeare **CONCLUÍDO**
- Emprestar recursos de emergência para aumentar a capacidade extra **CONCLUÍDO** •
Ativar capacitor de fluxo para equilibrar a carga entre clusters (Bug 5554823) **TODO**

Linha do **tempo do incidente** (mais recente primeiro: os horários estão em UTC)

- 21/10/2015 15:28 UTC jennifer
 - Aumentando a capacidade de atendimento globalmente em
- 2x • 2015-10-21 15:21 UTC jennifer
 - Direcionar todo o tráfego para o cluster sacrificial USA-2 e drenar o tráfego de outros clusters para que eles possam se recuperar de falhas em cascata enquanto executam mais tarefas
- Trabalho de índice MapReduce concluído, aguardando replicação do Bigtable para todos os clusters
- 21/10/2015 15:10 UTC martym
 - Adicionando novo soneto ao corpus de Shakespeare e iniciando o índice MapReduce
- 21/10/2015 15:04 UTC martym
 - Obtém o texto do soneto recém-descoberto de shakespeare-discuss@ mailing
 - Lista
- 2015-10-21 15:01 UTC docbrown
 - Incidente declarado devido a falha em cascata
- 21/10/2015 14:55 UTC docbrown
 - Tempestade de pager, ManyHttp500s em todos os clusters

APÊNDICE D

Exemplo Postmortem

Shakespeare Soneto++ Postmortem (incidente #465)

Data: 21-10-2015

Autores: jennifer, martym, agoogler

Status: Concluído, itens de ação em andamento

Resumo: Shakespeare Pesquise por 66 minutos durante o período de muito interesse em Shakespeare devido à descoberta de um novo soneto.

Impacto: 1 Estimativa de 1,21 bilhão de consultas perdidas, sem impacto na receita.

Raiz: vazamento¹ de recursos para o cluster de sacrifício falha devido a expor que não era essa causa corpus de Shakespeare.

O soneto recém-descoberto usava uma palavra que nunca havia aparecido em uma das obras de Shakespeare, que por acaso era o termo que os usuários procuravam. Em circunstâncias normais, a taxa de falhas de tarefas devido a vazamentos de recursos é baixa o suficiente para passar despercebida.

Gatilho: Bug latente acionado por aumento repentino no tráfego.

Resolução: tráfego direcionado para o cluster de sacrifício e capacidade 10x adicionada para mitigar a falha em cascata. Índice atualizado implantado, resolvendo a interação com o bug latente.

Mantendo a capacidade extra até que o interesse público em novos sonetos passe.

Vazamento de recursos identificado e correção implantada.

1 Impacto é o efeito sobre usuários, receita, etc.

2 Uma explicação das circunstâncias em que este incidente aconteceu. Muitas vezes é útil usar uma técnica como os 5 Porquês [Ohn88] para entender os fatores que contribuem.

Detecção: Borgmon detectou alto nível de HTTP 500s e paginou de plantão.

Itens de ação: 3

Item de ação	Tipo	Proprietário	Incomodar
Atualize o manual com instruções para responder a falhas em cascata	mitigar	jennifer evitar	n/a CONCLUÍDO
Use o capacitor ux para equilibrar a carga entre os clusters	martym Bug 5554823	TODO	
Agende o teste de falha em cascata durante o próximo DiRT	processo	docbrown n/a	TODO
Investigue o índice de execução MR/fusão continuamente	prevenir	jennifer	Bug 5554824 TODO
Conecte o vazamento do descritor de le no subsistema de classificação de pesquisa	evitar	agoogler Bug 5554825	CONCLUÍDO
Adicione recursos de redução de carga à pesquisa de Shakespeare	evitar	agoogler Bug 5554826	TODO
Crie testes de regressão para garantir que os servidores respondam de forma sensata a consultas de morte	prevenir	clarac	Bug 5554827 TODO
Implante o subsistema de classificação de pesquisa atualizado para produzir	prevenir	jennifer	n/a CONCLUÍDO
Congele a produção até 20/11/2015 devido ao esgotamento do orçamento por erro, ou procure exceção devido ao grotesco, inacreditável, bizarro e sem precedentes circunstâncias	outro	docbrown n/a	TODO

Lições aprendidas

O que foi bem

- O monitoramento rapidamente nos alertou sobre a alta taxa (atingindo ~100%) de HTTP 500s
- Corpus de Shakespeare atualizado rapidamente distribuído para todos os clusters

O que deu errado

- Estamos sem prática em responder a falhas em cascata
- Excedemos nosso orçamento de erro de disponibilidade (em várias ordens de magnitude) devido ao excepcional aumento de tráfego que, essencialmente, resultou em falhas

3 As IAs “incríveis” muitas vezes acabam sendo muito extremas ou caras para implementar, e pode ser necessário julgamento para redefiní-las em um contexto mais amplo. Existe o risco de otimizar demais para um problema específico, adicionando monitoramento específico toring/alertando quando mecanismos confiáveis como testes de unidade podem detectar problemas muito mais cedo no desenvolvimento processo.

Onde tivemos sorte⁴

- A lista de discussão dos aficionados de Shakespeare tinha uma cópia do novo soneto disponível • Os logs do servidor tinham rastreamentos de pilha apontando para a exaustão do descritor de arquivo como causa para batida
- A consulta de morte foi resolvida ao enviar um novo índice contendo pesquisa popular prazo

Linha do tempo 5

21/10/2015 (todos os horários UTC)

- 14:51 Notícias informam que um novo soneto shakespeariano foi descoberto em um porta-luvas do Delorean
- 14:53 O tráfego para a pesquisa de Shakespeare aumenta em 88x após a postagem em /r/shakespeare apontar para o mecanismo de pesquisa de Shakespeare como local para encontrar um novo soneto (exceto que ainda não temos o soneto)
- 14:54 INÍCIO DE INTERRUPÇÃO — Os backends de pesquisa começam a derreter sob carga • 14:55 docbrown recebe tempestade de pager, **ManyHttp500s** de todos os clusters • 14:57 Todo o tráfego para a pesquisa de Shakespeare está falhando: consulte [http://monitor/shakespeare?
end_time=20151021T145700](http://monitor/shakespeare?end_time=20151021T145700)
- 14:58 docbrown começa a investigar, encontra taxa de falhas de back-end muito alta • 15:01 **INCIDENTE COMEÇA** docbrown declara o incidente #465 devido a falha em cascata, coordenação em #shakespeare, nomeia jennifer comandante do incidente • 15:02 alguém coincidentemente envia um e-mail para shakespeare-discussão@resonnet discovery, que por acaso está no topo da caixa de entrada de marty • 15:03 jennifer notifica shakespeare-incidents@ lista do incidente • 15:04 marty rastreia o texto do novo soneto e procura documentação sobre atualização do corpus
- 15:06 docbrown descobre que os sintomas de falha são idênticos em todas as tarefas em todas as cluÿters, investigando a causa com base nos logs do aplicativo

⁴ Esta seção é realmente para quase acidentes, por exemplo, "O teletransportador de cabra estava disponível para uso de emergência com outros animais apesar da falta de certificação."

⁵ Um "roteiro" do incidente; use a linha do tempo do incidente do documento de Gerenciamento de Incidentes para começar a preencher a linha do tempo da autópsia e, em seguida, complementar com outras entradas relevantes.

- 15:07 marty m encontra documentação, inicia o trabalho de preparação para atualização do corpus •
- 15:10 marty m adiciona soneto às obras conhecidas de Shakespeare, inicia trabalho de indexação • 15:12
docbrown contata clarac & agoogler (da equipe de desenvolvimento de Shakespeare) para ajudar
com o exame da base de código para possíveis causas
- 15:18 Clarac encontra uma arma fumegante nos logs apontando para a exaustão do descritor de arquivo, confirma
contra o código que existe vazamento se o termo fora do corpus for pesquisado
- 15:20 marty m's index MapReduce job é concluído • 15:21 jennifer e
docbrown decidem aumentar a contagem de instâncias o suficiente para descarregar a carga em instâncias que eles
são capazes de fazer um trabalho apreciável antes de morrer e serem reiniciados
- 15:23 docbrown load balanceia todo o tráfego para o cluster USA-2, permitindo instância
aumento de contagem em outros clusters sem que os servidores falhem imediatamente
- 15:25 marty m começa a replicar o novo índice para todos os clusters • 15:28
docbrown inicia 2x aumento na contagem de instâncias
- 15:32 jennifer altera o balanceamento de carga para aumentar o tráfego para clusters não sacrificais • 15:33
tarefas em clusters não sacrificais começam a falhar, os mesmos sintomas de antes • 15:34 encontrou erro de
ordem de magnitude em cálculos de quadro branco, por exemplo
aumento de contagem
- 15:36 jennifer reverte o balanceamento de carga para sacrificar o cluster USA-2 em preparação para um aumento
adicional global de 5 vezes na contagem de instâncias (para um total de 10 vezes a capacidade inicial)
- 15:36 **OUTAGE MITIGADO**, índice atualizado replicado para todos os clusters
- 15:39 docbrown inicia a segunda onda de aumento de contagem de instâncias para 10x inicial
capacidade
- 15:41 jennifer restabelece o balanceamento de carga em todos os clusters para 1% do tráfego
- Taxas HTTP 500 de clusters não sacrificial 15:43 em taxas nominais, falhas de tarefas intery
mittent em níveis baixos
- 15:45 jennifer equilibra 10% do tráfego em clusters não sacrificial • As taxas HTTP 500 de
clusters não sacrificial 15:47 permanecem dentro do SLO, sem falhas de tarefa
observado
- 15:50 30% do tráfego balanceado em clusters sem sacrifício
- 15:55 50% do tráfego balanceado em clusters sem sacrifício
- **FIM DE INTERRUPÇÃO** 16:00 , todo o tráfego equilibrado em todos os
clusters • **FIM DE INCIDENTE 16:30**, atingiu o critério de saída de 30 minutos nominais
atuação

Informações de apoio: 6

- Painel de monitoramento,

http://monitor/shakespeare?end_time=20151021T160000&duration=7200

6 Informações úteis, links, logs, screenshots, gráficos, logs de IRC, logs de IM, etc.

APÊNDICE E

Lista de verificação de coordenação de lançamento

Esta é a Lista de Verificação de Coordenação de Lançamento original do Google, por volta de 2005, um pouco resumida para fins de brevidade:

Arquitetura

- Esboço de arquitetura, tipos de servidores, tipos de solicitações de clientes •
Solicitações programáticas de clientes

Máquinas e datacenters

- Máquinas e largura de banda, datacenters, redundância N+2, QoS de rede • Novos nomes de domínio, balanceamento de carga DNS

Estimativas de volume, capacidade e desempenho

- Tráfego HTTP e estimativas de largura de banda, lançamento "pico", combinação de tráfego, 6 meses fora •
Teste de carga, teste de ponta a ponta, capacidade por datacenter com latência máxima • Impacto em outros serviços com os quais mais nos importamos • Capacidade de armazenamento

Confiabilidade e failover do sistema

- O que acontece quando:
 - A máquina morre, o rack falha ou o cluster fica offline
 - Falha na rede entre dois datacenters

- Para cada tipo de servidor que se comunica com outros servidores (seus backends):
 - Como detectar quando os back-ends morrem e o que fazer quando eles morrem
 - Como encerrar ou reiniciar sem afetar clientes ou usuários
 - Balanceamento de carga, limitação de taxa, tempo limite, repetição e comportamento de tratamento de erros
- Backup/restauração de dados, recuperação de desastres

Monitoramento e gerenciamento de servidores

- Monitorando o estado interno, monitorando o comportamento de ponta a ponta, gerenciando alertas
- Monitorando o monitoramento
- Alertas e logs financeiramente importantes
- Dicas para executar servidores em ambiente de cluster
- Não trave servidores de e-mail enviando a si mesmo alertas de e-mail em seu próprio código de servidor

Segurança

- Revisão de design de segurança, auditoria de código de segurança, risco de spam, autenticação, SSL
- Visibilidade/controle de acesso de pré-lançamento, vários tipos de listas negras

Automação e tarefas manuais

- Métodos e controle de alterações para atualizar servidores, dados e configurações
- Processo de lançamento, compilações repetíveis, canários sob tráfego ativo, lançamentos em etapas

Problemas de crescimento

- Capacidade sobressalente, crescimento de 10x, alertas de crescimento
- Gargalos de escalabilidade, dimensionamento linear, dimensionamento com hardware, alterações necessárias
- Cache, fragmentação/refragmentação de dados

Dependências externas

- Sistemas de terceiros, monitoramento, rede, volume de tráfego, picos de lançamento
- Degradação graciosa, como evitar a superação acidental de serviços de terceiros
- Jogando bem com parceiros sindicalizados, sistemas de e-mail, serviços no Google

Planejamento de cronograma e lançamento

- Prazos rígidos, eventos externos, segundas ou sextas-feiras •

Procedimentos operacionais padrão para este serviço, para outros serviços

APÊNDICE F

Exemplo de Ata de Reunião de Produção

Data: 23/10/2015

Participantes: agoogler, clarac, docbrown, jennifer, martym

Anúncios:

- Grande interrupção (#465), estourou o orçamento de erro

Revisão do item de ação anterior

- Certificar o teletransportador de cabras para uso com gado (bug 1011101)
 - Não linearidades na aceleração de massa agora previsíveis, devem ser capazes de atingir com precisão em poucos dias.

Revisão de interrupção

- Novo Soneto (interrupção 465)
 - 1,21B de consultas perdidas devido a falha em cascata após interação entre bug latente (descriptor de arquivo vazado em pesquisas sem resultados) + não ter novo soneto no corpus + volume de tráfego sem precedentes e inesperado
 - Bug de vazamento do descriptor de arquivo corrigido (bug 5554825) e implantado para produzir
 - Analisando o uso do capacitor de fluxo para balanceamento de carga (bug 5554823) e o uso de rejeição de carga (bug 5554826) para evitar a recorrência
 - Orçamento de erro de disponibilidade aniquilado; empurra para produzir congelado por 1 mês, a menos que o docbrown possa obter uma exceção com base no fato de que o evento foi bizarro e imprevisível (mas o consenso é que a exceção é improvável)

Eventos de paginação

- AnnotationConsistencyTooEventual: paginado 5 vezes esta semana, provavelmente devido a atraso de replicação entre regiões entre Bigtables.
 - A investigação ainda está em andamento, veja o bug 4821600 — Nenhuma correção esperada em breve, aumentará o limite de consistência aceitável para reduzir alertas não acionáveis

Eventos sem paginação

- Nenhum

Monitoramento de Mudanças e/ou Silêncios

- AnnotationConsistencyTooEventual, limite de atraso aceitável aumentado de 60s a 180s, veja o bug 4821600; TODO (martim).

Mudanças de produção planejadas

- Cluster USA-1 ficando offline para manutenção entre 29/10/2015 e 2015-11-02.
 - Nenhuma resposta necessária, o tráfego será roteado automaticamente para outros clusters em região.

Recursos

- Os recursos emprestados para responder ao incidente do soneto++ serão reduzidos instâncias do servidor e retornar recursos na próxima semana
- Utilização em 60% da CPU, 75% de RAM, 44% de disco (acima de 40%, 70%, 40% no último semana)

Principais métricas de serviço

- Latência **OK 99ile**: 88 ms < 100 ms SLO alvo [30 dias finais] • Disponibilidade **RUIM** : 86,95% < 99,99% SLO alvo [30 dias finais]

Discussão / Atualizações do Projeto

- Lançamento do Projeto Molière em duas semanas.

Novos itens de ação

- TODO(martym): Aumenta o limite AnnotationConsistencyTooEventual . • TODO(docbrown):
Retorna a contagem de instâncias ao normal e retorna os recursos.

Bibliografia

- [Ada15] Bram Adams, Stephany Bellomo, Christian Bird, Tamara Marshall-Keim, Foutse Khomh e Kim Moir, “The Practice and Future of Release Engineering: A Roundtable with Three Release Engineers”, IEEE Soware, vol. 32, não. 2 (março/abril de 2015), pp. 42–49.
- [Agu10] MK Aguilera, “Tropeçando na pesquisa de consenso: mal-entendidos e questões”, em Replicação, Notas de Palestra em Ciência da Computação 5959, 2010.
- [All10] J. Allspaw e J. Robbins, Web Operations: Keeping the Data on Time: O'Reilly, 2010.
- [All12] J. Allspaw, “Blameless PostMortems and a Just Culture”, postagem no blog, 2012.
- [All15] J. Allspaw, “Trade-Offs Under Pressure: Heuristics and Observations of Teams Resolving Internet Service Outages”, Tese de Mestrado, Universidade de Lund, 2015.
- [Ana07] S. Anantharaju, “Automatizando testes de segurança de aplicativos da web”, postagem no blog, julho de 2007.
- [Ana13] R. Ananatharayyan et al., “Photon: Fault-tolerant and Scalable Joining of Fluxos de dados contínuos”, em SIGMOD '13, 2013.
- [And05] A. Andrieux, K. Czajkowski, A. Dan, et al., “Web Services Agreement Speciÿ (Acordo WS)”, Setembro de 2005.
- [Bai13] P. Bailis e A. Ghodsi, “Consistência Eventual Hoje: Limitações, Extensões e Além”, em ACM Queue, vol. 11, não. 3, 2013.
- [Bai83] L. Bainbridge, “Ironias da Automação”, em Automatica, vol. 19, não. 6 de novembro 1983.
- [Bak11] J. Baker et al., “Megastore: Fornecendo Armazenamento Escalável e Altamente Disponível para Serviços Interativos”, em Proceedings of the Conference on Innovative Data System Research, 2011.

- [Bar11] LA Barroso, “Warehouse-Scale Computing: Entrando na década da adolescência”, palestra no 38º Simpósio Anual de Arquitetura de Computadores, vídeo disponível online, 2011.
- [Bar13] LA Barroso, J. Clidaras e U. Hölzle, **The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition**, Morgan & Claypool, 2013.
- [Ben12] C. Bennett e A. Tseitlin, “Macaco do caos lançado na natureza”, blog postagem, julho de 2012.
- [Bla14] M. Bland, “Goto Fail, Heartbleed, and Unit Testing Culture”, postagem no blog, junho 2014.
- [Boc15] L. Bock, **Regras de Trabalho!**, Doze livros, 2015.
- [Bol11] WJ Bolosky, D. Bradshaw, RB Haagens, NP Kusters e P. Li, “**Paxos Replicated State Machines as the Basis of a High-Performance Data Store**”, em Proc. INDE 2011, 2011.
- [Boy13] PG Boysen, “**Just Culture: A Foundation for Balanced Accountability and Segurança do paciente**”, no The Ochsner Journal, outono de 2013.
- [Bra15] VM Brasseur, “**Falha: Por que acontece e como se beneficiar disso**”, YAPC 2015.
- [Bre01] E. Brewer, “**Lições de serviços em escala gigante**”, em IEEE Internet Computing, volume 5, não. 4 de julho/agosto de 2001.
- [Bre12] E. Brewer, “**CAP Doze Anos Depois: Como as “Regras” Mudaram**”, em Computador, v. 45, não. 2 de fevereiro de 2012.
- [Bro15] M. Brooker, “**Recuo Exponencial e Jitter**”, no blog de arquitetura da AWS, março de 2015.
- [Bro95] FP Brooks Jr., “Nenhuma bala de prata — Essência e acidentes de engenharia de software”, em The Mythical Man-Month, Boston: Addison-Wesley, 1995, pp. 180–186.
- [Bru09] J. Brutlag, “**Velocidade Importa**”, no Google Research Blog, junho de 2009.
- [Bul80] GM Bull, The Dartmouth Time-sharing System: Ellis Horwood, 1980.
- [Bur99] M. Burgess, Princípios de Administração de Redes e Sistemas: Wiley, 1999.
- [Bur06] M. Burrows, “**O Serviço Chubby Lock para Sistemas Distribuídos Loosely-Coupled**”, em OSDI '06: Sétimo Simpósio sobre Projeto e Implementação de Sistemas Operacionais, novembro de 2006.
- [Bur16] B. Burns, B. Grant, D. Oppenheimer, E. Brewer e J. Wilkes, “**Borg, Omega e Kubernetes**” em ACM Queue, vol. 14, não. 1, 2016.
- [Cas99] M. Castro e B. Liskov, “**Prática de tolerância a falhas bizantinas**”, em Proc. OSDI 1999, 1999.

- [Cha10] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw e N. Weizenbaum, “FlumeJava: Easy, Efficient Data-Parallel Pipelines”, em ACM SIG-PLAN Conference on Programming Language Design and Implementation, 2010.
- [Cha96] TD Chandra e S. Toueg, “Detetores de falhas não confiáveis para sistemas distribuídos confiáveis”, em J. ACM, 1996.
- [Cha07] T. Chandra, R. Griesemer e J. Redstone, “Paxos Made Live—An Engineering Perspective”, em PODC '07: 26th ACM Symposium on Principles of Distributed Computing, 2007.
- [Cha06] F. Chang et al., “Bigtable: A Distributed Storage System for Structured Data”, em OSDI '06: Sétimo Simpósio sobre Projeto e Implementação de Sistemas Operacionais, novembro de 2006.
- [Chr09] GP Chrousous, “Estresse e Distúrbios do Sistema de Estresse”, em Nature Reviews Endocrinology, vol 5., no. 7 de 2009.
- [Clos53] C. Clos, “Um estudo de redes de comutação sem bloqueio”, em Bell System Technical Journal, vol. 32, não. 2, 1953.
- [Con15] C. Contavalli, W. van der Gaast, D. Lawrence e W. Kumari, “Client Subnet in DNS Queries”, IETF Internet-Dra [Con63] ME Conway, “Design of a Separable Transition-Diagram Compiler”, em Comun. ACM 6, 7 (julho de 1963), 396-408.
- [Con96] P. Conway, “Preservação no Mundo Digital”, relatório publicado pelo Conselho de Bibliotecas e Recursos de Informação, 1996.
- [Coo00] RI Cook, “How Complex Systems Fail”, em Operações Web: O'Reilly, 2010.
- [Cor12] JC Corbett et al., “Spanner: Google's Globally-Distributed Database”, em OSDI '12: Décimo Simpósio sobre Projeto e Implementação de Sistemas Operacionais, outubro de 2012.
- [Cra10] J. Cranmer, “Visualizando a cobertura de código”, postagem no blog, março de 2010.
- [Dea13] J. Dean e LA Barroso, “The Tail at Scale”, em Comunicações da ACM, vol. 56, 2013.
- [Dea04] J. Dean e S. Ghemawat, “MapReduce: Processamento de Dados Simplificado em Grandes Clusters”, em OSDI'04: Sexto Simpósio sobre Projeto e Implementação de Sistemas Operativos, dezembro de 2004.
- [Dea07] J. Dean, “Conselhos de Engenharia de Software da Construção de Sistemas Distribuídos em Grande Escala”, Palestra da classe Stanford CS297, primavera de 2007.
- [Dek02] S. Dekker, “Reconstruindo as contribuições humanas para acidentes: a nova visão sobre erro e desempenho”, em Journal of Safety Research, vol. 33, nº. 3, 2002.
- [Dek14] S. Dekker, The Field Guide to Understanding “Human Error”, 3ª edição: Ashgate, 2014.

- [Dic14] C. Dickson, “[Como abraçar a liberação contínua reduziu a complexidade da mudança](#)”, apresentação no USENIX Release Engineering Summit West 2014, vídeo disponível online.
- [Dur05] J. Durmer e D. Dinges, “[Neurocognitive Consequences of Sleep Deprivation](#)”, em Seminários em Neurologia, vol. 25, não. 1, 2005.
- [Eis16] DE Eisenbud et al., “[Maglev: um balanceador de carga de rede de software rápido e confiável](#)”, in NSDI '16: 13th USENIX Symposium on Networked Systems Design and Implementation, março de 2016.
- [Ere03] JR Erenkrantz, “[Gerenciamento de Liberação em Projetos de Código Aberto](#)”, in Proceedings of the 3rd Workshop on Open Source Software Engineering, Portland, Oregon, maio de 2003.
- [Fis85] MJ Fischer, NA Lynch e MS Paterson, “[Impossibilidade de consenso distribuído com um processo falho](#)”, J. ACM, 1985.
- [Fit12] BW Fitzpatrick e B. Collins-Sussman, Team Geek: A Software Developer's Guide para trabalhar bem com os outros: O'Reilly, 2012.
- [Flo94] S. Floyd e V. Jacobson, “[A Sincronização de Mensagens de Roteamento Periódico](#)”, em Transações IEEE/ACM em Rede, vol. 2, número 2, abril de 1994, pp. 122–136.
- [For10] D. Ford et al, “[Disponibilidade em Sistemas de Armazenamento Globalmente Distribuídos](#)”, em Anais do 9º Simpósio USENIX sobre Projeto e Implementação de Sistemas Operacionais, 2010.
- [Fox99] A. Fox e EA Brewer, “[Harvest, Yield, and Scalable Tolerant Systems](#)”, in Proceedings of the 7th Workshop on Hot Topics in Operating Systems, Rio Rico, Arizona, março de 1999.
- [Fow08] M. Fowler, “[Arquiteturas GUI](#)”, postagem no blog, 2006.
- [Gal78] J. Gall, SYSTEMANTICS: How Systems Really Work and How They Fail, 1ª ed., Pocket, 1977.
- [Gal03] J. Gall, The Systems Bible: The Beginner's Guide to Systems Large and Small, 3ª ed., General Systemantics Press/Liberty, 2003.
- [Gaw09] A. Gawande, The Checklist Manifesto: How to Get Things Right: Henry Holt and Company, 2009.
- [Ghe03] S. Ghemawat, H. Gobioff e ST. Leung, “[O sistema de arquivos do Google](#)”, em 19º Simpósio ACM sobre Princípios de Sistemas Operacionais, outubro de 2003.
- [Gil02] S. Gilbert e N. Lynch, “[Conjectura de Brewer e a Viabilidade de Serviços Web Consistentes, Disponíveis e Tolerantes a Partições](#)”, em ACM SIGACT News, vol. 33, nº. 2, 2002.
- [Gla02] R. Glass, Fatos e Falácias da Engenharia de Software, Addison-Wesley Profissional, 2002.

- [Gol14] W. Golab et al., "Eventually Consistente: Not What You Were Expecting?", dentro Fila ACM, vol. 12, não. 1, 2014.
- [Gra09] P. Graham, "Cronograma do Fabricante, Cronograma do Gerente", postagem no blog, julho de 2009.
- [Gup15] A. Gupta e J. Shute, "Alta disponibilidade em escala massiva: construindo a infraestrutura de dados do Google para anúncios", em Workshop de Business Intelligence para a Real Time Enterprise, 2015.
- [Ham07] J. Hamilton, "Sobre Projetar e Implantar Serviços em Escala de Internet", em Anais da 21ª Conferência de Administração de Sistemas de Grandes Instalações, novembro 2007.
- [Han94] S. Hanks, T. Li, D. Farinacci e P. Traina, "Encapsulamento de roteamento genérico sobre redes IPv4", RFC Informativo da IETF, 1994.
- [Hic11] M. Hickins, "Tape Rescues Google in Lost Email Scare", em dígitos, Wall Street Diário, 1 de março de 2011.
- [Hix15a] D. Hixson, "Planejamento de Capacidade", em ;login:, vol. 40, não. 1 de fevereiro de 2015.
- [Hix15b] D. Hixson, "O lado da engenharia de sistemas da engenharia de confiabilidade do site", em ;login: vol. 40, não. 3 de junho de 2015.
- [Hod13] J. Hodges, "Notas sobre sistemas distribuídos para sangues jovens", postagem no blog, 14 Janeiro de 2013.
- [Hol14] L. Holmwood, "Aplicando Técnicas de Gerenciamento de Alarme Cardíaco ao Seu Atendimento", postagem do blog, 26 de agosto de 2014.
- [Hum06] J. Humble, C. Read, D. North, "The Deployment Production Line", em Proceedings da IEEE Agile Conference, julho de 2006.
- [Hum10] J. Humble e D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation: Addison-Wesley, 2010.
- [Hun10] P. Hunt, M. Konar, FP Junqueira e B. Reed, "ZooKeeper: Coordenação sem espera para sistemas à escala da Internet", em USENIX ATC, 2010.
- [IAEA12] Agência Internacional de Energia Atômica, "Segurança de Usinas Nucleares: Projeto, SSR-2/1", 2012.
- [Jai13] S. Jain et al., "B4: Experiência com uma WAN Definida por Software Implementada Globalmente ", no SIGCOMM '13.
- [Jon15] C. Jones, T. Underwood e S. Nukala, "Contratando Engenheiros de Confiabilidade do Local", em ;login:, vol. 40, não. 3 de junho de 2015.
- [Jun07] F. Junqueira, Y. Mao, e K. Marzullo, "Paxos Clássicos vs. Paxos Rápidos: Caveat Emptor", em Proc. HotDep '07, 2007.
- [Jun11] FP Junqueira, BC Reid e M. Serafini, "Zab: transmissão de alto desempenho para sistemas de backup primário.", em Sistemas e Redes Confiáveis (DSN), 2011 IEEE/IFIP 41ª Conferência Internacional em 27 de junho de 2011: 245–256.

- [Kah11] D. Kahneman, Pensando, Rápido e Lento: Farrar, Straus e Giroux, 2011.
- [Kar97] D. Karger et al., “Hashing consistente e árvores aleatórias: protocolos de cache distribuídos para aliviar pontos quentes na World Wide Web”, em Proc. STOC '97, 29º simpósio anual da ACM sobre teoria da computação, 1997.
- [Kem11] C. Kemper, “Build in the Cloud: How the Build System Works”, Postagem no blog do Google Engineering Tools, agosto de 2011.
- [Ken12] S. Kendrick, “O que nos derruba?”, em ;login:, vol. 37, nº. 5 de outubro de 2012.
- [Kin09] Kincaid, Jason. “Desastre T-Mobile Sidekick: os servidores do Danger falharam e eles não têm backup.” Techcrunch. np, 10 de outubro de 2009. Web. 20 de janeiro de 2015, <http://techcrunch.com/2009/10/10/t-mobile-sidekick-microso s-servers crashed-and-they-dont-have-a-backup>.
- [Kin15] K. Kingsbury, “O problema com carimbos de data/hora”, postagem no blog, 2013.
- [Kir08] J. Kirsch e Y. Amir, “Paxos for System Builders: Uma Visão Geral”, em Proc. LADIS '08, 2008.
- [Kla12] R. Klau, “Como o Google define metas: OKRs”, postagem do blog, outubro de 2012.
- [Kle06] DV Klein, “Uma Análise Forense de um Ataque de Spam Baseado na Web de Dois Estágios Distribuído”, em Anais da 20ª Conferência de Administração de Sistemas de Grandes Instalações, dezembro de 2006.
- [Kle14] DV Klein, DM Betser e MG Monroe, “Tornando o Push On Green um Realidade”, em ;login:, vol. 39, nº. 5 de outubro de 2014.
- [Kra08] T. Krattenmaker, “Tornar cada reunião importante”, em Harvard Business Review, 27 de fevereiro de 2008.
- [Kre12] J. Kreps, “Conhecendo a confiabilidade do sistema distribuído”, postagem no blog, 19 março de 2012.
- [Kri12] K. Krishan, “Enfrentando o Inesperado”, em Comunicação da ACM, volume 55, nº. 11 de novembro de 2012.
- [Kum15] A. Kumar et al., “BwE: Alocação de largura de banda flexível e hierárquica para Computação Distribuída WAN”, no SIGCOMM '15.
- [Lam98] L. Lamport, “O Parlamento em tempo parcial”, em ACM Transactions on Computer Systems 16, 2, maio de 1998.
- [Lam01] L. Lamport, “Paxos Simples”, em ACM SIGACT News 121, dezembro 2001.
- [Lam06] L. Lamport, “Fast Paxos”, em Computação Distribuída 19.2, outubro de 2006.
- [Lim14] TA Limoncelli, SR Chalup e CJ Hogan, The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems, Volume 2: Addison-Wesley, 2014.

- [Loo10] J. Loomis, "How to Make Failure Beautiful: The Art and Science of Postmortems", in Web Operations: O'Reilly, 2010.
- [Lu15] H. Lu et al, "Consistência Existencial: Medindo e Compreendendo a Consistência no Facebook", em SOSP'15, 2015.
- [Mao08] Y. Mao, FP Junqueira e K. Marzullo, "Mencius: Construindo Máquinas de Estado Replicadas Eficientes para WANs", em OSDI '08, 2008.
- [Mas43] AH Maslow, "Uma Teoria da Motivação Humana", em Psychological Review 50(4), 1943.
- [Mau15] B. Maurer, "Fail at Scale", em ACM Queue, vol. 13, não. 12, 2015.
- [maio09] M. Mayer, "Este site pode danificar seu computador em todos os resultados de pesquisa?!?", postagem do blog, janeiro de 2009.
- [McI86] MD McIlroy, "A Research Unix Reader: Excertos anotados do Projeto Manual de gramática, 1971-1986".
- [McN13] D. McNutt, "Manter a consistência em um ambiente massivamente paralelo", apresentação no USENIX Configuration Management Summit 2013, vídeo disponível online.
- [McN14a] D. McNutt, "Acelerando o caminho do Dev para o DevOps", em ;login:, vol. 39, nº. 2 de abril de 2014.
- [McN14b] D. McNutt, "Os 10 Mandamentos da Engenharia de Liberação", apresentação no 2nd International Workshop on Release Engineering 2014, abril de 2014.
- [McN14c] D. McNutt, "Distribuindo Software em um Ambiente Massivamente Paralelo", apresentação na USENIX LISA 2014, vídeo disponível online.
- [Mic03] Microsoft TechNet, "O que é SNMP?", modificado pela última vez em 28 de março de 2003, <https://technet.microsoft.com/en-us/library/cc776379%28v=ws.10%29.aspx>.
- [Mea08] D. Meadows, Pensando em Sistemas: Chelsea Green, 2008.
- [Men07] P. Menage, "Adicionando Containers de Processos Genéricos ao Kernel Linux", em Proc. do Simpósio Linux de Ottawa, 2007.
- [Mer11] N. Merchant, "Culture Trumps Strategy, Every Time", em Harvard Business Review, 22 de março de 2011.
- [Moc87] P. Mockapetris, "Nomes de Domínio - Implementação e Especificação", Padrão de Internet IETF, 1987.
- [Mol86] C. Moler, "Matrix Computation on Distributed Memory Multiprocessors", em Hypercube Multiprocessors 1986, 1987.
- [Mor12a] I. Moraru, DG Andersen, e M. Kaminsky, "Paxos igualitários", Relatório Técnico do Laboratório de Dados Paralelos da Carnegie Mellon University CMU-PDL-12-108, 2012.
- [Mor14] I. Moraru, DG Andersen e M. Kaminsky, "Paxos Quorum Leases: Fast Reads Without Sacrificing Writes", em Proc. SOCC '14, 2014.

- [Mor12b] JD Morgenthaler, M. Gridnev, R. Sauciuc e S. Bhansali, “[Searching for Build Debt: Experiences Managing Technical Debt at Google](#)”, em Anais do 3º Workshop Internacional de Gestão da Dívida Técnica, 2012.
- [Nar12] C. Narla e D. Salas, “[Servidores Herméticos](#)”, postagem no blog, 2012.
- [Nel14] B. Nelson, “[Os Dados sobre Diversidade](#)”, em Comunicações da ACM, vol. 57, 2014.
- [Nic12] K. Nichols e V. Jacobson, “[Controlando o atraso da fila](#)”, em ACM Queue, vol. 10, não. 5, 2012.
- [Oco12] P. O'Connor e A. Kleyner, Engenharia de Confiabilidade Prática, 5ª edição: Wiley, 2012.
- [Ohn88] T. Ohno, Toyota Production System: Beyond Large-Scale Production: Productivity Press, 1988.
- [Ong14] D. Ongaro e J. Ousterhout, “[Em busca de um consenso comprehensível Algoritmo \(Versão Estendida\)](#)”.
- [Pen10] D. Peng e F. Dabek, “[Processamento Incremental em Grande Escala Usando Transações Distribuídas e Notificações](#)”, em Proc. do 9º Simpósio USENIX sobre Projeto e Implementação de Sistemas Operacionais, novembro de 2010.
- [Per99] C. Perrow, Acidentes normais: vivendo com tecnologias de alto risco, Princeton Editora Universitária, 1999.
- [Per07] AR Perry, “[Engineering Reliability into Web Sites: Google SRE](#)”, em Proc. do LinuxWorld 2007, 2007.
- [Pik05] R. Pike, S. Dorward, R. Griesemer, S. Quinlan, “[Interpretando os Dados: Análise Paralela com Sawzall](#)”, in Scientific Programming Journal vol. 13, não. 4, 2005.
- [Pot16] R. Potvin e J. Levenberg, “[The Motivation for a Monolithic Codebase: Por que o Google armazena bilhões de linhas de código em um único repositório](#)”, em Communications of the ACM, em julho de 2016. Vídeo disponível no [YouTube](#).
- [Roo04] JJ Rooney e LN Vanden Heuvel, “[Análise de causa raiz para iniciantes](#)”, em Progresso da Qualidade, julho de 2004.
- [Sai39] A. de Saint Exupéry, Terre des Hommes (Paris: Le Livre de Poche, 1939, in tradução de Lewis Galantière como Vento, Areia e Estrelas.
- [Sam14] RR Sambasivan, R. Fonseca, I. Shafer e GR Ganger, “[Então, você quer rastrear seu sistema distribuído? Principais insights de design de anos de experiência prática](#)”, Relatório Técnico do Laboratório de Dados Paralelos da Carnegie Mellon University CMU PDL-14-102, 2014.
- [San11] N. Santos e A. Schiper, “[Tuning Paxos for High-Throughput with Batching and Pipelining](#)”, na 13ª Conferência Internacional em Computação Distribuída e Redes, 2012.

[Sar97] NB Sarter, DD Woods e CE Billings, "Automation Surprises", em *Handbook of Human Factors & Ergonomics*, 2^a edição, G. Salvendy (ed.), Wiley, 1997.

[Sch14] E. Schmidt, J. Rosenberg e A. Eagle, *Como o Google funciona*: Editora Grand Central, 2014.

[Sch15] B. Schwartz, "Os Fatores que Impactam a Disponibilidade, Visualizados", postagem no blog, 21 dezembro de 2015.

[Sch90] FB Schneider, "Implementing Fault Tolerant Services Using the State Machine Approach: A Tutorial", em *ACM Computing Surveys*, vol. 22, não. 4, 1990.

[Sec13] Comissão de Valores Mobiliários, "Order In the Matter of Knight Capital Americas LLC", arquivo 3-15570, 2013.

[Sha00] G. Shao, F. Berman, e R. Wolski, "Master/Slave Computing on the Grid", no Workshop de Computação Heterogênea, 2000.

[Shu13] J. Shute et al., "F1: Um banco de dados SQL distribuído que escala", em *Proc. VLDB* 2013, 2013.

[Sig10] BH Sigelman et al., "Dapper, um Rastreamento de Sistemas Distribuídos em Grande Escala A infraestrutura", Relatório técnico do Google, 2010.

[Sin15] A. Singh et al., "Ascensão de Júpiter: uma década de topologias de fechamento e controle centralizado na rede de datacenter do Google", no SIGCOMM '15.

[Skel13] M. Skelton, "A operacionalidade pode melhorar se os desenvolvedores escreverem um livro de rascunhos ", postagem do blog, 16 de outubro de 2013.

[Slo11] B. Treynor Sloss, "Gmail de volta em breve para todos", postagem no blog, 28 de fevereiro 2011.

[Tat99] S. Tatham, "Como Relatar Bugs Efetivamente", 1999.

[Ver15] A. Verma, L. Pedrosa, MR Korupolu, D. Oppenheimer, E. Tune e J. Wilkes, "Gerenciamento de cluster em grande escala no Google com Borg", em *Proceedings of the European Conference on Computer Systems*, 2015.

[Wal89] DR Wallace e RU Fujii, "Verificação e validação de software: uma visão geral", IEEE Soware, vol. 6, não. 3 (maio de 1989), pp. 10, 17.

[War14] R. Ward e B. Beyer, "BeyondCorp: A New Approach to Enterprise Security", em *:login:*, vol. 39, nº. 6 de dezembro de 2014.

[Whi12] JA Whittaker, J. Arbon e J. Carollo, *How Google Tests Software*: Addison-Wesley, 2012.

[Woo96] A. Wood, "Prevendo a confiabilidade do software", em *Computador*, v. 29, não. 11, 1996.

[Wri12a] HK Wright, "Processos de Engenharia de Lançamento, Suas Falhas e Falhas", (seção 7.2.2.2) Tese de Doutorado, Universidade do Texas em Austin, 2012.

[Wri12b] HK Wright e DE Perry, “**Release Engineering Practices and Pitfalls**”, em Anais da 34ª Conferência Internacional de Engenharia de Software (ICSE '12). (IEEE, 2012), pp. 1281-1284.

[Wri13] HK Wright, D. Jasper, M. Klimek, C. Carruth, Z. Wan, “**Large-Scale Automated Refactoring Using ClangMR**”, in Proceedings of the 29th International Conference on Software Maintenance (ICSM '13), (IEEE, 2013), pp. 548–551.

[Zoo14] Projeto ZooKeeper (Fundação Apache), “**ZooKeeper Recipes and Solutions**”, na documentação do ZooKeeper 3.4, 2014.

Índice

Símbolos

/varz manipulador HTTP, 109

Um comportamento abusivo do cliente, 382 controle de acesso, 89 semântica de armazenamento de dados ACID, 287, 341 reconhecimentos, limitação adaptativa xxi-xxiv , 250 banco de dados de anúncios, 73-75 AdSense, 31 equação de disponibilidade agregada, 27, 477 agregação, 114, 180 agilidade vs. estabilidade, 97 (veja também simplicidade do software)

Serviço Alertmanager, 119 alertas definidos, 56 falso-positivos, 180

software para, 18 (veja também Borgmon; time-series monitor)

ing)
anacron, 316
Apache Mesos, 15 App Engine, 146 arquivos vs. backups, 343 consenso distribuído assíncrono, 289 sistemas de transmissão atômica, 295 política de atribuição, automação xx

aplicando a turnups de cluster, 75-81 vs. sistemas autônomos, 67 benefícios de, 67-70 melhores práticas para gerenciamento de mudanças, 10

Exemplo Borg, 81

lições intersetoriais, 467 exemplo de banco de dados, 73-75 Exemplo Diskeraze, 85 foco na confiabilidade, 83 Abordagem do Google para, 70 hierarquia de classes de automação, 72 recomendações para promulgação, 84 aplicações especializadas de, 79 casos de uso para, 70-73 ferramentas de automação, 194 sistemas autônomos, 67

Estudo de caso Auxon, 207-209, 211-213 disponibilidade, 38, 341 (consulte também disponibilidade de serviço) tabela de disponibilidade, 477

B

Rede B4, 15 servidores back-end, 19, 231 back-ends, falsos, 204 backups (ver integridade dos dados) Bandwidth Enforcer (BwE), 17 ferramentas de barreira, 193, 195, 293 pipelines de processamento em lote, 350 lotes, 282, 302, 329 Bazel, 90 melhores práticas de planejamento de capacidade, 482 para gerenciamento de mudanças, 10 orçamentos de erros, 481 falhas, 479 feedbacks, 174 para gerenciamento de incidentes, 166

- monitoramento, 481
- sobrecargas e falhas, 483 post-mortems , 171-172, 482 sistemas de recompensa, 174 funções dos engenheiros de lançamento, 88 lançamentos, 480 objetivos de nível de serviço, 480 formação de equipe, 483 bibliografia, 501
- Big Data, 327**
- Bigtable, 17, 31, 65**
- latência bimodal, 273
- monitoramento de caixa preta, 55, 59, 120
- culturas sem culpa, 170
- Ferramenta de construção Blaze, 90
- Blobstore, 17, 342**
- Borg, 14-16, 81-83, 329**
- Serviço de Nomenclatura Borg (BNS), 16
- Borgmon, 108-123 (veja também monitoramento de séries temporais) alerta, configuração 18, 118 , função 121 rate(), 116 regras, fragmentação 114-118 , arena 119 séries temporais, 111
- vetores, 112-113
- mecanismos de quebra de vidro, 201
- ambientes de construção, 190
- continuidade de negócios, 337
- Falhas bizantinas, 290, 304
- campi C , 14
- canários, 34, 155, 189, 380
- teorema CAP, 286
- CAPA (ação corretiva e preventiva), 465 abordagens de planejamento de capacidade para, 105 melhores práticas para, 482
- Exemplo de Diskerase, 85
- sistemas de consenso distribuídos e 307
- desvantagens de "consultas por segundo", 248
- desvantagens de planos tradicionais, 208 leituras adicionais, 106 etapas obrigatórias baseadas em intenção (consulte planejamento de capacidade baseado em intenção), 11 evitando sobrecarga do servidor com , 266
- lançamentos de produtos e, 376
- abordagem tradicional para, 207
- endereçamento de falhas em cascata, 280-283 causas de, 260-265 definidas, 259, 308 fatores desencadeantes, 276 visão geral de, 283 prevenção de sobrecarga do servidor, 265-276 testes para, 278-280 (veja também manipulação de sobrecarga) gerenciamento de mudanças, 10 (veja também automação) emergências induzidas por mudanças, 153-155 listas de mudanças (CLs), 20
- Chaos Monkey, 196**
- checkpoint state, 195
- cherry picking, 89
- Serviço de bloqueio gordinho , 18.291 interrupções planejadas, 39.47
- tarefas do cliente, 231 estrangulamentos do lado do cliente, 249 clientes, 19 desvios de relógio, 287
- Malha de rede Clos, 14**
- estratégias de integridade de dados do ambiente de nuvem, 341, 356 definição de integridade de dados, 340 evolução de aplicativos, 341 desafios técnicos, 344 clusters aplicando automação a turnups, 75-81 solução de gerenciamento de cluster, 329 definidos, 14 código amostras, xx estado de fluxo cognitivo, 409 cold caching, 274 instalações de colocation (colos), 85
- Colossus, 16**
- postos de comando, 164
- postmortems de comunicação e colaboração sem culpa, 171 estudos de caso, 432-439 importância de, 440 com Outalator, 181 fora da equipe SRE, 437 posição de SRE no Google, 425 reuniões de produção (ver reunião de produção).
- ings)

- dentro da equipe SRE, 430
- testes de resiliência em toda a empresa, 106
- estrutura de compensação, 128 otimização computacional, 209 gerenciamento de configuração, 93, 153, 201, 277 testes de configuração, 188 algoritmos de consenso
- Paxos igualitários, 302
 - Fast Paxos, 301, 320
 - melhorando o desempenho de, 296
 - Multi-Paxos, 303
 - Paxos, 290, 303
 - Balsa, 298, 302
 - Zab, 302
 - (veja também sistemas de consenso distribuído)
- consistência eventual, 287 por meio de automação, 67
- hash consistente, 227
- restrições, 208
- Consul, 291
- serviços ao consumidor, identificando a tolerância ao risco de, 28-31
- Ferramenta de compilação e implantação
- contínua Blaze, 90 ramificações, 90
 - destinos de compilação, 90 de gerenciamento de configuração, 93 de implantação, 93 de empacotamento, 91
 - Sistema de liberação rápida, 90, 91
 - testes, 90 processos de liberação típicos, 92 contribuidores, xxii-xxiv corrotinas, 327 corporativos segurança de rede, 106 garantias de correção, 335 correlação vs. causalidade, 136
- custos
- metas de disponibilidade e, 30, 32
 - diretas, 4 de falha em aceitar o risco, 25 indiretas, 4 de abordagem de gerenciamento de administração de sistema, 4 de consumo de CPU, 248, 262, 383
- algoritmos de falha de falha versus recuperação de falha, 290
- cron
- em grande escala, 325
 - edifício no Google, 319-325
- idempotência, 316
- implantação em larga escala de, 317
- líder e seguidores, 321 visão geral de, 326
- Algoritmo Paxos e, 320-325 propósito de, 315 aplicações de confiabilidade de, 316 resolvendo falhas parciais, 322 armazenando estado, 324 rastreando estado de tarefa cron, 319 usos para, 315 lições entre indústrias
- Apollo 8, xvii
- perguntas comparativas apresentadas, 459
- habilidades de tomada de decisão, 469-470
- Aplicação do Google de, 470 líderes do setor contribuindo, 460 temas-chave abordados, 459 cultura postmortem, 465-467 preparação e testes de desastres, 462-465 trabalho repetitivo/sobrecarga operacional, 467 estado atual, exposição, 138
- ## D
- Camada de armazenamento D, 16 painéis
- benefícios de, 57
 - definidos, 55
- análise de dados, com Outalator, 181
- backups de integridade de dados vs.
- arquivos, 343 estudos de caso em, 360-366 condições que levam à falha, 346 definidos, 339 definição expandida de, 340 modos de falha, 349 de usuários perspectiva, 345 visão geral de, 368 estratégia de seleção para, 341-343, 356
- abordagem de SRE para, 349-360 objetivos de SRE para, 344-349 princípios de SRE aplicados a, 367-368 requisitos rigorosos, 340 desafios técnicos de, 344 pipelines de processamento de dados continuidade de negócios e, 337 desafios de distribuição desigual de trabalho, 328 desafios ao padrão periódico, 328 desvantagens do periódico, 329-332

- efeito de big data, [328](#) problemas de monitoramento, [331-332](#) origem de, [327](#)
- visão geral de, [338](#) profundidade de pipeline, [328](#) pipelines simples vs. multifásicos, [328](#)
- sistema de fluxo de trabalho, [333, 335](#)
- recuperação de dados, [359](#) datacenters backbone network para, [15](#) validação de dados, [357](#)
- balanceamento de carga, [231-246](#) topologia de, [14](#)
- datastores ACID e BASE, [287, 341, 347](#) replicados confiáveis, [292](#) Decider, [74](#) habilidades de tomada de decisão, [469](#) defesa em profundidade, para integridade de dados, [349, 361, 367](#)
- visão de demanda, [11](#)
- hierarquias de dependência, [58, 263](#)
- implantação, [93](#) (veja também compilação e implantação contínua) ambiente de desenvolvimento, [19](#)
- divisão de desenvolvimento/operações, [4](#)
- DevOps, [7](#)
- Resposta Direta do Servidor (DSR), [228](#)
- ferramentas de recuperação de desastres, [195](#)
- simulação de desastres, [401](#) testes de desastres, [462-465](#)
- Testes de Desastres e Recuperação (DIRT), [462](#) acessos ao disco, [303](#)
- Processo Diskerase, [85](#)
- distração, [411](#) benefícios de sistemas de consenso distribuídos, [285](#)
- coordenação, uso em, [293](#) implantação, [304-312](#) travamento, uso em, [286](#)
- monitoramento, [312](#) necessidade de, [285](#)visão geral de, [313](#) padrões para, [291-295](#) desempenho de, [296-304](#)
- princípios, [289](#) composição de quórum, [310](#) técnica de locação de quórum, [299](#)
- (veja também algoritmos de consenso)
- escalonamento periódico distribuído (veja cron)
- DNS (Sistema de Nomes de Domínio)
- Extensão EDNS0, balanceamento de carga [225](#) usando, [224-227](#)
- DoubleClick for Publishers (DFP), drenos [437-439, 277](#)
- Arquivos de comunicação DTSS, situação de [327](#) proponentes em duelo, [298](#) durabilidade, [38](#)
- E**
- deteção precoce para integridade de dados, [356](#) (consulte também integridade de dados)
- Early Engagement Model, [448-451](#) algoritmos “embarrasosamente paralelos”, [328](#) engenheiros embarcados, [417-423](#) preparação para emergências, [361](#)
- lições intersetoriais, [462](#) emergências induzidas por mudanças de resposta a emergências, [153-155](#)
- elementos essenciais de, [151](#)
- Cinco Porquês, [140, 487](#)
- diretrizes para, [10](#) respostas iniciais, [151](#) lições aprendidas, [158](#)visão geral de, [159](#)
- emergências induzidas por processo, [155](#) disponibilidade de solução, [158](#)
- emergências induzidas por teste, [152](#)
- encapsulamento, [228](#) endpoints, em depuração, [138](#) compromissos (consulte modelo de engajamento SRE) benefícios de orçamentos de erro, [35](#) melhores práticas para, [481](#) formação, [34](#) diretrizes para, [8](#) motivação para, [33](#) taxas de erro, [38, 60](#)
- Escada rolante, [178](#)
- Pipelines ETL, [327](#)
- consistência eventual, média de carga de [287](#) executores, [253](#)
- Falhas F, práticas recomendadas para, [479](#) (consulte também falhas em cascata) back-ends falsos, [204](#) alertas falso-positivos, [180](#)

estruturas de sinalizador de recurso, 381

descriptores de arquivo, 263

Cinco Porquês, 140, 487

controle de fluxo, 233

Resultado de impossibilidade de FLP, 290

Flume, 328

fragmentação, 229

Operações fechadas G , 89

Encapsulamento de Roteamento Genérico (GRE), 228

GFE (Google Frontend), 21, 232

GFS (Sistema de Arquivos do Google), 76, 293, 318-319, 354

sobrecarga global, 248

Balanceador de carga de software global (GSLB), 18

Gmail, 65, 360

Google Apps for Work, 29

Google Compute Engine, 38

Práticas recomendadas do ambiente de

produção do Google para complexidade

de 479-484 , topologia de datacenter

205 , ambiente de desenvolvimento

14 , hardware 19 , 13

Serviço de busca Shakespeare, infraestrutura

de software 20-22 , software de sistema

19 , 15-19

Sistema Google Workflow como

padrão de controle de visualização de modelo,

334 continuidade de negócios e, 337 garantias

de correção, 335 desenvolvimento de, 333

estágios de execução em, 335

degradação graciosa, 267

GTape, 360

H

Hadoop Distributed File System (HDFS), 16 handoffs, 164

problemas de “hanging chunk”, 329 falhas de

gerenciamento de hardware, 15 softwares que “organizam”,

15-19 terminologia usada para, 13 verificações de

integridade, 281 Healthcare.gov, 103 compilações

herméticas , 89 quórums hierárquicos, 311

abordagem de alta velocidade, 24, 88

hotspotting, 236

I operações idempotentes, 78, 316

gerenciamento de incidentes

melhores práticas para, 166

eficazes, 161 protocolos

formais para, 130 processo de

gerenciamento de incidentes, 153, 163 resposta a

incidentes, 104 exemplo de incidente gerenciado,

165 funções, 163 modelo para, 485 exemplo de

incidente não gerenciado, 161 quando declarar um

incidente, 166 serviços de infraestrutura

identificando a tolerância ao risco de, 31

SRE aprimorado por meio de automação, 69

propostas de integração, 89 testes de integração, 186,

201 planejamento de capacidade baseado em intenção

Implementação Auxon, 211-213 premissa

básica de, 209 benefícios de, 209

definidos, 209 implantação de aproximação,

214 condução de adoção de, 215-217

precursores de intenção, 210 requisitos e

implementação, 213 seleção de nível de

intenção, 210 dinâmica de equipe, 218

interrupções cognitivas estado de fluxo e, 409

lidando com, 407 lidando com grandes volumes,

412 determinando a abordagem de manuseio, 408

distração e, 411 gerenciando carga operacional, 408

engenheiros de plantão e, 412 responsabilidades

contínuas, 413 polarizando tempo, 411 reduzindo,

413 atribuições de tickets , 413

IRC (Internet Relay Chat), 164

J

empregos, 16

Tecido de rede Júpiter, 14

L

Labelsets, 112
lame duck state, 234
latência definida, 341
medição, 38
monitoramento para,
60 checklist de
coordenação de lançamento,
373-380, 493 engenharia
(LCE), 370, 384-387 (veja também
lançamentos de produtos) exclusão
preguiçosa, 349 líder eleição, 286, 292 sistemas
de arrendamento, 294

Política de Round Robin menos carregado, 243
níveis de serviço, 37 (consulte também objetivos
de nível de serviço (SLOs)) documentos de incidentes
vivos, 164 datacenter de平衡amento de carga

serviços e tarefas de datacenter, 231
controle de fluxo, 233
Aplicação do Google de, 231
sobrecarga de manuseio, 247 uso
ideal da CPU, 232, 248 estado de
pato lame, 234 pools de conexões
limitantes, encapsulamento de pacotes
235-240 , 228 políticas para, 240-246

Dinâmica de engenharia de software SRE, 218
sistemas de consenso distribuídos e, 307 soluções
ótimas de front-end para, 223 usando DNS, 224-227
endereços IP virtuais (VIPs), 227 políticas

Round Robin menos carregado, 243
Round Robin, 241
Round Robin ponderado, 245 cortes
de carga, 267 testes de carga, 383 serviços
de bloqueio, 18, 293 registros, 138

Lustro, 16

M

máquinas
definidas, 13, 56
gerenciando com software, 15

quóruns da maioria, 304

MapReduce, 328
tempo médio
entre falhas (MTBF), 184, 199 para falha
(MTTF), 10 para reparar (MTTR), 10, 68, 184
exaustão de memória, 263 algoritmo Mencius,
302 meta-software, 70 Midas Package Manager
(MPM), 91 model- padrão view-controller, 334
modularidade, 100 padrão de carga Moiré em
pipelines, 331 monitoramento de sistemas
distribuídos evitando complexidade, 62 benefícios
do monitoramento, 56, 107 melhores práticas
para, 481 blackbox vs. whitebox, 59, 120 estudos
de caso, 65-66 desafios de, 64, 107 emergências
induzidas por mudanças, 154 criando regras
para, 63 quatro sinais dourados de, 60
diretrizes para, 9 instrumentação e
desempenho, 61 filosofia de monitoramento,
64 resolução, 62 definição de expectativas
para, 57 curto versus longo disponibilidade
de termo, 66 software para, 18 sintomas vs.
causas, 58 terminologia, 55 saídas de
monitoramento válidas, 10 (veja também
Borgmon;

ing)

Protocolo Multi-Paxos, 297, 303
(veja também algoritmos de consenso)
equipes multi-site, 127 matrizes
multidimensionais, 112 pipelines multifásicos,
328
Migração
do MySQL, 73-75, 437
emergências induzidas por teste e, 152

N

Configuração N + 2, 22, 210-212, 266, 482 resultados
negativos, 144
Tradução de Endereço de Rede, 228 latência
de rede, 300

balanceador de carga de rede, 227

partições de rede, 287

Qualidade de serviço de rede (QoS), segurança de rede 157, 252 , rede 106 , 17

Site NORAD Tracks Santa, 369 número de "nove", 38, 477

O

lançamentos mais antigos, reconstrução, 89 de plantão

plantão equilibrado, 127

benefícios de, 132 melhores

práticas para, 393, 400-405 estrutura de remuneração, 128 educação continuada,

406 práticas de educação, 392, 395

protocolos formais de gerenciamento de

incidentes, 130 cargas operacionais inadequadas, 130

experiências iniciais de aprendizado, 394 listas de verificação de aprendizado, 403 visão geral de, 125, 406 recursos para,

129 cronogramas de rotação, 126 sombra de plantão, 405

técnicas de redução de estresse, 128 volume de evento

alvo, 8 trabalho de projeto direcionado, 397 formação de

equipe, 391 requisitos de tempo, 128 treinamento para ,

395-401 materiais de treinamento, 397 atividades típicas,

126 pipelines monofásicos, 328 sistema aberto de

comentários/anotações, 171 aulas de carga operacional

interseitorial, 467 gerenciamento, 408 responsabilidades

contínuas, 408 tipos de, 407 sobrecarga operacional, 130

subcarga operacional , 132 trabalho operacional (ver labuta)

verificações e contrapesos fora de banda, 342, 357 sistemas

de comunicação fora de banda, 154 linhas de base de

rastreamento de interrupção e rastreamento de progresso,

177 benefícios de, 182

Outalador, 178-182

Outalador

agregação, 180 benefícios

de, 178 construindo o seu

próprio, 179 análise de

incidentes, 181 processo de

notificação, 178 relatórios e

comunicação, 181 marcação, 180 sobrecarga,

49 abordagens de manipulação de sobrecarga, 247 melhores práticas para, 483 limitação do lado

do cliente, 249 carga de conexões, 257 erros de

sobrecarga, 253 visão geral de, 258 orçamento

de repetição por cliente, 254 limites por cliente,

248 orçamento de repetição por solicitação,

254 lançamentos de produtos e, 383 solicitação

de criticidade, 251 solicitações de repetição ,

254 (consulte também novas tentativas , RPC)

sinais de utilização, 253 (veja

também falhas em cascata)

gerenciadores de pacotes P ,

encapsulamento de 91 pacotes, 228

Algoritmo de consenso Paxos

Algoritmo clássico Paxos, 301 acesso

ao disco e, 303

Algoritmo de consenso igualitário Paxos, 302

Algoritmo de consenso rápido Paxos, 301, 320

Protocolo Paxos de Lamport, 290

(veja também algoritmos de consenso)

eficiência de desempenho e, 12 monitoramento, 61

testes de desempenho, 186 pipelines periódicos,

328 agendamento periódico (veja cron)

armazenamento persistente, 303

Photon, 305

pipelines, 302

mudanças planejadas, 277

políticas e procedimentos, aplicação, 89 análises

post hoc, 58

Escada rolante, 178

- benefícios
 - postmortems de, 169
 - melhores práticas para, 171-174, 482
 - colaboração e compartilhamento, 171 conceito de, 169 lições intersetoriais, 465-467 exemplos postmortem, 487-490 revisão formal e publicação de, 171
 - A filosofia do Google para, 169 diretrizes para, 8 introdução de culturas postmortem, 172 engenharia de plantão e 400 melhorias contínuas, 175 participação recompensadora, 174 gatilhos para, 170 privacidade, 341 testes proativos, 159 relatórios de problemas, 136 morte de processo, 276 verificações de integridade de processos, 281 atualizações de processos, 277 emergências induzidas por processos, 155
 - Protocol Data Units, 229
 - provisionamento, diretrizes para, 11
 - Modelo PRR (Production Readiness Review), 442, 444-448
 - frequência de push, 34 gerentes de push, 413
 - safe_load do Python, 201
 - Modelo Q "consultas por segundo", 248
 - Query of Death, 276 filas de atraso controlado, 267 first-in, first-out, 267 last in, first-out, 267 management of, 266, 294 queue-as-work-distribution pattern, 295 quorum (veja sistemas de consenso distribuídos)
-
- ## R
- Protocolo de consenso de balsa, 298, 302
 - (veja também algoritmos de consenso)
 - REIDE, 354
 - Sistema de liberação automatizada rápida , carga de trabalho de 90, 91 leituras, dimensionamento, 298 backups reais, 343 de colaboração em tempo real, 171 de capacidade de recuperação, 347 de recuperação, 359 sistemas de recuperação, 345 recursividade (consulte recursão) servidores DNS recursivos, 225 recursividade de separação de responsabilidades, 163 redundância, 347, 354
 - Códigos de eliminação Reed-Solomon, 354
 - testes de regressão, 186 desafios de engenharia de lançamento, 87 construção e implantação contínuas, 90-94 definidos, 87 instituintes, 95 filosofia de, 88-90 o papel dos engenheiros de lançamento, 87 aplicações mais amplas de, 95 confiabilidade quantidade de testes necessária, 184 benefícios de, 204 mecanismos de quebra de vidro, 201 testes canários, 189

testes de configuração, 188
 coordenação de, 197 criação de ambientes de teste e construção, 190 orçamentos de erros, 8, 33-35, 481 esperando falha de teste, 199-200 versões backend falsas, 204 objetivos de, 183 importância de, testes de integração xvi , 186, 201

MTTR e, 184 testes de desempenho, 186 proativos, 159 sondas de produção, 202 testes de produção, 187 testes de regressão, 186 metas de confiabilidade, 25 testes de sanidade, 186 ambientes segregados e, 198 testes de fumaça, 186 velocidade de, 196 testes estatísticos, 196

testes de estresse, 188 testes de sistema, 186 testes em escala, 192-204 tempo de, 187 testes de unidade, 185 armazenamentos de dados replicados confiáveis, 292 Chamada de procedimento remoto (RPC), 19, 138, 252 bimodal, 273 prazos ausentes, 271 propagando, 267, 272 gerenciamento de filas, 266, 294 selecionando, 271 tentativas, 268-271

Criticidade de RPC, 251 (consulte também manipulação de sobrecarga) adição de réplicas, 307 desvantagens de réplicas líderes, 308 localização de, 306, 310 número implantado, 304 logs replicados, 305 máquina de estado replicado (RSM), 291 replicação, 347, 354 latência de solicitação, 38, 60 alterações de perfil de solicitação, 277 taxa de sucesso de solicitação, 27 testes de resiliência, 106

Recursos

alocação de, 14, 16 exaustão, 261 limites, 278 (veja também planejamento de capacidade) restaurações, 355 retenções, 348 tentativas, RPC evitando, 254 falhas em cascata devido a, 268 considerações para automático, 270 diagnósticos de interrupções devido a, 271 manipulação de erros de sobrecarga e, 254 orçamentos de repetição por cliente, 254 orçamentos de repetição por solicitação, 254 engenharia reversa, 398 proxies reversos, 157 histórico de revisão, 351 gerenciamento de risco equilibrando risco e inovação, 25 custos de, 25 orçamentos de erro, 33-36, 481 chave insights, 36 medindo o risco do serviço, 26 tolerância ao risco dos serviços, 28-33 procedimentos de reversão, 153 lançamentos, 277, 379, 480

causa raiz análise de, 105, 169 (veja também postmortems) definido, 56 Política Round Robin, 241 tempos de ida e volta (RTT), 300 linhas, avaliação de 14 regras, em sistemas de monitoramento, 114-118

S

Safari® Books Online, teste de sanidade xxi , saturação de 186 , escala de 60 definido, 341 problemas em, 347 segurança na engenharia de lançamento, 89 nova abordagem para, 106 modelo de autoatendimento, 88 separação de responsabilidades, 163 servidores

- vs. clientes, 19
- definidos, 13
- cenário de sobrecarga, 260
- prevenindo sobrecarga, 265-276
- tabela de disponibilidade de disponibilidade
 - de serviço, 477 fatores de custo, 30, 32 definidos, 38 alvo para serviços ao consumidor, 29 alvo para serviço de infraestrutura, 31 equação baseada em tempo, 26 tipos de falhas de serviço ao consumidor, 29 tipos de falhas de serviços de infraestrutura, 32 verificações de integridade de serviço, 281 latência de serviço mais flexível, 31
- monitoramento para, 60 acordos de nível de serviço (SLAs), 39 indicadores de nível de serviço (SLIs) agregando medições brutas, 41 coleta de indicadores, 41 definidos, 38 indicadores padronizados, 43 acordos de objetivos de nível de serviço (SLOs) na prática, 47 melhores práticas para, 480 escolha, 37-39 medidas de controle, 46 definidos, 38 objetivos definindo, 43 selecionando indicadores relevantes, 40 falácias estatísticas e, 43 seleção de destino, 45 expectativas do usuário e, 39, 46 abordagem abrangente de gerenciamento de serviços para, xvi abordagem do Google para, 5-7 abordagem sysadmin para, 3, 67
- Arquitetura Orientada a Serviços (SOA), 81
- Serviço de pesquisa Shakespeare, exemplo de alerta, 137 aplicando SRE a, 20-22 exemplo de falha em cascata, 259-283 depuração, 139 engajamento, 283, 445 gerenciamento de incidentes, 485 postmortem, 487-490 reunião de produção, 497-499 implantações fragmentadas, 307
- Valor de criticidade SHEDDABLE_PLUS, 251 simplicidade, 97-101
- Estrutura de automação Sísifo, 93
- Atividades de Engenharia de Confiabilidade do Site (SRE) incluídas em, 103 abordagem para aprendizado, xix componentes básicos de, xv benefícios de, 6 desafios de, 6 definidos, xiii-xiv, 5 primeiros engenheiros, xvii
- Abordagem do Google para gerenciamento, 5-7, 425 crescimento de no Google, 473, 474 contratações, 5, 391 origens de, xvi abordagem sysadmin para gerenciamento, 3, 67 composição e habilidades de equipe, 5, 126, 473 princípios de, 7-12 atividades típicas de, 52 aplicações generalizadas de, xvi inicialização lenta, 274 testes de fumaça, 186
- SNMP (Simple Networking Monitoring Protocol), 111 soft delete, 350 software bloat, 99 engenharia de software em atividades SRE incluídas, 106
- hierarquia de confiabilidade de serviço recursos adicionais, 106 planejamento de capacidade, 105 desenvolvimento, 106 diagrama de, 103 resposta a incidentes, 104 monitoramento, 104 lançamento de produto, 106 análise de causa raiz, 105 testes, 105 indisponibilidade de serviço, 264
- Estudo de caso Auxon, 207-209 benefícios de, 222 encorajadores, 215 fomentadores, 218
- Foco do Google em, 205 importância de, 205 planejamento de capacidade baseado em intenção, 209-218 equipe e tempo de desenvolvimento, 219 dinâmica de equipe, 218 tolerância a falhas de software, 34

simplicidade do software
 evitando inchaço, 99
 modularidade, 100
 previsibilidade e, 98
 simplicidade de lançamento,
 100 confiabilidade e, 101
 limpezas de código-fonte, 98
 estabilidade do sistema versus agilidade, 97
 escrita de APIs mínimas, 99

Chave inglesa, 17, 32, 337

Aspectos do modelo de
 engajamento SRE abordados por, 443
 Early Engagement Model, 448-451 frameworks
 e plataformas em, 451-455 importância de, 441

Revisão de Prontidão de Produção, 442, 444-448

Ferramentas SRE
 ferramentas de automação,
 194 ferramentas de barreira,
 193, 195 ferramentas de recuperação
 de desastres, 195 testes, 193
 escritas, 202 SRE Way, 12
 estabilidade versus agilidade, 97 (veja
 também simplicidade de software) líderes
 estáveis, 302 testes estatísticos, 196
 pilha de armazenamento, 16 estresse
 testes, 188 processo líder forte, 297 Stubby,
 19 subconjuntos definidos, 235
 determinísticos, 238 processos de, 235
 aleatórios, 237 selecionando subconjuntos,
 236 consenso síncrono, 289 administradores
 de sistema (administradores de sistemas),
 3, 67 software de sistema gerenciando
 falhas com, 15 gerenciando máquinas,
 15 armazenamento, 16 testes de
 sistema, 186 taxa de transferência do
 sistema, 38 administradores de
 sistemas (sysadmins), 3, 67 engenharia de
 sistemas, 390

Marcação T ,
 180 erros de "tarefa sobre carregada",
 253 tarefas
 back-end, 231
 clientes, 231
 definidos, 16

Protocolo de comunicação TCP/IP, 300 benefícios
 de team building da abordagem do Google para ,
 6.473 práticas recomendadas para, 483 foco no
 desenvolvimento, 6 dinâmicas de engenharia de
 software SRE, 218 eliminação de complexidade, 98
 foco em engenharia, 6, 7, 52, 126-127 , 474 equipes multi-
 site, 127 autossuficiência, 88 habilidades necessárias, 5
 equipe e tempo de desenvolvimento, 219 composição de
 equipe, 5 campi de terminologia (específicos do Google),
 14 clientes, 19 clusters, 14 datacenters, 14 frontend/
 backend, 19 jobs, 16 máquinas, 13 buffers de protocolo
 (protobufs), 19 racks, 14 linhas, 14 servidores, 13, 19
 tarefas, 16 ambientes de teste, 190 (veja também testes
 de confiabilidade) emergências induzidas por testes, 152 testes
 (veja testes de confiabilidade) texto logs, 138 threads
 starvation, 263 throttling adaptáveis, 250 client-side, 249
 problemas "thundering herd", 331, 383 equação de
 disponibilidade baseada em tempo, 26, 477

Banco de dados de série temporal (TSDB), 112
 alertas de monitoramento de série temporal,
 118 monitoramento de caixa preta, 120

Sistema de monitoramento Borgmon, 108

- coleta de dados exportados, 110
 - instrumentação de aplicativos, 109 manutenção da configuração Borgmon, 121 topologia de monitoramento, 119 abordagem prática para, 108 avaliação de regras, 114-118 dimensionamento, 122 armazenamento de dados de séries temporais, 111-113 ferramentas para, 108 tempo para -ao vivo (TTL), 225 timestamps, 292 labuta
 - benefícios de limitação, 24, 51 cálculo, 51 características de, 49 lições intersetoriais, 467 definidas, 49 desvantagens de, 52 vs. trabalho de engenharia, 52
 - análise de tráfego, 21-22, 60 treinos, 392, 395-397 processo de triagem, 137 Trivial File Transfer Protocol (TFTP), 157 solução de problemas Estudo de caso do App Engine, 146-149 abordagens para, 133 armadilhas comuns, 135 problemas de cura, 145 problemas de diagnóstico, 139-142 exame de componentes do sistema, 138 registro, 138 modelo de, 134 armadilhas, 135-136 relatórios de problemas, 136 diagrama de processo, 135 simplificando, 150 abordagem sistemática para, 150 testando e tratando problemas, 142-145 triagem, 137 automação de turndown, 157, 277 convenções tipográficas, xix
 - testes de unidade U , 185 Pipe UNIX, 327 paralisações não planejadas, 26 tempo de atividade, 341 solicitações de usuários com valores de criticidade atribuídos, 251 tarefas e organização de dados, 22 falhas de monitoramento, 60 solicitações de latência, 38 solicitações de monitoramento de latência, 60 tentativas, 254 serviços de, 21 métricas de taxa de sucesso, 27 análise de tráfego, 22, 60
 - sinais de utilização, 253
 - Expressões de variáveis V , 113 vetores, 112 velocidades, 341
 - Projeto Viceroy, 432-437 endereços IP virtuais (VIPs), 227
- C**
- "Salas de Guerra", 164
 - Política de Round Robin Ponderada, 245
 - Exercício Wheel of Misfortune, 173
 - monitoramento de caixa branca, 55, 59, 108
 - cargas de trabalho, 296
 - rendimento Y , 38
 - YouTube, 29
- Z**
- Consenso Zab, 302
 - Zookeeper, 291

sobre os autores

Betsy Beyer é redatora técnica do Google em Nova York, especializada em engenharia de confiabilidade de sites. Ela já escreveu documentação para as equipes de operações de hardware e data center do Google em Mountain View e em seus datacenters distribuídos globalmente. Antes de se mudar para Nova York, Betsy foi professora de redação técnica na Universidade de Stanford. A caminho de sua carreira atual, Betsy estudou Relações Internacionais e Literatura Inglesa, e possui diplomas de Stanford e Tulane.

Chris Jones é engenheiro de confiabilidade do site do Google App Engine, um produto de plataforma de nuvem como serviço que atende a mais de 28 bilhões de solicitações por dia. Baseado em San Francisco, ele foi anteriormente responsável pelo cuidado e alimentação das estatísticas de publicidade do Google, armazenamento de dados e sistemas de suporte ao cliente. Em outras vidas, Chris trabalhou em TI acadêmica, analisou dados para campanhas políticas e se envolveu em alguns hackers leves de kernel BSD, obtendo diplomas em Engenharia da Computação, Economia e Política de Tecnologia ao longo do caminho. Ele também é um engenheiro profissional licenciado.

Jennifer Petoff é gerente de programa da equipe de engenharia de confiabilidade de sites do Google e está sediada em Dublin, na Irlanda. Ela gerenciou grandes projetos globais em vários domínios, incluindo pesquisa científica, engenharia, recursos humanos e operações de publicidade. Jennifer ingressou no Google depois de passar oito anos na indústria química. Ela é PhD em Química pela Universidade de Stanford e bacharel em Química e bacharel em Psicologia pela Universidade de Rochester.

Niall Murphy lidera a equipe de engenharia de confiabilidade do site de anúncios no Google Ireland. Ele está envolvido na indústria da Internet há cerca de 20 anos e atualmente é presidente do INEX, o hub de peering da Irlanda. Ele é autor ou coautor de vários artigos técnicos e/ou livros, incluindo IPv6 Network Administration para O'Reilly, e vários RFCs. Ele está atualmente co-escrevendo uma história da Internet na Irlanda e é titular de diplomas em Ciência da Computação, Matemática e Estudos de Poesia, o que certamente é algum tipo de erro. Ele mora em Dublin com sua esposa e dois filhos.

Colofão

O animal na capa da Site Reliability Engineering é o lagarto-monitor ornamentado, um réptil nativo da África Ocidental e Central. Até 1997, era considerado uma subespécie do lagarto-monitor do Nilo (*Varanus niloticus*), mas agora é classificado como um polimorfo de *Varanus stellatus* e *Varanus niloticus* devido aos seus diferentes padrões de pele. Ele também tem um alcance menor do que o monitor do Nilo, preferindo um habitat de floresta tropical de planície.

Monitores ornamentados são grandes lagartos, capazes de crescer até 6 a 7 pés de comprimento. Eles são mais coloridos do que os monitores do Nilo, com pele verde-oliva mais escura e menos faixas de manchas amarelas brilhantes que vão do ombro à cauda. Como todos os lagartos-monitores, este animal tem um corpo musculoso e robusto, garras afiadas e uma cabeça alongada. Suas narinas são

colocados no alto do focinho, permitindo-lhes passar tempo na água. São excelentes nadadores e alpinistas, o que lhes permite manter uma dieta de peixes, sapos, ovos, insetos e pequenos mamíferos.

Os lagartos-monitores são frequentemente mantidos como animais de estimação, embora exijam muitos cuidados e não sejam adequados para iniciantes. Eles podem ser perigosos quando se sentem ameaçados (chicoteando suas caudas poderosas, arranhando ou mordendo), mas é possível domá-los um pouco com o manuseio regular e ensiná-los a associar a presença de seu guardião com a entrega de comida.

Muitos dos animais nas capas da O'Reilly estão ameaçados de extinção; todos eles são importantes para o mundo. Para saber mais sobre como você pode ajudar, acesse animals.oreilly.com.

A imagem da capa é do Brockhaus Lexicon. As fontes da capa são URW Typewriter e Guardian Sans. A fonte do texto é Adobe Minion Pro; a fonte do título é Adobe Myriad Condensed; e a fonte do código é Ubuntu Mono da Dalton Maag.