

Engenharia de Confiabilidade do Google

COMO O GOOGLE ADMINISTRA SEUS SISTEMAS DE PRODUÇÃO

Site Reliability Engineering (SRE)

novatec

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

Engenharia de Confabilidade do Google

COMO O GOOGLE ADMINISTRA SEUS SISTEMAS DE PRODUÇÃO

Editado por

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

O'REILLY®
Novatec

Authorized Portuguese translation of the English edition of Site Reliability Engineering, ISBN 9781491929124 © 2016 Betsy Beyer, Chris Jones, Jennifer Petoff, Niall Richard Murphy. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Tradução em português autorizada da edição em inglês da obra Site Reliability Engineering, ISBN 9781491929124 © 2016 Betsy Beyer, Chris Jones, Jennifer Petoff, Niall Richard Murphy. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora de todos os direitos para publicação e venda desta obra.

© Novatec Editora Ltda. 2016.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Tradução: Lúcia A. Kinoshita

Revisão gramatical: Sandro Andretta

Assistente editorial: Priscila A. Yoshimatsu

Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-752-7

Histórico de edições impressas:

Agosto/2016 Primeira edição

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

Email: novatec@novatec.com.br

Site: www.novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

Sumário

[Apresentação](#)

[Prefácio](#)

[Parte I ■ Introdução](#)

[Capítulo 1 ■ Introdução](#)

[A abordagem com administradores de sistemas para gerenciamento de serviços](#)

[A abordagem do Google para o gerenciamento de serviços: Site Reliability Engineering](#)

[Princípios da SRE](#)

[Garantindo um foco durável em engenharia](#)

[Buscando a máxima rapidez nas mudanças sem violar o SLO de um serviço](#)

[Monitoração](#)

[Resposta a emergências](#)

[Gerenciamento de mudanças](#)

[Previsão de demanda e planejamento de capacidade](#)

[Provisionamento](#)

[Eficiência e desempenho](#)

[O fim do começo](#)

[Capítulo 2 ■ O ambiente de produção do Google do ponto de vista de um SRE](#)

[Hardware](#)

[Sistema de software que “organiza” o hardware](#)

[Administrando as máquinas](#)

[Armazenagem](#)

[Rede](#)

[Outros softwares de sistemas](#)

[Serviço de lock](#)

[Monitoração e alertas](#)
[Nossa infraestrutura de software](#)
[Nosso ambiente de desenvolvimento](#)
[Shakespeare: um exemplo de serviço](#)
[Vida de uma requisição](#)
[Organização de jobs e dados](#)

[Parte II ■ Princípios](#)

[Outras leituras da SRE do Google](#)

[Capítulo 3 ■ Aceitando os riscos](#)

[Administrando riscos](#)
[Mensurando os riscos a um serviço](#)
[Tolerância dos serviços aos riscos](#)
[Identificando a tolerância a riscos dos serviços para consumidores](#)
[Identificando a tolerância a riscos dos serviços de infraestrutura](#)
[Motivação para provisão de erros](#)
[Calculando sua provisão para erros](#)
[Vantagens](#)

[Capítulo 4 ■ Objetivos do nível de serviço](#)

[Terminologia do nível de serviço](#)

[Indicadores](#)
[Objetivos](#)
[Acordos](#)

[Indicadores na prática](#)

[Com o que você e seus usuários se importam?](#)
[Coletando indicadores](#)
[Agregação](#)
[Padronize os indicadores](#)

[Objetivos na prática](#)

[Definindo objetivos](#)
[Definindo as metas](#)
[Medidas de controle](#)
[Os SLOs definem expectativas](#)
[Acordos na prática](#)

Capítulo 5 ■ Eliminando tarefas penosas

Definição de tarefas penosas

Por que menos tarefas penosas é melhor

O que é qualificado como engenharia?

As tarefas penosas são sempre ruins?

Conclusão

Capítulo 6 ■ Monitorando sistemas distribuídos

Definições

Por que monitorar?

Definindo expectativas razoáveis para monitoração

Sintomas versus causas

Caixa-preta versus caixa-branca

Os quatro sinais de ouro

Preocupando-se com a cauda (ou instrumentação e desempenho)

Escolhendo uma resolução apropriada para as medições

O mais simples possível, não mais simples que isso

Juntando esses princípios

Monitoração no longo prazo

SRE do Bigtable: uma história de excesso de alertas

Gmail: respostas previsíveis de seres humanos, possíveis de estar em um script

O longo prazo

Conclusão

Capítulo 7 ■ A evolução da automação no Google

O valor da automação

Consistência

Uma plataforma

Correções mais rápidas

Ação mais rápida

Economia de tempo

O valor da SRE no Google

Os casos de uso para automação

Casos de uso para automação pela SRE do Google

Uma hierarquia de classes de automação

[Automatizar a você mesmo para deixar de executar uma tarefa: automatize TUDO!](#)

[Reducindo o sofrimento: aplicando a automação em ativação de clusters](#)

[Detectando inconsistências com o Prodtest](#)

[Resolvendo inconsistências de forma idempotente](#)

[A tendência em especializar](#)

[Ativação de cluster orientada a serviços](#)

[Borg: nascimento do computador em escala de warehouse](#)

[Confiabilidade é a característica fundamental](#)

[Recomendações](#)

[Capítulo 8 ■ Engenharia de release](#)

[O papel de um engenheiro de release](#)

[Filosofia](#)

[Modelo autônomo](#)

[Alta velocidade](#)

[Builds herméticas](#)

[Garantindo o cumprimento de políticas e de procedimentos](#)

[Build e implantação contínuas](#)

[Construção](#)

[Branching](#)

[Testes](#)

[Empacotamento](#)

[Rapid](#)

[Implantação](#)

[Gerenciamento de configuração](#)

[Conclusões](#)

[Não serve apenas para os Googlers](#)

[Use a engenharia de release desde o princípio](#)

[Capítulo 9 ■ Simplicidade](#)

[Estabilidade versus agilidade do sistema](#)

[A virtude do tédio](#)

[Não abrirei mão do meu código!](#)

[A métrica “linhas de código negativas”](#)

[APIs mínimas](#)

[Modularidade](#)

[Simplicidade em releases](#)

[Uma conclusão simples](#)

[Parte III ■ Práticas](#)

[Monitoração](#)

[Resposta a incidentes](#)

[Postmortem e análise de causas-raízes](#)

[Testes](#)

[Planejamento de capacidade](#)

[Desenvolvimento](#)

[Produto](#)

[Outras leituras da SRE do Google](#)

[Capítulo 10 ■ Alertas práticos a partir de dados de séries temporais](#)

[O surgimento do Borgmon](#)

[Instrumentação das aplicações](#)

[Coleta de dados exportados](#)

[Armazenagem na arena de séries temporais](#)

[Rótulos e vetores](#)

[Avaliação de regras](#)

[Geração de alertas](#)

[Fragmentando a topologia de monitoração](#)

[Monitoração caixa-preta](#)

[Mantendo a configuração](#)

[Dez anos se passaram...](#)

[Capítulo 11 ■ De plantão](#)

[Introdução](#)

[Vida de um engenheiro de plantão](#)

[Plantão equilibrado](#)

[Equilíbrio quanto à quantidade](#)

[Equilíbrio quanto à qualidade](#)

[Pagamentos](#)

[Sentindo-se seguro](#)

Evitando uma carga operacional inadequada

Sobrecarga operacional

Um inimigo traiçoeiro: pouca carga operacional

Conclusões

Capítulo 12 ■ Resolvendo problemas de modo eficiente

Teoria

Na prática

Relato do problema

Triagem

Análise

Diagnóstico

Teste e tratamento

Resultados negativos são mágicos

Cura

Estudo de caso

Facilitando a resolução de problemas

Conclusão

Capítulo 13 ■ Resposta a emergências

O que fazer quando os sistemas falham

Emergência induzida por testes

Detalhes

Resposta

Descobertas

Emergência induzida por alterações

Detalhes

Resposta

Descobertas

Emergência induzida por processo

Detalhes

Resposta

Descobertas

Todos os problemas têm soluções

Aprenda com o passado. Não deixe que ele se repita.

Mantenha um histórico das interrupções de serviço

[Faça as perguntas importantes e até mesmo improváveis: E se...?](#)

[Incentive testes proativos](#)

[Conclusão](#)

[Capítulo 14 ■ Administrando incidentes](#)

[Incidentes não administrados](#)

[A anatomia de um incidente não administrado](#)

[Foco centrado no problema técnico](#)

[Comunicação precária](#)

[Trabalho sem coordenação](#)

[Elementos do processo de gerenciamento de incidentes](#)

[Separação recursiva de responsabilidades](#)

[Um posto de comando reconhecido](#)

[Documento vivo do estado do incidente](#)

[Passagem de responsabilidade clara e direta](#)

[Um incidente administrado](#)

[Quando declarar que há um incidente](#)

[Resumindo](#)

[Capítulo 15 ■ Cultura de postmortem: aprendendo com o fracasso](#)

[A filosofia de postmortem no Google](#)

[Colabore e compartilhe conhecimentos](#)

[Introduzindo uma cultura de postmortem](#)

[Conclusão e melhorias contínuas](#)

[Capítulo 16 ■ Monitorando interrupções de serviço](#)

[Escalator](#)

[Outalator](#)

[Agregação](#)

[Atribuição de rótulos](#)

[Análise](#)

[Benefícios inesperados](#)

[Capítulo 17 ■ Testes voltados à confiabilidade](#)

[Tipos de testes de software](#)

[Testes tradicionais](#)

[Testes de produção](#)
[Criando um ambiente de teste e de build](#)
[Testando em escala](#)
[Testando ferramentas escaláveis](#)
[Testando para desastres](#)
[A necessidade de ser rápido](#)
[Atualizando versões em produção](#)
[Falha esperada em testes](#)
[Integração](#)
[Sondas na produção](#)
[Conclusão](#)

[Capítulo 18 ■ Engenharia de software em SRE](#)
[Por que a engenharia de software na SRE é importante?](#)
[Estudo do caso Auxon: contexto do projeto e domínio do problema](#)
[Planejamento de capacidade tradicional](#)
[Nossa solução: planejamento de capacidade baseado em intenção](#)
[Planejamento de capacidade baseado em intenção](#)
[Precursors da intenção](#)
[Introdução ao Auxon](#)
[Requisitos e implementação: sucessos e lições aprendidas](#)
[Aumentando a divulgação e levando à adoção](#)
[Dinâmica das equipes](#)
[Promovendo a engenharia de software na SRE](#)
[Construindo uma cultura de engenharia de software com sucesso em SRE: composição da equipe e tempo de desenvolvimento](#)
[Chegando lá](#)
[Conclusões](#)

[Capítulo 19 ■ Distribuição de carga no frontend](#)
[Capacidade não é a resposta](#)
[Distribuição de carga usando DNS](#)
[Distribuição de carga no endereço IP virtual](#)
[Capítulo 20 ■ Distribuição de carga no datacenter](#)
[O caso ideal](#)
[Identificando tarefas ruins: controle de fluxo e estado de incapacidade](#)

- [Uma abordagem simples para tarefas não saudáveis: controle de fluxo](#)
- [Uma abordagem robusta para tarefas não saudáveis: estado de incapacidade](#)
- [Limitando o pool de conexões com a criação de subconjuntos](#)
 - [Escolhendo o subconjunto correto](#)
 - [Um algoritmo de seleção de subconjunto: criação aleatória de subconjuntos](#)
 - [Um algoritmo de seleção de subconjuntos: criação determinística de subconjuntos](#)
- [Políticas de distribuição de carga](#)
 - [Round Robin Simples](#)
 - [Least-Loaded Round Robin](#)
 - [Weighted Round Robin](#)
- [Capítulo 21 ■ Tratando sobrecarga](#)
 - [As armadilhas das “consultas por segundo”](#)
 - [Limites por cliente](#)
 - [Throttling do lado cliente](#)
 - [Criticidade](#)
 - [Sinais de utilização](#)
 - [Tratando erros de sobrecarga](#)
 - [Decidindo fazer uma nova tentativa](#)
 - [Carga de conexões](#)
 - [Conclusões](#)
- [Capítulo 22 ■ Tratando falhas em cascata](#)
 - [Causas de falhas em cascata e design para evitá-las](#)
 - [Sobrecarga de servidores](#)
 - [Esgotamento de recursos](#)
 - [Indisponibilidade do serviço](#)
 - [Evitando a sobrecarga nos servidores](#)
 - [Gerenciamento de filas](#)
 - [Rejeição de carga e degradação elegante](#)
 - [Retentativas](#)
 - [Latência e tempos de espera](#)
 - [Inicialização lenta e caching frio](#)
 - [Sempre desça na pilha](#)
 - [Condições para disparo de falhas em cascata](#)

Morte de processo
Atualizações de processos
Novos rollouts
Crescimento orgânico
Mudanças, drenagens e desativações planejadas

Testes para falhas em cascata

Teste até falhar, e um pouco além
Teste clientes populares
Teste backends não críticos

Passos imediatos para tratar falhas em cascata

Aumente os recursos
Interrompa as falhas de verificação de sanidade/mortes
Reinic peace os servidores
Desacerte tráfego
Entre no modo de degradação
Elimine a carga em batch
Elimine o tráfego ruim

Observações finais

Capítulo 23 ■ Administrando estados críticos: consenso distribuído para confiabilidade

Motivação para o uso do consenso: falha na coordenação de sistemas distribuídos

Estudo de caso 1: o problema do split-brain
Estudo de caso 2: failover exige intervenção humana
Estudo de caso 3: algoritmo de pertencimento a grupo com falha

Como o consenso distribuído funciona

Visão geral do Paxos: um protocolo de exemplo

Padrões de arquitetura de sistema para o consenso distribuído

Máquinas de estado replicadas confiáveis
Bancos de dados e repositórios de configuração replicados e confiáveis
Processamento altamente disponível usando eleição de líder
Coordenação distribuída e serviços de locking
Enfileiramento distribuído e troca de mensagem confiáveis

Desempenho do consenso distribuído

Multi-Paxos: fluxo de mensagens detalhado

[Escalando cargas de trabalho intensas em leitura](#)
[Leases de quórum](#)
[Desempenho do consenso distribuído e latência de rede](#)
[Pensando no desempenho: Fast Paxos](#)
[Líderes estáveis](#)
[Batching](#)
[Acesso a disco](#)
[Implantando sistemas baseados em consenso distribuído](#)
[Número de réplicas](#)
[Localização das réplicas](#)
[Capacidade e distribuição de carga](#)
[Monitorando sistemas de consenso distribuído](#)
[Conclusão](#)
[**Capítulo 24 ■ Escalonamento periódico e distribuído com o cron**](#)
[Cron](#)
[Introdução](#)
[Ponto de vista da confiabilidade](#)
[Cron jobs e idempotência](#)
[Cron em larga escala](#)
[Infraestrutura estendida](#)
[Requisitos estendidos](#)
[Desenvolvendo o cron no Google](#)
[Monitorando o estado dos cron jobs](#)
[O uso do Paxos](#)
[Os papéis de líder e de seguidor](#)
[Armazenando o estado](#)
[Executando o cron em larga escala](#)
[Resumo](#)
[**Capítulo 25 ■ Pipelines de processamento de dados**](#)
[Origem do padrão de projeto pipeline](#)
[Efeito inicial do Big Data no padrão pipeline simples](#)
[Desafios com o padrão de pipeline periódico](#)
[Problemas causados por distribuição de carda irregular](#)
[Desvantagens de pipelines periódicos em ambientes distribuídos](#)

Monitorando problemas em pipelines periódicos

Problemas de “thundering herd”

Padrão de carga Moiré

Introdução ao Google Workflow

Workflow como um padrão Modelo-Visão-Controlador

Estágios de execução no Workflow

Garantia de que o Workflow está correto

Garantindo a continuidade do negócio

Resumo e considerações finais

Capítulo 26 ■ Integridade de dados: o que você lê é o que você escreveu

Requisitos rigorosos da integridade de dados

Escolhendo uma estratégia para uma integridade de dados superior

Backups versus arquivamentos

Requisitos do ambiente de nuvem em perspectiva

Objetivos da SRE do Google na manutenção da integridade de dados e da disponibilidade

A integridade dos dados é o meio, a disponibilidade é a meta

Entregando um sistema de recuperação, e não um sistema de backup

Tipos de falha que levam à perda de dados

Desafios de manter a integridade de dados ampla e profunda

Como a SRE do Google enfrenta o desafio da integridade de dados

As 24 combinações de modos de falha de integridade de dados

Primeira camada: remoção soft

Segunda camada: backups e seus métodos relacionados de recuperação

Camada abrangente: replicação

1T versus 1E: não é “apenas” um backup maior

Terceira camada: detecção precoce

Saber se a recuperação de dados funcionará

Estudos de caso

Gmail – fevereiro de 2011: restauração a partir do GTape

Google Music – março de 2012: detecção de remoção furtiva

Princípios gerais de SRE conforme aplicados à integridade de dados

Mente de principiante

Confie, mas verifique

Esperança não é uma estratégia
Defesa em profundidade

Conclusão

Capítulo 27 ■ Lançamento de produtos confiáveis em escala

Launch Coordination Engineering

O papel do Launch Coordination Engineer

Definindo um processo de lançamento

A checklist de lançamento

Levando à convergência e à simplificação

Lançando o inesperado

Desenvolvendo uma checklist de lançamento

Arquitetura e dependências

Integração

Planejamento de capacidade

Modos de falha

Comportamento do cliente

Processos e automação

Processo de desenvolvimento

Dependências externas

Planejamento do rollout

Técnicas selecionadas para lançamentos confiáveis

Rollouts graduais e em fases

Frameworks de flag para funcionalidades

Lidando com comportamentos abusivos de clientes

Comportamento de sobrecarga e testes de carga

Desenvolvimento da LCE

Evolução da checklist de LCE

Problemas que a LCE não resolveu

Conclusão

Parte IV ■ Gerenciamento

Outras leituras da SRE do Google

Capítulo 28 ■ Acelerando os SREs para chegar ao plantão e além

Você contratou seus próximos SREs; e agora?

Experiências iniciais de aprendizado: o caso da estrutura sobre o caos
Caminhos de aprendizagem cumulativos e organizados
Trabalho específico em projeto, e não tarefas braçais
Criando profissionais espetaculares em engenharia reversa e em raciocínio de improviso
Engenharia reversa: descobrindo como os sistemas funcionam
Pensar de modo estatístico e comparativo: guardiões do método científico sob pressão
Artistas do improviso: quando o inesperado acontece
Reunindo tudo: engenharia reversa de um serviço em produção
Cinco práticas para os engenheiros que aspiram ao plantão
Fome de falhas: lendo e compartilhando postmortems
Interpretando papéis em situações de desastre
Provoque falhas reais, faça correções reais
Documentação como aprendizado
Acompanhando o plantão com antecedência e frequência
De plantão e além: ritos de passagem e colocando a educação contínua em prática
Considerações finais

Capítulo 29 ■ Lidando com interrupções
Administrando a carga operacional
Fatores que determinam como as interrupções são tratadas
Máquinas imperfeitas
Estado de fluxo cognitivo
Fazer bem uma tarefa
É sério, diga-me o que devo fazer
Reducindo as interrupções

Capítulo 30 ■ Incluindo um SRE para se recuperar de uma sobrecarga operacional
Fase 1: conheça o serviço e saiba qual é o contexto
Identifique as principais causas de estresse
Identifique os fatores que causam estresse
Fase 2: compartilhando o contexto
Escreva um bom postmortem para a equipe

Classifique os fatores que causam estresse de acordo com o tipo
Fase 3: conduzindo as mudanças

Comece pelo básico

Consiga ajuda para eliminar as causas de estresse

Explique o seu raciocínio

Faça perguntas inteligentes

Conclusão

Capítulo 31 ■ Comunicação e colaboração em SRE

Comunicações: reuniões de produção

Agenda

Participação

Colaboração na SRE

Composição da equipe

Técnicas para trabalhar de modo eficiente

Estudo de caso sobre colaboração em SRE: o Viceroy

O surgimento do Viceroy

Desafios

Recomendações

Colaboração fora da SRE

Estudo de caso: migração do DFP para F1

Conclusão

Capítulo 32 ■ O modelo de engajamento da SRE em evolução

Engajamento da SRE: o quê, como e por quê

O modelo PRR

O modelo de engajamento da SRE

Suporte alternativo

Revisões de Prontidão para Produção: Modelo Simples de PRR

Engajamento

Análise

Melhorias e refatoração

Treinamento

Integração à SRE

Melhorias contínuas

Evolução do Modelo Simples de PRR: Engajamento Precoce

Candidatos a um Engajamento Precoce
Vantagens do Modelo de Engajamento Precoce
Desenvolvimento de serviços em evolução: frameworks e plataforma de SRE
Lições aprendidas
Fatores externos que afetam a SRE
Em direção a uma solução estrutural: Frameworks
Novo serviço e vantagens do gerenciamento
Conclusão

Parte V ■ Conclusões

Capítulo 33 ■ Lições aprendidas com outros mercados

Conheça os veteranos do mercado

Preparo e testes para desastre

Foco organizacional incansável em segurança

Atenção aos detalhes

Capacidade de alternância

Simulações e treinamentos ao vivo

Treinamento e certificação

Foco na coleta de requisitos detalhados e no design

Defesa em profundidade e em largura

Cultura de postmortem

Automatizando tarefas repetitivas e o overhead operacional

Tomada de decisão estruturada e racional

Conclusões

Capítulo 34 ■ Conclusão

Apêndice A ■ Tabela de disponibilidade

Apêndice B ■ Um conjunto de melhores práticas para serviços em produção

Falhe de forma saudável

Rollouts progressivos

Defina SLOs como um usuário

Provisões de erro

Monitoração

[Postmortems](#)

[Planejamento de capacidade](#)

[Sobrecargas e falhas](#)

[Equipes de SRE](#)

[Apêndice C ■ Exemplo de documento de estado do incidente](#)

[Apêndice D ■ Exemplo de postmortem](#)

[Lições aprendidas](#)

[Linha do tempo](#)

[Informações para suporte](#)

[Apêndice E ■ Checklist da coordenação de lançamentos](#)

[Apêndice F ■ Exemplo de minutas de reunião de produção](#)

[Bibliografia](#)

[Sobre os autores](#)

[Colofão](#)

Elogios a Engenharia de Confiabilidade do Google¹

Os SREs do Google prestaram um enorme serviço ao nosso mercado escrevendo os princípios, as práticas e os padrões – arquiteturais e culturais – que permitem às suas equipes combinar entregas contínuas com uma confiabilidade de padrão mundial em uma escala absurda. Ler este livro e experimentar essas ideias por conta própria é algo que você deve a si mesmo e à sua empresa.

— Jez Humble, coautor de *Entrega contínua* e *Lean Enterprise*

Eu me lembro de quando o Google começou a falar em conferências de administração de sistemas. Era como ouvir uma palestra de um especialista em monstro-de-gila em um show de répteis. Certamente era interessante ouvir falar de um mundo bem diferente, mas no final, o público retornava aos seus mundos.

Atualmente vivemos em um universo diferente, em que as práticas operacionais do Google não estão tão distantes daqueles que trabalham em uma escala menor. De repente, as melhores práticas de SRE, aprimoradas com o passar dos anos, passaram a ser de profundo interesse para o restante de nós. Para aqueles que encaram desafios relacionados a escala, confiabilidade e operações, este livro veio em boa hora.

— David N. Blank-Edelman, Diretor, membro do conselho administrativo da USENIX, fundador e coorganizador da SREcon

Fiquei esperando este livro desde que saí do castelo encantado do Google. É uma doutrina que prego aos meus colegas de trabalho.

— Björn Rabenstein, Líder de Equipe de Engenharia de Produção do SoundCloud, desenvolvedor do Prometheus e SRE do Google até 2013

Uma discussão abrangente sobre Site Reliability Engineering pela empresa que inventou o conceito. Inclui não só os detalhes técnicos, mas também o processo de raciocínio, os objetivos, os princípios e as lições aprendidas com o tempo. Se quiser saber o que a SRE realmente significa, comece por aqui.

— Russ Allbery, SRE e Engenheiro de Segurança

Neste livro, os funcionários do Google compartilham os processos que seguiram, incluindo os passos incorretos, e que permitiram que os serviços do Google se expandissem em escala massiva e com alto grau de confiabilidade. Recomendo fortemente a todos que queiram criar um conjunto de serviços integrados, e que esperam que ele vá ser escalado, que leiam este livro. O livro é um guia escrito por pessoas de dentro da empresa para desenvolver serviços possíveis de manter.

— Rik Farrow, USENIX

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

— Urs Hölzle, SVP de Infraestrutura Técnica, Google

¹ N.T.: Utilizaremos no livro o título original *Site Reliability Engineering* e sua sigla correspondente *SRE*, por estarem muito consolidados no mercado.

Apresentação

A história do Google é uma história de crescimento. É uma das histórias de maior sucesso no mercado de informática, assinalando uma mudança em direção a negócios centrados em TI. O Google foi uma das primeiras empresas a definir na prática o que significa um alinhamento entre negócios e TI, e prosseguiu divulgando o conceito de DevOps a uma comunidade de TI mais ampla. Este livro foi escrito por uma variada amostra das próprias pessoas que fizeram dessa transição uma realidade.

O Google cresceu em uma época em que o papel tradicional do administrador de sistemas estava em transformação. A empresa questionou a administração de sistemas, como se dissesse: não podemos nos apegar à tradição como se fosse uma autoridade; devemos pensar de modo inovador, e não temos tempo para esperar que todos os demais nos acompanhem. Na introdução do livro *Princípios de administração de redes e sistemas* [Bur99], argumentei que a administração de sistemas era uma forma de engenharia entre seres humanos e computadores. Essa ideia foi fortemente rejeitada por alguns analistas que disseram que “ainda não estamos no estágio em que podemos chamar a isso de engenharia”. Na época, senti que a área havia se perdido, presa em sua própria cultura encantada, e não era capaz de enxergar um caminho para avançar. Então o Google traçou uma linha na areia, forçando aquele destino a se tornar realidade. A nova função foi chamada de SRE, ou Site Reliability Engineer (Engenheiro de Confiabilidade de Sites). Alguns de meus amigos estavam entre os primeiros dessa nova geração de engenheiros: formalizaram a ideia usando software e automação. Inicialmente, eles trabalhavam de modo altamente sigiloso, e o que acontecia dentro e fora do Google era bem diferente: a experiência do Google era única. Com o tempo, as informações e os métodos fluíram em ambas as direções. Este livro mostra uma disposição para deixar que as ideias de SRE saiam das sombras.

Nesta obra, não só veremos como o Google construiu sua infraestrutura

lendária, como também de que modo a empresa estudou, aprendeu e mudou sua mentalidade acerca das ferramentas e das tecnologias no caminho. Nós também podemos encarar desafios assustadores com a mente aberta. A natureza tribal da cultura de TI muitas vezes faz com que os praticantes fiquem entrincheirados em posições dogmáticas que impedem o mercado de avançar. Se o Google foi capaz de vencer essa inércia, nós também podemos.

Este livro é uma coletânea de artigos de uma empresa, com uma única visão em comum. O fato de as contribuições estarem alinhadas em torno do objetivo de uma única empresa é o que o torna especial. Há temas e personagens (sistemas de software) comuns, recorrentes em vários capítulos. Vemos opções a partir de diferentes pontos de vista e sabemos que elas estão correlacionadas para resolver interesses em conflito. Os artigos não são textos acadêmicos rigorosos; são relatos pessoais, escritos com orgulho, em uma variedade de estilos próprios, do ponto de vista de conjuntos individuais de habilidades. Foram escritos com coragem e uma honestidade intelectual que é revigorante e incomum na literatura do mercado. Algumas pessoas dizem “nunca faça isso, sempre faça aquilo”, enquanto outras são mais filosóficas e dispostas a experimentar, refletindo a variedade de personalidades em uma cultura de TI, e como isso também desempenha um papel na história. Nós, por outro lado, lemos os artigos com a humildade de observadores que não fizeram parte da jornada e que não têm todas as informações sobre a diversidade dos desafios conflitantes. Nossas várias perguntas são o verdadeiro legado da obra: Por que eles fizeram X? O que teria acontecido se tivessem feito Y? Como veremos isso daqui a alguns anos? É comparando nossas ideias com o raciocínio que está no livro que podemos avaliar nossos próprios pensamentos e experiências.

O aspecto mais impressionante de todo este livro é a sua própria existência. Atualmente, vemos uma cultura descarada do “simplesmente me mostre o código”. Uma cultura de “não fazer perguntas” se desenvolveu em torno do código aberto, em que se valoriza a comunidade, e não o expertise. O Google é uma empresa que ousou pensar nos problemas desde os princípios iniciais e empregar os melhores talentos, com uma elevada proporção de PhDs. As ferramentas eram apenas componentes nos processos, funcionando em cadeias de softwares, pessoas e dados. Nada do que está aqui nos diz como

resolver problemas de forma universal, mas essa é a questão. Histórias como essas são muito mais valiosas que os códigos 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.

Essa, então, é a história de como uma empresa fez isso. O fato de haver muitas histórias que se sobrepõem nos mostra que escalar é muito mais que apenas uma ampliação fotográfica de uma arquitetura de computadores em um livro didático. Tem a ver com escalar um processo de negócios, e não apenas os equipamentos. Essa lição sozinha vale o seu peso em papel eletrônico.

Não nos envolvemos muito em análises autocríticas no mundo de TI; por isso, há muita reinvenção e repetição. Durante vários anos, havia apenas a comunidade da conferência USENIX LISA discutindo infraestrutura de TI, além de algumas conferências sobre sistemas operacionais. Hoje em dia é bem diferente, mas, apesar disso, este livro ainda nos dá a impressão de ser uma oferta rara: uma documentação detalhada do passo do Google em direção a uma época que é um divisor de águas. A narrativa não serve para ser copiada – mas talvez para ser emulada –, porém, pode inspirar o próximo passo para todos nós. Há uma honestidade intelectual sem igual nessas páginas, que expressa tanto liderança quanto humildade. São histórias de esperanças, medos, sucessos e falhas. Eu saúdo a coragem dos autores e editores em permitir tanta franqueza para que nós, que não fazemos parte das experiências diretas, possamos também nos beneficiar das lições aprendidas, protegidos por um casulo.

— Mark Burgess
autor de *In Search of Certainty*
Oslo, março de 2016

Prefácio

A engenharia de software tem um aspecto em comum com ter filhos: o trabalho de parto *antes* do nascimento é doloroso e difícil, porém é no trabalho *depois* do nascimento que você investe a maior parte de seus esforços. Entretanto, a engenharia de software como disciplina gasta muito mais tempo falando do primeiro período em oposição ao segundo, apesar das estimativas de que 40 a 90% dos custos totais de um sistema ocorrem após o nascimento.¹ O modelo popular do mercado que concebe o software operacional implantado como “estabilizado” em produção e, desse modo, precisando de muito menos atenção dos engenheiros de software, está incorreto. Através dessas lentes, então, vemos que, se a engenharia de software tende a se concentrar no design e na construção de sistemas de software, deve haver outra disciplina que enfoque o ciclo de vida *todo* dos objetos de software, desde a concepção, passando pela implantação e operação, pelos ajustes e, em algum momento, pela desativação pacífica. Essa disciplina usa – e deve usar – uma grande variedade de habilidades, porém tem preocupações distintas de outros tipos de engenharias. Atualmente, nossa resposta é a disciplina que o Google chama de Site Reliability Engineering (Engenharia de Confiabilidade de Sites).

Então, o que é exatamente a Site Reliability Engineering (SRE)? Admitimos que esse não é um nome particularmente claro para o que fazemos – praticamente qualquer engenheiro de confiabilidade do Google é indagado sobre o significado exato disso e o que eles fazem realmente no dia a dia.

Analizando o termo por partes, antes de tudo, os SREs são *engenheiros*. Aplicamos os princípios de ciência da computação e de engenharia no design e no desenvolvimento de sistemas computacionais: em geral, sistemas grandes e distribuídos. Às vezes, nossa tarefa é escrever o software para esses sistemas juntamente com nossas contrapartidas na equipe de desenvolvimento de produto; outras vezes, nossa tarefa é construir todas as

partes adicionais necessárias a esses sistemas, como backups ou distribuição de carga, para que, de modo ideal, possam ser reutilizadas em sistemas diferentes; em algumas ocasiões, nossa tarefa é descobrir como aplicar soluções existentes a novos problemas.

Em seguida, vamos nos concentrar na *confiabilidade* (reliability) do sistema. Ben Treynor Sloss, VP de Operações 24/7 do Google, criador do termo SRE, argumenta que confiabilidade é a característica mais fundamental de qualquer produto: um sistema não será muito útil se ninguém puder usá-lo! Como a confiabilidade² é muito crítica, os SREs têm como foco descobrir maneiras de melhorar o design e a operação dos sistemas para deixá-los mais escaláveis, mais confiáveis e mais eficientes. No entanto, investimos esforços nessa direção apenas até certo ponto: quando os sistemas são “confiáveis o suficiente”, passamos a investir nossos esforços em acrescentar funcionalidades ou desenvolver novos produtos.³

Por fim, os SREs enfocam serviços em operação, desenvolvidos com base em nossos sistemas computacionais distribuídos, independentemente de esses serviços serem para armazenagens de escala mundial, emails para centenas de milhões de usuários ou aquele com que o Google começou: pesquisas web. O “site” em nosso nome originalmente se referia ao papel dos SREs em manter o site *google.com* executando, embora, atualmente, operamos vários outros serviços, muitos dos quais não são sites – variam de infraestruturas internas, como o Bigtable, a produtos para desenvolvedores externos, como o Google Cloud Platform.

Apesar de termos representado a SRE como uma disciplina ampla, não é de se surpreender que ela tenha surgido no mundo de rápidas transformações dos web services e, talvez, quanto à origem, deva algo às peculiaridades de nossa infraestrutura. Também não é nenhuma surpresa que, de todas as características da pós-implantação de software para as quais poderíamos ter dedicado atenção especial, a confiabilidade é aquela que consideramos como principal.⁴ O domínio dos web services, seja porque o processo de melhorar e mudar o software do lado do servidor é comparativamente contido, seja porque o gerenciamento de alterações por si só está fortemente ligado a falhas de todos os tipos, é uma plataforma natural a partir da qual nossa abordagem

poderia ter nascido.

Apesar de ter surgido no Google, e na comunidade web de modo mais geral, achamos que essa disciplina tem lições aplicáveis a outras comunidades e empresas. Este livro é uma tentativa de explicar como trabalhamos: serve tanto para que outras empresas possam fazer uso do que aprendemos quanto para podermos definir melhor os papéis e os significados dos termos. Para isso, organizamos o livro de modo que os princípios gerais e as práticas mais específicas estejam separados sempre que possível e, quando for apropriado discutir um assunto em particular com informações específicas do Google, acreditamos que o leitor nos dará permissão para fazer isso e não temerá tirar conclusões úteis sobre o seu próprio ambiente.

Também incluímos alguns materiais para orientação – uma descrição do ambiente de produção do Google e um mapeamento entre alguns de nossos softwares internos e os softwares publicamente disponíveis –, o que poderá ajudar a contextualizar o que dissermos, deixando a informação mais utilizável de forma direta.

Em última instância, é claro, softwares e uma engenharia de sistemas mais orientados à confiabilidade são inherentemente bons. No entanto, reconhecemos que empresas menores podem estar se perguntando qual é a melhor maneira de usar a experiência apresentada aqui: de modo muito semelhante à segurança, quanto antes você se preocupar com a confiabilidade, melhor será. Isso implica que, apesar de uma empresa pequena ter muitas preocupações imediatas e as escolhas de software que você fizer poderem ser distintas daquelas feitas pelo Google, ainda assim vale a pena implantar um suporte leve para a confiabilidade com antecedência, pois expandir uma estrutura mais tarde custará menos que introduzir um sistema que não esteja presente. A Parte IV contém algumas boas práticas para treinamento, comunicação e reuniões que achamos que funcionam bem para nós, muitas das quais serão imediatamente utilizáveis em sua empresa.

Porém, para tamanhos entre uma startup e uma multinacional, é provável que já exista alguém em sua empresa fazendo o trabalho de SRE, sem que essa tarefa seja necessariamente chamada com esse nome ou reconhecida como tal. Outra maneira de começar a trilhar o caminho da melhoria de

confiabilidade em sua empresa é reconhecer formalmente esse trabalho ou encontrar esse tipo de profissional e promover o que eles fazem – pagando-os. São pessoas que estão nos extremos entre uma maneira ou outra de ver o mundo: como Newton, que às vezes é chamado não de primeiro físico, mas de último alquimista do mundo.

Adotando o ponto de vista histórico, quem, então, poderia ser o primeiro SRE, se olharmos para trás?

Gostamos de pensar que Margaret Hamilton, emprestada do MIT para trabalhar no programa Apollo, tinha todos os traços significativos da primeira SRE.⁵ Em suas próprias palavras, “parte da cultura consistia em aprender de tudo e de todos, incluindo daqueles de quem menos esperávamos”.

Segundo um episódio ilustrativo, ela trouxe sua filhinha Lauren para o trabalho um dia, quando parte da equipe estava executando cenários da missão no computador híbrido de simulação. Como fazem as crianças, Lauren saiu para explorar e fez uma “missão” falhar ao selecionar teclas do DSKY de forma inesperada, alertando a equipe sobre o que poderia acontecer se o programa de pré-lançamento P01 fosse inadvertidamente selecionado por um verdadeiro astronauta, durante uma missão real, no meio de um percurso. (Iniciar P01 inadvertidamente em uma missão real seria um grande problema, pois ele limpava os dados de navegação, e o computador não estava equipado para pilotar a espaçonave sem dados de navegação.)

Com os instintos de uma SRE, Margaret submeteu uma requisição de mudança do programa para que um código especial de verificação de erro fosse acrescentado no software de voo, caso um astronauta, por acidente, selecionasse P01 durante o voo. Porém, essa ação foi considerada desnecessária pelas pessoas “lá de cima” da NASA: é claro que isso jamais aconteceria! Então, em vez de adicionar um 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, muitos participantes do projeto zombaram da atualização, pois várias vezes lhes foi dito que os astronautas não cometiam nenhum erro – afinal de contas, eles eram treinados para serem perfeitos.)

Bem, a proteção sugerida por Margaret foi considerada desnecessária

somente até a próxima missão, na Apollo 8, apenas alguns dias depois da atualização das especificações. Durante o percurso no quarto dia de voo com os astronautas Jim Lovell, William Anders e Frank Borman a bordo, Jim Lovell selecionou P01 por engano – por acaso, no dia de Natal –, gerando muita confusão para todos os envolvidos. Foi um problema crítico, pois, na ausência de uma solução alternativa, não ter dados de navegação significava que os astronautas jamais voltariam para casa. Felizmente, a atualização na documentação havia definido explicitamente essa possibilidade, e teve um valor inestimável para determinar como fazer o upload de dados utilizáveis e recuperar a missão, sem que houvesse muito desperdício de tempo.

Como disse Margaret, “uma compreensão total de como operar os sistemas não era suficiente para evitar erros humanos”, e a solicitação de alteração para incluir um software para detecção de erro e recuperação no programa de pré-lançamento P01 foi aprovada logo depois disso.

Embora o incidente com a Apollo 8 tenha ocorrido há décadas, há muitos aspectos nos parágrafos anteriores que são diretamente relevantes para as vidas dos engenheiros atualmente, e muito disso continuará a ser relevante no futuro. Do mesmo modo, para os sistemas sob sua responsabilidade, nos grupos em que trabalhar ou nas empresas que estiver construindo, por favor, tenha em mente o Modo SRE: meticulosidade e dedicação, acreditar no valor da preparação e da documentação e ter consciência do que pode dar errado, em conjunto com um forte desejo de evitar isso. Bem-vindo à nossa profissão emergente!

Como ler este livro

Este livro contém uma série de artigos escritos por membros e alunos da organização Site Reliability Engineering do Google. A obra se assemelha mais a palestras de conferências do que a um livro-padrão de um autor ou de um pequeno grupo deles. Cada capítulo tem como propósito ser lido como parte de um todo coerente, mas você estará fazendo um bom negócio se ler qualquer assunto que seja de seu interesse. (Se houver outros artigos que deem suporte ou informações sobre o texto, eles serão referenciados para que você possa dar prosseguimento de forma apropriada.)

Não é preciso ler em nenhuma sequência em particular, embora sugerimos, no mínimo, começar pelos Capítulos 2 e 3, que descrevem o ambiente de produção do Google e mostram como a SRE aborda riscos, respectivamente. Em muitos aspectos, o risco é a qualidade fundamental de nossa profissão.) Ler o livro de ponta a ponta, é claro, também é útil e possível; nossos capítulos estão agrupados por temas em Princípios (Parte II), Práticas (Parte III) e Gerenciamento (Parte IV). Cada parte tem uma rápida introdução que destaca do que se tratam as partes individuais, além de referenciar outros artigos publicados pelos SREs do Google, abordando assuntos específicos com mais detalhes. Além disso, o site da empresa para este livro, <https://g.co/SREBook>, tem diversos recursos úteis.

Esperamos que este livro seja tão útil e interessante a você quanto foi compilá-lo para nós.

— Os editores

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

Indica termos novos, URLs, endereços de email, nomes e extensões de arquivos.

Largura constante

Usada para listagens de programas, assim como em parágrafos para se referir a elementos de programas, como nomes de variáveis ou de funções, bancos de dados, tipos de dados, variáveis de ambiente, comandos e palavras-chave.

Largura constante em negrito

Mostra comandos ou outro texto que devam ser digitados literalmente pelo usuário.

Largura constante em itálico

Mostra o texto que deve ser substituído por valores fornecidos pelo usuário

ou determinados pelo contexto.



Este elemento significa uma dica ou sugestão.



Este elemento significa uma observação geral.



Este elemento significa um aviso ou uma precaução.

Uso de exemplos de código de acordo com a política da O'Reilly

Materiais suplementares estão disponíveis em <https://g.co/SREBook>.

Este livro está aqui para ajudá-lo a fazer seu trabalho. De modo geral, se este livro incluir exemplos de código, você poderá usá-los em seus programas e em sua documentação. Você não precisa nos contatar para pedir permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que use diversas partes de código deste livro não requer permissão. Porém, vender ou distribuir um CD-ROM de exemplos de livros da O'Reilly requer permissão. Responder a uma pergunta mencionando este livro e citar o código de exemplo não requer permissão. Em contrapartida, incluir uma quantidade significativa de código de exemplos deste livro na documentação de seu produto requer permissão.

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

Se você achar que o seu uso dos exemplos de código está além do razoável ou da permissão concedida, sinta-se à vontade para nos contatar em permissions@oreilly.com.

Como entrar em contato com a Novatec

Envie seus comentários e suas dúvidas sobre este livro à editora escrevendo para: novatec@novatec.com.br.

Temos uma página web para este livro na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição em português:

<http://www.novatec.com.br/catalogo/7522517-reliability-engineering>

- Página da edição original em inglês:

<http://bit.ly/site-reliability-engineering>

Para obter mais informações sobre os livros da Novatec, acesse nosso site em <http://www.novatec.com.br>.

Agradecimentos

Este livro não teria sido possível sem os esforços incansáveis de nossos autores e escritores técnicos. Gostaríamos também de agradecer aos seguintes revisores internos por oferecerem feedbacks especialmente valiosos: 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 operar serviços de larga escala foram essenciais para que este livro se tornasse realidade.

Gostaríamos de enviar agradecimentos especiais a Rik Farrow, o editor de *:login:*, pela parceria conosco em diversas contribuições para pré-publicação por meio do USENIX.

Embora tenhamos agradecido especificamente aos autores nos capítulos, gostaríamos de aproveitar para reconhecer aqueles que contribuíram em cada capítulo oferecendo ideias, discussões e análises aprofundadas.

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, Štěpán 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 colaboradores a seguir, que ofereceram material significativo, fizeram um excelente trabalho de revisão, concordaram em ser entrevistados, forneceram expertise ou recursos significativos ou produziram outro excelente efeito 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 Palatin, 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.

Apreciamos muito os feedbacks inteligentes e profundos recebidos 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 original do livro e igualmente conspirador, que saiu do Google antes que esse projeto estivesse concluído, mas que exerceu profunda influência sobre ele, e a Margaret Hamilton, que afavelmente nos permitiu mencionar sua história em nosso prefácio. Além disso, gostaríamos de estender nossos agradecimentos especiais a Shylaja Nukala, que generosamente nos concedeu o tempo de seus escritores técnicos e deu apoio aos seus esforços necessários e valiosos com todo o seu coração.

Os editores também gostariam de agradecer pessoalmente às seguintes pessoas:

Betsy Beyer: para minha avó (minha heroína pessoal), por proporcionar inúmeras conversas incentivadoras ao telefone e pelo suprimento de pipoca; e a Riba, por me fornecer as calças de moletom necessárias para me abastecer por várias noites adentro. Isso, é claro, além dos agradecimentos à equipe de estrelas de SREs, que foram colaboradores realmente afáveis.

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

Jennifer Petoff: para meu marido Scott, por ter me apoiado incrivelmente durante o processo de dois anos de escrita deste livro e por manter os editores com um suprimento abundante de açúcar em nossa “Dessert Island”.

Niall Murphy: para Léan, Oisín e Fiachra, que foram consideravelmente mais pacientes do que eu tinha o direito de esperar, com um pai e um marido muito mais ranzinza que o normal durante anos. Para Dermot, pela oferta de transferência.

¹ O próprio fato de haver essa variação tão grande nas estimativas nos diz algo sobre a engenharia de software como disciplina, mas consulte, por exemplo, [Gla02], para ver mais detalhes.

² Em nosso caso, a confiabilidade é “a probabilidade de [um sistema] executar uma função necessária sem falhas, em condições definidas por um período de tempo específico”, de acordo com a definição em [Oco12].

³ Os sistemas de software em que estamos interessados são, em geral, sites e serviços semelhantes; não discutiremos as preocupações com confiabilidade relacionadas a softwares voltados para instalações de energia nuclear, aeronaves, equipamentos médicos ou outros sistemas de segurança crítica. Porém, vamos comparar nossas abordagens àquelas usadas em outros mercados no Capítulo 33.

⁴ Nesse aspecto, nos diferenciamos do termo de mercado DevOps, pois embora, definitivamente, consideremos a infraestrutura como código, temos a *confiabilidade* como nosso foco principal. Além do mais, estamos fortemente orientados a acabar com a necessidade de operações – consulte o Capítulo 7 para ver mais detalhes.

⁵ Além dessa ótima história, argumenta-se fortemente que ela tenha popularizado o termo “engenharia de software”.

PARTE I

Introdução

Esta seção apresenta algumas explicações gerais sobre o que é SRE e por que ela é diferente das práticas mais convencionais de TI do mercado.

Ben Treynor Sloss, VP sênior responsável pelas operações técnicas do Google – e criador do termo “Site Reliability Engineering” –, apresenta sua visão do que significa SRE, como ela funciona e como se compara a outras maneiras de fazer o trabalho no mercado, no Capítulo 1.

Incluímos um guia para o ambiente de produção do Google no Capítulo 2 como uma maneira de ajudar você a conhecer a variedade de novos termos e sistemas que está prestes a ser apresentada no restante do livro.

CAPÍTULO 1

Introdução

Escrito por Benjamin Treynor Sloss¹

Editado por Betsy Beyer

Esperança não é uma estratégia.

— Ditado tradicional de SRE

É uma verdade universalmente aceita que os sistemas não se operam por si sós. Então, como um sistema – particularmente, um sistema computacional complexo que opera em larga escala – *deveria* ser executado?

A abordagem com administradores de sistemas para gerenciamento de serviços

Historicamente, as empresas têm empregado administradores de sistemas para operar sistemas computacionais complexos.

Essa abordagem com administradores de sistemas (ou sysadmins) envolve reunir componentes de software existentes e implantá-los para que funcionem em conjunto de modo a gerar um serviço. Aos administradores de sistema, então, é atribuída a tarefa de operar o serviço e responder a eventos e a atualizações quando ocorrerem. À medida que a complexidade e o volume de tráfego do sistema aumentam, gerando um aumento correspondente de eventos e de atualizações, a equipe de administração de sistemas cresce para absorver o trabalho adicional. Como o papel de administrador de sistema exige um conjunto específico de habilidades em comparação com aquele exigido dos desenvolvedores de um produto, os desenvolvedores e os administradores de sistema são divididos em equipes distintas: “desenvolvimento” e “operações” ou “ops”.

O modelo com administradores de sistema para gerenciamento de serviços tem diversas vantagens. Para as empresas que estão decidindo como operar um serviço e montar uma equipe para isso, essa abordagem é relativamente fácil de implementar: por ser um paradigma conhecido no mercado, há muitos exemplos com os quais aprender e que podemos emular. Um grupo relevante de talentos já está amplamente disponível. Um conjunto de ferramentas, componentes de software (de prateleira ou não) e empresas de integração já estão disponíveis para ajudar a operar os sistemas montados, portanto uma equipe iniciante de administradores de sistema não precisará reinventar a roda nem fazer o design de um sistema do zero.

A abordagem com administradores de sistema e a separação entre desenvolvimento/ops que a acompanha apresentam algumas desvantagens e armadilhas. De modo geral, elas se enquadram em duas categorias: custos diretos e custos indiretos.

Os custos diretos não são nem sutis nem ambíguos. Operar um serviço com uma equipe que dependa de intervenção manual tanto para administrar mudanças quanto para tratar eventos se torna custoso à medida que o serviço e/ou o tráfego para o serviço aumentam, pois o tamanho da equipe é necessariamente escalado de acordo com a carga gerada pelo sistema.

Os custos indiretos da divisão desenvolvimento/ops podem ser sutis, mas frequentemente são mais altos para a empresa do que os custos diretos. Esses custos surgem do fato de as duas equipes terem experiências anteriores, conjunto de habilidades e pagamento de incentivos bem diferentes. Elas usam um vocabulário diferente para descrever as situações, além de fazerem pressuposições diferentes sobre risco e possibilidades de soluções técnicas e sobre o nível visado para a estabilidade do produto. A separação entre os grupos pode facilmente passar a ser não só em relação a pagamentos de incentivos, mas também a comunicação, metas e, após um tempo, até mesmo em confiança e respeito. O resultado é uma patologia.

Desse modo, equipes tradicionais de operações e suas contrapartidas em desenvolvimento de produtos com frequência acabam em conflito, em que o mais visível diz respeito à rapidez com que o software pode ser disponibilizado para produção. Em sua essência, as equipes de

desenvolvimento querem lançar novas funcionalidades e vê-las adotadas pelos usuários. As equipes de operações, em *sua essência*, querem garantir que o serviço não falhe enquanto estiverem de posse do pager. Como a maioria das interrupções de serviço é causada por algum tipo de mudança – uma nova configuração, o lançamento de uma nova funcionalidade ou um novo tipo de tráfego de usuário –, as metas das duas equipes entram fundamentalmente em conflito.

Os dois grupos entendem que é inaceitável definir seus interesses nos termos mais francos possíveis (“Queremos lançar tudo, a qualquer momento, sem impedimentos” *versus* “Não queremos jamais mudar nada no sistema depois que ele estiver funcionando”). Como seu vocabulário e as pressuposições quanto aos riscos diferem, os dois grupos muitas vezes recorrem a uma forma familiar de guerra de trincheiras para fazer seus interesses prevalecerem. A equipe de operações tenta proteger o sistema em execução contra o risco de mudanças introduzindo pontos de verificação (gates) para lançamentos e mudanças. Por exemplo, revisões para lançamento podem conter uma verificação explícita de *todos* os problemas que *alguma vez* já causaram uma interrupção de serviço no passado – pode ser uma lista arbitrariamente longa, em que nem todos os elementos têm o mesmo valor. A equipe de desenvolvimento rapidamente aprende a responder. Essas equipes fazem menos “lançamentos” e criam mais “flags de ativação”, fazem “atualizações incrementais” ou “cherry-picks”². Adotam táticas como dividir o produto em partes para que menos funcionalidades estejam sujeitas à revisão de lançamento.

A abordagem do Google para o gerenciamento de serviços: Site Reliability Engineering

O conflito não é uma parte inevitável no oferecimento de um serviço de software. No Google optamos por operar nossos sistemas usando uma abordagem diferente: nossas equipes de Site Reliability Engineering (Engenharia de Confiabilidade de Sites) se concentram em contratar engenheiros de software para operar nossos produtos e criar sistemas para realizar o trabalho que, de outro modo, seria realizado por administradores de

sistemas, com frequência, manualmente.

O que é exatamente Site Reliability Engineering, conforme passou a ser definido pelo 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 me juntei ao Google em 2003 e me foi atribuída a responsabilidade de operar uma “Equipe de Produção” com sete engenheiros, minha vida toda até então havia sido em engenharia de software. Portanto, projetei e administrei o grupo do modo como *eu* gostaria que ele funcionasse se eu mesmo trabalhasse como um SRE. Esse grupo, desde então, amadureceu até se tornar a equipe de SRE do Google como é hoje, que permanece fiel às suas origens conforme a visão de um engenheiro de software de longa data.

Um bloco de construção principal da abordagem do Google ao gerenciamento de serviços é a composição de cada equipe de SRE. Como um todo, a SRE pode ser dividida em duas categorias principais.

De 50 a 60% são Engenheiros de Software do Google (Google Software Engineers) 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 ficaram muito próximos das qualificações de Engenheiros de Software do Google (isto é, com 85 a 99% do conjunto de habilidades exigido) e que, *além disso*, tinham um conjunto de habilidades técnicas úteis à SRE, mas raro na maioria dos engenheiros de software. De longe, expertise no funcionamento interno do sistema Unix e em redes (Camadas 1 a 3) são os dois tipos mais comuns de habilidades técnicas alternativas que procuramos.

Comum a todos os SREs é a crença e a aptidão para desenvolver sistemas de software a fim de resolver problemas complexos. Na SRE monitoramos o progresso na carreira de ambos os grupos bem de perto e, até agora, não percebemos nenhuma diferença prática de desempenho entre os engenheiros dos dois lados. Com efeito, a experiência prévia, de certo modo, diferente, da equipe de SRE com frequência resulta em sistemas de alta qualidade e inteligentes, que são claramente o produto da síntese de vários conjuntos de habilidades.

O resultado de nossa abordagem na contratação de SREs é que acabamos com uma equipe de pessoas que (a) ficará rapidamente entediada realizando tarefas manualmente e (b) tem o conjunto de habilidades necessário para escrever um software de modo a substituir seu trabalho anteriormente manual, mesmo quando a solução for complicada. Os SREs também acabam compartilhando experiências acadêmicas e intelectuais anteriores com o restante da equipe de desenvolvimento. Desse modo, a SRE consiste essencialmente em fazer o trabalho que, historicamente, era feito por uma equipe de operações, porém usando engenheiros com expertise em software, e contando com o fato de que esses engenheiros, de forma inerente, são predispostos e têm a habilidade para fazer o design e implementar automações com um software para substituir o trabalho humano.

Por design, é fundamental que as equipes de SRE estejam focadas em engenharia. Sem uma engenharia constante, a carga das operações aumenta e as equipes precisarão de mais pessoas somente para acompanhar esse aumento da carga de trabalho. Em algum momento no futuro, um grupo tradicional focado em operações escalará linearmente conforme o volume do serviço: se os produtos aos quais o serviço dá suporte 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 responsável pelo gerenciamento de um serviço deve escrever código; do contrário, ficará sobrecarregada. Desse modo, o Google coloca *um limite de 50% no trabalho agregado de “ops” para todos os SREs* – tickets, plantão, tarefas manuais etc. Esse limite garante que a equipe de SRE tenha tempo suficiente em seus cronogramas para deixar o serviço estável e operacional. Esse é um limite superior; com o tempo, se deixados por conta própria, a equipe de SRE deverá acabar com pouca carga operacional e se envolverá quase totalmente em tarefas de desenvolvimento, pois o serviço basicamente opera e se corrige sozinho: queremos sistemas que sejam *automáticos*, e não apenas *automatizados*. Na prática, a escala e novas funcionalidades sustentam os SREs.

A regra geral do Google é que uma equipe de SRE deve gastar os outros 50% de seu tempo fazendo desenvolvimento. Então, como garantimos o

cumprimento desse limite? Em primeiro lugar, precisamos mensurar como o tempo dos SREs é gasto. Com esse dado em mãos, garantimos que as equipes que estejam gastando menos de 50% de seu tempo, de forma consistente, em trabalhos de desenvolvimento mudem suas práticas. Com frequência, isso quer dizer passar parte da carga de operações de volta à equipe de desenvolvimento ou aumentar a equipe sem lhe atribuir novas responsabilidades operacionais. Manter esse equilíbrio entre tarefas de ops e de desenvolvimento de forma consciente nos permite garantir que os SREs tenham tempo para se envolver em uma engenharia criativa e autônoma, ao mesmo tempo que preservam a sabedoria adquirida do lado das operações de um serviço.

Descobrimos que a abordagem de SRE do Google para operar sistemas de larga escala tem muitas vantagens. Como os SREs modificam diretamente o código em sua busca por fazer com que os sistemas do Google executem sozinhos, as equipes de SRE têm como características tanto a inovação rápida quanto um alto grau de aceitação de mudanças. Essas equipes têm custo relativamente baixo – 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 necessário para operar, manter e melhorar um sistema escala de forma proporcionalmente menor ao tamanho do sistema. Por fim, a SRE não só contorna a falta de funcionalidade da divisão dev/ops, como também essa estrutura melhora nossas equipes de desenvolvimento de produtos: transferências fáceis entre as equipes de desenvolvimento de produtos e de SRE possibilitam um treinamento cruzado de todo o grupo e melhoram as habilidades dos desenvolvedores que, de outro modo, poderiam ter dificuldades em aprender a construir um sistema distribuído de um milhão de núcleos de CPU (cores).

Apesar desses ganhos líquidos, o modelo de SRE é caracterizado pelo seu próprio conjunto distinto de desafios. Um desafio contínuo que o Google enfrenta é contratar os SREs: o SRE não só concorre com os mesmos candidatos do processo de contratação da equipe de desenvolvimento de produtos, como também o fato de definirmos a qualificação para contratação em um nível muito alto no que diz respeito às habilidades tanto de programação quanto de engenharia de sistemas implica que nossas opções de

candidatos à contratação são necessariamente reduzidas. Como nossa área é relativamente nova e única, não há muitas informações no mercado sobre como montar e administrar uma equipe de SRE (espero, porém, que este livro dê alguns passos nessa direção!). Depois que uma equipe de SRE estiver montada, suas abordagens possivelmente não ortodoxas à administração de serviços exigem um forte apoio da gerência. Por exemplo, a decisão de interromper os lançamentos de novas versões pelo restante do trimestre depois que uma provisão para erros (error budget) tenha sido consumida pode não ser aceita por uma equipe de desenvolvimento de produto, a menos que seja imposta pela gerência.

DevOps ou SRE?

O termo “DevOps” surgiu no mercado no final de 2008 e, na época desta publicação (início de 2016), continuava em mudança. Seus princípios essenciais – envolvimento da função de TI em cada fase do design e do desenvolvimento de um sistema, alta dependência de automação em comparação com esforços humanos, aplicação de práticas e ferramentas de engenharia em tarefas de operação – são consistentes com muitas das práticas e dos princípios de SRE. Poderíamos ver o DevOps como uma generalização de vários princípios essenciais de SRE para uma variedade maior de organizações, estruturas gerenciais e recursos humanos. Do mesmo modo, poderíamos ver a SRE como uma implementação específica de DevOps, com algumas extensões idiossincráticas.

Princípios da SRE

Embora as nuances de fluxos de trabalho, prioridades e operações cotidianas variem de equipe de SRE para equipe de SRE, todas compartilham um conjunto básico de responsabilidades para o(s) serviço(s) ao(s) qual(is) dão suporte e respeitam os mesmos princípios essenciais. Em geral, uma equipe de SRE é responsável pela *disponibilidade, latência, desempenho, eficiência, gerenciamento de mudanças, monitoração, respostas a emergências e planejamento da capacidade* de seu(s) serviço(s). Definimos regras de envolvimento e princípios para o modo como as equipes de SRE interagem

com seus ambientes – não só com o ambiente de produção, mas também com as equipes de desenvolvimento de produtos, as equipes de testes, os usuários, e assim por diante. Essas regras e práticas de trabalho nos ajudam a manter o foco em tarefas de engenharia, em oposição a mantê-lo em tarefas de operação.

A próxima seção discute cada um desses princípios essenciais da SRE do Google.

Garantindo um foco durável em engenharia

Como já discutimos, o Google limita as tarefas operacionais dos SREs em 50% de seu tempo. O tempo restante deve ser gasto usando suas habilidades de programação em tarefas de projeto. Na prática, isso é feito monitorando a quantidade de tarefas operacionais feitas pelos SREs e redirecionando o excesso dessas tarefas para as equipes de desenvolvimento de produtos: atribuindo bugs e tickets de volta aos gerentes de desenvolvimento, (re)integrando os desenvolvedores nas escalas de plantão com pagers, e assim por diante. O redirecionamento termina quando a carga operacional reduzir para 50% ou menos novamente. Isso também proporciona um sistema eficiente de feedback, orientando os desenvolvedores a criar sistemas que não precisem de intervenções manuais. Essa abordagem funciona bem quando toda a empresa – as equipes de SRE e de desenvolvimento, igualmente – entende por que o sistema de válvula de segurança existe e dá suporte à meta de não ter eventos de transbordamento (overflow) porque o produto não gera carga operacional suficiente para exigir-la.

Quando se concentram em tarefas de operação, em média, os SREs devem receber um máximo de dois eventos em um turno de 8 a 12 horas de plantão. Esse volume visado dá tempo suficiente a um engenheiro de plantão para tratar o evento de forma rápida e precisa, limpar e restaurar os serviços de volta ao normal e então conduzir um postmortem. Se houver mais de dois eventos regularmente por turno de plantão, os problemas não poderão ser investigados de forma completa e os engenheiros estarão sobrecarregados a ponto de não poderem aprender com esses eventos. Um cenário com muitas solicitações de pager também não melhorará com um aumento da equipe. Por

outro lado, se os SREs de plantão receberem menos de um evento por turno de forma consistente, mantê-los assim será um desperdício de seu tempo.

Os postmortems devem ser escritos para todos os incidentes significativos, não importa se houve um acionamento de pager; os postmortems que não acionaram um pager são mais valiosos ainda, pois é provável que apontem para lacunas claras na monitoração. Essa investigação deve definir o que aconteceu em detalhes, identificar todas as causas-raízes do evento e atribuir ações para corrigir o problema ou melhorar o modo como será abordado da próxima vez. O Google opera com uma *cultura de postmortem sem acusações*, com o objetivo de expor falhas e aplicar engenharia para corrigi-las, em vez de evitá-las ou minimizá-las.

Buscando a máxima rapidez nas mudanças sem violar o SLO de um serviço

As equipes de desenvolvimento de produtos e de SRE podem desfrutar de um relacionamento profissional produtivo se eliminarem os conflitos estruturais em suas respectivas metas. O conflito estrutural se encontra entre o ritmo da inovação e a estabilidade do produto e, conforme descrito antes, muitas vezes é expresso de forma indireta. Em SRE trazemos esse conflito para o primeiro plano e então o resolvemos com a introdução de uma provisão para erros (*error budget*).

A provisão para erros originou-se da observação de que *100% é a meta de confiabilidade incorreta para basicamente tudo* (marca-passos e freios ABS são exceções relevantes). Em geral, para qualquer serviço ou sistema de software, 100% não é a meta correta de confiabilidade porque nenhum usuário poderá dizer qual é a diferença entre um sistema 100% disponível ou 99,999% disponível. Há muitos outros sistemas no caminho entre o usuário e o serviço (seu notebook, o WiFi doméstico, o ISP, a rede de energia elétrica etc.), e esses sistemas em conjunto têm disponibilidade bem menor que 99,999%. Assim, a diferença marginal entre 99,999% e 100% é perdida nos ruídos provocados por outras faltas de disponibilidade, e o usuário não terá nenhuma vantagem do enorme esforço exigido para acrescentar aquele último 0,001% de disponibilidade.

Se 100% é a meta de confiabilidade incorreta para um sistema, então qual é a meta correta? Essa, na verdade, não é uma pergunta técnica – é uma pergunta relacionada ao produto e que deve levar em consideração as seguintes questões:

- Qual é o nível de disponibilidade que deixará os usuários satisfeitos, dado o modo como eles usam o produto?
- Quais alternativas estão disponíveis aos usuários que não estiverem satisfeitos com a disponibilidade do produto?
- O que acontece com a utilização do produto pelos usuários em diferentes níveis de disponibilidade?

O negócio ou o produto devem definir a meta de disponibilidade do sistema. Depois que essa meta for definida, a provisão para erros será um menos a meta de disponibilidade. Um serviço que esteja 99,99% disponível estará 0,01% indisponível. Esse 0,01% permitido de indisponibilidade é a *provisão para erros* (error budget) *do serviço*. Podemos gastar a provisão no que quisermos, desde que não gastemos mais que o seu valor.

Então, como queremos gastar a provisão para erros? A equipe de desenvolvimento quer lançar funcionalidades e atrair novos usuários. O ideal seria gastarmos toda a nossa provisão para erros assumindo riscos com o que lançarmos para fazer o lançamento rapidamente. Essa premissa básica descreve o modelo completo das provisões para erros. Assim que as atividades de SRE são conceituadas nesse modelo, usar a provisão de erros com táticas como rollouts em fases e experimentos de 1% pode otimizar o processo para permitir lançamentos mais rápidos.

O uso de uma provisão para erros resolve o conflito estrutural de pagamento de incentivos entre desenvolvimento e SRE. A meta de SRE deixa de ser “nenhuma interrupção de serviço”; em vez disso, os SREs e os desenvolvedores de produto têm como objetivo gastar a provisão de erros para obter o máximo de rapidez no lançamento de funcionalidades. Essa mudança faz toda a diferença. Uma interrupção de serviço deixa de ser algo “ruim” – é uma parte esperada do processo de inovação, e uma ocorrência que as equipes tanto de desenvolvimento quanto de SRE devem administrar, em vez de temer.

Monitoração

A monitoração é um dos principais meios pelos quais os proprietários de serviços controlam a saúde e a disponibilidade de um sistema. Desse modo, uma estratégia de monitoração deve ser criada de modo bem planejado. Uma abordagem clássica e comum para a monitoração consiste em observar um valor ou uma condição específicos e, então, enviar um email de alerta quando esse valor for excedido ou a condição ocorrer. No entanto, esse tipo de alerta por email não é uma solução eficaz: um sistema que exija uma pessoa para ler um email e decidir se algum tipo de ação deve ser executado em resposta tem uma falha fundamental. A monitoração jamais deve exigir um ser humano para interpretar qualquer parte do domínio do alerta. Em vez disso, um software deve fazer a interpretação e os seres humanos devem ser notificados somente quando houver a necessidade de tomar uma atitude.

Há três tipos de saídas válidas para a monitoração:

Alertas

Significam que um ser humano deve tomar uma atitude imediata em resposta a algo que esteja acontecendo ou que está prestes a ocorrer, para que a situação melhore.

Tickets

Significam que um ser humano deve tomar uma atitude, mas não de imediato. O sistema não é capaz de tratar a situação de modo automático, mas se um ser humano tomar uma atitude em alguns dias, não haverá nenhum dano resultante.

Logging

Ninguém precisa ver essa informação, mas ela será registrada para diagnóstico e investigação forense. A expectativa é que ninguém leia os logs, a menos que outro fato exija que isso seja feito.

Resposta a emergências

A confiabilidade é uma função do MTTF (Mean Time To Failure, ou Tempo Médio para Falhas) e do MTTR (Mean Time To Repair, ou Tempo Médio

para Correção) [Sch15]. A métrica mais relevante na avaliação da eficiência da resposta a emergências é a rapidez com que a equipe de resposta é capaz de deixar o sistema novamente saudável, isto é, o MTTR.

Seres humanos acrescentam latência. Mesmo que um dado sistema tenha mais falhas *propriamente ditas*, um sistema capaz de evitar emergências que exijam intervenção humana terá mais disponibilidade que um sistema que exija uma intervenção manual. Quando há necessidade da intervenção de seres humanos, percebemos que pensar nas melhores práticas e registrá-las com antecedência em um “manual de regras” resulta em uma melhoria de aproximadamente três vezes no MTTR se comparada à estratégia de “improvisar”. O engenheiro herói de plantão, “pau pra toda obra”, faz o serviço, porém o engenheiro de plantão treinado, de posse de um manual de regras, faz um trabalho bem melhor. Embora nenhum manual de regras, independentemente de sua abrangência, seja um substituto para engenheiros inteligentes, capazes de raciocinar diante da situação, passos e dicas claros e completos para resolver problemas são valiosos para responder a uma chamada de importância crítica ou em que o tempo seja crucial. Desse modo, a SRE do Google conta com manuais de regras para os que estão de plantão, além de exercícios como “Wheel of Misfortune” (Roda do Azar)³ para preparar os engenheiros a reagirem a eventos quando estiverem de plantão.

Gerenciamento de mudanças

A SRE descobriu que, em geral, 70% das interrupções de serviço se devem a mudanças em um sistema ativo. As melhores práticas nesse domínio utilizam automação para fazer o seguinte:

- implementar rollouts progressivos;
- detectar problemas de forma rápida e precisa;
- fazer rollback das mudanças de modo seguro quando surgirem problemas.

Esse trio de práticas minimiza de forma eficiente o número total de usuários e operações expostos a mudanças ruins. Ao remover os seres humanos do circuito, essas práticas evitam os problemas comuns de cansaço, familiaridade/menosprezo e falta de atenção a tarefas altamente repetitivas. Como resultado, há um aumento tanto da velocidade na disponibilização de

uma nova versão quanto da segurança.

Previsão de demanda e planejamento de capacidade

Previsão de demanda e planejamento de capacidade podem ser vistos como um meio de garantir que haja capacidade suficiente e redundância para servir a uma demanda futura projetada, com a disponibilidade necessária. Não há nada particularmente especial nesses conceitos, exceto que um número surpreendente de serviços e equipes não executa os passos necessários para garantir que a capacidade exigida esteja implantada na ocasião em que se tornar necessária. O planejamento de capacidade deve levar em consideração tanto o crescimento orgânico (proveniente da adoção natural do produto e do uso pelos clientes) quanto o crescimento inorgânico (que resulta de eventos como lançamentos de funcionalidades, campanhas de marketing ou outras mudanças orientadas ao negócio).

Vários passos são obrigatórios no planejamento de capacidade:

- uma previsão exata da demanda orgânica, que se estenda para além do tempo de antecedência exigido para aquisição de capacidade;
- uma incorporação precisa das fontes de demanda inorgânica na previsão de demanda;
- testes regulares de carga do sistema para correlacionar a capacidade bruta (servidores, discos e assim por diante) com a capacidade do serviço.

Como a capacidade é crítica para a disponibilidade, a implicação natural é que a equipe de SRE deve ser responsável pelo planejamento de capacidade, o que significa que ela também deve ser responsável pelo provisionamento.

Provisionamento

O provisionamento combina tanto o gerenciamento de mudanças quanto o planejamento de capacidade. Em nossa experiência, o provisionamento deve ser conduzido de forma rápida e somente quando necessário, pois a capacidade tem custo alto. Esse exercício também deve ser feito de forma correta; caso contrário, a capacidade não estará operacional quando for necessária. Acrescentar mais capacidade com frequência envolve ativar uma

nova instância ou localização, fazer modificações significativas em sistemas existentes (arquivos de configuração, distribuidores de carga, rede) e validar se a nova capacidade está operacional e fornece os resultados corretos. Desse modo, é uma operação de mais risco que a transferência de carga, que com frequência é feita várias vezes por hora, e deve ser tratada com um grau extra de cuidado correspondente.

Eficiência e desempenho

O uso eficiente de recursos é importante sempre que um serviço se preocupa com dinheiro. Como a SRE, em última instância, controla o provisionamento, ela também deve se envolver em qualquer tarefa relacionada à utilização, pois a utilização é uma função de como um dado serviço opera e como é provisionado. Isso implica que prestar bastante atenção na estratégia de provisionamento de um serviço e, desse modo, em sua utilização, proporciona uma grande vantagem nos custos totais do serviço.

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

Os sistemas de software se tornam mais lentos à medida que sua carga aumenta. Um serviço mais lento é equivalente a uma perda de capacidade. Em algum momento, um sistema que se tornou mais lento para de servir, o que corresponde a uma lentidão infinita. Os SREs fazem um provisionamento para atender a uma meta de capacidade *a uma velocidade específica de resposta* e, desse modo, estão profundamente interessados no desempenho de um serviço. Os SREs e os desenvolvedores de produto vão (e devem) monitorar e modificar um serviço a fim de melhorar o seu desempenho, aumentando assim a capacidade e melhorando a eficiência.⁴

O fim do começo

A Site Reliability Engineering representa uma ruptura significativa das melhores práticas de mercado existentes para administrar serviços grandes e complicados. Motivada originalmente pela familiaridade – “como engenheiro

de software, é assim que eu gostaria de investir o meu tempo para realizar um conjunto de tarefas repetitivas” –, ela se tornou muito mais: um conjunto de princípios, práticas e incentivos, e um campo de empreendimento dentro da disciplina mais ampla de engenharia de software. O restante do livro explora o Modo SRE em detalhes.

1 Vice-presidente, Engenheiro do Google, fundador do Google SRE.

2 N.T.: Levar apenas alguns commits específicos de um branch para outro.

3 Veja a seção “Interpretando papéis em situações de desastre”.

4 Para outras discussões sobre como essa colaboração pode funcionar na prática, consulte a seção “Comunicações: reuniões de produção”.

CAPÍTULO 2

O ambiente de produção do Google do ponto de vista de um SRE

Escrito por JC van Winkel

Editado por Betsy Beyer

Os datacenters do Google são bem diferentes dos datacenters mais convencionais e de server farms (fazendas de servidores) de pequena escala. Essas diferenças representam, ao mesmo tempo, problemas extras e oportunidades. Este capítulo discute os desafios e as oportunidades que caracterizam os datacenters do Google e apresenta a terminologia usada neste livro.

Hardware

A maior parte dos recursos computacionais do Google está em datacenters projetados por essa empresa, com distribuição de energia, refrigeração, rede e hardware de processamento proprietários (veja [Bar13]). De modo diferente de datacenters “padrões” com colocation (compartilhamento de localização), o hardware computacional em um datacenter projetado pelo Google é o mesmo em toda a instalação.¹ Para acabar com a confusão entre hardware e software de servidores, utilizamos a terminologia a seguir no livro:

Máquina

Um hardware (ou talvez uma VM).

Servidor

Um software que implementa um serviço.

As máquinas podem executar qualquer servidor, portanto não dedicamos

máquinas específicas a programas específicos de servidor. Não há nenhuma máquina específica que execute o nosso servidor de emails, por exemplo. Em vez disso, a alocação de recursos é tratada por nosso sistema operacional de cluster, o *Borg*.

Sabemos que esse uso da palavra *servidor* é incomum. O uso comum da palavra combina “binário que aceita uma conexão de rede” com *máquina*, mas diferenciá-los é importante quando falamos de computação no Google. Depois que você se acostumar com o nosso uso da palavra *servidor*, o motivo pelo qual faz sentido usar essa terminologia especializada, não apenas no Google, mas também no restante deste livro, se tornará evidente.

A Figura 2.1 mostra a topologia de um datacenter do Google:

- Dezenas de máquinas são colocadas em um *rack*.
- Os racks são colocados em uma *fila*.
- Uma ou mais filas formam um *cluster*.
- Geralmente um prédio de *datacenter* abriga vários clusters.
- Vários prédios de datacenter próximos formam um *campus*.

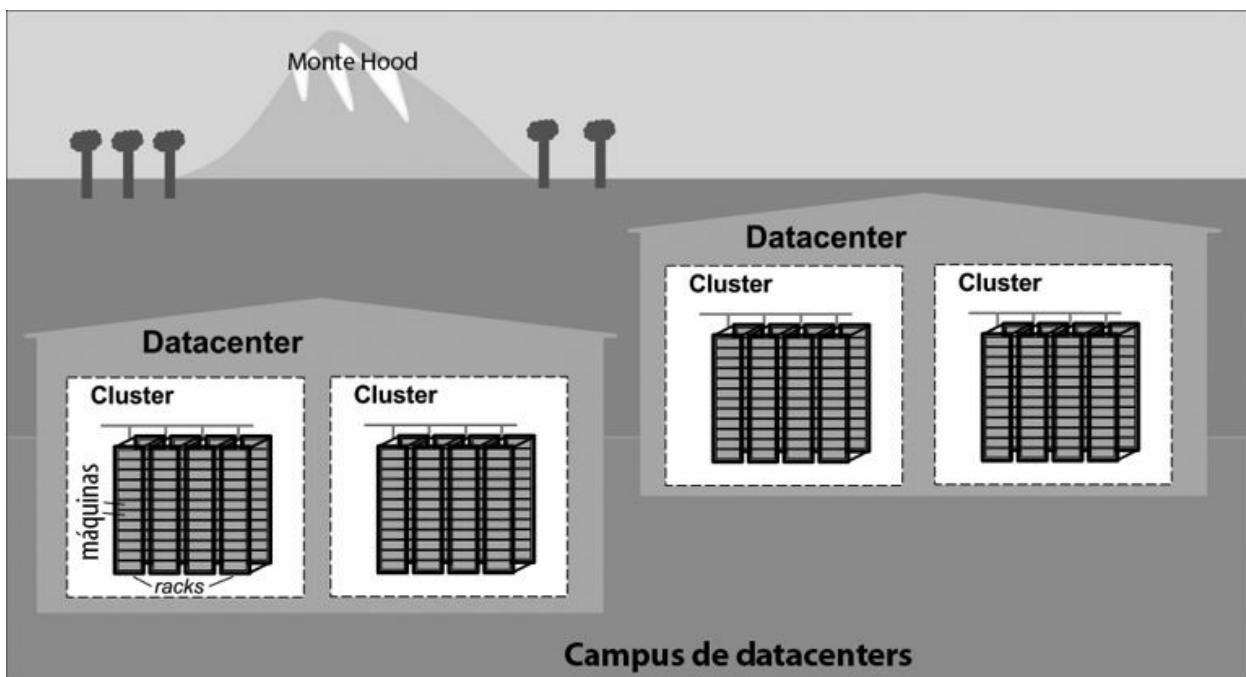


Figura 2.1 – Exemplo da topologia de um campus de datacenters do Google.
As máquinas em um determinado datacenter devem ser capazes de conversar

umas com as outras, portanto criamos um switch virtual bem rápido, com dezenas de milhares de portas. Fizemos isso conectando centenas de switches criados pelo Google em uma estrutura de rede Clos [Clos53] chamada *Jupiter* [Sin15]. Em sua maior configuração, o Jupiter suporta uma largura de banda de duas seções de 1,3 Pbps entre os servidores.

Os datacenters são conectados uns aos outros com nossa rede de backbone *B4* de abrangência mundial [Jai13]. A *B4* é uma arquitetura de rede definida por software (e usa o protocolo de comunicações OpenFlow de padrão aberto). Ela fornece uma largura de banda massiva a um número modesto de sites e utiliza alocação elástica de banda para maximizar a banda média [Kum15].

Sistema de software que “organiza” o hardware

Nosso hardware deve ser controlado e administrado por um software capaz de tratar escalas massivas. Falhas de hardware são um problema notório que administrámos 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 são, de certo modo, de tirar o fôlego. Desse modo, queremos abstrair esses problemas dos usuários, e as equipes que operam nossos serviços, de modo semelhante, não querem ser incomodadas por problemas de hardware. Cada campus de datacenters tem equipes dedicadas à manutenção do hardware e da infraestrutura do datacenter.

Administrando as máquinas

O *Borg*, ilustrado na Figura 2.2, é um sistema operacional distribuído de cluster [Ver15] semelhante ao Apache Mesos.² O *Borg* administra seus jobs no nível de cluster.

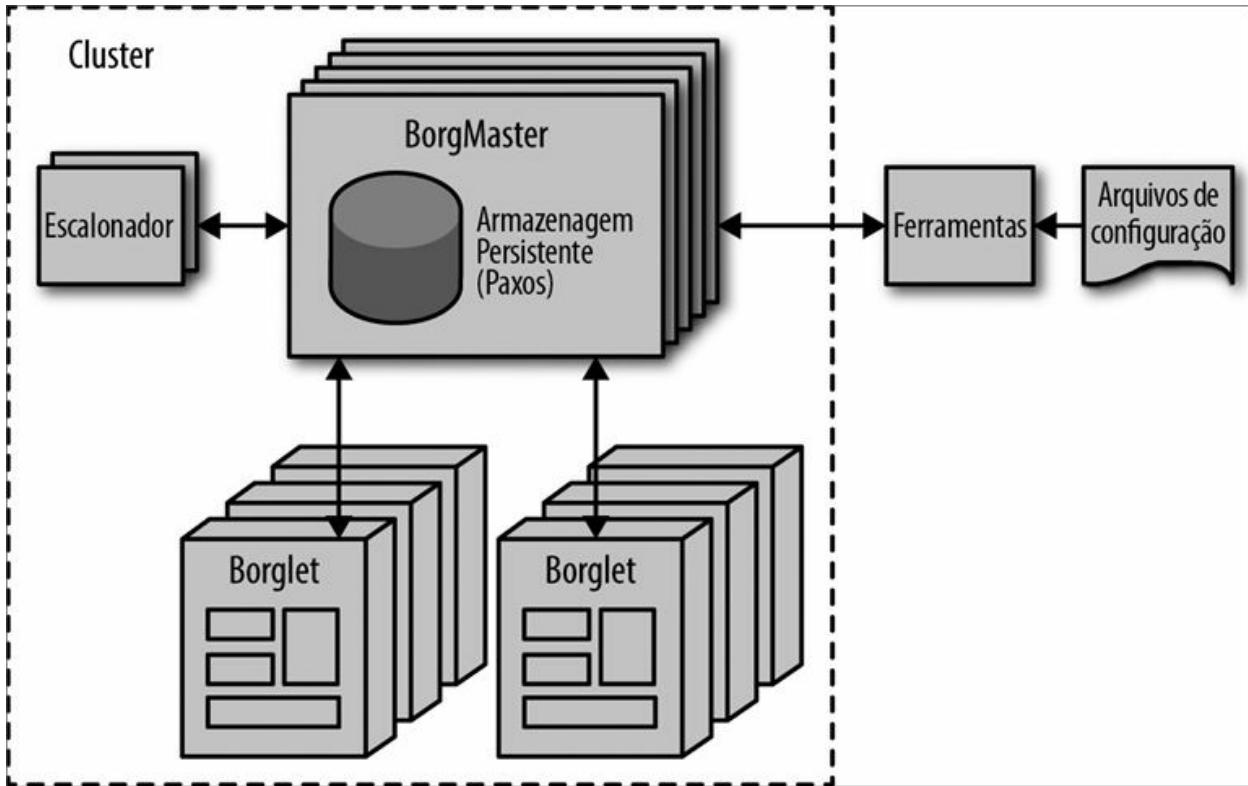


Figura 2.2 – Arquitetura de alto nível de clusters do Borg.

O Borg é responsável por executar os *jobs* dos usuários, que podem ser servidores executando indefinidamente ou processos em batch como um MapReduce [Dea04]. Os jobs podem ser constituídos de mais de uma *tarefa* (task) idêntica (às vezes, milhares), tanto por questões de confiabilidade quanto porque um único processo geralmente não é capaz de tratar todo o tráfego do cluster. Quando o Borg inicia um job, ele encontra as máquinas para executar as tarefas e diz a elas para iniciar o programa servidor. O Borg então monitora continuamente essas tarefas. Se uma tarefa não funcionar bem, ele a mata e a reinicia, possivelmente em uma máquina diferente.

Como as tarefas são alocadas de forma fluida entre as máquinas, não podemos simplesmente contar com os endereços IP e os números das portas para fazer referência às tarefas. Resolvemos esse problema com um nível extra de acesso indireto: quando iniciamos um job, o Borg aloca um nome e um número de índice para cada tarefa usando o *BNS* (Borg Naming Service). 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 um número de porta pelo BNS. Por exemplo, o path do

BNS pode ser uma string como `/bns/<cluster>/<usuário>/<nome do job>/<número da tarefa>`, que será resolvido como `<endereço IP >:<porta>`.

O Borg também é responsável pela alocação de recursos aos jobs. Todo job precisa especificar os recursos de que necessita (por exemplo, 3 núcleos de CPU, 2 GiB de RAM). Ao usar a lista de requisitos de todos os jobs, o Borg é capaz de distribuir as tarefas pelas máquinas de forma otimizada, levando também em consideração os domínios de falhas (por exemplo, o Borg não executará todas as tarefas de um job no mesmo rack, pois fazer isso significa que o switch nesse rack será um ponto único de falha para esse job).

Se uma tarefa tentar usar mais recursos do que solicitou, o Borg matará a tarefa e a reiniciará (pois uma tarefa que falhe lentamente de modo cíclico, em geral, é preferível a uma que nem sequer tenha iniciado).

Armazenagem

As tarefas podem usar o disco local das máquinas como um rascunho, mas temos várias opções de armazenagem no cluster para armazenamento permanente (e até mesmo o espaço de rascunho em algum momento será transferido para o modelo de armazenagem de cluster). São comparáveis ao Lustre e ao HDFS (Hadoop Distributed File System) – ambos são sistemas de arquivos de código aberto para clusters.

A camada de armazenagem é responsável por oferecer aos usuários um acesso fácil e confiável à área de armazenagem disponível a um cluster. Como mostra a Figura 2.3, a armazenagem tem várias camadas:

1. A camada mais baixa se chama D (de *disk* [disco], embora D utilize tanto discos giratórios quanto armazenamento em flash). D é um servidor de arquivos que executa em quase todas as máquinas de um cluster. No entanto, os usuários que quiserem acessar seus dados não vão querer se lembrar em que máquina eles estão armazenados, e é aí que a próxima camada entra em cena.
2. Uma camada sobre a camada D, chamada *Colossus*, cria um sistema de arquivos que se estende por um cluster; ele oferece a semântica usual de um sistema de arquivos, assim como replicação e criptografia. O Colossus é o sucessor do GFS, que é o Google File System [Ghe03].

3. Há vários serviços do tipo banco de dados sobre o Colossus:

- a. O Bigtable [Cha06] é um sistema de banco de dados NoSQL capaz de tratar bancos de dados com petabytes de tamanho. Um Bigtable é um mapa multidimensional ordenado, esparso, distribuído e persistente, indexado por chave de linha, chave de coluna e timestamp; cada valor do mapa é um array de bytes não interpretado. O Bigtable tem suporte para replicação com consistência eventual (eventually consistent) entre datacenters.
- b. O Spanner [Cor12] oferece uma interface do tipo SQL para os usuários que exijam uma verdadeira consistência pelo mundo.
- c. Vários outros sistemas de bancos de dados, por exemplo, o *Blobstore*, estão disponíveis. Cada uma dessas opções tem seu próprio conjunto de pontos positivos e negativos (veja o Capítulo 26).

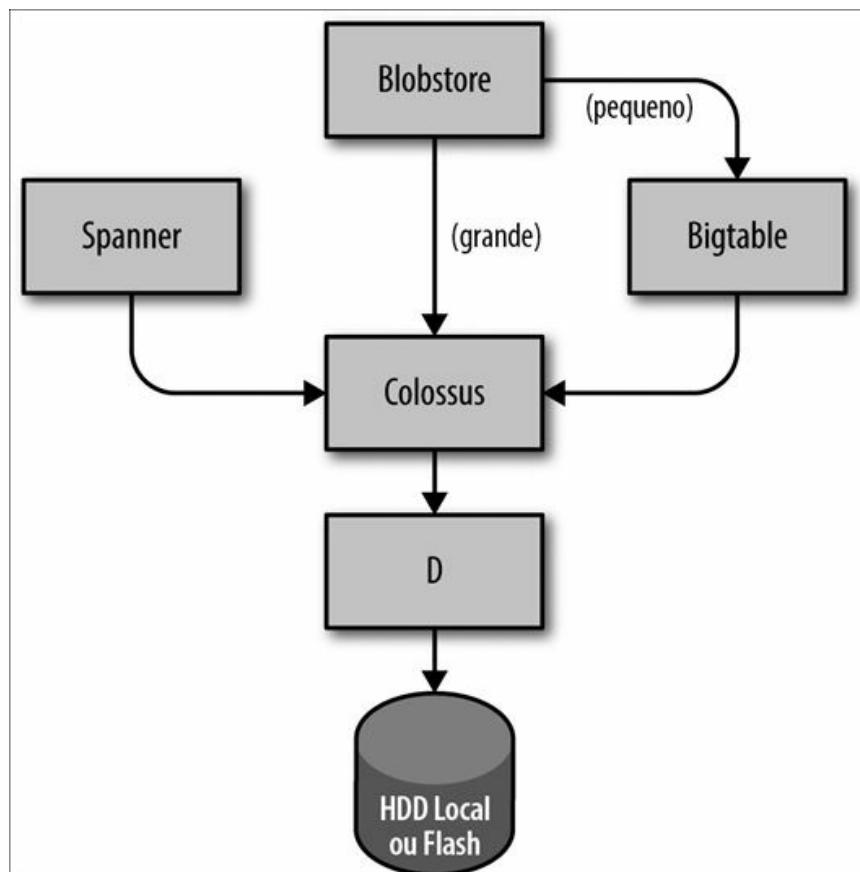


Figura 2.3 – Partes da pilha de armazenagem do Google.

Rede

O hardware de rede do Google é controlado de várias maneiras. Conforme discutimos antes, usamos uma rede baseada em OpenFlow definida por software. Em vez de usar um hardware “inteligente” de roteamento, contamos com componentes de switching “burros”, mais baratos, combinados com um controlador central (duplicado) que faz o pré-processamento dos melhores caminhos pela rede. Desse modo, podemos remover dos roteadores as decisões de roteamento que exigem bastante processamento e usar um hardware de switching mais simples.

A largura de banda da rede deve ser alocada de forma inteligente. Assim como o Borg limita os recursos computacionais que uma tarefa pode utilizar, o BwE (Bandwidth Enforcer) administra a banda a fim de maximizar a banda média disponível. A otimização de banda não diz respeito apenas ao custo: uma engenharia de tráfego centralizada tem mostrado que é capaz de resolver vários problemas que, tradicionalmente, seriam muito difíceis de solucionar, por meio de uma combinação de roteamento distribuído e engenharia de tráfego [Kum15].

Alguns serviços têm jobs que executam em vários clusters distribuídos pelo mundo. Para minimizar a latência dos serviços distribuídos globalmente, queremos direcionar os usuários ao datacenter mais próximo com capacidade disponível. Nossa GSLB (*Global Software Load Balancer*, ou Distribuidor Global de Carga de Software) faz uma distribuição de carga em três níveis:

- Distribuição de carga geográfica para requisições de DNS (por exemplo, para www.google.com), descrita no Capítulo 19.
- Distribuição de carga no nível de um serviço de usuário (por exemplo, YouTube ou Google Maps).
- Distribuição de carga no nível de RPC (Remote Procedure Call, ou Chamada de Procedimento Remoto), descrita no Capítulo 20.

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

Outros softwares de sistemas

Vários outros componentes em um datacenter também são importantes.

Serviço de lock

O serviço de lock *Chubby* [Bur06] oferece uma API semelhante a um sistema de arquivos para manutenção de locks. O Chubby trata esses locks entre as diferentes localidades de datacenters. Ele utiliza o protocolo Paxos para Consensus assíncrono (veja o Capítulo 23).

O Chubby também desempenha um papel importante na eleição do mestre. Quando um serviço tem cinco réplicas de um job executando por questões de confiabilidade, mas apenas uma das réplicas pode realizar o trabalho propriamente dito, o Chubby é usado para selecionar *qual* réplica deve prosseguir.

Dados que devam ser consistentes são bastante adequados ao armazenamento no Chubby. Por esse motivo, o BNS usa o Chubby para armazenar mapeamentos entre pares de paths BNS e Endereço IP:porta.

Monitoração e alertas

Queremos garantir que todos os serviços executem conforme necessário. Desse modo, executamos várias instâncias de nosso programa de monitoração *Borgmon* (veja o Capítulo 10). O Borgmon “extrai” métricas regularmente dos servidores monitorados. Essas métricas podem ser usadas instantaneamente para alertas, além de poderem ser armazenadas para uso em dados gerais históricos (por exemplo, em gráficos). Podemos usar a monitoração de diversas maneiras:

- Definir alertas para problemas sérios.
- Comparar comportamentos: uma atualização de software deixou o servidor mais rápido?
- Analisar como o comportamento do consumo de recursos evolui com o passar 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 possível de nossa infraestrutura de hardware. Nosso código é altamente multithreaded, portanto uma tarefa pode facilmente utilizar vários núcleos de CPU. Para facilitar a apresentação de painéis de controle, a monitoração e a depuração (debugging), todo servidor tem um servidor HTTP que oferece diagnósticos e estatísticas para uma dada tarefa.

Todos os serviços do Google se comunicam usando uma infraestrutura de RPC (Remote Procedure Call, ou Chamada de Procedimento Remoto) chamada *Stubby*; uma versão de código aberto, o gRPC, está disponível.³ Com frequência, uma chamada de RPC é feita mesmo quando uma chamada a uma sub-rotina no programa local precisa ser executada. Isso facilita refatorar a chamada em um servidor diferente caso haja necessidade de mais modularidade, ou quando a base de código de um servidor aumentar. O GSLB é capaz de distribuir a carga das RPCs da mesma maneira que distribui a carga dos serviços visíveis externamente.

Um servidor recebe requisições de RPC de seu *frontend* e envia as RPCs para o seu *backend*. Em termos tradicionais, o frontend é chamado de cliente e o backend, de servidor.

Os dados são transferidos de e para uma RPC usando *buffers de protocolo*⁴, geralmente abreviados como “protobufs”, que são semelhantes ao Thrift do Apache. Os buffers de protocolo apresentam muitas vantagens em relação ao XML para serialização de dados estruturados: são mais simples de usar, de 3 a 10 vezes menores, de 20 a 100 vezes mais rápidos e são menos ambíguos.

Nosso ambiente de desenvolvimento

A velocidade de desenvolvimento é muito importante no Google, portanto construímos um ambiente de desenvolvimento completo para utilizar a nossa infraestrutura [Mor12b].

Com exceção de alguns grupos que têm seus próprios repositórios de código aberto (por exemplo, o Android e o Chrome), os Engenheiros de Software do Google trabalham a partir de um único repositório compartilhado [Pot16].

Isso tem algumas implicações práticas importantes para os nossos fluxos de trabalho:

- Se encontrarem um problema em um componente fora de seu projeto, os engenheiros podem corrigi-lo, enviar as alterações propostas (“changelist” ou *CL*) ao proprietário para revisão e submeter a *CL* na linha principal.
- Alterações no código-fonte no próprio projeto de um engenheiro exigem revisão. Todo software é revisado antes de ser submetido.

Quando um software é desenvolvido, a requisição de build é enviada para os servidores de build em um datacenter. Mesmo builds maiores são executadas rapidamente, pois muitos servidores de build são capazes de fazer a compilação em paralelo. Essa infraestrutura também é usada para testes contínuos. Sempre que uma *CL* é submetida, os testes são executados em todos os softwares que possam depender dessa *CL*, seja direta ou indiretamente. Se o framework determinar que a mudança provavelmente provocou falhas em outras partes do sistema, o dono da alteração submetida será notificado. Alguns projetos utilizam um sistema push-on-green, em que uma nova versão é automaticamente enviada para produção depois de passar pelos testes.

Shakespeare: um exemplo de serviço

Para oferecer um modelo de como um serviço seria hipoteticamente implantado no ambiente de produção do Google, vamos analisar um serviço de exemplo que interage com várias tecnologias dessa empresa. Suponha que queremos oferecer um serviço que permita determinar em que lugar uma dada palavra é usada em todas as obras de Shakespeare.

Podemos dividir esse sistema em duas partes:

- Um componente em lote (batch) que leia todos os textos de Shakespeare, crie um índice e escreva esse índice em um Bigtable. Esse job deve ser executado uma só vez ou, talvez, com uma frequência muito baixa (pois nunca se sabe se um novo texto poderá ser descoberto!).
- Um frontend de aplicação que trate requisições dos usuários finais. Esse job estará sempre ativo, pois os usuários em todos os fusos horários vão

querer fazer pesquisas nos livros de Shakespeare.

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

A fase de mapeamento lê os textos de Shakespeare e os separa em palavras individuais. Isso será mais rápido se for feito em paralelo por várias tarefas de trabalho.

A fase de ordenação organiza as tuplas por palavra.

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

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

Vida de uma requisição

A Figura 2.4 mostra como uma requisição de usuário é tratada: em primeiro lugar, 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 requisição, em última instância, acaba no servidor DNS do Google, que conversa com o GSLB. Como o GSLB controla a carga de tráfego entre os servidores de frontend nas várias regiões, ele escolhe qual endereço IP de servidor deve ser enviado a esse usuário.

O navegador se conecta ao servidor HTTP nesse IP. Esse servidor (chamado de Google Frontend, ou GFE) é um proxy reverso que termina a conexão TCP (2). O GFE consulta qual é o serviço exigido (pesquisa web, mapas ou, neste caso, Shakespeare). Novamente usando o GSLB, o servidor encontra um servidor de frontend do Shakespeare disponível e envia a esse servidor uma RPC contendo a requisição HTTP (3).

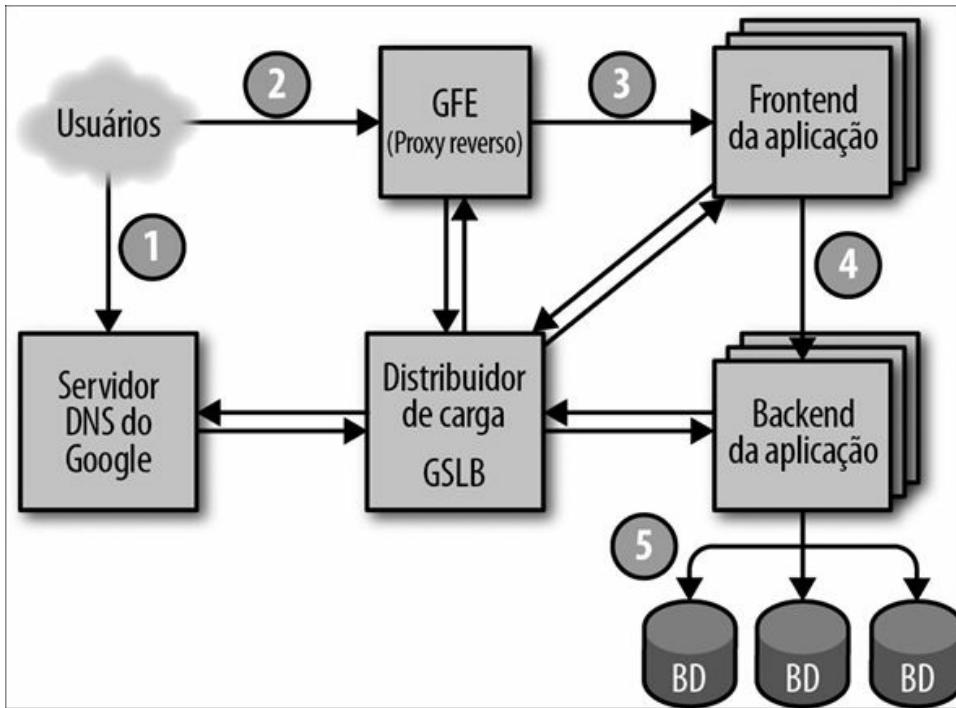


Figura 2.4 – A vida de uma requisição.

O servidor Shakespeare analisa a requisição HTTP e constrói um protobuf contendo a palavra a ser buscada. O servidor de frontend do Shakespeare agora precisa entrar em contato com o servidor de backend do Shakespeare: o servidor de frontend entra em contato com o GSLB a fim de obter o endereço BNS de um servidor de backend adequado e que não esteja sobrecarregado (4). Esse servidor de backend do Shakespeare agora entra em contato com um servidor de Bigtable para obter os dados solicitados (5).

A resposta é escrita no protobuf de resposta e é devolvida ao servidor de backend do Shakespeare. O backend entrega um protobuf contendo o resultado para o servidor de frontend do Shakespeare, que monta o HTML e devolve a resposta ao usuário.

Essa cadeia toda de eventos é executada em um piscar de olhos – demora apenas algumas centenas de milissegundos! Como muitas partes estão envolvidas, há vários pontos de falha em potencial; particularmente, um GSLB com falha causaria muita confusão. No entanto, as políticas do Google de testes rigorosos e rollouts cuidadosos, além dos métodos proativos de recuperação de erros como degradação elegante, nos permitem oferecer o serviço confiável que nossos usuários passaram a esperar. Afinal de contas,

as pessoas normalmente usam www.google.com para verificar se suas conexões de internet estão definidas de forma correta.

Organização de jobs e dados

Os testes de carga determinaram que nosso servidor de backend é capaz de tratar aproximadamente cem consultas por segundo (QPS, ou Queries Per Second). Experimentos realizados com um conjunto limitado de usuários nos levaram a esperar uma carga de pico de aproximadamente 3.470 QPS; portanto, precisamos de, no mínimo, 35 tarefas. No entanto, as considerações a seguir indicam que precisamos de pelo menos 37 tarefas no job, isto é, $N + 2$:

- Durante as atualizações, uma tarefa de cada vez estará indisponível, restando 36 tarefas.
- Uma falha de máquina pode ocorrer durante uma atualização de tarefa, restando apenas 35 tarefas, que são suficientes para atender a uma carga de pico.⁵

Uma análise mais detalhada do tráfego de usuários mostra que nosso uso no pico é distribuído globalmente: 1.430 QPS da América do Norte, 290 da América do Sul, 1.400 da Europa e da África e 350 da Ásia e da Austrália. Em vez de colocar todos os backends em um só local, nós os distribuímos pelos Estados Unidos, América do Sul, Europa e Ásia. Permitir uma redundância de $N + 2$ por região significa que acabamos com 17 tarefas nos Estados Unidos, 16 na Europa e 6 na Ásia. Entretanto decidimos usar 4 tarefas (e não 5) na América do Sul para reduzir o overhead de $N + 2$ para $N + 1$. Nesse caso, estamos dispostos a tolerar um pequeno risco de mais latência em troca de custos mais baixos de hardware: se o GSLB redirecionar o tráfego de um continente para outro quando nosso datacenter sul-americano estiver acima de sua capacidade, poderemos economizar 20% dos recursos que gastaríamos com hardware. Nas regiões maiores, espalharemos as tarefas entre dois ou três clusters para ter mais resiliência.

Como os backends precisam entrar em contato com o Bigtable que armazena os dados, devemos também fazer o projeto desse elemento de armazenagem de forma estratégica. Um backend na Ásia que entre em contato com um

Bigtable nos Estados Unidos acrescenta um volume significativo de latência, portanto replicamos o Bigtable em cada região. A replicação do Bigtable nos ajuda de duas maneiras: oferece resiliência caso um servidor de Bigtable falhe e diminui a latência no acesso de dados. Embora o Bigtable ofereça apenas uma consistência eventual, isso não é um grande problema, pois não precisamos atualizar o conteúdo com frequência.

Introduzimos muitos termos novos neste capítulo; embora você não precise se lembrar de todos, será conveniente fazer isso para compreender muitos dos outros sistemas aos quais vamos nos referenciar mais adiante.

¹ Bem, é *aproximadamente* o mesmo. Na maior parte. Exceto por aquilo que é diferente. Alguns datacenters acabam com várias gerações de hardware de computadores e, às vezes, expandimos os datacenters depois que são construídos. Porém, na maior parte das vezes, o hardware de nosso datacenter é homogêneo.

² Alguns leitores podem ter mais familiaridade com o descendente do Borg, o Kubernetes – um framework de código aberto para coordenação de Container Clusters, iniciado pelo Google em 2014; veja <http://kubernetes.io> e [Bur16]. Para ver mais detalhes sobre as semelhanças entre o Borg e o Apache Mesos, consulte [Ver15].

³ Veja <http://grpc.io>.

⁴ Buffers de protocolo são um mecanismo extensível, independente de linguagem e de plataforma, para serialização de dados estruturados. Para ver mais detalhes, acesse <https://developers.google.com/protocol-buffers/>.

⁵ Estamos supondo que a probabilidade de duas falhas de tarefa simultâneas em nosso ambiente é baixa a ponto de poder ser negligenciável. Pontos únicos de falha, como switches sobre um rack ou distribuição de energia, podem fazer com que essa suposição seja inválida em outros ambientes.

PARTE II

Princípios

Esta seção analisa os *princípios* subjacentes ao modo como as equipes de SRE geralmente trabalham – os padrões, os comportamentos e as áreas de preocupação que influenciam o domínio geral das operações de SRE.

O primeiro capítulo desta seção, e a leitura mais importante se você quiser ver a imagem a partir do maior ângulo possível a fim de saber exatamente o que a SRE faz e como pensar nela, é o Capítulo 3, *Aceitando os riscos*. A SRE é vista através das lentes dos riscos – a avaliação, o gerenciamento e o uso de provisões de erro (error budgets) para oferecer abordagens neutras e úteis ao gerenciamento de serviços.

Os objetivos do nível de serviço são outra unidade conceitual fundamental da SRE. O mercado comumente agrupa conceitos bem diferentes sob o rótulo genérico de acordos de nível de serviço – uma tendência que dificulta pensar nesses conceitos de forma clara. O Capítulo 4, *Objetivos do nível de serviço*, tenta separar os conceitos de indicadores, objetivos e acordos, analisa como a SRE utiliza cada um desses termos e apresenta algumas recomendações sobre como identificar métricas úteis para suas próprias aplicações.

Eliminar tarefas penosas é uma das atividades mais importantes da SRE, e será assunto do Capítulo 5, *Eliminando tarefas penosas*. Definimos uma *tarefa penosa* (toil) como um trabalho operacional mundano e repetitivo, que não resulta em valores duráveis, e que escala linearmente de acordo com o crescimento do serviço.

Seja no Google, seja em outros lugares, a monitoração é um componente absolutamente essencial para fazer o trabalho certo em produção. Se você não puder monitorar um serviço, não saberá o que está acontecendo; se não enxergar o que está acontecendo, não será possível confiar em você. Leia o

Capítulo 6, *Monitorando sistemas distribuídos*, para ver algumas recomendações sobre o que deve ser monitorado e como fazê-lo, além de conhecer algumas das melhores práticas independentes de implementação.

No Capítulo 7, *A evolução da automação no Google*, examinaremos a abordagem da SRE para a automação e analisaremos alguns casos de estudo sobre como a SRE implementou a automação, com e sem sucesso.

A maioria das empresas trata a engenharia de release como um pensamento *a posteriori*. No entanto, como veremos no Capítulo 8, *Engenharia de release*, a engenharia de release não é crítica apenas para a estabilidade do sistema como um todo – pois a maior parte das interrupções de serviço resulta da implantação de alterações de algum tipo. Também é a melhor maneira de garantir que os releases sejam consistentes.

Um princípio fundamental de qualquer engenharia de software eficiente, e não só da engenharia voltada à confiabilidade, é a simplicidade, que é uma qualidade que, uma vez perdida, pode ser extremamente difícil de ser recuperada. Apesar disso, como diz um velho ditado, um sistema complexo que funciona necessariamente evoluiu de um sistema simples que funcionava. O Capítulo 9, *Simplicidade*, discute esse assunto com detalhes.

Outras leituras da SRE do Google

Aumentar a velocidade de disponibilização do produto de forma segura é um princípio fundamental de qualquer empresa. Em “Making Push On Green a Reality” (Fazendo do “push on green” uma realidade) [Kle14], publicado em outubro de 2014, mostramos que tirar os seres humanos do processo de release pode reduzir, de modo paradoxal, as tarefas penosas dos SREs, ao mesmo tempo que *aumenta* a confiabilidade do sistema.

CAPÍTULO 3

Aceitando os riscos

Escrito por Marc Alvidrez

Editado por Kavita Guliani

Talvez você espere que o Google tente criar serviços que sejam 100% confiáveis – serviços que jamais falhem. Contudo, o fato é que, depois de certo ponto, aumentar a confiabilidade é pior para um serviço (e para seus usuários)! Uma confiabilidade extrema tem um custo: maximizar a estabilidade limita a velocidade com que novas funcionalidades podem ser desenvolvidas e a rapidez com que os produtos podem ser entregues aos usuários, além de aumentar de forma drástica os seus custos, o que, por sua vez, reduz o número de funcionalidades que uma equipe é capaz de oferecer. Além do mais, os usuários geralmente não percebem a diferença entre uma confiabilidade alta e uma confiabilidade extrema em um serviço, pois a experiência do usuário é dominada por componentes menos confiáveis, como a rede de celulares ou o dispositivo com o qual o usuário está trabalhando. Falando de modo simples, um usuário em um smartphone com 99% de confiabilidade não é capaz de perceber a diferença entre um serviço com confiabilidade de 99,99% e outro com 99,999%! Com isso em mente, em vez de simplesmente maximizar o uptime, a Site Reliability Engineering procura equilibrar o risco da falta de disponibilidade com as metas de inovação rápida e operações eficientes dos serviços, para que a satisfação dos usuários em geral – com funcionalidades, serviços e desempenho – seja otimizada.

Administrando riscos

Sistemas não confiáveis podem rapidamente minar a confiança dos usuários, portanto queremos reduzir as chances de falhas no sistema. No entanto, a

experiência mostra que, à medida que desenvolvemos sistemas, o custo não aumenta de forma linear com o aumento de confiabilidade – uma melhoria incremental na confiabilidade pode custar cem vezes mais que o incremento anterior. A composição do custo tem duas dimensões:

O custo de máquinas/recursos computacionais redundantes

O custo associado a equipamentos redundantes que, por exemplo, nos permitem deixar os sistemas offline para manutenções de rotina ou manutenções imprevistas, ou que nos oferecem espaço para armazenar blocos com códigos de paridade a fim de oferecer uma garantia mínima de durabilidade dos dados.

O custo de oportunidade

O custo em que uma empresa incorre quando aloca recursos de engenharia para desenvolver sistemas ou funcionalidades que reduzam os riscos em vez de recursos que sejam diretamente visíveis ou utilizáveis pelos usuários finais. Esses engenheiros não trabalham mais com novas funcionalidades e produtos para os usuários finais.

Em SRE administramos a confiabilidade dos serviços principalmente administrando riscos. Conceituamos o risco como um *continuum*. Atribuímos a mesma importância tanto para descobrir como fazer um trabalho de engenharia de modo a aumentar a confiabilidade dos sistemas do Google quanto para identificar o nível apropriado de tolerância para os serviços que operamos. Fazer isso nos permite realizar uma análise de custo/benefício a fim de determinar, por exemplo, em que ponto do *continuum* (não linear) de riscos devemos colocar os serviços Search, Ads, Gmail ou Photos. Nossa objetivo é alinhar explicitamente o risco assumido por um dado serviço com o risco que o negócio está disposto a suportar. Nós nos esforçamos para deixar um serviço confiável o suficiente, mas não *mais* confiável do que seria necessário. Ou seja, quando definimos uma meta de disponibilidade de 99,99%, queremos excedê-la, mas não por muito mais: isso seria perder uma oportunidade para adicionar novas funcionalidades no sistema, acabar com as dívidas técnicas ou reduzir seus custos operacionais. Em certo sentido, vemos a meta de disponibilidade tanto como um mínimo quanto como um máximo.

A principal vantagem desse modelo é que ele permite assumir riscos de forma explícita e bem planejada.

Mensurando os riscos a um serviço

Como prática-padrão no Google, com frequência nos sairemos melhor se identificarmos uma métrica objetiva para representar a propriedade que queremos otimizar em um sistema. Ao definir uma meta, podemos avaliar nosso desempenho atual e monitorar as melhorias ou as degradações ao longo do tempo. Para riscos a serviços, não é claro de imediato de que modo podemos reduzir todos os potenciais fatores a uma única métrica. As falhas de serviço podem ter muitos efeitos em potencial, incluindo falta de satisfação, danos ou perda de confiança pelos usuários, perda direta ou indireta de receita, impacto na marca ou na reputação e uma cobertura indesejável da imprensa. Claramente, alguns desses fatores são difíceis de mensurar. Para que esse problema possa ser tratado e seja consistente entre os vários tipos de sistemas que operamos, o nosso foco está no *downtime não planejado*.

Para a maioria dos serviços, a maneira mais simples de representar a tolerância a riscos é em termos do nível aceitável de downtime não planejado. O downtime não planejado é representado pelo nível desejado de *disponibilidade do serviço*, em geral expresso em termos do número de “noves” que gostaríamos de oferecer: disponibilidade de 99,9%, 99,99% ou 99,999%. Cada nove adicional corresponde a uma melhoria na ordem de magnitude em direção a 100% de disponibilidade. Para sistemas que oferecem serviços, essa métrica é tradicionalmente calculada com base na proporção de uptime do sistema (veja a Equação 3.1).

Equação 3.1 – Disponibilidade baseada em tempo

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

Ao usar essa fórmula pelo período de um ano, podemos calcular o número aceitável de minutos de downtime para alcançar um dado número de noves de disponibilidade. Por exemplo, um sistema com uma meta de

disponibilidade de 99,99% pode estar inativo por até 52,56 minutos em um ano e manter sua meta de disponibilidade; veja o Apêndice A, que apresenta uma tabela.

No Google, porém, uma métrica baseada em tempo para a disponibilidade geralmente não é significativa porque estamos diante de serviços distribuídos globalmente. Nossa abordagem para isolamento de falhas faz com que seja bem provável que estejamos servindo a pelo menos um subconjunto do tráfego para um dado serviço em algum lugar do mundo, em qualquer dado instante (isto é, estamos pelo menos parcialmente “up” o tempo todo). Desse modo, em vez de usar métricas que dependam do uptime, definimos a disponibilidade em termos da *taxa de requisições bem-sucedidas*. A Equação 3.2 mostra como essa métrica baseada em produtividade é calculada em uma janela móvel (isto é, a proporção de requisições bem-sucedidas em uma janela de um dia).

Equação 3.2 – Disponibilidade agregada

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

Por exemplo, um sistema que atenda a 2,5 milhões de requisições em um dia com uma meta de disponibilidade diária de 99,99% pode ter até 250 erros e, ainda assim, atingir sua meta para esse dia.

Em uma aplicação típica, nem todas as requisições são iguais: falhar em uma requisição de cadastramento de usuário é diferente de falhar em uma requisição que faça polling em segundo plano para saber se há um novo email. Em muitos casos, porém, a disponibilidade calculada como a taxa de requisições bem-sucedidas em relação a todas as requisições é uma aproximação razoável para o downtime não planejado, do ponto de vista do usuário final.

Quantificar o downtime não planejado como uma taxa de requisições bem-sucedidas também deixa essa métrica de disponibilidade mais propícia a ser usada em sistemas que não sirvam tipicamente aos usuários finais de forma direta. A maioria dos sistemas que não oferecem serviços aos usuários (por exemplo, sistemas batch, pipeline, sistemas de armazenagem e transacionais) tem um conceito bem definido para unidades de trabalho bem-sucedidas ou

não. Na verdade, embora os sistemas discutidos neste capítulo sejam primordialmente sistemas que ofereçam serviços aos consumidores e à infraestrutura, muitos dos mesmos princípios também se aplicam a sistemas que não são desse tipo, com um mínimo de modificação.

Por exemplo, um processo batch que extraia, transforme e insira o conteúdo de um de nossos bancos de dados de clientes em um data warehouse (armazém ou depósito de dados) para permitir uma análise futura pode ser configurado para executar de forma periódica. Se usarmos uma taxa de requisições bem-sucedidas definida em termos de registros processados com e sem sucesso, podemos calcular uma métrica de disponibilidade útil, apesar do fato de o sistema batch não executar de forma constante.

Com mais frequência, definimos metas de disponibilidade trimestrais para um serviço e monitoramos nosso desempenho em relação a essas metas com periodicidade semanal ou até mesmo diária. Essa estratégia nos permite administrar o serviço com um alto nível de disponibilidade como meta; para isso, procuramos, rastreamos e corrigimos os desvios significativos à medida que eles inevitavelmente surgirem. Veja o Capítulo 4, que tem mais detalhes.

Tolerância dos serviços aos riscos

O que significa identificar a tolerância de um serviço aos riscos? Em um ambiente formal, ou no caso de sistemas críticos quanto à segurança, a tolerância do serviço aos riscos geralmente está incluída diretamente na definição básica do produto ou do serviço. No Google, a tolerância dos serviços aos riscos tende a ser menos claramente definida.

Para identificar a tolerância de um serviço aos riscos, os SREs devem trabalhar com os donos dos produtos para transformar um conjunto de objetivos de negócio em objetivos explícitos nos quais possamos aplicar a engenharia. Nesse caso, os objetivos de negócio em que estamos interessados têm um impacto direto no desempenho e na confiabilidade do serviço oferecido. Na prática, é mais fácil falar dessa tradução do que fazê-la. Enquanto os serviços aos consumidores geralmente têm proprietários explícitos para o produto, é incomum que serviços de infraestrutura (por exemplo, sistemas de armazenagem ou uma camada de caching HTTP de

propósito geral) tenham uma estrutura semelhante de identificação do proprietário do produto. Discutiremos os casos de consumidores e de infraestrutura, um de cada vez.

Identificando a tolerância a riscos dos serviços para consumidores

Nossos serviços para consumidores com frequência têm uma equipe de produto que atua como o dono do negócio para uma aplicação. Por exemplo, Search, Google Maps e Google Docs têm seus próprios gerentes de produto. Esses gerentes de produto têm a responsabilidade de entender os usuários e o negócio, além de moldar o produto para que ele tenha sucesso no mercado. Quando há 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 desenvolvem o sistema muitas vezes exercem esse papel, de forma consciente ou não.

Existem muitos fatores que devem ser considerados ao avaliar a tolerância dos serviços aos riscos, como:

- Qual é o nível de disponibilidade exigido?
- Tipos diferentes de falhas têm efeitos diferentes no serviço?
- Como podemos usar o custo do serviço para ajudar a localizar o serviço no *continuum* de risco?
- Que outras métricas do serviço são importantes para serem levadas em consideração?

Meta para o nível de disponibilidade

A meta para o nível de disponibilidade para um dado serviço do Google geralmente depende da funcionalidade que ele oferece e de como o serviço está posicionado no mercado. A lista a seguir inclui algumas questões a serem consideradas:

- Qual é o nível de serviço que os usuários esperarão?
- Esse serviço está diretamente ligado à receita (seja à nossa receita, seja à

receita de nossos clientes)?

- É um serviço pago ou é gratuito?
- Se houver concorrentes no mercado, qual é o nível de serviço que esses concorrentes oferecem?
- Esse serviço está voltado aos consumidores ou às empresas?

Considere os requisitos para o Google Apps for Work. A maior parte de seus usuários são empresas, algumas grandes e outras pequenas. Essas empresas dependem dos serviços do Google Apps for Work (por exemplo, Gmail, Calendar, Drive, Docs) para oferecer ferramentas que permitam aos seus funcionários realizar suas tarefas diárias. Falando de outra maneira, uma interrupção em um serviço do Google Apps for Work será uma interrupção não só para o Google, mas também para todas as empresas que essencialmente dependem de nós. Para um serviço típico do Google Apps for Work, podemos definir uma meta de disponibilidade trimestral externa de 99,9% e dar suporte a essa meta com uma meta de disponibilidade interna mais alta e um contrato que estipule penalidades caso falhemos em proporcionar a meta externa.

O YouTube apresenta um conjunto contrastante de considerações. Quando o Google comprou o YouTube, tivemos que tomar uma decisão sobre a meta de disponibilidade apropriada para o site. Em 2006, o YouTube tinha os consumidores como foco e estava em uma fase bem diferente de seu ciclo de vida de negócios em relação ao Google naquela época. Apesar de o YouTube já ter um ótimo produto, ele ainda estava em um processo de mudanças e crescimento rápidos. Definimos uma meta de disponibilidade menor para o YouTube em comparação com nossos produtos corporativos porque um desenvolvimento rápido de funcionalidades era relativamente mais importante.

Tipos de falhas

O aspecto esperado das falhas para um dado serviço é outra consideração importante. Quão resiliente é o nosso negócio em relação ao downtime do serviço? O que é pior para o serviço: uma taxa baixa constante de falhas ou uma interrupção ocasional do site todo? Os dois tipos de falhas podem

resultar no mesmo número absoluto de erros, mas podem ter impactos extremamente diferentes no negócio.

Um exemplo ilustrativo da diferença entre interrupções completas e parciais surge naturalmente em sistemas que oferecem informações privadas. Considere uma aplicação de gerenciamento de contatos, e a diferença entre falhas intermitentes que façam com que as fotos dos perfis deixem de ser apresentadas *versus* um caso de falha que resulte nos contatos privados de um usuário serem mostrados para outro usuário. O primeiro caso evidentemente constitui uma experiência ruim para o usuário, e os SREs trabalharão para remediar o problema de forma rápida. No segundo caso, porém, o risco de expor dados privados poderia facilmente minar a confiança básica do usuário de maneira significativa. Como resultado, tirar o serviço do ar totalmente seria apropriado durante a depuração e a potencial fase de limpeza no segundo caso.

Na outra extremidade dos serviços oferecidos pelo Google, às vezes é aceitável ter interrupções regulares durante algumas janelas de manutenção. Alguns anos atrás, o Ads Frontend costumava ser um desses serviços. Ele é usado por anunciantes e criadores de sites para definir, configurar, executar e monitorar suas campanhas publicitárias. Como a maior parte desse trabalho ocorre durante o horário comercial normal, determinamos que interrupções de serviço ocasionais, regulares e agendadas na forma de janelas de manutenção seriam aceitáveis, e contabilizamos essas interrupções agendadas como downtime planejado, e não como downtime não planejado.

Custo

O custo geralmente é o fator essencial para determinar a meta de disponibilidade apropriada a um serviço. O Ads está particularmente em uma boa posição para fazer essa negociação, pois sucessos e falhas em requisições podem ser diretamente traduzidos em ganhos ou perdas de receita. Para determinar a meta de disponibilidade para cada serviço, fazemos perguntas como:

- Se fôssemos desenvolver e operar esses sistemas com um nove a mais de disponibilidade, qual seria o nosso aumento na receita?

- Essa receita adicional supera o custo para alcançar esse nível de confiabilidade?

Para deixar essa equação de negociação mais concreta, considere a análise de custo/benefício a seguir para um serviço de exemplo em que toda requisição tem o mesmo valor:

Melhoria proposta na meta de disponibilidade: 99,9% → 99,99%

Aumento de confiabilidade proposto: 0,09%

Receita do serviço: 1 milhão de dólares

Valor da confiabilidade melhorada: 1 milhão de dólares * 0,0009 = 900 dólares

Nesse caso, se o custo para melhorar a disponibilidade em um nove for menor que 900 dólares, o investimento valerá a pena. Se o custo for maior que 900 dólares, ele excederá o aumento de receita projetado.

Pode ser mais difícil definir essas metas quando não temos uma função simples de tradução entre confiabilidade e receita. Uma estratégia útil pode ser considerar a taxa de erros em background dos ISPs na internet. Se as falhas forem medidas do ponto de vista do usuário final e se for possível conduzir a taxa de erros do serviço a um valor inferior à taxa de erros em background, esses erros se acomodarão no ruído da conexão de internet de um dado usuário. Embora haja diferenças significativas entre ISPs e protocolos (por exemplo, TCP *versus* UDP, IPv4 *versus* IPv6), medimos a taxa típica de erros em background dos ISPs e observamos que ela está entre 0,01% e 1%.

Outras métricas de serviços

Analisar a tolerância dos serviços aos riscos em relação a métricas além da disponibilidade muitas vezes é produtivo. Entender quais métricas são importantes e quais não são nos proporciona alguns graus de liberdade quando tentarmos assumir riscos planejados.

A latência do serviço para nossos sistemas Ads oferece um exemplo ilustrativo. Quando o Google lançou inicialmente o Web Search, uma das características distintivas essenciais do serviço era a velocidade. Quando

introduzimos o AdWords, que exibe anúncios ao lado dos resultados de pesquisa, um requisito essencial do sistema era que os anúncios não deveriam deixar a experiência de pesquisa mais lenta. Esse requisito direcionou as metas de engenharia em todas as gerações dos sistemas AdWords e é tratado como uma invariante.

O AdSense – o sistema de anúncios do Google que oferece anúncios contextuais em resposta a requisições de códigos JavaScript que os anunciantes inserem em seus sites – tem uma meta de latência bem diferente. A meta de latência para o AdSense consiste em evitar que a apresentação das páginas de terceiros se torne lenta quando anúncios contextuais são inseridos. A meta de latência específica então depende da velocidade com que a página de determinado anunciante é apresentada. Isso significa que os anúncios do AdSense, em geral, podem ser servidos centenas de milissegundos mais lentamente que os anúncios do AdWords.

Esse requisito menos rígido de latência para o serviço nos permitiu fazer muitas negociações inteligentes no provisionamento (isto é, para determinar a quantidade e as localizações dos recursos que usamos para oferecer o serviço), que nos fazem economizar custos de forma substancial em comparação com um provisionamento mais ingênuo. Em outras palavras, dada a relativa falta de sensibilidade do serviço AdSense a mudanças moderadas no desempenho da latência, podemos consolidar o oferecimento do serviço em menos localizações geográficas, reduzindo nosso overhead operacional.

Identificando a tolerância a riscos dos serviços de infraestrutura

Os requisitos para construir e operar componentes de infraestrutura diferem dos requisitos de produtos voltados a consumidores de várias maneiras. Uma diferença fundamental é que, por definição, os componentes de infraestrutura têm vários clientes, geralmente com necessidades variadas.

Meta para o nível de disponibilidade

Considere o Bigtable [Cha06], que é um sistema de armazenagem distribuído de escala massiva para dados estruturados. Alguns serviços voltados aos

consumidores servem dados diretamente do Bigtable no tratamento de uma requisição de usuário. Esses serviços precisam de baixa latência e alta confiabilidade. Outras equipes usam o Bigtable como um repositório de dados utilizados para análises offline (por exemplo, MapReduce) feitas regularmente. Essas equipes tendem a estar mais preocupadas com throughput do que com confiabilidade. A tolerância a riscos nesses dois casos de uso é bem diferente.

Uma abordagem para atender às necessidades dos dois casos de uso é conceber todos os serviços de infraestrutura de modo que sejam superconfiáveis. Dado o fato de que esses serviços de infraestrutura também tendem a agregar quantidades enormes de recursos, uma abordagem desse tipo geralmente é muito cara na prática. Para entender as diferentes necessidades dos diferentes tipos de usuários, podemos observar o estado desejado da fila de requisições para cada tipo de usuário do Bigtable.

Tipos de falhas

O usuário que espera baixa latência quer que as filas de requisições do Bigtable estejam (quase sempre) vazias para que o sistema possa processar cada requisição pendente em sua chegada, de modo imediato. (Com efeito, um enfileiramento ineficiente com frequência é a causa de latências altas em casos extremos.) O usuário preocupado com análises offline está mais interessado no throughput do sistema, de modo que os usuários vão querer que as filas de requisições jamais estejam vazias. Para otimizar em relação ao throughput, o sistema Bigtable jamais deve ficar ocioso enquanto espera pela próxima requisição.

Como podemos ver, sucesso e falha são opostos para esses conjuntos de usuários. O sucesso para o usuário que espera baixa latência é uma falha para o usuário interessado em análises offline.

Custo

Uma maneira de satisfazer a essas restrições concorrentes de maneira eficaz quanto ao custo é particionar a infraestrutura e oferecê-la em vários níveis independentes de serviço. No exemplo do Bigtable, podemos construir dois tipos de clusters: clusters para baixa latência e clusters para throughput. Os

clusters para baixa latência são projetados para serem operados e utilizados por serviços que precisem de baixa latência e alta confiabilidade. Para garantir tamanhos pequenos de fila e satisfazer aos requisitos mais rigorosos de isolamento de clientes, o sistema Bigtable pode ser provisionado com um volume substancial de capacidade adicional para reduzir a contenção e aumentar a redundância. Os clusters para throughput, por outro lado, podem ser provisionados para executar a todo vapor, com menos redundância, otimizando o throughput em detrimento da latência. Na prática, somos capazes de satisfazer a essas necessidades menos rigorosas com um custo bem menor, talvez com 10 a 50% do custo de um cluster para baixa latência. Dada a escala massiva do Bigtable, essas economias de custo se tornam rapidamente significativas.

A estratégia principal, no que diz respeito à infraestrutura, é proporcionar serviços com níveis explicitamente definidos de serviço, permitindo, assim, que os clientes façam as negociações corretas quanto aos riscos e custos quando construírem seus sistemas. Com níveis de serviço definidos de forma explícita, os provedores de infraestrutura podem externalizar de modo eficiente a diferença de custo exigida para oferecer serviços em um dado nível aos clientes. Expor o custo dessa maneira motiva os clientes a escolher o nível de serviço com o menor custo que atenda às suas necessidades. Por exemplo, o Google+ pode decidir colocar os dados que sejam importantes para garantir a privacidade dos usuários em um sistema de armazenagem de dados consistente, global e de alta disponibilidade (por exemplo, um sistema do tipo SQL replicado globalmente, como o Spanner [Cor12]), enquanto coloca dados opcionais (dados que não são críticos, mas melhoram a experiência do usuário) em um banco de dados mais barato, menos confiável, menos atualizado e com consistência eventual (por exemplo, um armazenamento NoSQL com replicação baseada em best-effort [melhor esforço], como o Bigtable).

Observe que podemos operar várias classes de serviços usando hardware e software idênticos. Podemos oferecer garantias de serviço extremamente diferentes ajustando diversas características do serviço, como a quantidade de recursos, o grau de redundância, as restrições de provisionamento geográfico e, de modo essencial, a configuração do software da infraestrutura.

Exemplo: infraestrutura de frontend

Para demonstrar que esses princípios de avaliação de tolerância a riscos não se aplicam apenas à infraestrutura de armazenagem, vamos dar uma olhada em outra classe grande de serviços: a infraestrutura de frontends do Google. A infraestrutura de frontends é constituída de sistemas de proxy reverso e de distribuição de carga executados nas proximidades da fronteira de nossa rede. São os sistemas que, entre outras coisas, servem como endpoint para as conexões com os usuários finais (por exemplo, terminam o TCP do navegador do usuário). Dado o seu papel crítico, esses sistemas são planejados de modo a fornecer um nível extremamente alto de confiabilidade. Enquanto os serviços aos consumidores com frequência podem limitar a visibilidade da falta de confiabilidade nos backends, esses sistemas de infraestrutura não têm tanta sorte assim. Se uma requisição jamais chegar ao servidor de frontend do serviço da aplicação, ela será perdida.

Exploramos maneiras de identificar a tolerância a riscos tanto dos serviços aos consumidores quanto de infraestrutura. Agora discutiremos o uso desse nível de tolerância para administrar a falta de confiabilidade por meio de provisões de erro (error budgets).

Motivação para provisão de erros¹

Escrito por Mark Roth

Editado por Carmela Quinito

Outros capítulos deste livro discutem de que modo podem surgir tensões entre as equipes de desenvolvimento de produtos e as equipes de SRE, considerando que elas são, em geral, avaliadas de acordo com métricas diferentes. O desempenho da equipe de desenvolvimento de produtos, em sua maior parte, é avaliado de acordo com a velocidade do produto, o que gera incentivos para que novos códigos sejam implantados o mais rápido possível. Enquanto isso, o desempenho de SRE é avaliado (o que não é de se surpreender) de acordo com a confiabilidade de um serviço, o que implica um incentivo para se contrapor a uma taxa alta de mudanças. A assimetria de informações entre as duas equipes amplifica mais ainda essa tensão inerente.

Os desenvolvedores de produto têm mais visibilidade quanto ao tempo e aos esforços envolvidos para escrever e lançar seu código, enquanto os SREs têm mais visibilidade quanto à confiabilidade do serviço (e ao estado do ambiente de produção em geral).

Com frequência, essas tensões se refletem em opiniões diferentes sobre o nível de esforço que deve ser aplicado nas práticas de engenharia. A lista a seguir apresenta algumas tensões típicas:

Tolerância a falhas do software

O quanto inflexível devemos deixar o software a eventos inesperados? Se for flexível demais, teremos um produto frágil e não utilizável. Se for pouco flexível, teremos um produto que ninguém vai querer usar (mas que executa de modo muito estável).

Testes

Novamente, não faça testes o suficiente e você terá interrupções de serviço constrangedoras, vazamentos de dados privados ou vários outros eventos dignos de serem publicados na mídia. Teste demais, e você poderá perder o seu mercado.

Frequência de implantação de versão

Toda implantação de uma nova versão é arriscada. Quanto devemos trabalhar para reduzir esse risco em detrimento de realizar outras tarefas?

Duração e tamanho do “canary”²

Testar uma nova versão em um pequeno subconjunto de uma carga de trabalho típica é uma boa prática, muitas vezes chamada de *canarying*. Quanto tempo devemos esperar e qual é o tamanho do canary?

Normalmente, equipes existentes já identificaram algum tipo de equilíbrio informal entre esses itens a fim de determinar onde estão as fronteiras entre risco/esforço. Infelizmente, é muito raro poder provar que esse equilíbrio seja ideal, e não apenas uma função das habilidades de negociação dos engenheiros envolvidos. Essas decisões também não devem ser determinadas por política, medo ou esperança. (Com efeito, o ditado não oficial da SRE do Google é “Esperança não é uma estratégia”.) Em vez disso, nossa meta é

definir uma métrica objetiva, com a qual os dois lados tenham concordado, possível de ser usada para orientar as negociações de forma reproduzível. Quanto mais baseada em dados puder ser uma decisão, melhor será.

Calculando sua provisão para erros

Para que essas decisões sejam calcadas em dados objetivos, as duas equipes em conjunto definem uma provisão de erros trimestral baseada no SLO (Service Level Objective, ou Objetivo de Nível de Serviço) do serviço (veja o Capítulo 4). A provisão de erros oferece uma métrica clara e objetiva, que determina até que ponto um serviço pode deixar de ser confiável em um único trimestre. Essa métrica retira a política das negociações entre os SREs e os desenvolvedores de produto quando decidirem qual será o nível de risco permitido.

Nossa prática, então, pode ser descrita assim:

- A Gerência de Produto define um SLO, que estabelece uma expectativa quanto ao uptime que o serviço deve ter por trimestre.
- O uptime real é medido por um terceiro neutro: o nosso sistema de monitoração.
- A diferença entre esses dois números é a “provisão” (budget) para a “falta de confiabilidade” restante no trimestre.
- Desde que o uptime medido esteja acima do SLO – em outras palavras, desde que haja uma provisão de erros restante –, novas versões poderão ser implantadas.

Por exemplo, suponha que o SLO de um serviço seja atender a 99,999% de todas as consultas com sucesso a cada trimestre. Isso significa que a provisão de erros para o serviço corresponde a uma taxa de falhas de 0,001% para um dado trimestre. Se um problema nos fizer falhar em 0,0002% das consultas esperadas para o trimestre, o problema terá gasto 20% da provisão de erros trimestral do serviço.

Vantagens

A principal vantagem de uma provisão para erros é que ela oferece um

incentivo comum que permite tanto ao desenvolvimento de produtos quanto à SRE se concentrar em encontrar o equilíbrio correto entre inovação e confiabilidade.

Muitos produtos usam esse circuito de controle para administrar a velocidade da disponibilização de novas versões: desde que os SLOs do sistema sejam atendidos, o lançamento de novas versões pode continuar. Se ocorrerem violações frequentes de SLO a ponto de gastar a provisão para erros, o lançamento de novas versões será temporariamente interrompido, ao mesmo tempo que recursos adicionais serão investidos em testes e desenvolvimento do sistema para deixá-lo mais resiliente, melhorar o seu desempenho e assim por diante. Abordagens mais sutis e eficazes do que essa técnica simples de ligar/desligar estão disponíveis:³ por exemplo, reduzir a velocidade dos lançamentos ou fazer rollback quando a provisão para erros estiver próxima de ser consumida por causa da violação de SLO.

Por exemplo, se o desenvolvimento de produto quiser fazer um teste reduzido ou aumentar a velocidade de implantação de versões e a SRE resistir a isso, a provisão de erros orientará a decisão. Quando a provisão for grande, os desenvolvedores de produto poderão assumir mais riscos. Quando a provisão estiver quase esgotada, os próprios desenvolvedores de produto farão pressão para que mais testes sejam feitos ou reduzirão a velocidade de implantação de novas versões, pois não vão querer correr o risco de acabar com a provisão e impedir lançamentos. Com efeito, a equipe de desenvolvimento de produtos passa a se autopoliciar. Ela conhece a provisão e é capaz de administrar seus próprios riscos. (É claro que esse resultado depende de uma equipe de SRE ter a autoridade para realmente impedir lançamentos se o SLO for violado.)

O que acontecerá se uma interrupção de serviço de rede ou uma falha de datacenter reduzir o SLO medido? Eventos como esses também consomem a provisão para erros. Como resultado, o número de novas atualizações de versão poderá ser reduzido pelo restante do trimestre. A equipe toda apoia essa redução, pois todos compartilham a responsabilidade pelo uptime.

A provisão também ajuda a enfatizar alguns dos custos de metas de confiabilidade demasiadamente altas, em termos tanto de falta de flexibilidade quanto de inovação lenta. Se a equipe está tendo problemas para

lançar novas funcionalidades, poderá optar por deixar o SLO menos rigoroso (aumentando assim a provisão para erros) a fim de ter mais inovação.

Principais insights

- Administrar a confiabilidade dos serviços, em sua maior parte, tem a ver com o gerenciamento de riscos, e esse gerenciamento pode ter um custo alto.
- 100% provavelmente jamais será uma meta correta de confiabilidade: não só é um valor impossível de atingir, como também é uma confiabilidade acima daquela que os usuários de um serviço querem ou percebem. Combine o perfil do serviço com o risco que o negócio está disposto a assumir.
- Uma provisão para erros (error budget) alinha incentivos e enfatiza uma responsabilidade conjunta entre SRE e desenvolvimento de produtos. Provisões para erros facilitam decidir a taxa de lançamento de versões e amenizam de modo eficiente as discussões sobre interrupções de serviço com os stakeholders, além de permitir que várias equipes cheguem à mesma conclusão sobre riscos à produção sem que haja rancores.

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

² N.T.: A *canary release* (implantação canário) consiste na implantação de uma nova versão de software para apenas um subconjunto de usuários a fim de reduzir o risco da implantação dessa versão em todo o ambiente de produção. O nome faz alusão à técnica de levar canários a uma mina de carvão; em caso de haver gases tóxicos, o canário morreria antes, alertando os mineradores do risco.

³ Conhecido como controle “bang/bang” – veja https://en.wikipedia.org/wiki/Bang–bang_control.

CAPÍTULO 4

Objetivos do nível de serviço

Escrito por Chris Jones, John Wilkes e Niall Murphy com Cody Smith

Editado por Betsy Beyer

É impossível administrar um serviço corretamente, muito menos administrá-lo bem, sem entender quais comportamentos realmente são importantes para esse serviço e como podemos mensurar e avaliar esses comportamentos. Para isso, gostaríamos de definir e proporcionar um dado *nível de serviço* aos nossos usuários, independentemente de eles usarem uma API interna ou um produto público.

Usamos intuição, experiência e uma compreensão do que os usuários querem para definir *SLIs* (Service Level Indicators, ou Indicadores de Nível de Serviço), *SLOs* (Service Level Objectives, ou Objetivos de Nível de Serviço) e *SLAs* (Service Level Agreements, ou Acordos de Nível de Serviço). Essas medidas descrevem propriedades básicas de métricas importantes, quais valores queremos que essas métricas tenham e como reagiremos se não pudermos oferecer o serviço esperado. Em última instância, escolher métricas apropriadas ajuda a tomar a atitude correta caso algo dê errado, além de deixar uma equipe de SRE confiante de que um serviço está saudável.

Este capítulo descreve o modelo que usamos para enfrentar problemas de modelagem, seleção e análise de métricas. Boa parte dessas explicações seriam muito abstratas sem um exemplo, portanto utilizaremos o serviço Shakespeare descrito na seção “Shakespeare: um exemplo de serviço”, para ilustrar nossos pontos principais.

Terminologia do nível de serviço

Muitos leitores provavelmente terão familiaridade com o conceito de SLA,

mas os termos *SLI* e *SLO* também merecem uma definição cuidadosa, pois, no uso comum, o termo *SLA* está sobrecarregado e assumiu vários significados que dependem do contexto. Preferimos separar esses significados por questões de 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 requisição* – quanto tempo demora para que uma resposta seja devolvida a uma requisição – como um *SLI* essencial. Outros *SLIs* comuns incluem a taxa de erro, com frequência expressa na forma de uma fração de todas as requisições recebidas, e o *throughput do sistema*, em geral medido em requisições por segundo. As medidas muitas vezes são agregadas, isto é, dados brutos são coletados em uma janela de medição e então são transformados em uma taxa, uma média ou um percentil.

O ideal é que o *SLI* mensure diretamente um nível de serviço de interesse, mas, às vezes, somente um valor aproximado estará disponível, pois a medida desejada pode ser difícil de ser obtida ou interpretada. Por exemplo, a latência do lado do cliente com frequência é a métrica mais relevante ao usuário, mas talvez seja possível medir a latência somente no servidor.

Outro tipo de *SLI* importante aos SREs é a *disponibilidade*, isto é, a fração do tempo em que um serviço é utilizável. Com frequência, ela é definida em termos da fração de requisições bem formadas que tiveram sucesso, às vezes chamada de *yield* (produção). (A *durabilidade* – que é a probabilidade de os dados serem retidos por um longo período de tempo – é igualmente importante para os sistemas de armazenagem de dados.) Embora 100% de disponibilidade seja impossível, uma disponibilidade próxima a 100% muitas vezes é viável de imediato, e o mercado normalmente expressa valores de alta disponibilidade em termos do número de “noves” no percentual de disponibilidade. Por exemplo, disponibilidades de 99% e de 99,999% podem ser referenciadas como disponibilidades de “2 naves” e “5 naves” respectivamente, e a meta atual divulgada para a disponibilidade do Google

Compute Engine é de “três noves e meio” – ou seja, 99,95% de disponibilidade.

Objetivos

Um SLO é um *objetivo de nível de serviço*: um valor ou um conjunto de valores que servem de meta para um nível de serviço medido por um SLI. Uma estrutura natural para os SLOs é, portanto, $SLI \leq \text{meta}$ ou $\text{limite inferior} \leq SLI \leq \text{limite superior}$. Por exemplo, podemos decidir que devolveremos os resultados da pesquisa no Shakespeare “rapidamente”, adotando um SLO segundo o qual nossa latência média de requisição de pesquisa deva ser menor que cem milissegundos.

Escolher um SLO apropriado é complicado. Para começar, nem sempre chegamos a escolher seu valor! Para requisições HTTP de entrada provenientes do mundo externo ao nosso serviço, a métrica QPS (Queries Per Second, ou Consultas por Segundo) é essencialmente determinada pelos desejos de seus usuários, e você não poderá realmente definir um SLO para isso.

Por outro lado, você *pode* dizer que deseja que a latência média por requisição esteja abaixo de cem milissegundos, e definir uma meta como essa, por sua vez, poderia motivá-lo a escrever seu frontend com vários tipos de comportamentos de baixa latência ou comprar determinados tipos de equipamento com baixa latência. (Obviamente, cem milissegundos é um valor arbitrário, mas, em geral, valores menores de latência são bons.) Há excelentes motivos para acreditar que rápido é melhor que lento, e que a latência acima de determinados valores experimentada pelos usuários realmente afasta as pessoas – consulte “Speed Matters” (Velocidade é importante) [Bru09] para ver mais detalhes.)

Novamente, isso é mais sutil do que poderia parecer à primeira vista, pois esses dois SLIs – QPS e latência – podem estar conectados de forma subjacente: QPS mais alto com frequência resulta em latências mais elevadas, e é comum que os serviços tenham uma queda de desempenho depois que ultrapassarem algum limite de carga.

Escolher e divulgar SLOs aos usuários gera expectativas sobre como será o

desempenho de um serviço. Essa estratégia pode reduzir reclamações sem fundamento aos proprietários de serviços, como, por exemplo, sobre o serviço estar lento. Sem um SLO explícito, os usuários muitas vezes 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 resultar em uma confiança excessiva no serviço, quando os usuários acreditam erroneamente que um serviço estará mais disponível do que ele está na realidade (como ocorre com o Chubby: veja a caixa de texto “A interrupção de serviço global e planejada do Chubby”), e em uma falta de confiança, quando usuários em prospecção acreditam que um sistema é mais frágil e menos confiável do que realmente é.

A interrupção de serviço global e planejada do Chubby

Escrito por Marc Alvidrez

O Chubby [Bur06] é o serviço de lock do Google para sistemas distribuídos com baixo acoplamento. No caso global, distribuímos instâncias do Chubby de modo que cada réplica esteja em uma região geográfica diferente. Com o tempo, descobrimos que as falhas na instância global do Chubby geravam interrupções de serviço de forma consistente, muitas das quais eram visíveis aos usuários finais. Porém, o fato é que verdadeiras interrupções globais de serviço do Chubby são tão raras que os proprietários de serviços começaram a adicionar dependências ao Chubby, supondo que ele jamais ficaria inativo. Sua alta confiabilidade oferecia uma falsa sensação de segurança porque os serviços não poderiam funcionar de modo apropriado se o Chubby estivesse indisponível, embora essa ocorrência fosse rara.

A solução para esse cenário com Chubby é interessante: o SRE garante que o Chubby global atenda ao seu objetivo de nível de serviço, mas não o excede de modo significativo. Em qualquer dado trimestre, se uma verdadeira falha não tiver reduzido a disponibilidade a um valor abaixo da meta, uma interrupção de serviço controlada será produzida, deixando o sistema inativo de forma intencional. Dessa maneira, somos capazes de acabar com dependências não razoáveis do Chubby assim que são

adicionadas. Fazer isso força os proprietários de serviços a enfrentar mais cedo a realidade dos sistemas distribuídos.

Acordos

Por fim, os SLAs são *acordos* de nível de serviço: um contrato explícito ou implícito com seus usuários, que inclui consequências por (deixar de) atender aos SLOs que eles contêm. As consequências são mais facilmente reconhecidas quando são financeiras – um desconto ou uma penalidade –, mas podem assumir outras formas. Um modo fácil de dizer qual é a diferença entre um SLO e um SLA é perguntar “o que acontecerá se os SLOs não forem atendidos?”: se não houver nenhuma consequência explícita, então é quase certo que você estará diante de um SLO.¹

A SRE normalmente não se envolve na definição de SLAs, pois eles estão intimamente ligados às decisões de negócio e de produto. A SRE, porém, se envolve em ajudar a evitar o acionamento das consequências de SLOs não atendidos. Os SREs também podem ajudar a definir os SLIs: obviamente, deve haver uma maneira objetiva de medir os SLOs no acordo; do contrário, haverá desentendimentos.

O Google Search é um exemplo de um serviço importante que não tem um SLA para o público: queremos que todos usem o Search do modo mais fluido e eficiente possível, mas não assinamos um contrato com o mundo todo. Mesmo assim, ainda há consequências caso o Search não esteja disponível – a falta de disponibilidade resulta em danos à nossa reputação, bem como em uma queda na receita com anúncios. Muitos outros serviços do Google, como o Google for Work, têm SLAs explícitos com seus usuários. Independentemente de um serviço em particular ter ou não um SLA, é importante definir SLIs e SLOs e usá-los para administrar o serviço.

Essa foi a teoria – vamos agora passar para a prática.

Indicadores na prática

Considerando que explicamos *por que* escolher métricas apropriadas para avaliar o seu serviço é importante, como você faz para identificar quais

métricas são significativas para o seu serviço ou sistema?

Com o que você e seus usuários se importam?

Você não deve usar todas as métricas que puder observar em seu sistema de monitoração como um SLI; uma compreensão sobre o que seus usuários querem do sistema dará base para uma seleção criteriosa de alguns indicadores. Escolher indicadores em demasia dificulta prestar o nível correto de atenção aos indicadores que são importantes, enquanto escolher poucos pode deixar comportamentos significativos de seu sistema sem análise. Geralmente percebemos que um punhado de indicadores representativos é suficiente para avaliar e tirar conclusões sobre a saúde de um sistema.

Os serviços tendem a se enquadrar em algumas categorias amplas no que diz respeito aos SLIs que acham relevantes:

- *Sistemas que oferecem serviços voltados ao usuário*, como os frontends de pesquisa do Shakespeare, geralmente se importam com *disponibilidade, latência e throughput*. Em outras palavras: Podemos responder à requisição? Quanto tempo demorou para responder? Quantas requisições podem ser tratadas?
- *Sistemas de armazenagem* com frequência enfatizam *latência, disponibilidade e durabilidade*. Em outras palavras: Quanto tempo demora para ler ou escrever dados? Podemos acessar os dados por demanda? Os dados continuarão presentes quando precisarmos deles? Veja o Capítulo 26, que apresenta uma discussão mais ampla sobre essas questões.
- *Sistemas de big data*, como pipelines de processamento de dados, tendem a se importar com *throughput* e *latência fim a fim*. Em outras palavras: Qual é o volume de dados que está sendo processado? Quanto tempo demora para os dados fluírem do recebimento até a conclusão? (Alguns pipelines também podem ter metas para latência em estágios individuais de processamento.)
- Todos os sistemas devem se importar com *correção*: a resposta correta foi devolvida, os dados corretos foram recuperados, a análise correta foi feita? É importante monitorar a correção como um indicador da saúde do

sistema, mesmo que, muitas vezes, ela seja uma propriedade dos dados do sistema, e não da infraestrutura *per se*, e, desse modo, normalmente não seja uma responsabilidade a ser cumprida pelo SRE.

Coletando indicadores

Muitas métricas de indicadores, em sua maior parte, são naturalmente coletadas do lado do servidor, usando um sistema de monitoração como o Borgmon (veja o Capítulo 10) ou o Prometheus, ou com análises periódicas de log – por exemplo, respostas HTTP 500 como uma fração de todas as requisições. Entretanto, alguns sistemas devem ser instrumentados para que a coleta seja feita do lado do *cliente*, pois não avaliar o comportamento no cliente pode fazer com que vários problemas que afetam os usuários, mas não afetam as métricas do lado do servidor, passem despercebidos. Por exemplo, concentrar-se na latência das respostas no backend da pesquisa de Shakespeare pode fazer com que uma latência ruim para o usuário devido a problemas com o JavaScript da página não seja detectada: nesse caso, calcular quanto tempo demora para uma página se tornar utilizável no navegador é uma aproximação melhor para aquilo que é realmente vivenciado pelo usuário.

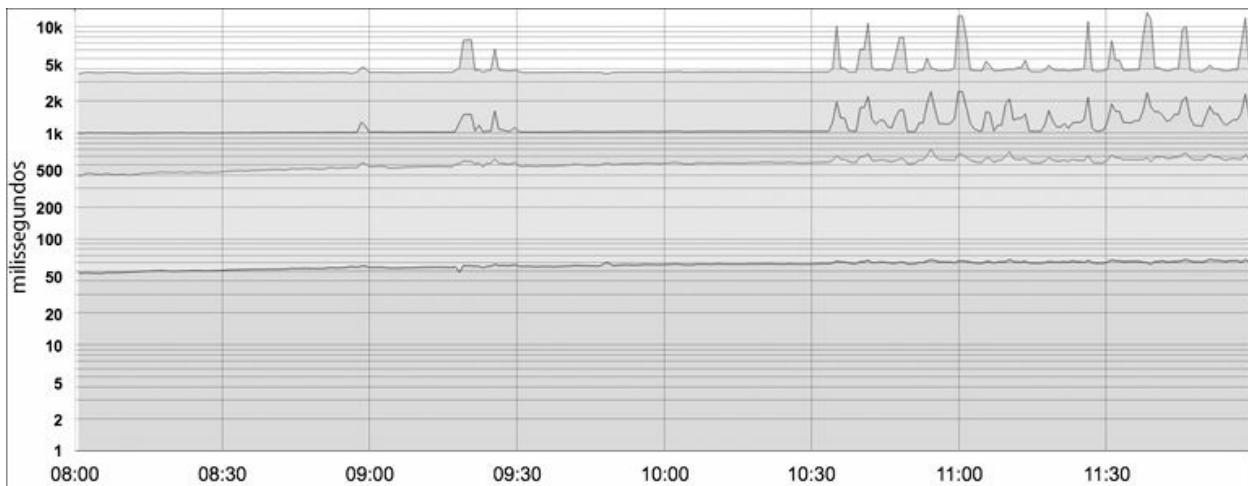
Agregação

Por questões de simplicidade e de usabilidade, com frequência agregamos medidas brutas. Isso deve ser feito com cuidado.

Algumas medidas parecem ser simples, como o número de requisições servidas *por segundo*, mas mesmo essa medida aparentemente simples agrupa dados de uma janela de medição de forma implícita. A medida é obtida uma vez por segundo ou pela média das requisições em um minuto? A última opção pode ocultar taxas instantâneas muito mais altas de requisições em picos que durem apenas alguns segundos. Considere um sistema que atenda a 200 requisições/s em segundos pares, e 0 nos demais. Ela tem a mesma carga média de um sistema que atenda a 100 requisições/s constantes, mas tem uma carga *instantânea* duas vezes maior que a carga média. De modo semelhante, calcular a média das latências das requisições pode parecer atraente, mas

oculta um detalhe importante: é totalmente possível que a maioria das requisições seja rápida, mas muitas requisições na cauda longa² sejam bem mais lentas.

A maioria das métricas pode ser mais bem compreendida como *distribuições*, e não como médias. Por exemplo, para um SLI de latência, algumas requisições serão servidas rapidamente, enquanto outras invariavelmente demorarão mais – às vezes, muito mais. Uma média simples pode ocultar essas latências na cauda, assim como mudanças nelas. A Figura 4.1 mostra um exemplo: embora uma requisição típica seja servida em aproximadamente 50 ms, 5% das requisições são 20 vezes mais lentas! Monitorar e gerar alertas com base apenas na latência média não evidenciará nenhuma mudança de comportamento no curso do dia, se houvesse mudanças significativas em latências na cauda (a linha mais acima).



*Figura 4.1 – Latências para os 50º, 85º, 95º e 99º percentis de um sistema.
Observe que o eixo Y tem uma escala logarítmica.*

Ao usar percentis como indicadores, podemos considerar o formato da distribuição e seus atributos diferenciadores: um percentil de alta ordem, como o 99º ou o 99,9º, mostra um valor plausível de pior caso, enquanto usar o 50º percentil (também conhecido como mediana) enfatiza o caso típico. Quanto maior a variação nos tempos de resposta, mais a experiência típica do usuário será afetada por comportamentos da cauda longa – um efeito exacerbado em carga alta por causa dos efeitos de enfileiramento. Estudos de usuários mostraram que as pessoas geralmente preferem um sistema um

pouco mais lento a um com uma grande variação no tempo de resposta, portanto algumas equipes de SRE se concentram somente nos valores de percentis altos, argumentando que se o comportamento do 99,9º percentil for bom, então a experiência típica com certeza também será.

Uma nota sobre falácia em estatística

Em geral, preferimos trabalhar com percentis em vez da média (média aritmética) de um conjunto de valores. Fazer isso possibilita considerar os pontos de dados na cauda longa que, com frequência, têm características significativamente diferentes (e mais interessantes) que a média. Por causa da natureza artificial dos sistemas de computação, os pontos de dados muitas vezes são distorcidos – por exemplo, nenhuma requisição pode ter uma resposta em menos de 0 ms e um timeout de 1.000 ms significa que não pode haver nenhuma resposta bem-sucedida com valores maiores que o timeout. Como resultado, não podemos supor que a média e a mediana sejam iguais – ou nem mesmo próximas!

Tentamos não supor que nossos dados estejam normalmente distribuídos sem antes verificá-los, caso algumas intuições e aproximações padrões não sejam válidas. Por exemplo, se a distribuição não estiver conforme o esperado, um processo que execute uma ação quando vir pontos fora da curva (por exemplo, reiniciar um servidor com latências altas de requisição) pode fazer isso com frequência demais, ou não fazê-lo com a frequência devida.

Padronize os indicadores

Recomendamos que você padronize definições comuns para SLIs para que você não precise sempre pensar neles partindo dos princípios básicos. Qualquer característica que esteja de acordo com os templates padrões de definição pode ser omitida da especificação de um SLI individual, por exemplo:

- Intervalos de agregação: “média calculada em um minuto”
- Regiões de agregação: “todas as tarefas em um cluster”

- Frequência com que as medições são feitas: “a cada dez segundos”
- Quais requisições estão incluídas: “HTTP GETs dos jobs de monitoração caixa-preta”
- Como os dados são adquiridos: “por meio de nossa monitoração, medidos no servidor”
- Latência para o acesso de dados: “tempo até o último byte”

Para economizar esforços, crie um conjunto de templates reutilizáveis de SLI para cada métrica comum; isso também facilita a todos compreender o que um SLI específico quer dizer.

Objetivos na prática

Comece pensando (ou descobrindo!) com o que seus usuários se importam, e não com o que você pode medir. Com frequência, o que é importante para os seus usuários é difícil ou impossível de medir; portanto, você acabará fazendo uma aproximação 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, às vezes percebemos que trabalhar a partir dos objetivos desejados de volta para indicadores específicos funciona melhor do que escolher indicadores e então definir as metas.

Definindo objetivos

Para ter o máximo de clareza, os SLOs devem especificar como são medidos e as condições em que são válidos. Por exemplo, podemos dizer o seguinte (a segunda linha é igual à primeira, mas conta com os defaults do SLI da seção anterior para eliminar redundância):

- 99% (média calculada em um minuto) das chamadas de RPC Get serão concluídas em menos de 100 ms (medidas em todos os servidores de backend).
- 99% das chamadas de RPC Get serão concluídas em menos de 100 ms.

Se o formato das curvas de desempenho for importante, você poderá

especificar várias metas de SLO:

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

Se você tiver usuários com cargas de trabalho heterogêneas, por exemplo, um pipeline que processa em grandes volumes, para o qual o throughput seja importante, e um cliente interativo que se importa com latência, talvez seja apropriado definir objetivos separados para cada classe de carga de trabalho:

- 95% das chamadas de RPC Set dos clientes interessados em throughput serão concluídas em < 1 s.
- 99% das chamadas de RPC Set com payloads < 1 kB dos clientes interessados em latência serão concluídas em < 10 ms.

Insistir que os SLOs sejam atendidos 100% do tempo não é desejável nem realista: fazer isso pode reduzir a taxa de inovação e de implantação, exigir soluções caras e demasiadamente conservadoras, ou ambos. Em vez disso, é melhor permitir uma provisão para erros – uma taxa com que os SLOs podem deixar de ser atendidos – e monitorá-la de modo diário ou semanal. A alta gerência provavelmente vai querer uma avaliação mensal ou trimestral também. (Uma provisão para erros é simplesmente um SLO para atender a outros SLOs!)

A taxa com que os SLOs deixam de ser atendidos é um indicador útil para a saúde do serviço conforme percebida pelos usuários. É conveniente monitorar os SLOs (e as violações de SLO) de forma diária ou semanal para ver tendências e obter avisos de problemas em potencial antes que eles ocorram. A alta gerência provavelmente vai querer uma avaliação mensal ou trimestral também nesse caso.

A taxa de violação de SLOs pode ser comparada à provisão para erros (veja a seção “Motivação para provisão de erros”), com a diferença usada como uma entrada para o processo que decide quando podemos fazer o rollout de novas versões.

Definindo as metas

Definir metas (SLOs) não é uma atividade puramente técnica por causa das implicações no produto e nos negócios, que devem se refletir tanto nos SLIs quanto nos SLOs (e talvez nos SLAs) selecionados. De modo semelhante, talvez seja necessário fazer uma negociação entre alguns atributos do produto e outros considerando as restrições impostas pelas equipes, o time-to-market, a disponibilidade de hardware e o orçamento. Embora a SRE deva fazer parte dessa conversa e aconselhar sobre os riscos e a viabilidade das diferentes opções, aprendemos algumas lições que podem ajudar a deixar essa discussão mais produtiva:

Não escolha uma meta baseada no desempenho atual

Embora entender os méritos e os limites de um sistema seja essencial, adotar valores sem refletir sobre eles pode obrigar você a dar suporte a um sistema que exija esforços heroicos para atender às suas metas, e que não poderá ser melhorado sem um replanejamento significativo.

Mantenha a simplicidade

Agregações complicadas em SLIs podem ocultar mudanças no desempenho do sistema; além disso, será mais difícil compreendê-las.

Evite absolutos

Apesar de ser tentador pedir que um sistema possa escalar sua carga “infinitamente” sem nenhum aumento de latência e que esteja “sempre” disponível, esse requisito não é realista. Mesmo um sistema que abrace esses ideais provavelmente exigiria muito tempo para ser projetado e criado, e teria um alto custo de operação – haveria uma grande chance de esse sistema acabar sendo desnecessariamente melhor do que aquele com o qual os usuários estariam satisfeitos (e até mesmo encantados).

Tenha o mínimo possível de SLOs

Escolha SLOs suficientes apenas para oferecerem uma boa cobertura dos atributos de seu sistema. Defenda os SLOs que você selecionar: se não puder jamais vencer uma conversa sobre prioridades citando um determinado SLO, é provável que não valha a pena ter esse SLO.³ No entanto, nem todos os atributos do produto podem ser capturados por SLOs:

é difícil especificar o “encantamento do usuário” com um SLO.

A perfeição pode esperar

Você sempre pode aperfeiçoar as definições e as metas de SLO com o tempo, à medida que conhecer melhor o comportamento de um sistema. É melhor começar com uma meta menos rigorosa, que possa ser elevada, do que escolher uma meta excessivamente rígida, que deva ser reduzida quando descobrir que ela é inatingível.

Os SLOs podem – e devem – ser um fator importante na determinação das prioridades das tarefas dos SREs e dos desenvolvedores de produto, pois refletem aquilo com que os usuários se importam. Um bom SLO é uma função incentivadora útil e legítima para uma equipe de desenvolvimento. Porém, um SLO mal planejado pode resultar em trabalho desperdiçado se uma equipe faz esforços heroicos para atender a um SLO demasiadamente agressivo, ou em um produto ruim se o SLO for brandão demais. Os SLOs são um forte incentivador: utilize-os sabiamente.

Medidas de controle

Os SLIs e os SLOs são elementos essenciais nos circuitos de controle utilizados para administrar sistemas:

1. Monitore e calcule os SLIs do sistema.
2. Compare os SLIs aos SLOs e decida se uma ação é necessária ou não.
3. Se uma ação for necessária, descubra o *que* deve acontecer para atender à meta.
4. Execute essa ação.

Por exemplo, se o passo 2 mostrar que a latência das requisições está aumentando, e o SLO não será atendido em algumas horas a menos que algo seja feito, o passo 3 poderá incluir um teste para a hipótese de que os servidores estejam limitados por CPU (CPU-bound) e decidir pela adição de mais CPUs a fim de distribuir a carga. Sem o SLO, você não saberia se deve tomar uma atitude (nem quando).

Os SLOs definem expectativas

Publicar os SLOs define expectativas para o comportamento do sistema. Os usuários (e os potenciais usuários) com frequência querem saber o que podem esperar de um serviço para entender se ele é apropriado para seus casos de uso. Por exemplo, uma equipe que queira desenvolver um site de compartilhamento de fotos talvez queira evitar o uso de um serviço que prometa uma durabilidade muito longa e um custo baixo em troca de uma disponibilidade um pouco menor, embora o mesmo serviço possa ser uma opção perfeita para um sistema de gerenciamento de registros para arquivamento.

Para definir expectativas realistas aos seus usuários, você pode considerar o uso de uma ou de ambas as táticas a seguir:

Manter uma margem de segurança

Usar um SLO interno mais rigoroso que o SLO anunciado aos usuários permite que você tenha espaço para responder a problemas crônicos antes que eles se tornem visíveis externamente. Um buffer de SLO também possibilita acomodar reimplementações que negociem desempenho em troca de outros atributos, como custos ou facilidade de manutenção, sem ter que desapontar os usuários.

Não exagere na realização

Os usuários fazem desenvolvimentos com base na realidade que você oferece, e não naquilo que você diz que vai fornecer, particularmente para serviços de infraestrutura. Se o desempenho real de seu serviço for bem melhor do que o que está definido no SLO, os usuários passarão a contar com o seu desempenho atual. Você pode evitar a dependência em excesso deixando o sistema ocasionalmente offline de forma deliberada (o serviço Chubby do Google introduziu interrupções de serviço planejadas em resposta ao fato de estar disponível em demasia)⁴, criando um gargalo para algumas requisições ou fazendo o design do sistema de modo que ele não seja mais rápido com menos carga.

Entender a eficiência com que um sistema atende às expectativas ajuda a decidir se devemos investir em deixar o sistema mais rápido, mais disponível e mais robusto. De modo alternativo, se o serviço estiver se saindo bem,

talvez o tempo dos funcionários devesse ser gasto em outras prioridades, tais como pagar débitos técnicos, acrescentar novas funcionalidades ou introduzir outros produtos.

Acordos na prática

Redigir um SLA exige equipes de negócios e equipes jurídicas para definir as consequências e as penalidades apropriadas devido a uma falha. O papel da SRE é ajudá-los a compreender a probabilidade e a dificuldade de atender aos SLOs contidos no SLA. Boa parte dos conselhos na definição de SLOs também se aplica aos SLAs. Ser conservador em relação àquilo que é divulgado aos usuários é uma atitude sábia, pois quanto maior a clientela, mais difícil será alterar ou apagar SLAs que se mostrem imprudentes ou com os quais seja difícil de trabalhar.

¹ A maioria das pessoas realmente quer dizer SLO quando diz “SLA”. Uma pista: se alguém falar de uma “violação de SLA”, quase sempre estará falando de um SLO não atendido. Uma verdadeira violação de SLA pode dar origem a um caso judicial por quebra de contrato.

² N.T.: “Cauda longa (do inglês long tail) é um termo utilizado na Estatística para identificar distribuições de dados como a curva de Pareto, onde o volume de dados é classificado de forma decrescente. Quando comparada a uma distribuição normal, ou Gaussiana, a cauda longa apresenta uma quantidade muito maior de dados ao longo da cauda.” Fonte: https://pt.wikipedia.org/wiki/Cauda_longa.

³ Se você jamais puder vencer uma conversa sobre SLOs, provavelmente não valerá a pena ter uma equipe de SRE para o produto.

⁴ A injeção de falhas [Ben12] tem um propósito diferente, mas também pode ajudar a definir expectativas.

CAPÍTULO 5

Eliminando tarefas penosas

Escrito por Vivek Rau

Editado por Betsy Beyer

Se um operador humano precisa estar em contato com seu sistema durante as operações normais, você tem um bug.

A definição de normal muda à medida que seu sistema cresce.

— Carla Geisser, SRE do Google

Em SRE, queremos gastar tempo com trabalhos de longo prazo em projetos de engenharia, e não com tarefas operacionais. Como o termo *tarefa operacional* pode ser interpretado erroneamente, utilizamos uma expressão específica: *tarefas penosas* (toil).

Definição de tarefas penosas

As tarefas penosas não são apenas as “tarefas que não gosto de fazer”. Também não são simplesmente o equivalente às tarefas administrativas ou trabalhos sujos. As preferências quanto aos tipos de trabalho que sejam satisfatórios e agradáveis variam de pessoa para pessoa, e algumas pessoas até mesmo gostam de trabalhos manuais e repetitivos. Há também tarefas administrativas que precisam ser feitas, mas não devem ser classificadas como tarefas penosas: são as *atividades extras* (overhead). As atividades extras com frequência são tarefas não ligadas diretamente à operação de um serviço de produção, e incluem atividades como reuniões da equipe, definição e atribuição de notas às metas¹, snippets² e papelada de RH. Trabalhos sujos às vezes podem ter valor no longo prazo e, nesse caso, também não são considerados tarefas penosas. Limpar toda a configuração de alertas de seu serviço e remover o excesso pode ser um trabalho sujo, mas não é uma tarefa penosa.

Então, o que é uma tarefa penosa? A tarefa penosa é um tipo de tarefa ligada à operação de um serviço de produção que tende a ser manual, repetitiva, possível de ser automatizada, tática, desprovida de valor durável e que escala linearmente com o crescimento do serviço. Nem todas as tarefas consideradas penosas têm todos esses atributos, mas quanto mais uma tarefa se assemelhar a uma ou mais das descrições a seguir, maiores serão as chances de ela ser considerada uma tarefa penosa:

Manual

Inclui trabalhos como executar manualmente um script que automatize alguma tarefa. Executar um script pode ser mais rápido que executar cada passo do script manualmente, mas o tempo que um ser humano gasta com a *mão na massa* executando esse script (e não o tempo decorrido) continua sendo o tempo de uma tarefa penosa.

Repetitiva

Se você estiver executando uma tarefa pela primeira vez, ou até mesmo pela segunda, essa não será uma tarefa penosa. A tarefa penosa é uma tarefa que você executa repetidamente. Se você estiver resolvendo um problema novo ou inventando uma nova solução, essa não será uma tarefa penosa.

Passível de automação

Se uma máquina puder realizar a tarefa tão bem quanto um ser humano, ou a necessidade da tarefa puder ser eliminada por design, ela será uma tarefa penosa. Se o discernimento humano for essencial para a tarefa, há uma boa chance de ela não ser uma tarefa penosa.³

Tática

As tarefas penosas são orientadas a interrupções e são reativas, e não orientadas a estratégias e proativas. Tratar alertas de pagers é uma tarefa penosa. Talvez não sejamos capazes de jamais eliminar esse tipo de trabalho completamente, mas devemos trabalhar continuamente no sentido de minimizá-lo.

Sem valor durável

Se seu serviço permanece no mesmo estado depois que você termina uma tarefa, essa tarefa provavelmente é uma tarefa penosa. Se a tarefa produziu uma melhoria permanente em seu serviço, é provável que não tenha sido uma tarefa penosa, mesmo que um pouco de trabalho sujo – por exemplo, mergulhar em código legado e em configurações e deixá-los mais robustos – estivesse envolvido.

O(n) com o crescimento do serviço

Se o trabalho envolvido em uma tarefa escala linearmente com o tamanho do serviço, o volume de tráfego ou o número de usuários, essa tarefa provavelmente é uma tarefa penosa. Um serviço administrado e projetado de modo ideal é capaz de crescer pelo menos em uma ordem de magnitude sem trabalhos adicionais, além da aplicação de alguns esforços exigidos apenas uma vez para a adição de recursos.

Por que menos tarefas penosas é melhor

Nossa organização SRE tem uma meta divulgada de manter o trabalho operacional (isto é, as tarefas penosas) abaixo de 50% do tempo de cada SRE. No mínimo, 50% do tempo de cada SRE deve ser gasto em trabalho de projetos de engenharia que reduzirão as tarefas penosas futuras ou acrescentarão funcionalidades ao serviço. O desenvolvimento de funcionalidades geralmente tem como foco melhorar a confiabilidade, o desempenho ou a utilização, o que com frequência reduz as tarefas penosas como um efeito secundário.

Compartilhamos essa meta de 50% porque as tarefas penosas tendem a se expandir se não forem controladas e podem rapidamente ocupar 100% do tempo de todos. O trabalho de reduzir as tarefas penosas e escalar os serviços é a “Engenharia” do termo Site Reliability Engineering (Engenharia de Confiabilidade de Sites). O trabalho de engenharia é o que permite à organização SRE escalar de forma proporcionalmente menor que o tamanho do serviço e administrar serviços de modo mais eficiente que uma equipe puramente de Dev ou puramente de Ops.

Além do mais, quando contratamos novos SREs, prometemos a eles que a

SRE não é uma organização típica de Ops, citando a regra de 50% que acabamos de mencionar. Precisamos cumprir essa promessa não permitindo que a organização SRE ou qualquer subequipe dela se transforme em uma equipe de Ops.

Calculando as tarefas penosas

Se procuramos limitar o tempo que um SRE gasta em tarefas penosas a 50%, como esse tempo é gasto?

Há um limite inferior para a quantidade de tarefas penosas que qualquer SRE deve tratar se estiver de plantão. Um SRE típico tem uma semana de plantão primário e uma semana de plantão secundário a cada ciclo (para uma discussão sobre turnos de plantão primário e secundário, veja o Capítulo 11). Isso implica que, em um rodízio de seis pessoas, pelo menos duas a cada seis semanas são dedicadas a turnos de plantão e tratamento de interrupções, o que significa que o limite inferior de tarefas penosas em potencial é $2/6 = 33\%$ do tempo de um SRE. Em um rodízio de oito pessoas, o limite inferior será $2/8 = 25\%$.

De modo consistente com esses dados, os SREs informam que sua principal fonte de tarefas penosas são as interrupções (isto é, mensagens e emails não urgentes relacionados ao serviço). A próxima fonte principal é a resposta (urgente) de plantão, seguida de releases e atualizações de versão (pushes). Mesmo que nosso processo de release e de atualizações de versão geralmente seja tratado com uma boa dose de automação, ainda há bastante espaço para melhorias nessa área.

Avaliações trimestrais dos SREs do Google mostram que o tempo médio gasto em tarefas penosas é de aproximadamente 33%; portanto, estamos bem melhor que a nossa meta geral de 50%. No entanto, a média não captura os pontos fora da curva: alguns SREs afirmam que têm 0% de tarefas penosas (apenas projetos de desenvolvimento, sem trabalho de plantão), enquanto outros argumentam que têm 80% de tarefas penosas. Quando SREs individuais informam excesso de tarefas penosas, com frequência isso indica uma necessidade de os gerentes distribuírem a

carga de tarefas penosas de modo mais uniforme na equipe e incentivarem esses SREs a encontrar projetos de engenharia satisfatórios.

O que é qualificado como engenharia?

O trabalho de engenharia é novo e exige intrinsecamente um discernimento humano. Produz uma melhoria permanente em seu serviço e é orientado por uma estratégia. Frequentemente é criativo e inovador, assumindo uma abordagem orientada a 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 com mais serviços, com o mesmo tamanho de equipe.

Atividades típicas de SRE se enquadram nas seguintes categorias aproximadas:

Engenharia de software

Envolve escrever ou modificar códigos, além de qualquer trabalho associado de design e de documentação. Exemplos incluem escrever scripts de automação, criar ferramentas ou frameworks, adicionar funcionalidades ao serviço para escalabilidade e confiabilidade ou modificar o código da infraestrutura para deixá-la mais robusta.

Engenharia de sistemas

Envolve configurar sistemas de produção, modificar configurações ou documentar sistemas de forma a produzir melhorias duradouras resultantes de um esforço aplicado uma só vez. Exemplos incluem configurações e atualizações para monitoração, configuração de distribuição de carga, configuração de servidores, ajuste de parâmetros do sistema operacional e configuração do distribuidor de carga. A engenharia de sistemas também inclui consultorias em arquitetura, design e processo de transformação em produto (productionization) para as equipes de desenvolvedores.

Tarefas penosas

Tarefas diretamente ligadas à operação de um serviço que sejam repetitivas, manuais etc.

Tarefas extras (overhead)

Tarefas administrativas não ligadas diretamente à operação de um serviço. Exemplos incluem contratação, papelada de RH, reuniões da equipe/empresa, limpeza de fila de bugs, snippets, revisões com pares, autoavaliações e cursos de treinamento.

Todo SRE deve gastar no mínimo 50% de seu tempo em trabalhos de engenharia, quando a média for calculada em um período de alguns trimestres ou em um ano. As tarefas penosas tendem a ter picos; portanto, um tempo constante de 50% gasto em engenharia pode não ser realista para algumas equipes de SRE, e elas podem estar bem abaixo dessa meta em alguns trimestres. Contudo, se a fração de tempo gasta em projetos tiver uma média significativamente abaixo de 50% no longo prazo, a equipe afetada deve dar um passo para trás e descobrir o que está errado.

As tarefas penosas são sempre ruins?

As tarefas penosas não deixam todos insatisfeitos o tempo todo, especialmente se forem em pequena quantidade. Tarefas previsíveis e repetitivas podem ser bem tranquilizantes. Elas geram um senso de realização e de vitórias rápidas. Podem ser atividades de baixo risco e pouco estresse. Algumas pessoas são atraídas por tarefas penosas e podem até mesmo gostar desse tipo de trabalho.

As tarefas penosas não são sempre nem invariavelmente ruins, e todos precisam saber de forma absolutamente clara que um pouco de tarefas penosas é inevitável na função de SRE e, na verdade, em qualquer função de engenharia. Em doses pequenas, as tarefas penosas não são um problema, e se você estiver satisfeito com essas pequenas doses, então tudo bem. As tarefas penosas se tornam tóxicas quando feitas em grandes quantidades. Se você estiver sobrecarregado de muitas tarefas penosas, deverá ficar bastante preocupado e reclamar em voz alta. Entre os vários motivos pelos quais ter muitas tarefas penosas é ruim, considere o seguinte:

Estagnação na carreira

O progresso de sua carreira se tornará mais lento ou será interrompido se

você gastar pouco tempo em projetos. O Google recompensa trabalhos sujos quando são inevitáveis e exercem um grande impacto positivo, mas você não pode fazer uma carreira com base em trabalhos sujos.

Baixo moral

As pessoas têm limites diferentes para o volume de tarefas penosas que podem tolerar, mas todos têm um limite. Tarefas penosas em demasia resultam em esgotamento, tédio e insatisfação.

Além disso, gastar tempo demais em tarefas penosas em detrimento de gastá-lo em engenharia prejudica uma organização SRE das seguintes maneiras:

Gera confusão

Trabalhamos arduamente para garantir que todos que trabalhem na organização SRE ou junto a ela entendam que somos uma organização de engenharia. Os indivíduos ou as equipes na SRE que se envolvam com tarefas penosas em excesso comprometem a clareza dessa mensagem e confundem as pessoas em relação às nossas funções.

Progresso lento

Tarefas penosas em demasia deixam uma equipe menos produtiva. A velocidade de introdução de novas funcionalidades em um produto se tornará mais lenta se a equipe de SRE estiver ocupada demais com tarefas manuais e apagando incêndios para que seja possível fazer o rollout de novas funcionalidades prontamente.

Gera precedentes

Se você estiver disposto demais a assumir tarefas penosas, suas contrapartidas em Dev serão incentivadas a sobrecarregar você com mais tarefas penosas, às vezes transferindo tarefas operacionais que deveriam ser realizadas por Devs para a SRE. Outras equipes também poderão começar a esperar que os SREs assumam essas tarefas, o que seria ruim por motivos óbvios.

Promove atrito

Mesmo que você não esteja pessoalmente insatisfeito com tarefas penosas,

seus colegas de trabalho atuais ou futuros podem gostar bem menos delas. Se você incluir muitas tarefas penosas nos procedimentos de sua equipe, estará motivando os melhores engenheiros da equipe a começarem a procurar empregos mais satisfatórios em outros lugares.

Causa quebra de confiança

Pessoas recém-contratadas ou transferidas de outras áreas que se juntaram à SRE com a promessa de trabalhar em projetos se sentirão enganadas, o que é ruim para o moral.

Conclusão

Se todos nos comprometermos em eliminar um pouco das tarefas penosas a cada semana com um bom trabalho de engenharia, faremos uma limpeza em nossos serviços de forma constante e poderemos deslocar nossos esforços coletivos para usar a engenharia a fim de escalar, planejar a próxima geração de serviços e criar cadeias de ferramentas disponíveis a todos os SREs. Vamos inventar mais e fazer menos tarefas penosas.

¹ Usamos o sistema Objectives and Key Results (Objetivos e resultados principais), cujo pioneiro foi Andy Grove na Intel; veja [Kla12].

² Os Googlers registram pequenos resumos de formato livre, ou “snippets”, contendo aquilo com que trabalhamos a cada semana.

³ Devemos tomar cuidado ao dizer que uma tarefa “não é penosa porque precisa de discernimento humano”. Devemos pensar com cuidado se a natureza da tarefa exige intrinsecamente um discernimento humano e não pode ser tratada por um design melhor. Por exemplo, poderíamos construir (e alguns o fizeram) um serviço que alerte seus SREs várias vezes ao dia, em que cada alerta exija uma resposta complexa envolvendo muito discernimento humano. Um serviço desse tipo tem um design precário, com uma complexidade desnecessária. O sistema deve ser simplificado e reconstruído para eliminar as condições de falha subjacentes ou lidar com essas condições de modo automático. Até que o novo design e a nova implementação tenham sido concluídos e o rollout do serviço melhorado seja feito, o trabalho de aplicar discernimento humano para responder a cada alerta, definitivamente, será uma tarefa penosa.

CAPÍTULO 6

Monitorando sistemas distribuídos

Escrito por Rob Ewaschuk

Editado por Betsy Beyer

As equipes de SRE do Google têm alguns princípios básicos e boas práticas para construir sistemas bem-sucedidos de monitoração e alertas. Este capítulo apresenta diretrizes sobre quais problemas devem interromper um ser humano com um page e como lidar com problemas que não sejam sérios o suficiente a ponto de acionar um page.

Definições

Não há um vocabulário uniformemente compartilhado para discutir todos os assuntos relacionados à monitoração. Até mesmo dentro do Google, o uso dos termos a seguir varia, mas as interpretações mais comuns estão listadas a seguir:

Monitoração

Coletar, processar, agregar e exibir dados quantitativos de tempo real sobre um sistema; por exemplo, contadores e tipos de consultas, contadores e tipos de erros, tempos de processamento e tempos de vida dos servidores.

Monitoração caixa-branca

Monitoração baseada em métricas expostas por partes internas do sistema, incluindo logs, interfaces como a Java Virtual Machine Profiling Interface ou um handler HTTP que gere estatísticas internas.

Monitoração caixa-preta

Testar comportamentos visíveis externamente, como um usuário os veria.

Painel de controle (dashboard)

Uma aplicação (geralmente baseada em web) que apresenta uma visão resumida das principais métricas de um serviço. Um painel de controle pode ter filtros, seletores e outros itens, mas é pré-configurado para expor as métricas que sejam mais importantes aos seus usuários. O painel de controle também pode exibir informações da equipe, como o tamanho da fila de tickets, uma lista dos bugs de alta prioridade, o engenheiro de plantão no momento para uma dada área de responsabilidade ou atualizações de versão (pushes) recentes.

Alerta

Uma notificação cujo propósito é ser lida por um ser humano e é enviada a um sistema; por exemplo, uma fila de bugs ou tickets, um alias de email ou um pager. Respectivamente, esses alertas são classificados como *tickets*, *alertas de email*¹ e *pages*.

Causa-raiz

Um defeito em um sistema de software ou humano que, se corrigido, instila confiança de que esse evento não acontecerá novamente do mesmo modo. Um dado incidente pode ter várias causas-raízes: por exemplo, talvez tenha sido causado por uma combinação de automação insuficiente de processos, software que falhou por causa de dados espúrios de entrada e insuficiência de testes no script usado para gerar a configuração. Cada um desses fatores pode representar uma causa-raiz por si só, e cada um deles deve ser corrigido.

Nó e máquina

Usados de modo intercambiável para indicar uma única instância de um kernel em execução em um servidor físico, em uma máquina virtual ou em um contêiner. Pode haver vários serviços que valham a pena ser monitorados em uma única máquina. Os serviços podem:

- estar relacionados um ao outro: por exemplo, um servidor de caching e um servidor web;
- ser serviços não relacionados que compartilham o hardware: por exemplo,

um repositório de código e um mestre (master) para um sistema de configuração como o Puppet (<https://puppetlabs.com/puppet/puppet-open-source>) ou o Chef (<https://www.chef.io/chef/>).

Atualização de versão (push)

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

Por que monitorar?

Existem vários motivos para monitorar um sistema, incluindo:

Analisar tendências de longo prazo

Qual é o tamanho do meu banco de dados e qual é a rapidez com que ele está crescendo? Com que velocidade o meu contador de usuários ativos diariamente (daily-active users) está crescendo?

Fazer comparações ao longo do tempo ou com grupos experimentais

As consultas são mais rápidas com o Bucket of Bytes 2.72 da empresa X em comparação com o Ajax DB 3.14? Qual é a melhoria na taxa de hits de cache em memória com um nó extra? Meu site está mais lento do que estava na semana passada?

Gerar alertas

Algo está errado e alguém precisa fazer a correção imediatamente! Ou algo poderá falhar em breve, portanto alguém deverá olhar logo.

Criar painéis de controle

Os painéis de controle devem responder a perguntas básicas sobre o seu serviço e normalmente incluem alguma forma dos quatro sinais de ouro (discutidos na seção “Os quatro sinais de ouro”).

Conduzir análises retrospectivas ad hoc (isto é, depuração)

Nossa latência simplesmente disparou; o que mais aconteceu aproximadamente na mesma hora?

A monitoração de sistemas também é útil para fornecer dados brutos à

aplicação de analytics² aos negócios e facilitar a análise de brechas de segurança. Como este livro tem como foco os domínios da engenharia, na qual a SRE tem expertise em particular, não discutiremos essas aplicações de monitoração aqui.

Monitorar e gerar alertas permite que um sistema nos diga quando tem falhas ou, quem sabe, nos informe o que está prestes a falhar. Quando o sistema não é capaz de se corrigir automaticamente, queremos que um ser humano investigue o alerta, determine se ele tem um problema de verdade em mãos, resolva o problema e determine a sua causa-raiz. A menos que você esteja realizando auditorias de segurança em componentes de um sistema com escopos bem restritos, você jamais deverá acionar um alerta simplesmente porque “parecia haver algo um pouco estranho”.

Acionar o pager de uma pessoa representa um uso bem caro do tempo de um funcionário. Se um funcionário estiver na empresa, um page interromperá seu fluxo de trabalho. Se estiver em casa, um page interromperá seu tempo pessoal e, quem sabe, até mesmo seu sono. Quando ocorrem pages com frequência demais, os funcionários começam a tentar adivinhar, fazem uma leitura superficial ou até mesmo deixam de ver alertas, ignorando inclusive um page “real” que acaba sendo mascarado pelo ruído. As interrupções de serviço podem se prolongar porque outros ruídos interferem em um diagnóstico e uma correção rápidos. Sistemas eficientes de alerta têm um bom sinal e pouco ruído.

Definindo expectativas razoáveis para monitoração

Monitorar uma aplicação complexa é um empreendimento significativo de engenharia por si só. Mesmo com uma infraestrutura substancial para instrumentação, coleta, exibição e geração de alertas implantada, uma equipe de SRE do Google com 10 a 12 membros geralmente tem um ou, às vezes, dois membros cuja atribuição principal é construir e manter os sistemas de monitoração para seu serviço. Esse número tem diminuído com o passar do tempo à medida que generalizamos e centralizamos a infraestrutura comum de monitoração, mas toda equipe de SRE geralmente tem pelo menos uma “pessoa para monitoração”. (Apesar do que foi dito, embora possa parecer

divertido ter acesso a painéis de controle com gráficos de tráfego e dados semelhantes, as equipes de SRE evitam cuidadosamente qualquer situação que exija alguém que fique “olhando para uma tela a fim de observar se há problemas”.)

Em geral, o Google tem a tendência de usar sistemas de monitoração mais simples e rápidos, com ferramentas melhores para uma análise *post hoc*. Evitamos sistemas “mágicos”, que tentem aprender limites ou detectar casualidades de forma automática. Regras que detectam mudanças inesperadas nas taxas de requisição dos usuários finais são um contraexemplo: embora essas regras ainda sejam mantidas do modo mais simples possível, elas fazem uma detecção muito rápida de uma anomalia bem simples, específica e grave. Outros usos dos dados de monitoração, como o planejamento de capacidade e a previsão de tráfego, podem tolerar mais fragilidade e, desse modo, mais complexidade. Experimentos de observação conduzidos em um horizonte de prazo bem distante (meses ou anos) com uma taxa de amostragem baixa (horas ou dias) com frequência também podem tolerar mais fragilidade, pois amostras perdidas ocasionalmente não ocultarão uma tendência de longo prazo.

A SRE do Google tem tido apenas um sucesso limitado com hierarquias de dependências complexas. Raramente utilizamos regras como “Se sei que o banco de dados está lento, gere um alerta para um banco de dados lento; caso contrário, gere um alerta informando que o site está lento de modo geral”. Regras que contam com dependências normalmente são pertinentes a partes muito estáveis de nosso sistema, como o nosso sistema para drenar o tráfego de usuários de um datacenter. Por exemplo, “Se o tráfego de um datacenter for drenado, não me alerte sobre sua latência” é uma regra de alerta comum dos datacenters. Poucas equipes no Google mantêm hierarquias complexas de dependências porque nossa infraestrutura tem uma taxa constante de refatoração contínua.

Algumas das ideias descritas neste capítulo ainda são uma aspiração: sempre há espaço para passar mais rapidamente do sintoma para a(s) causa(s)-raiz(es), principalmente em sistemas em constante mudança. Então, embora este capítulo defina algumas metas para os sistemas de monitoração e

algumas maneiras de alcançar essas metas, é importante que os sistemas de monitoração – em especial o caminho crítico do ponto inicial de um problema em produção até um page para um ser humano, passando pela triagem básica e uma depuração profunda – continuem simples e abrangentes e sejam mantidos assim por todos da equipe.

De modo semelhante, para deixar o nível de ruído baixo e o sinal claro, os elementos de seu sistema de monitoração que levam a um pager devem ser bem simples e robustos. As regras que geram alertas para os seres humanos devem ser simples de entender e devem representar uma falha clara.

Sintomas versus causas

Seu sistema de monitoração deve responder a duas perguntas: o que está errado e por quê?

“O que está errado?” indica o sintoma; “por quê?” indica uma causa (possivelmente intermediária). A Tabela 6.1 lista alguns sintomas hipotéticos e as causas correspondentes.

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 estão lentas	As CPUs estão sobrecarregadas com um bogosort*, ou um cabo de Ethernet está preso embaixo de um rack, o que é visível como uma perda parcial de pacotes.
Usuários na Antártida não estão recebendo GIFs animados de gatos.	Sua Rede de Distribuição de Conteúdo (Content Distribution Network) odeia cientistas e felinos e, desse modo, colocou os IPs de alguns clientes na lista negra.
Conteúdo privado está legível ao mundo	Uma nova atualização de software fez com que as ACLs fossem esquecidas e permitiu todas as requisições.

* N.T.: “Bogosort (também conhecido como *CaseSort*) é um algoritmo de ordenação extremamente ineficiente. É baseado na reordenação aleatória dos elementos. Não é utilizado na prática, mas pode ser usado no ensino de algoritmos mais eficientes.” (Fonte: <https://pt.wikipedia.org/wiki/Bogosort>.)

“O que” *versus* “por que” é uma das distinções mais importantes ao escrever um bom sistema de monitoração com o sinal no nível máximo e um mínimo de ruído.

Caixa-preta versus caixa-branca

Combinamos um uso intenso de monitoração caixa-branca com usos modestos, porém importantes, de monitoração caixa-preta. A maneira mais simples de pensar na monitoração caixa-preta *versus* a monitoração caixa-branca é que a monitoração caixa-preta é orientada a sintomas e representa problemas ativos – não previstos: “O sistema não está funcionando corretamente, neste instante”. A monitoração caixa-branca depende da capacidade de inspecionar a parte interna do sistema, como logs ou endpoints de HTTP, com instrumentação. A monitoração caixa-branca, desse modo, permite detectar problemas iminentes, falhas mascaradas por retentativas e assim por diante.

Observe que, em um sistema com várias camadas, o sintoma para uma pessoa é a causa para outra. Por exemplo, suponha que o desempenho de um banco de dados esteja ruim. Leituras lentas de banco de dados são um sintoma para o SRE do banco de dados que as detecta. No entanto, para o SRE do frontend que está observando um site lento, as mesmas leituras lentas do banco de dados são uma causa. Desse modo, a monitoração caixa-branca às vezes é orientada a sintomas e, outras vezes, é orientada a causas, dependendo simplesmente do quanto informativa seja a sua caixa-branca.

Ao coletar dados de telemetria para depuração, a monitoração caixa-branca é essencial. Se os servidores web parecerem lentos em requisições com uso intenso de banco de dados, você precisará saber o quanto rápido o servidor web percebe que o banco de dados está e o quanto rápido o banco de dados acredita que é. Do contrário, não será possível distinguir um servidor de banco de dados realmente lento de um problema de rede entre seu servidor web e o banco de dados.

No que diz respeito ao paging, a monitoração caixa-preta tem a vantagem principal de forçar uma disciplina em que um ser humano seja importunado somente quando um problema já estiver acontecendo e contribuindo com os

sintomas reais. Por outro lado, para problemas iminentes, mas que ainda não estejam ocorrendo, a monitoração caixa-preta não é muito útil.

Os quatro sinais de ouro

Os quatro sinais de ouro para monitoração são: latência, tráfego, erros e saturação. Se você puder calcular apenas quatro métricas de seu sistema voltado ao usuário, concentre-se nestas quatro:

Latência

O tempo que demora para servir a uma requisição. É importante distinguir entre a latência de requisições bem-sucedidas e a latência de requisições com falha. Por exemplo, um erro HTTP 500 disparado em virtude da perda de conexão com um banco de dados ou outro backend crítico pode ser servido rapidamente; contudo, como um erro HTTP 500 indica uma requisição com falha, incluir os 500 em sua latência geral pode resultar em cálculos errôneos. Por outro lado, um erro lento é pior que um erro rápido! Assim, é importante monitorar a latência dos erros, em oposição a apenas filtrá-los.

Tráfego

Uma medida do volume de demanda exigido de seu sistema, calculado segundo uma métrica de alto nível específica do sistema. Para um web service, essa medida geralmente é feita em requisições HTTP por segundo, talvez separada conforme a natureza das requisições (por exemplo, conteúdo estático *versus* dinâmico). Para um sistema de streaming de áudio, essa medida pode estar centrada na taxa de E/S de rede ou em sessões concorrentes. Para um sistema de armazenagem chave-valor, essa medida pode ser feita por meio de transações e leituras por segundo.

Erros

A taxa de requisições que falham, seja de modo explícito (por exemplo, HTTP 500), implícito (por exemplo, uma resposta HTTP 200 de sucesso, porém associada a um conteúdo incorreto) ou de acordo com uma política (por exemplo, “Se você se comprometeu com tempos de resposta de um

segundo, qualquer requisição que demore mais de um segundo será um erro”). Embora os códigos de resposta dos protocolos sejam insuficientes para expressar todas as condições de falha, protocolos secundários (internos) podem ser necessários para monitorar modos parciais de falhas. Monitorar esses casos pode ser extremamente diferente: capturar HTTP 500 em seu distribuidor de carga pode fazer um trabalho decente na identificação de todas as requisições que falharam completamente, enquanto apenas testes fim a fim de sistemas poderão detectar se você está servindo um conteúdo incorreto.

Saturação

Representa o quanto “cheio” está o seu serviço. Uma medida de uma fração de seu sistema, enfatizando os recursos que são mais limitados (por exemplo, em um sistema limitado por memória, mostre a memória; em um sistema limitado por E/S, mostre a E/S). Observe que muitos sistemas têm degradação de desempenho antes de atingirem 100% de utilização; portanto, ter uma meta de utilização é essencial.

Em sistemas complexos, a saturação pode ser suplementada por medidas de carga de nível mais alto: seu serviço é capaz de tratar o dobro do tráfego de forma apropriada, tratar apenas mais 10% de tráfego ou tratar menos tráfego ainda do que está recebendo no momento? Para serviços bem simples, que não tenham nenhum parâmetro para alterar a complexidade da requisição (por exemplo, “Dê-me um nonce³” ou “Preciso de um inteiro monotônico globalmente único”), que raramente mudam de configuração, um valor estático de um teste de carga pode ser adequado. Conforme discutido no parágrafo anterior, porém, a maioria dos serviços precisa usar sinais indiretos, como utilização de CPU ou largura de banda de rede, que tenham um limite superior conhecido. Aumentos de latência frequentemente são um indicador importante de saturação. Medir o tempo de resposta de seu 99º percentil em uma janela pequena (por exemplo, um minuto) pode oferecer um sinal de saturação com bastante antecedência.

Por fim, a saturação também se preocupa com previsões de saturação iminente; por exemplo, “Parece que seu banco de dados lotará seu disco rígido em quatro horas”.

Se você calcular todos os quatro sinais de ouro e acionar um ser humano com um page quando um sinal estiver problemático (ou, no caso da saturação, quase problemático), seu serviço estará minimamente coberto de forma decente pela monitoração.

Preocupando-se com a cauda (ou instrumentação e desempenho)

Quando construímos um sistema de monitoração do zero, é tentador fazer o design de um sistema com base na média de algumas quantidades: a latência média, o uso médio de CPU de seus nós ou a ocupação média de seus bancos de dados. O perigo representado pelos dois últimos casos é evidente: CPUs e bancos de dados podem ser facilmente utilizados de forma bem desequilibrada. O mesmo vale para a latência. Se você executar um web service com uma latência média de 100 ms com 1.000 requisições por segundo, 1% das requisições poderão facilmente demorar 5 segundos.⁴ Se seus usuários dependerem de vários desses web services para renderizar suas páginas, o 99º percentil de um backend pode se tornar facilmente a resposta mediana de seu frontend.

A maneira mais simples de diferenciar entre uma média lenta e uma “cauda” de requisições bem lenta é coletar contadores de requisições e agrupá-los por latência (adequados à criação de um histograma), em vez de coletar as latências propriamente ditas: quantas requisições eu atendi, que demoraram 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 as fronteiras do histograma de modo aproximadamente exponencial (nesse caso, por fatores próximos de 3) com frequência é uma maneira fácil de visualizar a distribuição de suas requisições.

Escolhendo uma resolução apropriada para as medições

Aspectos diferentes de um sistema devem ser mensurados com níveis distintos de granularidade. Por exemplo:

- Observar a carga de CPU pelo período de um minuto não revelará picos de longa duração uniformes que determinam as latências altas da cauda.
- Por outro lado, para um web service cuja meta seja não mais que 9 horas de downtime agregado por ano (99,9% de uptime anual), verificar um status 200 (sucesso) mais de uma ou duas vezes por minuto provavelmente será uma frequência desnecessária.
- De modo semelhante, verificar se o disco rígido está cheio para um serviço cuja meta seja uma disponibilidade de 99,9% mais de uma vez a cada 1 a 2 minutos provavelmente não será necessário.

Tome cuidado com o modo como você estrutura a granularidade de suas medidas. Coletar medidas de carga de CPU por segundo pode produzir dados interessantes, mas medidas com essa frequência podem ser muito caras para coletar, armazenar e analisar. Se sua meta de monitoração exigir uma resolução alta, mas não uma latência extremamente baixa, você poderá reduzir esses custos realizando uma amostragem interna no servidor e, então, configurar um sistema externo para coletar e agragar essa distribuição no tempo ou a partir de diferentes servidores. Você pode:

1. Registrar a utilização atual de CPU a cada segundo.
2. Usando grupos de 5% de granularidade, incrementar o grupo de utilização de CPU apropriado a cada segundo.
3. Fazer a agregação desses valores a cada minuto.

Essa estratégia permite observar altos níveis de atividades na CPU em intervalos breves, sem incorrer em custos muito altos devido à coleta e à retenção de dados.

O mais simples possível, não mais simples que isso

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

- Alertas em diferentes limites de latência, em percentis diferentes, em todos os tipos de métricas distintas.

- Código extra para detectar e expor possíveis causas.
- Painéis de controle associados a cada uma dessas possíveis causas.

As fontes das complexidades em potencial nunca se esgotam. Como todos os sistemas de software, a monitoração pode se tornar muito complexa, a ponto de ser frágil, complicada de mudar e um peso para a manutenção.

Desse modo, faça o design de seu sistema de monitoração com um olho na simplicidade. Ao escolher o que monitorar, tenha em mente as diretrizes a seguir:

- As regras que detectam incidentes reais com muita frequência devem ser tão simples, previsíveis e confiáveis quanto possívels.
- Coleta de dados, agregação e configuração de alertas que são raramente exercitadas (por exemplo, menos de uma vez por trimestre para algumas equipes de SRE) devem estar sujeitas à remoção.
- Sinais coletados, mas não expostos em nenhum painel de controle previamente configurado nem usados por nenhum alerta, são candidatos à remoção.

Na experiência do Google, coleta e agregação básicas de métricas, em conjunto com alertas e painéis de controle, têm funcionado bem como um sistema relativamente independente. (De fato, o sistema de monitoração do Google está dividido em vários binários, mas geralmente as pessoas aprendem sobre todos os aspectos desses binários.) Pode ser tentador combinar a monitoração com outros aspectos da inspeção de sistemas complexos, por exemplo, uma geração detalhada do perfil do sistema, depuração de um único processo, monitoração de detalhes sobre exceções ou falhas, testes de carga, coleta e análise de logs ou inspeção de tráfego. Embora a maioria desses assuntos compartilhe aspectos comuns com a monitoração básica, misturar informações demais resulta em sistemas excessivamente complexos e frágeis. Como ocorre com muitos outros aspectos da engenharia de software, manter sistemas distintos com pontos de integração claros, simples e com baixo acoplamento é uma estratégia melhor (por exemplo, usar APIs web para extrair dados resumidos em um formato que possa permanecer constante por um período de tempo longo).

Juntando esses princípios

Os princípios discutidos neste capítulo podem ser reunidos em uma filosofia de monitoração e geração de alertas que é amplamente endossada e seguida pelas equipes de SRE do Google. Embora essa filosofia de monitoração seja uma espécie de aspiração, é um bom ponto de partida para escrever ou revisar um novo alerta e pode ajudar sua empresa a fazer as perguntas certas, independentemente do tamanho dela ou da complexidade de seu serviço ou sistema.

Ao criar regras para monitoração e alertas, fazer as perguntas a seguir pode ajudar a evitar falsos positivos e um uso excessivo de pager:⁵

- Essa regra detecta uma *condição que, de outra maneira, não seria detectada*, que é urgente, para a qual é possível disparar uma ação e é visível de forma ativa ou iminente pelo usuário?⁶
- Serei capaz de ignorar esse alerta alguma vez, sabendo que é benigno? Quando e por que poderei ignorar esse alerta, e como posso evitar esse cenário?
- Esse alerta definitivamente indica que os usuários estão sendo afetados de modo negativo? Há casos possíveis de ser detectados, em que os usuários não estão sofrendo impactos negativos (por exemplo, quando o tráfego está sendo drenado ou em implantações de testes), que devam ser filtrados?
- Posso disparar uma ação em resposta a esse alerta? Essa ação é urgente ou pode esperar até amanhã de manhã? A ação poderia ser automatizada de modo seguro? Essa ação será uma correção de longo prazo ou apenas uma solução alternativa de curto prazo?
- Outras pessoas estão sendo acionadas por pages por causa desse problema, caso em que pelo menos um dos pages seria desnecessário?

Essas perguntas refletem uma filosofia fundamental sobre pages e pagers:

- Sempre que o pager é acionado, devo ser capaz de reagir com um senso de urgência. Só posso reagir com um senso de urgência algumas vezes ao dia sem ficar cansado.

- Todo page deve ser passível de uma ação.
- Toda resposta a um page deve exigir inteligência. Se um page simplesmente merecer uma resposta robótica, não deveria ser um page.
- Pages devem estar relacionados a um problema novo ou a um evento que ainda não tenha sido visto antes.

Uma perspectiva desse tipo acaba com determinadas distinções: se um page satisfaz os quatro itens anteriores, é irrelevante se o page é acionado por uma monitoração caixa-branca ou caixa-preta. Essa perspectiva também amplifica certas distinções: é melhor investir muito mais esforços capturando os sintomas do que as causas; quando se trata das causas, preocupe-se apenas com as causas muito iminentes e definidas.

Monitoração no longo prazo

Em sistemas de produção modernos, os sistemas de monitoração analisam um sistema sempre em evolução, com uma arquitetura de software, características de carga e metas de desempenho sempre em mudança. Um alerta que, no momento, é excepcionalmente raro e difícil de automatizar pode se tornar frequente, talvez até merecendo um script criado em conjunto para resolvê-lo. A essa altura, alguém deve encontrar e eliminar as causas-raízes do problema; se uma solução desse tipo não for possível, a resposta ao alerta merece ser totalmente automatizada.

É importante que decisões sobre monitoração sejam feitas com metas de longo prazo em mente. Todo page que acontece hoje provoca distrações em um ser humano, evitando que o sistema seja melhorado amanhã, portanto, com frequência, é possível que a disponibilidade ou o desempenho sofram no curto prazo para que a perspectiva de longo prazo melhore para o sistema. Vamos observar dois casos de estudo que mostram essa negociação.

SRE do Bigtable: uma história de excesso de alertas

A infraestrutura interna do Google geralmente é oferecida e avaliada em relação a um objetivo de nível de serviço (SLO, veja o Capítulo 4). Vários anos atrás, o SLO do serviço Bigtable era baseado em um sintético

desempenho médio bem comportado dos clientes. Por causa de problemas no Bigtable e em camadas mais baixas da pilha de armazenagem, o desempenho médio era determinado por uma cauda “larga”: os 5% das piores requisições eram, com frequência, significativamente mais lentos que o restante.

Alertas via email eram disparados quando nos aproximávamos do SLO, e alertas de paging eram acionados quando o SLO era ultrapassado. Os dois tipos de alertas eram disparados em grande número, consumindo um volume de tempo de engenharia inaceitável: a equipe gastava quantidades significativas de tempo fazendo a triagem dos alertas para identificar os poucos sobre os quais era realmente possível tomar uma atitude e, com frequência, deixávamos de perceber os problemas que realmente afetavam os usuários, pois poucos deles o faziam. Muitos dos pages não eram urgentes por causa de problemas bem compreendidos da infraestrutura, e recebiam respostas mecânicas ou nem recebiam respostas.

Para remediar a situação, a equipe usava uma abordagem de três vertentes: ao mesmo tempo em que fazíamos esforços enormes para melhorar o desempenho do Bigtable, também reduzimos temporariamente nossa meta de SLO usando a latência de requisição do 75º percentil. Também desabilitamos os emails de alerta, pois havia tantos deles que gastar tempo diagnosticando-os era inviável.

Essa estratégia nos deu espaço suficiente para respirar e realmente corrigir os problemas de mais longo prazo do Bigtable e das camadas mais baixas da pilha de armazenagem, em vez de corrigir problemas táticos de forma constante. Os engenheiros de plantão realmente conseguiam trabalhar quando não estavam ocupados com pages o tempo todo. Em última instância, afastar-nos temporariamente dos alertas nos permitiu fazer progressos mais rápidos em direção a um serviço melhor.

Gmail: respostas previsíveis de seres humanos, possíveis de estar em um script

Nos primórdios do Gmail, o serviço era baseado em um sistema de gerenciamento de processos distribuído e adaptado chamado Workqueue, originalmente criado para fazer processamento em batch de partes do índice

de pesquisa. O Workqueue havia sido “adaptado” para processos de longa duração e, em seguida, aplicado ao Gmail, porém certos bugs na base de código relativamente opaca do escalonador se mostraram difíceis de ser vencidos.

Naquela época, a monitoração do Gmail estava estruturada de modo que alertas eram disparados quando tarefas individuais eram “desescalonadas” pelo Workqueue. Essa configuração não era ideal, pois, mesmo naquela época, o Gmail tinha muitos e muitos milhares de tarefas, cada qual representando uma fração de um percentual de nossos usuários. Para nós, era muito importante oferecer uma boa experiência aos usuários do Gmail, mas era impossível dar manutenção em uma configuração de alertas desse tipo.

Para resolver esse problema, a equipe de SRE do Gmail construiu uma ferramenta que ajudava a “cutucar” o escalonador do modo correto para minimizar os impactos aos usuários. A equipe fez diversas discussões para saber se deveríamos ou não simplesmente automatizar o circuito todo, desde a detecção do problema até acionar o reescalonador, até que uma solução melhor de longo prazo fosse implementada, mas algumas pessoas achavam que esse tipo de solução alternativa atrasaria uma correção de verdade.

Esse tipo de tensão é comum em uma equipe e, muitas vezes, reflete uma falta de confiança subjacente em sua autodisciplina: enquanto alguns membros da equipe querem implementar um “hack” para ganhar tempo, a fim de fazer uma correção apropriada, outros se preocupam com a possibilidade de um hack ser esquecido ou, ainda, que a correção apropriada perca indefinidamente a prioridade. Essa preocupação faz sentido, pois é fácil construir camadas de débitos técnicos impossíveis de manter fazendo remendos sobre os problemas em vez de implementar correções verdadeiras. Os gerentes e os líderes técnicos desempenham um papel fundamental na implementação de correções verdadeiras, de longo prazo, ao darem suporte e prioridade às correções de longo prazo que poderão consumir tempo, mesmo quando o “sofrimento” inicial do paging diminuir.

Pages com respostas mecânicas e algorítmicas devem ser um sinal vermelho. A falta de disposição de sua equipe em automatizar esses pages implica que ela não tem confiança de que pode limpar seus débitos técnicos. Esse é um

problema importante, que vale a pena ser escalado.

O longo prazo

Um tema comum conecta os exemplos anteriores com o Bigtable e o Gmail: uma tensão entre a disponibilidade de curto prazo e a disponibilidade de longo prazo. Com frequência, um esforço verdadeiro pode ajudar um sistema precário a atingir uma alta disponibilidade, mas esse caminho geralmente tem vida curta e está repleto de estresse e de dependência de um pequeno número de membros heroicos da equipe. Assumir uma diminuição controlada e de curto prazo na disponibilidade muitas vezes é uma negociação difícil, porém estratégica para a estabilidade do sistema no longo prazo. É importante não pensar em todo page como um evento isolado, mas considerar se o *nível* geral de paging conduz a um sistema saudável, com disponibilidade apropriada e uma equipe saudável e viável, com perspectiva de longo prazo. Analisamos estatísticas sobre frequência de pages (normalmente expressas como incidentes por turno, em que um incidente pode ser composto de alguns pages relacionados) em relatórios trimestrais com a gerência, garantindo que os responsáveis pelas decisões acompanhem os dados sobre a carga de pages e a saúde de suas equipes em geral.

Conclusão

Um fluxo saudável de monitoração e alertas é simples e fácil de compreender. Ele tem como foco principal os sintomas para paging, reservando dados heurísticos orientados a causas como auxílios para problemas de depuração. Monitorar sintomas é mais fácil quanto mais “para cima” você empilhar o seu monitor, embora a monitoração de saturação e de desempenho de subsistemas como bancos de dados muitas vezes deva ser feita diretamente no próprio subsistema. Alertas via email têm valor bastante limitado e tendem a ficar rapidamente sobrecarregados de ruído; em vez disso, você deve favorecer um painel de controle que monitore todos os problemas não críticos no momento em busca do tipo de informação que normalmente acaba resultando em alertas via email. Um painel também pode ser combinado com um log para a análise de correlações históricas.

No longo prazo, conseguir um rodízio de plantões e um produto bem-sucedidos inclui optar por gerar alertas para sintomas ou problemas reais iminentes, adaptando suas metas com valores que sejam realmente atingíveis e garantindo que sua monitoração seja capaz de oferecer diagnósticos rápidos.

¹ Às vezes são chamados de “alerta spam”, pois raramente são lidos ou disparam uma ação.

² N.T.: *Analytics* é a descoberta, interpretação e informação de padrões significativos nos dados.
(Baseado em <https://en.wikipedia.org/wiki/Analytics>.)

³ N.T.: Em criptografia, um *nonce* é um número arbitrário que pode ser usado apenas uma vez.
(Baseado em https://en.wikipedia.org/wiki/Cryptographic_nonce.)

⁴ Se 1% de suas requisições demorarem 10x mais que a média, isso significa que o restante de suas requisições será aproximadamente duas vezes mais rápido que a média. Porém, se você não estiver analisando sua distribuição, a ideia de que a maior parte de suas requisições está próxima da média é apenas um pensamento esperançoso.

⁵ Veja *Applying Cardiac Alarm Management Techniques to Your On-Call* (Aplicando técnicas de gerenciamento de alarmes cardíacos em seu plantão) [Hol14], que apresenta um exemplo de “fadiga de alertas” em outro contexto.

⁶ Situações com redundância zero ($N + 0$) contam como iminentes, assim como partes “quase cheias” de seu serviço! Para ver mais detalhes sobre o conceito de redundância, acesse https://en.wikipedia.org/wiki/N%2B1_redundancy.

CAPÍTULO 7

A evolução da automação no Google

Escrito por Niall Murphy com John Looney e Michael Kacirek

Editado por Betsy Beyer

Além da magia negra, há apenas automação e mecanização.

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

Para a SRE, a automação é um multiplicador de forças, e não uma panaceia. É claro que simplesmente *multiplicar* forças não muda naturalmente a precisão do ponto em que essa força é aplicada: automatizar sem planejar pode criar tantos problemas quantos os que são resolvidos. Desse modo, embora acreditemos que uma automação baseada em software seja superior a uma operação manual na maioria das circunstâncias, melhor que as duas opções é um design de sistema de mais alto nível que não exija nenhuma delas — um sistema *autônomo*. Ou, falando de outra maneira, o valor da automação resulta tanto do que ela faz quanto de sua aplicação criteriosa. Discutiremos tanto o valor da automação quanto o modo como nossa atitude evoluiu com o tempo.

O valor da automação

Qual é exatamente o valor da automação?¹

Consistência

Embora a escala seja uma motivação óbvia para a automação, há vários outros motivos para usá-la. Tome o exemplo dos sistemas de computação das universidades, em que muitos engenheiros de sistemas iniciaram suas carreiras. Os administradores de sistemas com essa experiência anterior, em geral, tinham como responsabilidade operar um conjunto de máquinas com

algum software e estavam acostumados a executar manualmente várias ações no cumprimento dessas obrigações. Um exemplo comum está na criação de contas de usuários; outros incluem obrigações puramente operacionais, como garantir que os backups ocorram, administrar failovers² de servidores e fazer pequenas manipulações de dados, por exemplo, mudar os *resolv.conf* dos servidores DNS de upstream, dados de zona de servidores DNS e atividades semelhantes. Em última instância, porém, essa predominância de tarefas manuais não é satisfatória nem para as empresas e, na verdade, nem para as pessoas que fazem a manutenção dos sistemas dessa maneira. Para começar, qualquer ação realizada por um ou vários seres humanos centenas de vezes não será sempre executada do mesmo modo: mesmo com a melhor das intenções, poucos de nós serão tão consistentes quanto uma máquina. Essa falta de consistência inevitável resulta em erros, lapsos, problemas com a qualidade dos dados e, sim, problemas de confiabilidade. Nesse domínio – execução de procedimentos conhecidos, com escopos bem definidos –, o valor da consistência, em vários aspectos, é o mais importante na automação.

Uma plataforma

A automação não oferece apenas consistência. Sistemas automáticos, se projetados e criados de modo apropriado, também oferecem uma *plataforma* que pode ser estendida, aplicada a outros sistemas ou, quem sabe, até mesmo lançada com vistas ao lucro.³ (A alternativa, isto é, a ausência de automação, não é eficaz quanto ao custo nem extensível: em vez disso, representa uma taxa adicional cobrada na operação de um sistema.)

Uma plataforma também *centraliza erros*. Em outras palavras, um bug corrigido no código será corrigido aí definitivamente, de modo diferente de um grande conjunto de pessoas realizando o mesmo procedimento, conforme discutimos antes. Uma plataforma pode ser estendida para realizar tarefas adicionais de modo mais fácil do que instruir seres humanos a executá-las (ou, às vezes, até mesmo perceberem que elas precisam ser feitas). Conforme a natureza da tarefa, ela pode ser executada continuamente ou com muito mais frequência do que os seres humanos poderiam executá-la de forma apropriada, ou em momentos que sejam inconvenientes às pessoas. Além do mais, uma plataforma é capaz de exportar métricas sobre seu desempenho ou

ainda permitir que sejam descobertos detalhes sobre seu processo que você não conhecia antes, pois esses detalhes são mais facilmente mensuráveis no contexto de uma plataforma.

Correções mais rápidas

Há uma vantagem adicional em sistemas em que a automação é usada para resolver falhas comuns (uma situação frequente na automação criada por SREs). Se a automação executar regularmente e com sucesso suficiente, o resultado será um MTTR (Mean Time To Repair, ou Tempo Médio para Correção) reduzido para essas falhas comuns. Você poderá, então, investir seu tempo em outras tarefas, alcançando assim mais velocidade de desenvolvimento, pois não precisará gastar tempo investigando um problema ou (de modo mais comum) fazendo uma limpeza depois dele.

Como é de amplo conhecimento no mercado, quanto mais tarde no ciclo de vida do produto um problema é descoberto, mais cara será a correção; veja o Capítulo 17. Em geral, problemas que ocorrem no ambiente de produção são mais caros para corrigir, tanto em termos de tempo quanto de dinheiro, o que significa que um sistema automatizado que procure identificar problemas assim que eles surgirem tem boas chances de reduzir o custo total do sistema, desde que esse sistema seja suficientemente grande.

Ação mais rápida

Nas situações de infraestrutura em que a automação de SRE tende a ser implantada, os seres humanos geralmente não reagem tão rapidamente quanto as máquinas. Na maioria dos casos mais comuns, em que, por exemplo, um failover ou um desvio de tráfego podem ser bem definidos para uma aplicação em particular, não faz sentido exigir que um ser humano ocasionalmente pressione um botão chamado “Permitir que o sistema continue executando”. (Sim, é verdade que, às vezes, procedimentos automáticos podem acabar piorando uma situação já ruim, mas é por isso que esses procedimentos devem ter domínios bem definidos como escopo.) O Google tem um grande volume de automação; em muitos casos, os serviços aos quais damos suporte não poderiam sobreviver por muito tempo sem essa

automação, pois já cruzaram a fronteira da operação administrável manualmente há muito tempo.

Economia de tempo

Por fim, a economia de tempo é muito citada a favor da automação. Embora esse argumento costume ser o mais utilizado, em muitos aspectos, o benefício frequentemente é menos calculável de forma imediata. Os engenheiros muitas vezes têm dúvidas se vale a pena fazer uma automação ou escrever um código em particular, no que diz respeito aos esforços economizados por não exigir que uma tarefa seja executada manualmente *versus* o esforço exigido para escrever o código.⁴ É fácil esquecer-se do fato de que, uma vez que você tenha encapsulado alguma tarefa na automação, qualquer pessoa poderá executá-la. Desse modo, as economias de tempo se aplicam a todos que poderiam usar a automação de forma plausível. Desacoplar o operador da operação é muito eficaz.



Joseph Bironas, um SRE responsável pelos esforços de ativação de datacenters no Google por um tempo, argumentava, de forma veemente, que:

“Se estamos aplicando a engenharia em processos e soluções que não sejam automatizáveis, continuaremos precisando contratar pessoas para dar manutenção no sistema. Se precisarmos contratar pessoas para fazer o trabalho, estaremos alimentando as máquinas com sangue, suor e lágrimas de seres humanos. Pense em *Matrix* com menos efeitos especiais e mais Administradores de Sistemas irritados.”

O valor da SRE no Google

Todos esses benefícios e negociações se aplicam tanto a nós quanto a qualquer outra pessoa, e o Google tem uma forte tendência à automação. Parte de nossa preferência pela automação nasce de nossos desafios particulares de negócio: os produtos e os serviços de que cuidamos estão espalhados em escala mundial e, geralmente, não temos tempo para nos envolvermos no mesmo tipo de tarefas manuais com máquinas e serviços comuns em outras empresas.⁵ Para serviços realmente grandes, os fatores associados a consistência, rapidez e confiabilidade dominam a maior parte das conversações sobre os pontos negociados ao implementar uma

automação.

Outro argumento em favor da automação, particularmente no caso do Google, é nosso ambiente de produção complicado, porém surpreendentemente uniforme, descrito no Capítulo 2. Enquanto outras empresas podem ter um único equipamento importante sem uma API prontamente acessível, um software para o qual nenhum código-fonte esteja disponível ou outro impedimento para um controle total das operações de produção, o Google em geral evita esses cenários. Criamos APIs para sistemas quando não havia nenhuma API disponível do fornecedor. Mesmo que comprar um software para uma tarefa em particular fosse muito mais barato no curto prazo, optamos por escrever nossas próprias soluções, pois fazer isso gerava APIs com o potencial para ter vantagens muito maiores no longo prazo. Investimos bastante tempo para superar obstáculos ao gerenciamento de sistemas automático e, então, de forma resoluta, desenvolvemos esse gerenciamento. Considerando como o Google administra seus códigos-fontes [Pot16], a disponibilidade desses códigos para quase todos os sistemas com os quais os SREs têm contato também significa que nossa missão de “ser dono do produto na produção” é bem mais fácil porque controlamos a totalidade da pilha.

É claro que, embora o Google, ideologicamente, esteja 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 habilidade ou a inclinação para desenvolver automações em um determinado momento. Alguns sistemas essenciais começaram como protótipos rápidos, e não haviam sido projetados para durar nem fazer interface com a automação. Os parágrafos anteriores apresentaram uma visão radical de nossa posição, mas temos tido muito sucesso em colocá-la em ação no contexto do Google. Em geral, temos optado por criar plataformas quando possível ou nos *posicionar* de modo que possamos criar plataformas com o passar do tempo. Vemos essa abordagem baseada em plataforma como necessária para possibilitar a manutenção e a escalabilidade.

Os casos de uso para automação

No mercado, *automação* é o termo geralmente usado para escrever um código a fim de solucionar uma grande variedade de problemas, embora as motivações para escrever esse código e as próprias soluções muitas vezes sejam bem diferentes. De modo mais amplo, de acordo com essa visão, a automação é um “metasoftware” – um software que atua sobre um software.

Como mencionado anteriormente, há vários casos de uso para a automação. Eis uma lista não exaustiva de exemplos:

- Criação de contas de usuários
- Ativação e desativação de clusters para serviços
- Preparação para instalação e desativação de softwares ou hardwares
- Rollouts de novas versões de software
- Mudanças de configuração em tempo de execução
- Um caso especial de mudanças de configuração em tempo de execução: mudanças em suas dependências

Essa lista poderia se estender essencialmente *ad infinitum*.

Casos de uso para automação pela SRE do Google

No Google temos todos os casos de uso que acabamos de listar, além de outros.

No entanto, na SRE do Google, nossa afinidade principal tem sido normalmente com a infraestrutura em execução, em oposição a administrar a qualidade dos dados que passam por essa infraestrutura. Essa linha não está totalmente clara – por exemplo, nós nos preocuparemos profundamente se metade de um conjunto de dados desaparecer após uma atualização de versão, por isso geramos um alerta sobre diferenças significativas como essa, mas é raro escrevermos o equivalente a alterar as propriedades de algum subconjunto arbitrário de contas em um sistema. Desse modo, o contexto de nossa automação com frequência é aquele que visa a administrar o ciclo de vida dos sistemas, e não os seus dados: por exemplo, implantações de um serviço em um novo cluster.

Para tanto, os esforços de automação dos SREs não estão muito distantes do que muitas outras pessoas e empresas fazem, exceto que usamos ferramentas diferentes para administrá-la e temos um foco diferente (conforme será discutido).

Ferramentas amplamente disponíveis, como Puppet, Chef, cfengine e até mesmo Perl, que oferecem maneiras de automatizar determinadas tarefas, diferem principalmente em termos do nível de abstração dos componentes disponibilizados para ajudar no ato da automação. Uma linguagem completa como Perl disponibiliza recursos no nível de POSIX que, em teoria, oferecem um escopo essencialmente ilimitado de automação às APIs acessíveis ao sistema⁶, enquanto o Chef e o Puppet oferecem abstrações prontas para uso com as quais os serviços ou outras entidades de mais alto nível podem ser manipulados. A negociação aqui é clássica: abstração de mais alto nível é mais fácil de administrar e compreender, mas diante de uma “leaky abstraction”⁷ você falhará de forma sistemática, repetitiva e potencialmente inconsistente. Por exemplo, com frequência supomos que fazer uma atualização em um cluster com um novo binário é uma operação atômica; o cluster acabará com a versão antiga ou com a nova versão. No entanto, o comportamento no mundo real é mais complicado: a rede desse cluster pode falhar no meio do processo, máquinas podem falhar, a comunicação com a camada de gerenciamento do cluster pode falhar, deixando o sistema em um estado inconsistente; novos binários, dependendo da situação, poderiam ser disponibilizados para testes, mas não instalados, ou instalados mas não reiniciados, ou reiniciados mas não verificados. Pouquíssimas abstrações modelam esses tipos de resultados com sucesso e, de modo geral, acabam se interrompendo e exigindo intervenção. Sistemas realmente ruins de automação não fazem nem mesmo isso.

A SRE tem algumas filosofias e produtos no domínio da automação, alguns dos quais se parecem mais com ferramentas genéricas de rollout, sem uma modelagem particularmente detalhada das entidades de mais alto nível, enquanto outros se parecem mais com linguagens para descrever a implantação do serviço (e assim por diante) em um nível bastante abstrato. Nesse último caso, o trabalho efetuado tende a ser mais reutilizável e se assemelhar mais a uma plataforma comum se comparado ao primeiro caso,

mas a complexidade de nosso ambiente de produção às vezes implica que a primeira abordagem é a opção mais administrável de imediato.

Uma hierarquia de classes de automação

Embora todos esses passos de automação sejam importantes, e uma plataforma de automação seja realmente valiosa por si só, em um mundo ideal, não precisaríamos de automação exposta externamente. De fato, em vez de ter um sistema que precise *ter* uma lógica de colagem externa, seria melhor ainda se tivéssemos um sistema que não precisasse de *nenhuma lógica de colagem*, não só porque a internalização é mais eficiente (embora essa eficiência seja útil), mas também porque foi projetado para não precisar de lógica de colagem, antes de tudo. Fazer isso envolve tomar os casos de uso para lógica de colagem – em geral, manipulações de “primeira ordem” de um sistema, como adicionar contas ou realizar a ativação de um sistema – e encontrar uma maneira de lidar com esses casos de uso diretamente da aplicação.

Como exemplo mais detalhado, a maior parte da automação de ativação no Google é problemática porque acaba sendo mantida em separado do sistema principal e, desse modo, sofre de “bit rot” (degradação de software), isto é, não muda quando o sistema subjacente é alterado. Apesar das melhores intenções, tentar acoplar os dois mais fortemente (automação de ativação e o sistema principal) muitas vezes falha por causa de prioridades não alinhadas, pois os desenvolvedores de produto resistirão (não sem razão) a um requisito de teste de implantação para toda mudança. Em segundo lugar, uma automação crucial, porém executada somente a intervalos longos e, desse modo, difícil de testar, muitas vezes é particularmente frágil por causa do ciclo longo de feedback. Failover de cluster é um exemplo clássico de automação executada raramente: os failovers podem ocorrer somente a intervalos de alguns meses ou com uma frequência bem baixa, a ponto de inconsistências entre as instâncias serem introduzidas. A evolução da automação segue um caminho:

1) *Sem automação*

O failover no banco de dados mestre (master) é feito manualmente entre

localidades.

2) Automação específica a um sistema, mantida externamente

Um SRE tem um script de failover em seu diretório home.

3) Automação genérica mantida externamente

O SRE acrescenta suporte a banco de dados a um script de “failover genérico” que todos usam.

4) Automação específica a um sistema, mantida internamente

O banco de dados já vem com seu próprio script de failover.

5) Sistemas que não precisam de nenhuma automação

O banco de dados percebe os problemas e faz um failover automaticamente, sem intervenção humana.

A SRE detesta operações manuais, portanto, obviamente, tentamos criar sistemas que não as exijam. Às vezes, porém, as operações manuais são inevitáveis.

Além disso, há uma subvariedade de automação que aplica mudanças não em todo o domínio da configuração relacionada a um sistema específico, mas no domínio do ambiente de produção como um todo. Em um ambiente de produção proprietário altamente centralizado como o do Google, há um grande número de mudanças que têm um escopo não específico a um serviço – por exemplo, mudar os servidores de upstream do Chubby, uma mudança de flag na biblioteca cliente do Bigtable para deixar o acesso mais confiável, e assim por diante –; apesar disso, essas mudanças precisam ser administradas de modo seguro, e são passíveis de rollback, se for necessário. Acima de certo volume de alterações, será impraticável que mudanças que se estendam a todo o ambiente de produção sejam feitas manualmente e, em algum ponto antes desse instante, será um desperdício monitorar um processo manualmente, no qual uma grande parcela das mudanças seja trivial ou feita com sucesso por estratégias básicas de reinício e verificação.

Vamos usar estudos de casos internos para ilustrar alguns dos pontos anteriores com detalhes. O primeiro estudo de caso tem a ver com o modo como, por meio de um trabalho zeloso, visando ao longo prazo, conseguimos

alcançar o autoprofessado nirvana da SRE: automatizar a nós mesmos para deixar de executar uma tarefa.

Automatizar a você mesmo para deixar de executar uma tarefa: automatize TUDO!

Durante muito tempo, os produtos Ads do Google armazenavam seus dados em um banco de dados MySQL. Como os dados do Ads obviamente têm requisitos de alta confiabilidade, uma equipe de SRE tinha como responsabilidade cuidar de sua infraestrutura. De 2005 a 2008, o Ads Database executava, em sua maior parte, no que considerávamos ser um estado maduro e administrado. Por exemplo, tínhamos automatizado as piores tarefas rotineiras, mas nem todas, para substituições de réplicas padrões. Acreditávamos que o Ads Database era bem administrado e que tínhamos colhido a maior parte dos frutos ao alcance das mãos no que diz respeito a otimização e escala. Contudo, à medida que as operações diárias se tornaram confortáveis, os membros da equipe começaram a olhar para o próximo nível do desenvolvimento do sistema: migrar o MySQL para o sistema de escalonamento de cluster do Google, o Borg.

Esperávamos que essa migração trouxesse duas vantagens principais:

- Eliminaria *totalmente* a manutenção de máquina/réplica: o Borg trataria automaticamente a criação/reinicialização de tarefas novas e com falhas.
- Permitiria acomodar várias instâncias de MySQL no mesmo computador físico: o Borg possibilitaria um uso mais eficiente dos recursos da máquina por meio de Containers.

No final de 2008, implantamos com sucesso uma instância do MySQL no Borg como prova de conceito. Infelizmente, isso foi acompanhado de uma nova dificuldade significativa. Uma característica essencial de operação do Borg é que suas tarefas se movem automaticamente. As tarefas são comumente transferidas dentro do Borg de modo muito frequente – uma ou duas vezes por semana. Essa frequência era tolerável para nossas réplicas de banco de dados, mas inaceitável para nossos mestres (masters).

Naquela época, o processo para failover do mestre demorava entre 30 a 90

minutos por instância. Simplesmente porque executávamos em máquinas compartilhadas e estávamos sujeitos a reinicializações para upgrades de kernel, além da taxa normal de falhas de máquinas, tínhamos que esperar vários failovers não relacionados ao nosso caso toda semana. Esse fator, em conjunto com o número de partes em que nosso sistema estava hospedado, implicava que:

- Failovers manuais consumiriam um tempo substancial das pessoas e nos dariam uma disponibilidade, no melhor caso, de 99% de uptime, o que estava aquém dos requisitos de negócios do produto.
- Para atender a nossas provisões para erros, cada failover deveria utilizar menos de 30 segundos de downtime. Não havia jeito de otimizar um procedimento dependente de seres humanos para reduzir o downtime para menos de 30 segundos.

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

Em 2009, a SRE do Ads concluiu nosso daemon automatizado de failover, que batizamos de “Decider”. O Decider podia concluir failovers de MySQL para failovers planejados ou não em menos de 30 segundos em 95% das vezes. Com a criação do Decider, o MySQL no Borg (MoB) finalmente se tornou uma realidade. Passamos de otimizar nossa infraestrutura a fim de reduzir os failovers para abraçar a ideia de que a falha é inevitável e, desse modo, otimizar para ter uma recuperação rápida por meio de automação.

Embora a automação nos tenha permitido ter um MySQL altamente disponível em um mundo que nos迫使ava a ter até duas reinicializações por semana, essa solução veio acompanhada de seu próprio conjunto de custos. Todas as nossas aplicações tiveram que ser alteradas para incluir muito mais lógica para tratamento de falhas do que havia antes. Considerando que a norma no mundo de desenvolvimento de MySQL seja supor que a instância de MySQL é o componente mais estável da pilha, essa mudança implicou em personalizar softwares como o JDBC para que fossem mais tolerantes ao nosso ambiente suscetível a falhas. No entanto, as vantagens de migrar para o MoB com o Decider compensaram de longe esses custos. Uma vez no MoB, o tempo que nossa equipe gastava em tarefas operacionais mundanas caiu em

95%. Nossos failovers foram automatizados, portanto uma interrupção de uma única tarefa de banco de dados deixou de acionar o pager de um ser humano.

O principal ganho dessa nova automação foi termos muito mais tempo livre para investir em melhorar outras partes da infraestrutura. Essas melhorias tiveram um efeito em cascata: quanto mais tempo economizávamos, mais tempo podíamos gastar em otimizar e automatizar outras tarefas tediosas. Em algum momento, pudemos automatizar mudanças de esquemas, fazendo o custo de toda a manutenção operacional do Ads Database cair em aproximadamente 95%. Algumas pessoas podem dizer que nos automatizamos a nós mesmos com sucesso deixando de executar a tarefa. O lado do hardware de nosso domínio também conheceu melhorias. Migrar para o MoB liberou recursos consideráveis, pois podíamos escalar várias instâncias de MySQL nas mesmas máquinas, o que melhorou a utilização de nosso hardware. No total, conseguimos liberar aproximadamente 60% de nosso hardware. Nossa equipe agora tinha recursos de hardware e de engenharia em abundância.

Esse exemplo mostra a sabedoria de percorrer um quilômetro a mais a fim de oferecer uma plataforma em vez de substituir procedimentos manuais existentes. O próximo exemplo é proveniente do grupo de infraestrutura de clusters e mostra algumas das negociações mais difíceis que você poderá encontrar em seu caminho para automatizar *tudo*.

Reduzindo o sofrimento: aplicando a automação em ativação de clusters

Dez anos atrás, a equipe de SRE da Infraestrutura de Clusters parecia contratar um novo funcionário a cada intervalo de alguns meses. O fato é que essa era aproximadamente a mesma frequência com que ativávamos um novo cluster. Como ativar um serviço em um novo cluster expõe funcionários recém-contratados ao funcionamento interno de um serviço, essa tarefa parecia ser uma ferramenta de treinamento útil e natural.

Os passos seguidos para deixar um cluster pronto para uso eram semelhantes

a estes:

1. Preparar um prédio de datacenter com energia e refrigeração.
2. Instalar e configurar switches principais e conexões com o backbone.
3. Instalar alguns racks iniciais de servidores.
4. Configurar serviços básicos como DNS e instaladores, e então configurar um serviço de lock, de armazenagem e de processamento.
5. Implantar os racks de máquinas restantes.
6. Atribuir recursos aos serviços voltados aos usuários para que suas equipes possam configurar esses serviços.

Os passos 4 e 6 são extremamente complexos. Embora os serviços básicos como o DNS sejam relativamente simples, os subsistemas de armazenagem e processamento naquela época ainda passavam por intenso desenvolvimento e, assim, novas flags, componentes e otimizações eram adicionados semanalmente.

Alguns serviços tinham mais de cem subsistemas componentes diferentes, cada um com uma rede complexa de dependências. Falhar em configurar um subsistema, ou configurar um sistema ou componente de modo diferente de outras implantações, representa uma interrupção no serviço que impactará o cliente, esperando para acontecer.

Em um caso, um cluster de vários petabytes do Bigtable havia sido configurado para não usar o primeiro disco (logging) em sistemas de 12 discos, por questões de latência. Um ano depois, alguma automação supôs que se o primeiro disco de uma máquina não estivesse sendo usado, essa máquina não possuía nenhum sistema de armazenagem configurado; desse modo, seria seguro limpar a máquina e configurá-la do zero. Todos os dados do Bigtable foram limpos instantaneamente. Felizmente, tínhamos várias réplicas de tempo real do conjunto de dados, mas essas surpresas não são bem-vindas. A automação precisa ser cuidadosa ao depender de sinais de “segurança” implícitos.

As primeiras automações tinham como foco acelerar a disponibilização dos clusters. Essa abordagem tendia a contar com o uso criativo de SSH para problemas de distribuição lenta de pacotes e de inicialização de serviços. Essa

estratégia representou uma vitória inicial, mas esses scripts de formato livre se transformaram em colesterol para o débito técnico.

Detectando inconsistências com o Prodtest

À medida que o número de clusters aumentou, alguns deles passaram a exigir flags e configurações ajustadas manualmente. Como resultado, as equipes perdiam mais e mais tempo tentando identificar erros de configuração difíceis de localizar. Se uma flag que deixava o GFS mais responsivo a processamento de logs se propagasse até os templates defaults, as células com muitos arquivos poderiam ficar sem memória em situação de carga. Erros de configuração irritantes, que consumiam muito tempo, começaram a surgir praticamente em toda grande mudança de configuração.

Os shell scripts criativos – embora frágeis – que usávamos para configurar clusters não eram escaláveis para o número de pessoas que queriam fazer alterações nem para a enorme quantidade de substituições de clusters que precisavam ser feitas. Esses shell scripts também não eram capazes de resolver problemas mais significativos antes de declarar que um serviço estava adequado para tratar o tráfego voltado aos clientes, como:

- Todas as dependências do serviço estavam disponíveis e corretamente configuradas?
- Todas as configurações e pacotes estavam consistentes com outras implantações?
- A equipe poderia confirmar que toda exceção à configuração era desejada?

O Prodtest (Production Test) foi uma solução engenhosa para essas surpresas indesejadas. Estendemos o framework de testes de unidade de Python para que fosse possível fazer testes de unidade de serviços do mundo real. Esses testes de unidade têm dependências, permitindo que haja uma cadeia de testes, e uma falha em um teste interromperia rapidamente essa cadeia. Tome o teste mostrado na Figura 7.1 como exemplo.

O Prodtest de uma dada equipe recebia o nome do cluster e podia validar os serviços dessa equipe nesse cluster. Acréscimos posteriores nos permitiram

gerar um grafo dos testes de unidade e seus estados. Essa funcionalidade permitia a um engenheiro ver rapidamente se seu serviço estava corretamente configurado em todos os clusters e, se não estivesse, por quê. O grafo destacava o passo com falha e o teste de unidade Python com falha apresentava uma mensagem de erro mais extensa.

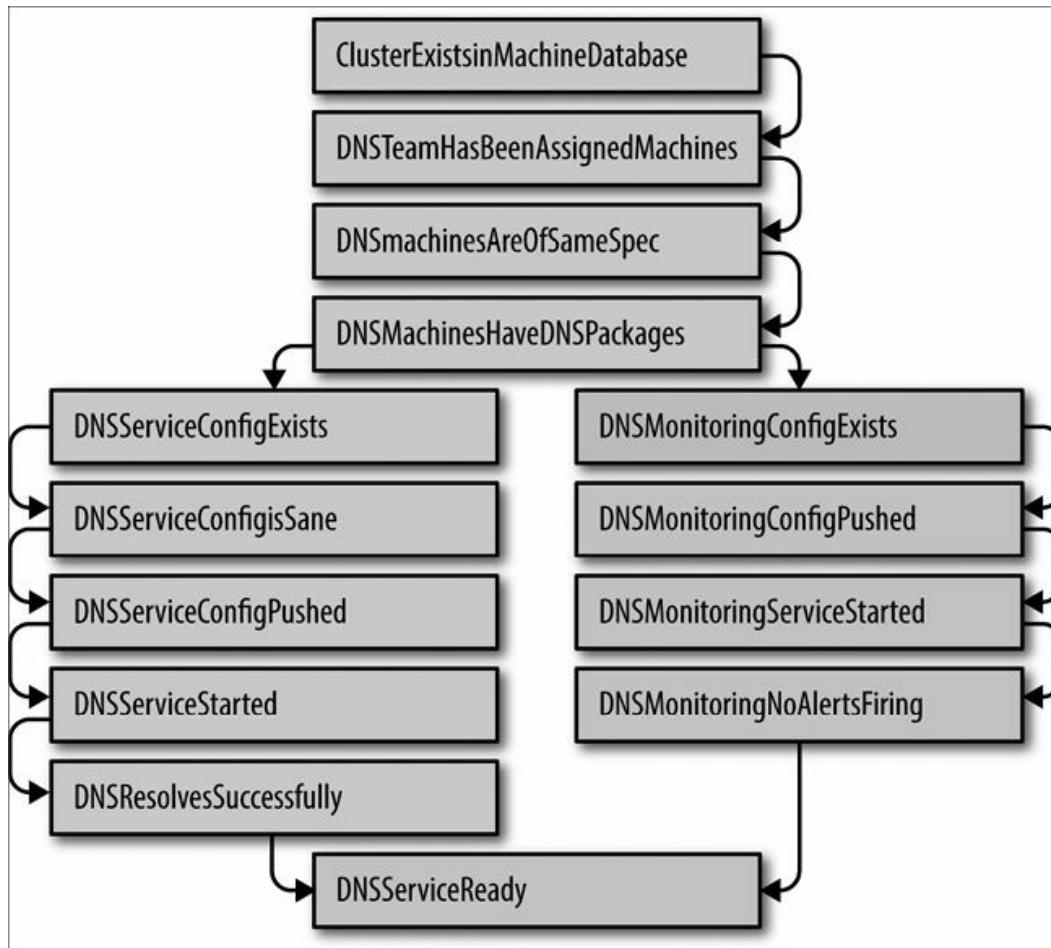


Figura 7.1 – ProdTest para o Seviço DNS, mostrando como um teste com falha interrompe a cadeia subsequente de testes.

Sempre que uma equipe se deparava com um atraso devido a um erro de configuração não esperado de outra equipe, um bug poderia ser registrado para estender seu Prodtest. Isso garantia que um problema semelhante seria descoberto com antecedência no futuro. Os SREs estavam orgulhosos de serem capazes de garantir aos seus clientes que todos os serviços – tanto serviços recém-ativados quanto serviços existentes com uma nova configuração – atenderiam ao tráfego de produção de forma confiável.

Pela primeira vez, nossos gerentes de projeto eram capazes prever quando um cluster poderia “ser ativado” e podiam entender completamente *por que* cada cluster demorava seis ou mais semanas para ir de “rede pronta” para “servindo a um tráfego ativo”. De repente, a SRE recebeu uma missão da gerência sênior: *em três meses, cinco novos clusters estarão com a rede pronta no mesmo dia. Por favor, ative-os em uma semana.*

Resolvendo inconsistências de forma idempotente

Uma “Ativação em uma semana” era uma missão assustadora. Tínhamos dezenas de milhares de linhas de shell script pertencentes a dezenas de equipes. Poderíamos rapidamente dizer quão despreparado estava qualquer dado cluster, mas corrigi-lo significava que dezenas de equipes deveriam registrar centenas de bugs e, então, tínhamos que esperar que esses bugs fossem corrigidos imediatamente.

Percebemos que evoluir de “Testes de unidade Python identificando erros de configuração” para “código Python corrigindo erros de configuração” nos permitiria corrigir esses erros de modo mais rápido.

O teste de unidade já sabia qual cluster estávamos analisando e o teste específico em falha, portanto combinamos cada teste com uma correção. Se cada correção fosse escrita para ser idempotente⁸ e pudesse supor que todas as dependências foram atendidas, resolver o problema seria fácil – e seguro. Exigir correções idempotentes significava que as equipes poderiam executar seu “script de correção” a cada 15 minutos sem temer que houvesse danos à configuração do cluster. Se o teste da equipe de DNS estivesse bloqueado na configuração da equipe de Machine Database de um novo cluster, assim que o cluster aparecesse no banco de dados, os testes e as correções da equipe de DNS começariam a funcionar.

Tome o teste mostrado na Figura 7.2 como exemplo. Se `TestDnsMonitoringConfigExists` falhar, conforme mostrado, podemos chamar `FixDnsMonitoringCreateConfig`, que extrai a configuração de um banco de dados e então faz check in de um esqueleto de arquivo de configuração em nosso sistema de controle de versões. Em seguida, `TestDnsMonitoringConfigExists` passa na próxima tentativa e o teste

TestDnsMonitoringConfigPushed pode ser feito. Se o teste falhar, o passo FixDnsMonitoringPushConfig é executado. Se uma correção falhar várias vezes, a automação supõe que a correção falhou e para, notificando o usuário.

De posse desses scripts, um pequeno grupo de engenheiros foi capaz de garantir que pudéssemos passar de “A rede funciona, e as máquinas estão listadas no banco de dados” para “Servindo a 1% do tráfego de websearch e de ads” em questão de uma ou duas semanas. Naquela época, isso parecia ser o ápice da tecnologia de automação.

Olhando para trás, essa abordagem tinha falhas profundas; a latência entre o teste, a correção e então um segundo teste introduzia testes *frágeis* que às vezes funcionavam e em outras ocasiões falhavam. Nem todas as correções eram naturalmente idempotentes; portanto, um teste frágil seguido de uma correção poderia deixar o sistema em um estado inconsistente.

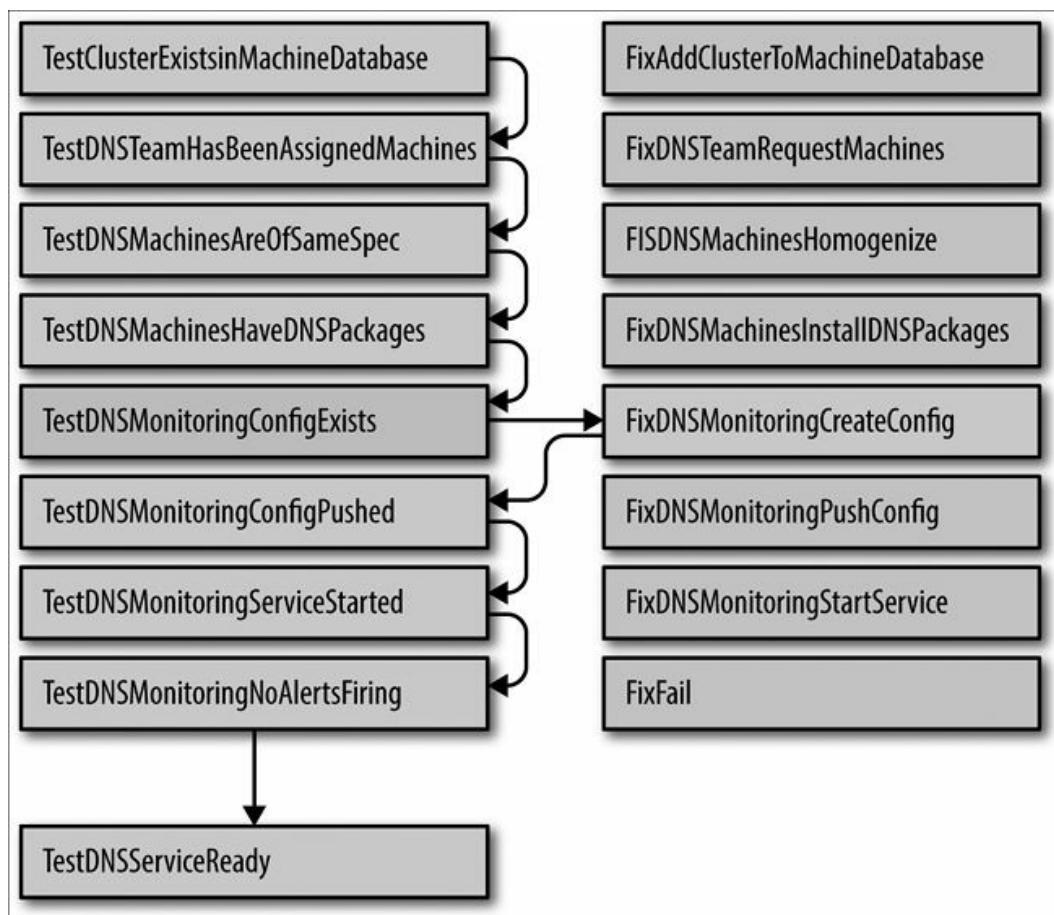


Figura 7.2 – ProdTest para o Serviço DNS, mostrando que um teste com falha resultou na execução de apenas uma correção.

A tendência em especializar

Os processos de automação podem variar de acordo com três aspectos:

- *Competência*, isto é, sua precisão.
- *Latência*, a rapidez com que todos os passos são executados quando iniciados.
- *Relevância*, ou a proporção do processo do mundo real incluído na automação.

Começamos com um processo altamente competente (mantido e operado pelos proprietários do serviço), com alta latência (os proprietários do serviço executavam o processo em seus tempos livres ou o atribuíam aos novos engenheiros) e muito relevante (os proprietários do serviço sabiam quando o mundo real mudava e podiam corrigir a automação).

Para reduzir a latência de ativação, muitas equipes donas de serviços instruíam uma única “equipe de ativação” a respeito de qual automação deveria ser executada. A equipe de ativação usava tickets para iniciar cada etapa da ativação de modo que pudéssemos monitorar as tarefas restantes e a quem elas haviam sido atribuídas. Se as interações humanas associadas aos módulos de automação ocorressem entre pessoas na mesma sala, as ativações de clusters podiam ocorrer em um período de tempo bem menor. Por fim, tínhamos nosso processo de automação competente, preciso e no tempo certo!

Porém, esse estado não durava muito tempo. O mundo real é caótico: software, configuração, dados etc. mudavam, resultando em mais de mil alterações diferentes por dia nos sistemas afetados. As pessoas mais afetadas pelos bugs de automação não eram mais especialistas no domínio, portanto a automação se tornou menos relevante (o que significava que novos passos deixavam de ser executados) e menos competente (novas flags podiam fazer a automação falhar). No entanto, demorou um pouco para que essa queda de qualidade tivesse impacto na velocidade.

O código de automação, assim como os códigos para teste de unidade, morre quando a equipe de manutenção não tem obsessão em manter o código em sincronia com a base de código que ela trata. O mundo muda em torno do

código: a equipe de DNS inclui novas opções de configuração, a equipe de armazenagem muda os nomes de seus pacotes e a equipe de rede precisa dar suporte a novos dispositivos.

Ao tirar das equipes que operavam os serviços a responsabilidade de manter e executar seus códigos de automação, criamos incentivos organizacionais ruins:

- Uma equipe cuja principal tarefa seja agilizar a ativação atual não tem nenhum incentivo para reduzir o débito técnico das equipes proprietárias de serviços que executam o serviço em produção mais tarde.
- Uma equipe que não execute automações não tem nenhum incentivo para criar sistemas que sejam fáceis de automatizar.
- Um gerente de produto cujo cronograma não seja afetado por uma automação de baixa qualidade sempre dará prioridade a novas funcionalidades em detrimento da simplicidade e da automação.

As ferramentas mais funcionais normalmente são escritas por aqueles que as usam. Um argumento semelhante se aplica à razão pela qual as equipes de desenvolvimento de produtos se beneficiam ao ter pelo menos algum conhecimento operacional de seus sistemas em produção.

As ativações novamente tinham alta latência, eram imprecisas e sem competência – o pior de todos os mundos. No entanto, uma exigência de segurança não relacionada ao contexto nos permitiu sair dessa armadilha. Boa parte da automação distribuída contava naquela época com SSH. Isso é ruim do ponto de vista da segurança, pois as pessoas precisam ter usuário root em muitas máquinas para executar a maior parte dos comandos. Uma conscientização crescente de ameaças de segurança avançadas e persistentes nos levou a reduzir os privilégios desfrutados pelos SREs ao mínimo absoluto necessário para suas tarefas. Tivemos que substituir nosso uso de sshd por um Local Admin Daemon autenticado, orientado por ACL e baseado em RPC, também conhecido como Admin Servers, que tinham permissões para fazer essas alterações locais. Como resultado, ninguém podia instalar nem modificar um servidor sem uma trilha possível de ser auditada. Mudanças no Local Admin Daemon e no Package Repo foram associadas às revisões de código, fazendo com que fosse muito difícil alguém exceder em sua

autoridade; dar a alguém o acesso para instalar pacotes não lhe permitia ver logs com localização compartilhada. O Admin Server fazia log de quem solicitou a RPC, de qualquer parâmetro e do resultado de todas as RPCs para melhorar a depuração e as auditorias de segurança.

Ativação de cluster orientada a serviços

Na iteração seguinte, os Admin Servers passaram a fazer parte dos fluxos de trabalho das equipes de serviços, tanto os Admin Servers específicos de máquina (para instalar pacotes e fazer reinicializações) quanto os Admin Servers no nível de clusters (para ações como drenagem de tráfego ou ativação de um serviço). Os SREs migraram de escrever shell scripts em seus diretórios home para criar servidores de RPC revisados por pares, com ACLs de alta granularidade.

Mais tarde, depois de perceber que os processos de ativação realmente deviam pertencer às equipes que eram donas dos serviços, vimos essa situação como uma maneira de abordar a ativação de clusters como um problema de SOA (Service-Oriented Architecture, ou Arquitetura orientada a serviços): os donos dos serviços seriam responsáveis por criar um Admin Server para tratar RPCs para ativação/desativação de clusters, enviadas pelo sistema que sabia quando os clusters estavam prontos. Por sua vez, cada equipe forneceria o contrato (API) necessário à automação de ativação, ao mesmo tempo que continuaria livre para alterar a implementação subjacente. Quando um cluster alcançava o estágio de “rede pronta”, a automação enviava uma RPC para cada Admin Server que desempenhasse uma função na ativação do cluster.

Agora temos um processo com baixa latência, competente e preciso; acima de tudo, esse processo tem permanecido robusto à medida que a taxa de alterações, o número de equipes e a quantidade de serviços parecem dobrar a cada ano.

Conforme mencionamos antes, nossa evolução na automação de ativação seguiu um caminho:

1. Ação manual disparada pelo operador (sem automação)
2. Automação específica a um sistema, escrita por um operador

3. Automação genérica mantida externamente
4. Automação específica a um sistema, mantida internamente
5. Sistemas autônomos que não exigem nenhuma intervenção humana

Embora essa evolução, de modo geral, tenha tido sucesso, o estudo de caso do Borg ilustra outra maneira com que passamos a pensar no problema da automação.

Borg: nascimento do computador em escala de warehouse

Outra maneira de entender o desenvolvimento de nossa atitude em relação à automação, e quando e onde essa automação é mais bem implantada, é considerar a história do desenvolvimento de nossos sistemas de gerenciamento de clusters.⁹ Assim como o MySQL no Borg, que mostrou o sucesso de converter operações manuais em operações automáticas, e o processo de ativação de clusters, que mostrou a desvantagem de não pensar cuidadosamente sobre onde e como a automação é implementada, o desenvolvimento de um gerenciamento de clusters também acabou oferecendo outra lição sobre como a automação deve ser feita. Como em nossos dois exemplos anteriores, algo bem sofisticado foi criado como o resultado eventual da evolução contínua de etapas iniciais mais simples.

No começo, os clusters do Google eram implantados de modo muito semelhante às redes pequenas de todo mundo na época: racks de máquinas com propósitos específicos e configurações heterogêneas. Os engenheiros faziam login em algum computador “mestre” bem conhecido para realizar tarefas administrativas; binários e configurações “de ouro” ficavam nesses mestres. Como tínhamos apenas um provedor de colocation (colo provider), a maior parte da lógica de nomenclatura supunha implicitamente essa localização. À medida que a produção se expandiu e começamos a usar vários clusters, diferentes domínios (nomes de clusters) passaram a compor o quadro geral. Tornou-se necessário ter um arquivo que descrevesse o que cada máquina fazia, agrupando as máquinas de acordo com uma estratégia pouco rigorosa de nomenclatura. Esse arquivo descritor, em conjunto com o

equivalente a um SSH paralelo, nos permitia reinicializar (por exemplo) todas as máquinas de pesquisa de uma só vez. Naquela época, era comum receber tickets como “pesquisa está sendo feita com a máquina x1, crawl pode ter a máquina agora”.

O desenvolvimento da automação havia começado. Inicialmente, ela era constituída de scripts Python simples para operações como:

- Gerenciamento de serviços: manter os serviços operando (por exemplo, reiniciando-os após falhas de segmentação [segfaults]).
- Monitorar quais serviços deveriam executar em quais máquinas.
- Fazer parse de mensagens de log: acessar todas as máquinas e procurar regexps.

Em algum momento, a automação se transformou em um banco de dados apropriado que monitorava o estado das máquinas, além de incorporar ferramentas mais sofisticadas de monitoração. Com a união das automações disponíveis, podíamos agora administrar automaticamente boa parte do ciclo de vida das máquinas: perceber quando as máquinas tinham falhas, remover os serviços, enviar as máquinas para conserto e restaurar a configuração quando elas voltavam.

Porém, dando um passo para trás, essa automação era útil, mas profundamente limitada pelo fato de as abstrações do sistema estarem implacavelmente vinculadas às máquinas físicas. Precisávamos de uma nova abordagem, e assim o Borg [Ver15] nasceu: um sistema que se afastou das atribuições relativamente estáticas de host/porta/job do mundo anterior, em direção ao tratamento de uma coleção de máquinas como um mar de recursos administrados. Central para o seu sucesso – e sua concepção – estava a noção de transformar o gerenciamento de clusters em uma entidade para a qual chamadas de API poderiam ser feitas, isto é, em um tipo de coordenador central. Isso resultou em dimensões extras de eficiência, flexibilidade e confiabilidade: de modo diferente do modelo anterior de “donos” de máquinas, o Borg permitia que as máquinas escalonassem, por exemplo, tarefas batch e tarefas voltadas aos usuários na mesma máquina.

Essa funcionalidade, em última instância, resultou em upgrades contínuos e

automáticos do sistema operacional com uma dose bem pequena de esforço constante¹⁰ – um esforço que *não* escala com o tamanho total das implantações em produção. Pequenos desvios no estado das máquinas atualmente são corrigidos de modo automático; o gerenciamento de máquinas quebradas e do ciclo de vida essencialmente não exige um SRE a essa altura. Milhares de máquinas nascem, morrem e vão para o conserto diariamente, sem nenhum esforço de SRE. Para ecoar as palavras de Ben Treynor Sloss: ao assumir a abordagem de que esse era um problema de software, a automação inicial nos deu tempo suficiente para transformar o gerenciamento de clusters em algo autônomo, em oposto a ser automatizado. Atingimos essa meta trazendo ideias relacionadas à distribuição de dados, APIs, arquiteturas hub-and-spoke e desenvolvimento de softwares de sistemas clássicos distribuídos para o domínio do gerenciamento de infraestrutura.

Uma analogia interessante é possível nesse caso: podemos fazer um mapeamento direto entre o caso de uma única máquina e o desenvolvimento das abstrações de gerenciamento de clusters. De acordo com essa visão, reescalonar em outra máquina é muito semelhante a um processo ser transferido de uma CPU para outra: é claro que esses recursos computacionais por acaso estão na outra extremidade de um link de rede, mas até que ponto isso realmente importa? Pensando nesses termos, reescalonar é semelhante a um recurso intrínseco do sistema, e não algo que alguém “automatizaria” – os seres humanos não poderiam reagir de modo suficientemente rápido, de qualquer modo. De modo semelhante, no caso da ativação de cluster: nessa metáfora, a ativação de cluster é simplesmente a adição de mais capacidade para reescalonar, um pouco como a adição de disco ou de RAM em um único computador. No entanto, de um computador que represente um único nó, em geral, não se espera que ele continue a funcionar quando um grande número de componentes falha. No caso do computador global, isso é esperado – ele *deve* se autocorrigir para funcionar depois que crescer além de determinado tamanho, por causa do grande número de falhas estatisticamente garantido que ocorrem a cada segundo. Isso implica que, à medida que movemos os sistemas para um ponto mais alto da hierarquia, da operação manual ao disparo automático de ações, até ser autônomo, alguma capacidade de autointrospecção será necessária para

sobreviver.

Confiabilidade é a característica fundamental

É claro que, para uma resolução de problemas eficiente, os detalhes da operação interna dos quais a introspecção depende também devem ser expostos aos seres humanos que administram o sistema em geral. Discussões análogas sobre o impacto da automação no domínio não computacional – por exemplo, na aviação¹¹ ou em aplicações industriais – com frequência apontam para as desvantagens de uma automação altamente eficiente¹²: operadores humanos são cada vez mais afastados do contato direto útil com o sistema à medida que a automação inclui cada vez mais atividades diárias com o tempo. Inevitavelmente, então, surge uma situação em que a automação falha, e os seres humanos agora são incapazes de operar o sistema com sucesso.

A fluidez de suas reações se perdeu devido à falta de prática, e seus modelos mentais do que o sistema *deveria* estar fazendo não refletem mais a realidade do que ele *faz*.¹³ Essa situação surge com mais frequência quando o sistema não é autônomo – isto é, quando a automação substitui as ações manuais e se presume que as ações manuais possam ser sempre realizadas e estarão disponíveis como estavam antes. Infelizmente, com o passar do tempo, isso deixa de ser verdadeiro: essas ações manuais nem sempre podem ser realizadas porque a funcionalidade para possibilitá-las não existe mais.

Nós também passamos por situações em que a automação foi realmente prejudicial em várias ocasiões – veja a seção “Automação: permitindo falhas em escala” –, mas, na experiência do Google, há mais sistemas para os quais a automação ou um comportamento autônomo não são mais itens extras opcionais. À medida que os sistemas escalam, esse certamente será o caso, mas continua havendo fortes argumentos para que haja mais comportamentos autônomos para os sistemas, independentemente de seu tamanho. A confiabilidade é a característica fundamental, e um comportamento autônomo e resiliente é uma forma conveniente de chegar lá.

Recomendações

Talvez você tenha lido os exemplos deste capítulo e decidido que precisa ter a escala do Google para ter algo a ver com algum aspecto da automação. Isso não é verdade por dois motivos: a automação proporciona mais do que apenas uma economia de tempo, portanto vale a plena implementá-la em mais casos do que um cálculo simples de tempo gasto *versus* tempo economizado poderia sugerir. Porém, a abordagem com o potencial mais elevado, na verdade, está na fase de design: fazer o lançamento de produtos e iterar rapidamente pode permitir que você implemente funcionalidades de modo mais rápido, mas raramente resulta em um sistema resiliente. A operação autônoma é difícil de ser adaptada de modo convincente em sistemas bem grandes, mas boas práticas padronizadas de engenharia de software podem ajudar de forma considerável: ter subsistemas desacoplados, introduzir APIs, minimizar efeitos colaterais, e assim por diante.

Automação: permitindo falhas em escala

O Google opera mais de uma dúzia de seus próprios datacenters de grande porte, mas também dependemos de máquinas em várias instalações de colocation (ou “colos”) de terceiros. Nossas máquinas nesses colos são usadas para terminar a maior parte das conexões de entrada, ou como um cache para nossa própria Content Delivery Network (Rede de Fornecimento de Conteúdo) a fim de reduzir a latência para o usuário final. A qualquer instante, vários desses racks estão sendo instalados e desativados; esses dois processos, em sua maior parte, são automatizados. Um passo durante a desativação envolve sobrescrever o conteúdo todo do disco de todas as máquinas do rack, depois do que um sistema independente verifica se os dados foram apagados com sucesso. Chamamos esse processo de “Diskerase”.

Certa vez, a automação responsável por desativar um rack em particular falhou, mas somente depois do passo de Diskerase ter sido concluído com sucesso. Mais tarde, o processo de desativação foi reiniciado do princípio, para depurar a falha. Nessa iteração, quando tentamos enviar o conjunto de máquinas do rack para o Diskerase, a automação determinou que o

conjunto de máquinas que ainda precisava ser submetido ao Diskerase estava (corretamente) vazio. Infelizmente, o conjunto vazio era usado como um valor especial, interpretado para significar “tudo”. Isso implicou no envio pela automação de quase todas as máquinas que tínhamos em todos os colos para o Diskerase.

Em minutos, o Diskerase altamente eficiente limpou os discos de todas as máquinas de nosso CDN, e as máquinas não eram mais capazes de ser o término das conexões dos usuários (nem fazer mais nada útil). Ainda éramos capazes de servir a todos os usuários a partir de nossos próprios datacenters e, depois de alguns minutos, o único efeito visível externamente era um leve aumento de latência. Até onde podíamos ver, poucos usuários perceberam o problema, graças ao bom planejamento de capacidade (pelo menos isso estava correto!). Enquanto isso, gastamos boa parte de dois dias reinstalando as máquinas dos racks afetados no colo; então, passamos as semanas seguintes fazendo auditorias e adicionando mais verificações de sanidade – incluindo limitação de taxa – em nossa automação, e deixando o nosso fluxo de trabalho de desativação idempotente.

¹ Se você acha que já entende exatamente o valor da automação, vá para a seção “O valor da SRE no Google”. No entanto, observe que nossa descrição contém algumas nuances que podem ser úteis para ter em mente ao ler o restante do capítulo.

² N.T.: *Failover* é o processo de alternar para um servidor, um sistema, um componente de hardware ou uma rede redundante ou standby na ocorrência de uma falha ou do término anormal de uma aplicação ou de algum dos itens mencionados anteriormente. (Baseado em <https://en.wikipedia.org/wiki/Failover>.)

³ O expertise adquirido ao implementar uma automação desse tipo também é valioso por si só; os engenheiros passam a compreender profundamente os processos existentes que foram automatizados e, mais tarde, podem automatizar novos processos mais rapidamente.

⁴ Veja o quadrinho XKCD em <http://xkcd.com/1205/>.

⁵ Veja, por exemplo, <http://blog.engineyard.com/2014/pets-vs-cattle>.

⁶ É claro que nem todo sistema que deve ser gerenciado realmente oferece APIs possíveis de serem chamadas para isso – forçando o uso de algumas ferramentas, por exemplo, chamadas de CLI ou cliques automáticos em sites.

⁷ N.T.: Em desenvolvimento de software, uma *leaky abstraction* é uma abstração que expõe detalhes e limitações de sua implementação subjacente que, de modo ideal, deveriam estar ocultos de seus usuários. (Baseado em: https://en.wikipedia.org/wiki/Leaky_abstraction.)

⁸ N.T.: “Em matemática e ciência da computação, a idempotência é a propriedade que algumas

operações têm de poderem ser aplicadas várias vezes sem que o valor do resultado se altere após a aplicação inicial.” (Fonte: <https://pt.wikipedia.org/wiki/Idempot%C3%AAncia>.)

9 Resumimos e simplificamos essa história para ajudar em sua compreensão.

10 Um número pequeno, que não se altera.

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

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

13 Esse é ainda um outro bom motivo para treinamentos regulares nas práticas; veja a seção “Interpretando papéis em situações de desastre”.

CAPÍTULO 8

Engenharia de release

Escrito por Dinah McNutt

Editado por Betsy Beyer e Tim Harvey

A engenharia de release é uma disciplina da engenharia de software relativamente nova e de rápido crescimento, que pode ser descrita de forma concisa como a construção e a entrega de softwares [McN14a]. Os engenheiros de release têm uma sólida compreensão (se não forem especialistas) em gerenciamento de códigos-fontes, compiladores, linguagem de configuração e ferramentas automatizadas para builds, gerenciadores de pacotes e instaladores. Seu conjunto de habilidades inclui profundo conhecimento de vários domínios: desenvolvimento, gerenciamento de configuração, integração de testes, administração de sistemas e suporte a clientes.

Executar serviços confiáveis exige processos de release confiáveis. Os SREs (Site Reliability Engineers, ou Engenheiros de Confiabilidade de Sites) devem saber que os binários e as configurações que usam são construídos de modo reproduzível e automatizado para que os releases sejam repetíveis, e não algo gerado somente naquela ocasião específica.

Mudanças em qualquer aspecto do processo de release devem ser intencionais, e não acidentais. Os SREs se importam com esse processo, desde o código-fonte até a implantação. A engenharia de release é um cargo profissional específico no Google. Os engenheiros de release trabalham com os engenheiros de software (SWEs) no desenvolvimento de produtos e com os SREs para definir todos os passos necessários para o release de software – desde o modo como o software é armazenado no repositório de códigos-fontes, passando pelas regras de build para compilação, até como os testes, o empacotamento e a implantação são conduzidos.

O papel de um engenheiro de release

O Google é uma empresa orientada a dados, e a engenharia de release acompanha essa realidade. Temos ferramentas que informam métricas variadas, por exemplo, quanto tempo demora para uma mudança de código ser implantada em produção (em outras palavras, a velocidade do lançamento de versões), e estatísticas sobre quais funcionalidades são usadas nos arquivos de configuração de builds [Ada15]. A maioria dessas ferramentas foi concebida e desenvolvida pelos engenheiros de release.

Os engenheiros de release definem as melhores práticas para usar nossas ferramentas a fim de garantir que os projetos sejam entregues com metodologias consistentes e repetíveis. Nossas melhores práticas incluem todos os elementos do processo de release. Exemplos incluem flags de compiladores, formatos para tags de identificação de builds e os passos necessários durante uma build. Garantir que nossas ferramentas se comportem de forma correta por padrão e estejam sempre adequadamente documentadas facilita às equipes manter o foco nas funcionalidades e nos usuários, em vez de gastar tempo reinventando a roda (de forma precária) quando se trata de release de softwares.

O Google tem um grande número de SREs responsáveis pela implantação segura dos produtos e por manter os serviços funcionando. Para garantir que nossos processos de release atendam aos requisitos de negócio, os engenheiros de release e os SREs trabalham juntos a fim de desenvolver estratégias para mudanças em canarying¹, implantar novas versões sem interromper os serviços e fazer rollback de funcionalidades que demonstrarem ter problemas.

Filosofia

A engenharia de release é orientada por uma filosofia de engenharia e de serviços expressa por meio de quatro princípios importantes, detalhados nas próximas seções.

Modelo autônomo

Para funcionar em escala, as equipes devem ser autossuficientes. A engenharia de release desenvolveu boas práticas e ferramentas que permitem às nossas equipes de desenvolvimento de produtos controlar e executar seus próprios processos de release. Embora tenhamos milhares de engenheiros e de produtos, conseguimos ter uma alta velocidade de release porque as equipes individuais são capazes de decidir a frequência para entregar novas versões de seus produtos e quando fazê-lo. Os processos de release podem ser automatizados até o ponto de exigirem um mínimo de envolvimento dos engenheiros, e muitos projetos são construídos automaticamente e entregues usando uma combinação de nosso sistema automatizado de build e nossas ferramentas de implantação. Os releases são realmente automáticos e só exigem o envolvimento de um engenheiro se e quando um problema surgir.

Alta velocidade

Os softwares voltados aos usuários (como muitos componentes do Google Search) são reconstruídos com frequência, pois visamos a fazer o rollout de funcionalidades voltadas aos clientes o mais rápido possível. Adotamos a filosofia de que releases frequentes resultam em menos alterações entre as versões. Essa abordagem facilita os testes e a resolução de problemas. Algumas equipes fazem builds a cada hora e então selecionam a versão que realmente será implantada em produção a partir do conjunto de builds resultante. A seleção é baseada nos resultados dos testes e nas funcionalidades contidas em uma dada build. Outras equipes adotaram o modelo de release “Push on Green” e implantam todas as builds que passarem em todos os testes [Kle14].

Builds herméticas

As ferramentas de build devem nos permitir garantir consistência e possibilidade de repetição. Se duas pessoas tentarem construir o mesmo produto com o mesmo número de versão no repositório de códigos-fontes em máquinas diferentes, esperamos ter resultados idênticos.² Nossas builds são herméticas, isto é, são insensíveis às bibliotecas e outros softwares instalados

na máquina de build. Em vez disso, as builds dependem de versões conhecidas de ferramentas de build, como compiladores, e de dependências, como as bibliotecas. O processo de build é autocontido e não deve depender de serviços externos ao ambiente de build.

Reconstruir versões mais antigas quando precisamos corrigir um bug em um software que está executando em produção pode ser um desafio. Fazemos isso reconstruindo a mesma versão da build original e incluindo alterações específicas que foram submetidas após esse ponto no tempo. Chamamos a essa tática de *cherry picking*. Nossas próprias ferramentas de build têm versões que estão no repositório de códigos-fontes para o projeto sendo construído. Desse modo, um projeto construído no mês passado não usará a versão de compilador deste mês se um cherry pick for necessário, pois essa versão pode conter funcionalidades incompatíveis ou indesejadas.

Garantindo o cumprimento de políticas e de procedimentos

Várias camadas de segurança e de controle de acesso determinam quem pode realizar operações específicas quando uma nova versão do projeto é lançada. As operações associadas incluem:

- Aprovar mudanças no código-fonte – essa operação é administrada por meio de arquivos de configuração espalhados pela base de código.
- Especificar as ações a serem executadas durante o processo de release.
- Criar uma nova versão.
- Aprovar a proposta inicial de integração (que é uma requisição para realizar uma build com um número específico de versão no repositório de códigos-fontes) e os cherry picks subsequentes.
- Implantar uma nova versão.
- Fazer alterações na configuração de build de um projeto.

Quase todas as mudanças na base de código exigem uma revisão de código, que é uma ação planejada, integrada ao nosso fluxo de trabalho normal de desenvolvimento. Nosso sistema automatizado de release gera um relatório de todas as alterações contidas em uma versão, que é arquivado com outros artefatos da build. Ao permitir que os SREs entendam quais mudanças estão

inclusas em uma nova versão de um projeto, esse relatório pode agilizar a resolução de problemas em uma nova versão.

Build e implantação contínuas

O Google desenvolveu um sistema automatizado de release chamado *Rapid*. O Rapid é um sistema que tira proveito de várias tecnologias do Google a fim de oferecer um framework que disponibilize versões escaláveis, herméticas e confiáveis. As próximas seções descrevem o ciclo de vida do software no Google e como ele é administrado usando o Rapid e outras ferramentas associadas.

Construção

O Blaze³ é a ferramenta de build preferida no Google. Essa ferramenta oferece suporte para a construção de binários a partir de várias linguagens, incluindo nossas linguagens padrões C++, Java, Python, Go e JavaScript. Os engenheiros usam o Blaze para definir os alvos de build (a saída de uma build, por exemplo, um arquivo JAR) e para especificar as dependências de cada alvo. Ao fazer uma build, o Blaze constrói automaticamente os alvos das dependências.

Os alvos de build para binários e testes de unidade são definidos nos arquivos de configuração de projeto do Rapid. Flags específicas de projeto, como um identificador único de build, são passadas ao Blaze pelo Rapid. Todos os binários aceitam uma flag que exibe a data da build, o número da revisão e o identificador de build, que nos permitem associar facilmente um binário a um registro de como ele foi construído.

Branching

Fazemos check in de todo o código no branch principal da árvore de códigos-fontes (mainline, ou linha principal). No entanto, a maioria dos projetos grandes não faz o release diretamente a partir da linha principal. Em vez disso, criamos um branch a partir dessa linha em uma revisão específica e jamais fazemos merge das alterações no branch de volta à linha principal. As correções de bug são submetidas à linha principal e então fazemos um cherry

pick para o branch a fim de incluí-las no release. Essa prática evita selecionar inadvertidamente alterações não relacionadas a esse release, submetidas na linha principal, desde que a build original ocorreu. Ao usar esse branch e o método de cherry pick, sabemos qual é o conteúdo exato de cada versão.

Testes

Um sistema de testes contínuos executa testes de unidade no código da linha principal sempre que uma alteração é submetida, permitindo detectar falhas de build e de testes rapidamente. A engenharia de release recomenda que os alvos de teste de builds contínuos correspondam aos mesmos alvos de teste associados ao release do projeto. Também recomendamos criar releases no número de revisão (versão) do último build de teste contínuo que tenha concluído todos os testes com sucesso. Essas medidas reduzem as chances de alterações subsequentes feitas na linha principal causarem falhas durante a build feita no momento do release.

Durante o processo de release, executamos novamente os testes de unidade usando o branch de release e criamos uma trilha de auditoria (audit trail) mostrando que todos os testes passaram. Esse passo é importante porque, se um release envolver cherry picks, o branch de release pode conter uma versão de código que não existe em nenhum lugar na linha principal. Queremos garantir que os testes passem no contexto do que está realmente sendo entregue.

Para complementar o sistema de testes contínuos, usamos um ambiente de testes independente, que executa testes no nível de sistema em artefatos de build empacotados. Esses testes podem ser iniciados manualmente ou a partir do Rapid.

Empacotamento

O software é distribuído em nossas máquinas de produção por meio do MPM (Midas Package Manager) [McN14c]. O MPM monta os pacotes com base em regras do Blaze que listam os artefatos de build a serem incluídos, juntamente com seus proprietários e as permissões. Os pacotes são nomeados (por exemplo, *search/shakespeare/frontend*), recebem um número de versão

com uma hash única e são assinados para garantir sua autenticidade. O MPM tem suporte para aplicar rótulos a uma versão em particular de um pacote. O Rapid aplica um rótulo contendo o ID de build, que garante que um pacote possa ser referenciado de forma exclusiva pelo nome do pacote e por esse rótulo.

Os rótulos podem ser aplicados a um pacote MPM para indicar a localização de um pacote no processo de release (por exemplo, dev, canary ou production). Se você aplicar um rótulo existente a um novo pacote, o rótulo será automaticamente transferido do pacote antigo para o novo pacote. Por exemplo, se um pacote tiver canary como rótulo, uma pessoa que instalar depois a versão canary desse pacote receberá automaticamente a versão mais recente 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 usados para definir alvos de build e de teste, regras de implantação e informações administrativas (como os proprietários do projeto). Listas de controle de acesso baseadas em funções determinam quem pode executar ações específicas em um projeto Rapid.

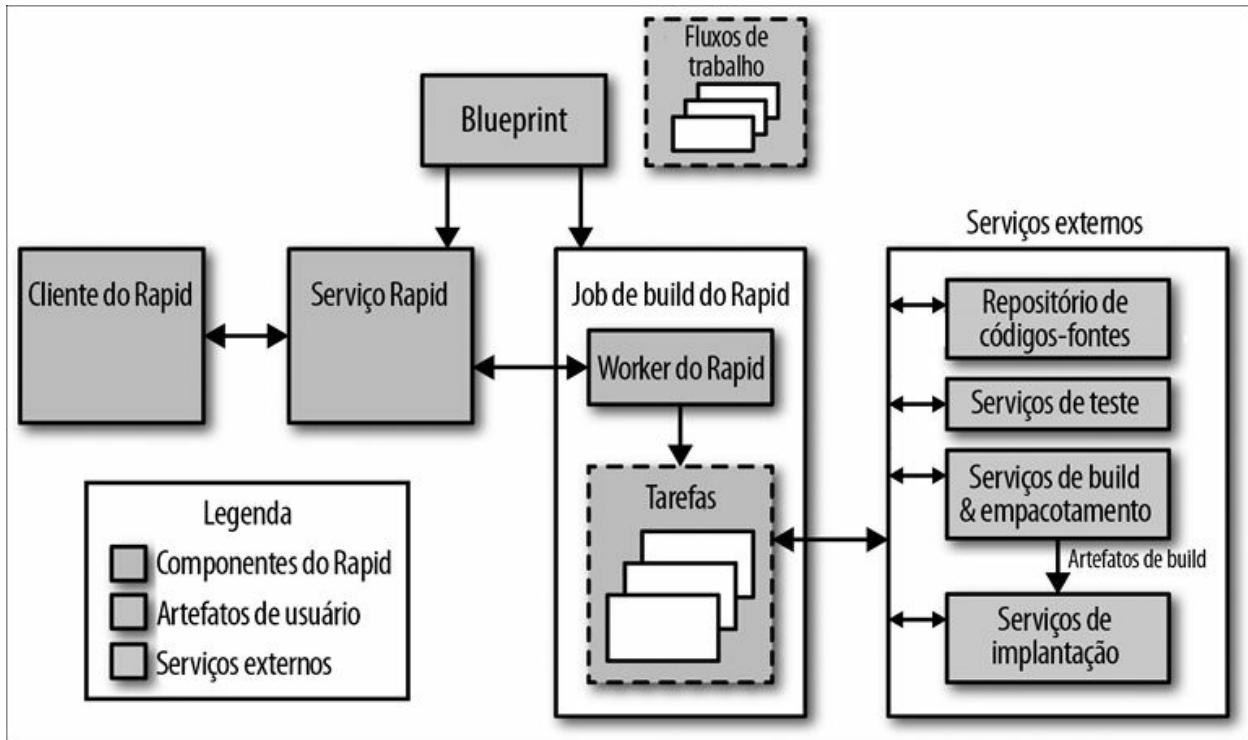


Figura 8.1 – Visão simplificada da arquitetura do Rapid mostrando os principais componentes do sistema.

Cada projeto Rapid tem fluxos de trabalho que definem as ações a serem executadas durante o processo de release. As ações do fluxo de trabalho podem ser executadas de forma serial ou em paralelo, e um fluxo de trabalho pode iniciar outros fluxos de trabalho. O Rapid despacha requisições de trabalho para tarefas que estão executando como um job do Borg em nossos servidores de produção. Como o Rapid utiliza nossa infraestrutura de produção, ele é capaz de lidar com milhares de requisições de release simultaneamente.

Um processo típico de release ocorre da seguinte maneira:

1. O Rapid utiliza o número de revisão da integração solicitada (com frequência, obtido automaticamente de nosso sistema de testes contínuos) para criar um branch de release.
2. O Rapid utiliza o Blaze para compilar todos os binários e executar os testes de unidade, muitas vezes executando esses dois passos em paralelo. A compilação e os testes ocorrem em ambientes dedicados a essas tarefas específicas, e não no job do Borg em que o fluxo de trabalho do Rapid

está executando. Essa separação nos permite trabalhar facilmente em paralelo.

3. Os artefatos da build são então disponibilizados para testes sistêmicos e implantações canário (canary deployments). Uma implantação canário típica envolve iniciar alguns jobs em nosso ambiente de produção depois que os testes de sistema foram concluídos.
4. Os resultados de cada passo do processo são registrados em log. Um relatório com todas as alterações desde o último release é criado.

O Rapid permite administrar nossos branches de release e os cherry picks; requisições individuais de cherry picks podem ser aprovadas ou rejeitadas para inclusão em uma nova versão.

Implantação

Com frequência, o Rapid é usado para conduzir implantações simples de forma direta. Ele atualiza os jobs do Borg para usar os pacotes MPM recém-construídos com base em definições de implantação nos arquivos blueprint e em executores de tarefa especializados.

Para implantações mais complexas, utilizamos o Sisyphus, que é um framework de automação de rollouts de propósito geral desenvolvido pela SRE. Um rollout é uma unidade de trabalho lógica composta de uma ou mais tarefas individuais. O Sisyphus oferece um conjunto de classes Python que pode ser estendido para dar suporte a qualquer processo de implantação. Ele tem um painel de controle que permite um controle mais detalhado de como o rollout é feito, além de oferecer uma maneira de monitorar o seu progresso.

Em uma integração típica, o Rapid cria um rollout em um job de longa duração do Sisyphus. O Rapid conhece o rótulo de build associado ao pacote MPM que ele criou, e pode especificar esse rótulo de build ao criar o rollout no Sisyphus. O Sisyphus utiliza o rótulo de build para especificar a versão dos pacotes MPM que deve ser implantada.

Com o Sisyphus, o processo de rollout pode ser tão simples ou tão complexo quanto for necessário. Por exemplo, ele pode atualizar todos os jobs associados imediatamente ou pode fazer o rollout de um novo binário em

clusters sucessivos durante um período de várias horas.

Nossa meta é ajustar o processo de implantação ao perfil de risco de um dado serviço. Em ambientes de desenvolvimento ou de pré-produção, podemos gerar builds a cada hora e implantar as versões automaticamente quando todos os testes passarem. Para serviços maiores voltados ao usuário, podemos fazer a atualização de versão iniciando em um cluster e expandindo exponencialmente até que todos os clusters estejam atualizados. Para partes mais sensíveis da infraestrutura, podemos estender o rollout por vários dias, entrelaçando-os entre instâncias em diferentes regiões geográficas.

Gerenciamento de configuração

O gerenciamento de configuração (configuration management) é uma área de colaboração particularmente intensa entre engenheiros de release e SREs. Embora o gerenciamento de configuração possa inicialmente parecer um problema ilusoriamente simples, mudanças em configuração são uma fonte em potencial de instabilidade. Como resultado, nossa abordagem para atualizações de versão e administração de configurações de sistemas e de serviços evoluiu de forma considerável com o tempo. Hoje em dia, usamos vários modelos para distribuir arquivos de configuração, conforme descrito nos parágrafos a seguir. Todos os esquemas envolvem armazenar a configuração em nosso repositório principal de códigos-fontes e impor um requisito rigoroso de revisão de código.

Utilize a linha principal (mainline) para configuração. Esse foi o primeiro método usado para configurar serviços no Borg (e nos sistemas anteriores ao Borg). Ao usar esse esquema, os desenvolvedores e os SREs modificam os arquivos de configuração no branch principal. As alterações são revisadas e então aplicadas ao sistema em execução. Como resultado, os releases de binários e as mudanças de configuração permanecem desacoplados. Embora seja simples do ponto de vista conceitual e procedural, essa técnica com frequência resulta em discrepâncias entre a versão dos arquivos de configuração para os quais o check in foi feito e a versão em execução do arquivo de configuração porque os jobs precisam ser atualizados para perceber as alterações.

Inclua os arquivos de configuração e os binários no mesmo pacote MPM. Para projetos com poucos arquivos de configuração ou projetos em que os arquivos (ou um subconjunto deles) mudam a cada ciclo de release, os arquivos de configuração podem ser incluídos no pacote MPM com os binários. Embora essa estratégia limite a flexibilidade ao vincular fortemente o binário e os arquivos de configuração, ela simplifica a implantação, pois exige a instalação apenas de um único pacote.

Empacote os arquivos de configuração em “pacotes de configuração” do MPM. Podemos aplicar o princípio de ser hermético ao gerenciamento de configuração. Como as configurações de binários tendem a estar fortemente vinculadas a versões particulares de binários, tiramos proveito dos sistemas de build e empacotamento para criar um snapshot (imagem instantânea) e lançar arquivos de configuração juntamente com seus binários. De modo semelhante ao nosso tratamento dos binários, podemos usar o ID de build para reconstruir a configuração em um ponto específico no tempo.

Por exemplo, uma mudança que implemente uma nova funcionalidade pode ser lançada com uma flag que configure essa funcionalidade. Ao gerar dois pacotes MPM, um para o binário e outro para a configuração, preservamos a capacidade de alterar cada pacote de forma independente, ou seja, se a funcionalidade foi lançada com uma flag de configuração `first_folio`, mas percebemos que ela deveria ser `bad_quarto`, podemos fazer um cherry pick dessa alteração e passá-la para o branch de release, reconstruir o pacote de configuração e implantá-lo. Essa abordagem tem a vantagem de não exigir uma nova build do binário.

Podemos tirar proveito do recurso de atribuição de rótulos do MPM para indicar quais versões dos pacotes MPM devem ser instaladas em conjunto. Um rótulo `much_ado` pode ser aplicado aos pacotes MPM descritos no parágrafo anterior, o que nos permite acessar os dois pacotes usando esse rótulo. Quando uma nova versão do projeto for construída, o rótulo `much_ado` será aplicado aos novos pacotes. Como essas tags são únicas no namespace de um pacote MPM, somente o pacote mais recente com essa tag será usado.

Leia os arquivos de configuração de uma área de armazenagem externa.

Alguns projetos têm arquivos de configuração que precisam mudar com frequência ou de forma dinâmica (isto é, enquanto o binário está executando). Esses arquivos podem ser armazenados no Chubby, no Bigtable ou em nosso sistema de arquivos baseado em códigos-fontes [Kem11].

Em suma, os proprietários dos projetos consideram as diferentes opções para distribuir e administrar os arquivos de configuração e decidem qual delas funciona melhor de acordo com cada caso.

Conclusões

Embora este capítulo tenha discutido especificamente a abordagem do Google para a engenharia de release e os modos como os engenheiros de release trabalham em colaboração com os SREs, essas práticas também podem ser aplicadas de forma mais ampla.

Não serve apenas para os Googlers

Quando equipados com as ferramentas corretas, uma automação apropriada e políticas bem definidas, os desenvolvedores e os SREs não deveriam se preocupar com releasing de software. Os releases podem ser tão simples quanto pressionar um botão.

A maioria das empresas lida com o mesmo conjunto de problemas de engenharia de release, independentemente de seus tamanhos ou das ferramentas que utilizam: Como você deve tratar a atribuição de versões em seus pacotes? Você deve usar um modelo de build e implementação contínuos ou fazer builds periódicas? Com que frequência você deve lançar uma nova versão? Quais políticas de gerenciamento de configuração você deve usar? Quais métricas de release são de seu interesse?

Os engenheiros de release do Google desenvolveram suas próprias ferramentas como resultado de suas necessidades, pois as ferramentas de código aberto ou fornecidas por terceiros não funcionavam na escala que precisávamos. Ferramentas personalizadas nos permitem incluir funcionalidades para dar suporte a (e até mesmo impor) políticas para processos de release. No entanto, essas políticas devem ser definidas antes

para que as funcionalidades apropriadas sejam acrescentadas às nossas ferramentas; todas as empresas devem se esforçar para definir seus processos de release, não importando se esses processos possam ser automatizados e/ou garantidos.

Use a engenharia de release desde o princípio

A engenharia de release com frequência tem sido um pensamento *a posteriori*, e esse modo de pensar deve mudar à medida que as plataformas e os serviços continuam a crescer em tamanho e em complexidade.

As equipes devem ter provisões para recursos de engenharia de release no início do ciclo de desenvolvimento de um produto. É mais barato implantar boas práticas e processos com antecedência, em vez de precisar adaptá-los ao seu sistema mais tarde.

É essencial que os desenvolvedores, os SREs e os engenheiros de release trabalhem em conjunto. O engenheiro de release deve entender qual é a intenção por trás de como o código deve ser construído e implantado. Os desenvolvedores não devem fazer uma build e “jogar o resultado por cima do muro” para que seja tratado pelos engenheiros de release.

As equipes de cada projeto decidem quando a engenharia de release deve se envolver em um projeto. Como a engenharia de release ainda é uma disciplina relativamente jovem, os gerentes nem sempre planejam e fazem uma provisão para a engenharia de release nos estágios iniciais de um projeto. Desse modo, ao considerar a maneira de incorporar as práticas de engenharia de release, não se esqueça de considerar sua função conforme aplicada em todo o ciclo de vida de seu produto ou de seu serviço – particularmente nos estágios iniciais.

Mais informações

Para mais informações sobre engenharia de release, veja as apresentações a seguir, cada uma com um vídeo online disponível:

- *How Embracing Continuous Release Reduced Change Complexity* (Como abraçar o release contínuo reduziu a complexidade das mudanças, <http://usenix.org/conference/ures14west/summit-program/presentation/dickson>), USENIX Release Engineering Summit West 2014, [Dic14]

- *Maintaining Consistency in a Massively Parallel Environment* (Mantendo a consistência em um ambiente com intenso paralelismo, <https://www.usenix.org/conference/ucms13/summit-program/presentation/mcnutt>), USENIX Configuration Management Summit 2013, [McN13]
 - *The 10 Commandments of Release Engineering* (Os dez mandamentos da engenharia de release, https://www.youtube.com/watch?v=RNmjYV_UsQ8), Segundo Workshop Internacional em Engenharia de Release 2014, [McN14b]
 - *Distributing Software in a Massively Parallel Environment* (Distribuindo software em um ambiente com intenso paralelismo, <https://www.usenix.org/conference/lisa14/conference-program/presentation/mcnutt>), LISA 2014, [McN14c]
-

¹ N.T.: A *canary release* (implantação canário) consiste na implantação de uma nova versão de software para apenas um subconjunto de usuários a fim de reduzir o risco da implantação dessa versão em todo o ambiente de produção. O nome faz alusão à técnica de levar canários a uma mina de carvão; em caso de haver gases tóxicos, o canário morreria antes, alertando os mineradores do risco.

² O Google utiliza um repositório de códigos-fontes unificado e monolítico; veja [Pot16].

³ O Blaze foi disponibilizado com código aberto como Bazel. Consulte “Bazel FAQ” no site do Bazel em <http://bazel.io/faq.html>.

CAPÍTULO 9

Simplicidade

*Escrito por Max Luebbe
Editado por Tim Harvey*

O preço da confiabilidade é perseguir a máxima simplicidade.

— C.A.R. Hoare, palestra no Turing Award

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, vamos parar de introduzir bugs. Se o hardware ou as bibliotecas subjacentes jamais mudarem, nenhum desses componentes introduzirá bugs. Se congelarmos a base atual de usuários, jamais precisaremos escalar o sistema. De fato, um bom resumo da abordagem de SRE ao gerenciamento de sistemas é: “No final das contas, nossa tarefa é manter a agilidade e a estabilidade em equilíbrio no sistema”.²

Estabilidade versus agilidade do sistema

Às vezes faz sentido sacrificar a estabilidade em detrimento da agilidade. Muitas vezes, já abordei um domínio de problema com o qual eu não tinha familiaridade conduzindo o que chamo de programação exploratória – definindo um prazo de validade explícito para qualquer código que eu escrever, ciente de que precisarei tentar e falhar uma vez para realmente entender a tarefa que devo executar. O código obtido na data de vencimento pode ser muito mais livre em termos de cobertura de testes e gerenciamento de releases, pois jamais será enviado para produção nem será visto pelos usuários.

Para a maioria dos sistemas de software em produção, queremos uma mistura equilibrada de estabilidade e agilidade. Os SREs trabalham para criar

procedimentos, práticas e ferramentas que deixem o software mais confiável. Ao mesmo tempo, os SREs garantem que esse trabalho tenha o mínimo possível de impacto na agilidade dos desenvolvedores. De fato, de acordo com a experiência da SRE, processos confiáveis tendem a aumentar a agilidade dos desenvolvedores: rollouts rápidos e confiáveis em produção facilitam as mudanças nesse ambiente. Como resultado, quando um bug surgir, menos tempo será necessário para encontrá-lo e corrigi-lo. Introduzir confiabilidade no desenvolvimento permite aos desenvolvedores focar sua atenção naquilo com que realmente nos importamos – a funcionalidade e o desempenho de seus softwares e sistemas.

A virtude do tédio

De modo diferente de praticamente tudo o mais na vida, o “tédio” 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 atinjam suas metas de negócio de forma previsível. Nas palavras do engenheiro Robert Muth, do Google, “Diferentemente de uma história de detetive, a falta de entusiasmo, suspense e quebra-cabeças, na verdade, é uma propriedade desejável do código-fonte”. As surpresas em produção são a nêmesis da SRE.

Como sugere Fred Brooks em seu artigo “No Silver Bullet” (Sem solução mágica) [Bro95], é muito importante considerar a diferença entre complexidade essencial e complexidade accidental. A complexidade essencial é a complexidade inerente a uma dada situação, que não pode ser removida da definição de um problema, enquanto a complexidade accidental é mais fluida e pode ser resolvida com esforços de engenharia. Por exemplo, escrever um servidor web implica lidar com a complexidade essencial de servir páginas web rapidamente. No entanto, se escrevermos um servidor web em Java, podemos introduzir uma complexidade accidental ao tentar minimizar o impacto da coleta de lixo (garbage collection) no desempenho.

Com o olhar voltado à minimização da complexidade accidental, as equipes de SRE devem:

- Retroceder quando uma complexidade accidental for introduzida nos

sistemas pelos quais são responsáveis.

- Esforçar-se constantemente para eliminar a complexidade dos sistemas com os quais estão envolvidas e pelos quais assumiram a responsabilidade operacional.

Não abrirei mão do meu código!

Como os engenheiros são seres humanos que muitas vezes desenvolvem um vínculo emocional com suas criações, os confrontos em relação a limpezas em larga escala da árvore de códigos-fontes não são incomuns. Alguns podem protestar: “E se precisarmos desse código mais tarde?”, “Por que não comentamos simplesmente o código para que possamos acrescentá-lo facilmente depois?”, ou “Por que não condicionamos o código a uma flag em vez de apagá-lo?”. Todas essas sugestões são horríveis. Os sistemas de controle de versões facilitam reverter mudanças, enquanto centenas de linhas de código comentado geram distração e confusão (especialmente se os arquivos-fontes continuarem a evoluir), e um código que jamais é executado, condicionado por uma flag que está sempre desabilitada, é uma bomba-relógio metafórica, esperando explodir, como vivenciado de forma dolorosa pela Knight Capital, por exemplo (veja “Order In the Matter of Knight Capital Americas LLC” (Decisão na questão da Knight Capital Americas LLC) [Sec13]).

Correndo o risco de parecer radical, quando você considerar um web service do qual se espera que esteja disponível 24/7, até certo ponto, toda linha de código nova escrita é uma responsabilidade. A SRE promove práticas que fazem com que seja mais provável que todo código tenha um propósito essencial, por exemplo, analisando detalhadamente o código para garantir que ele realmente conduza aos objetivos de negócio, removendo código morto de forma rotineira e inserindo detectores de excessos em todos os níveis de testes.

A métrica “linhas de código negativas”

O termo “software inchado” (software bloat) foi cunhado para descrever a

tendência do software em se tornar mais lento e maior com o tempo, como resultado de um fluxo constante de funcionalidades adicionais. Embora um software inchado pareça ser intuitivamente indesejável, seus aspectos negativos se tornam mais claros ainda quando considerados do ponto de vista da SRE: toda 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, de testar e, com frequência, tem menos defeitos. Com essa perspectiva em mente, talvez seja necessário manter uma certa reserva quando sentirmos um forte desejo de adicionar novas funcionalidades em um projeto. Algumas das programações mais satisfatórias que já fiz consistiram em apagar milhares de linhas de código de uma só vez quando já haviam deixado de ser úteis.

APIs mínimas

O poeta francês Antoine de Saint-Exupéry escreveu que “a perfeição é finalmente alcançada, não quando não há mais nada para adicionar, mas quando não houver mais nada para retirar” [Sai39]. Esse princípio também é aplicável no design e na construção de softwares. As APIs são uma expressão particularmente clara do motivo pelo qual essa regra deve ser seguida.

Escrever APIs claras e mínimas é um aspecto essencial para administrar a simplicidade de um sistema de software. Quanto menos métodos e argumentos oferecermos aos clientes da API, mais fácil será entendê-la, e mais esforços poderemos dedicar para deixar esses métodos os melhores possíveis. Mais uma vez, um tema recorrente aparece: a decisão consciente de não assumir determinados problemas nos permite manter o foco em nosso problema principal e melhorar de forma considerável a solução com a qual explicitamente nos comprometemos. Em software, menos é mais! Uma API pequena e simples normalmente é uma marca registrada de um problema bem compreendido.

Modularidade

Expandindo para além de APIs e binários únicos, muitas das regras gerais

que se aplicam à programação orientada a objetos também se aplicam ao design de sistemas distribuídos. A capacidade de fazer alterações em partes do sistema isoladamente é essencial para criar um sistema ao qual seja possível dar suporte. De modo específico, baixo acoplamento entre binários, ou entre binários e a configuração, é um padrão de simplicidade que, ao mesmo tempo, promove agilidade aos desenvolvedores e estabilidade ao sistema. Se um bug for descoberto em um programa que seja um componente de um sistema maior, esse bug poderá ser corrigido e uma atualização poderá ser feita em produção, independentemente do restante do sistema.

Embora a modularidade oferecida pelas APIs possa parecer simples, a noção de que ela também se estenda ao modo como as alterações nas APIs são introduzidas não é tão evidente. Uma única mudança em uma API pode forçar os desenvolvedores a reconstruir seu sistema todo e correr o risco de introduzir novos bugs. Atribuir versões às APIs permite que os desenvolvedores continuem a usar a versão da qual seu sistema dependa, enquanto o upgrade para uma versão mais recente é feito de forma segura e planejada. O ritmo de entrega de versões pode variar em um sistema, em vez de exigir uma atualização de versão completa de todo o sistema em produção sempre que uma funcionalidade for acrescentada ou melhorada.

À medida que um sistema cresce e se torna mais complexo, a separação de responsabilidades entre APIs e entre binários passa a ser cada vez mais importante. A analogia a seguir é uma comparação direta com o design de classes orientadas a objetos: assim como sabemos que escrever uma classe do tipo “saco de surpresas” contendo funções não relacionadas é uma prática ruim, o mesmo vale para criar e colocar em produção um binário “util” ou “misc”. Um sistema distribuído bem projetado é constituído de colaboradores, cada um com um propósito claro e um escopo bem definido.

O conceito de modularidade também se aplica aos formatos de dados. Um dos pontos fortes mais importantes dos buffers de protocolo³ do Google, e que também era uma meta de design, foi criar um formato de transmissão que fosse compatível para trás e para frente.

Simplicidade em releases

Releases simples geralmente são melhores que releases complicados. É muito mais fácil mensurar e compreender o impacto de uma única alteração, em vez de um lote de alterações lançado simultaneamente. Se lançarmos cem alterações não relacionadas em um sistema ao mesmo tempo e o desempenho piorar, entender quais mudanças tiveram impacto no desempenho e como elas fizeram isso exigirá um esforço considerável ou uma instrumentação adicional. Se o release for feito em lotes menores, podemos avançar mais rápido, com mais confiança, pois cada mudança de código poderá ser compreendida isoladamente no sistema maior. Essa abordagem aos releases pode ser comparada ao gradiente descendente em aprendizagem automática (machine learning), no qual encontramos uma solução ótima dando pequenos passos a cada vez e considerando se cada mudança resultou em uma melhoria ou em uma degradação.

Uma conclusão simples

Este capítulo repetiu um tema várias vezes: a simplicidade no software é um pré-requisito para a confiabilidade. Não estamos sendo preguiçosos quando consideramos a forma como podemos simplificar cada passo de uma dada tarefa. Em vez disso, estamos deixando claro o que, exatamente, queremos realizar, e como podemos fazer isso do modo mais fácil. Sempre que dissermos “não” a uma funcionalidade, não estamos limitando a inovação; estamos mantendo o ambiente livre de distrações para que o foco permaneça diretamente na inovação, e a verdadeira engenharia possa continuar.

¹ Com frequência, isso é válido para sistemas complexos em geral; veja [Per99] e [Coo00].

² De autoria do ex-gerente Johan Anderson, da época em que me tornei um SRE.

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

PARTE III

Práticas

Falando de modo simples, os SREs executam serviços – um conjunto de sistemas relacionados, operados com vistas aos usuários, que podem ser internos ou externos – e, em última instância, são responsáveis pela saúde desses serviços. Operar um serviço com sucesso implica diversas atividades: desenvolver sistemas de monitoração, planejar capacidade, responder a incidentes, garantir que as causas-raízes das interrupções de serviço sejam tratadas, e assim por diante. Esta seção aborda a teoria e a prática do trabalho cotidiano de um SRE: desenvolver e operar sistemas computacionais distribuídos de grande porte.

Podemos caracterizar a saúde de um serviço de modo muito semelhante à forma como Abraham Maslow classificou as necessidades humanas [Mas43] – dos requisitos mais básicos necessários para um sistema funcionar como um serviço até as funções de mais alto nível –, permitindo uma atualização de si mesmo e assumindo um controle ativo da direção do serviço, em vez de apagar incêndios de modo reativo. Essa compreensão é tão fundamental para o modo como avaliamos os serviços no Google que não foi explicitamente desenvolvida até vários SREs do Google, incluindo nosso antigo colega Mikey Dickerson¹, terem se juntado por um certo período à cultura radicalmente diferente do governo dos Estados Unidos a fim de ajudar no 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 a hierarquia mostrada na Figura III.1 para analisar os elementos que deixam um serviço confiável, dos mais básicos aos mais avançados.

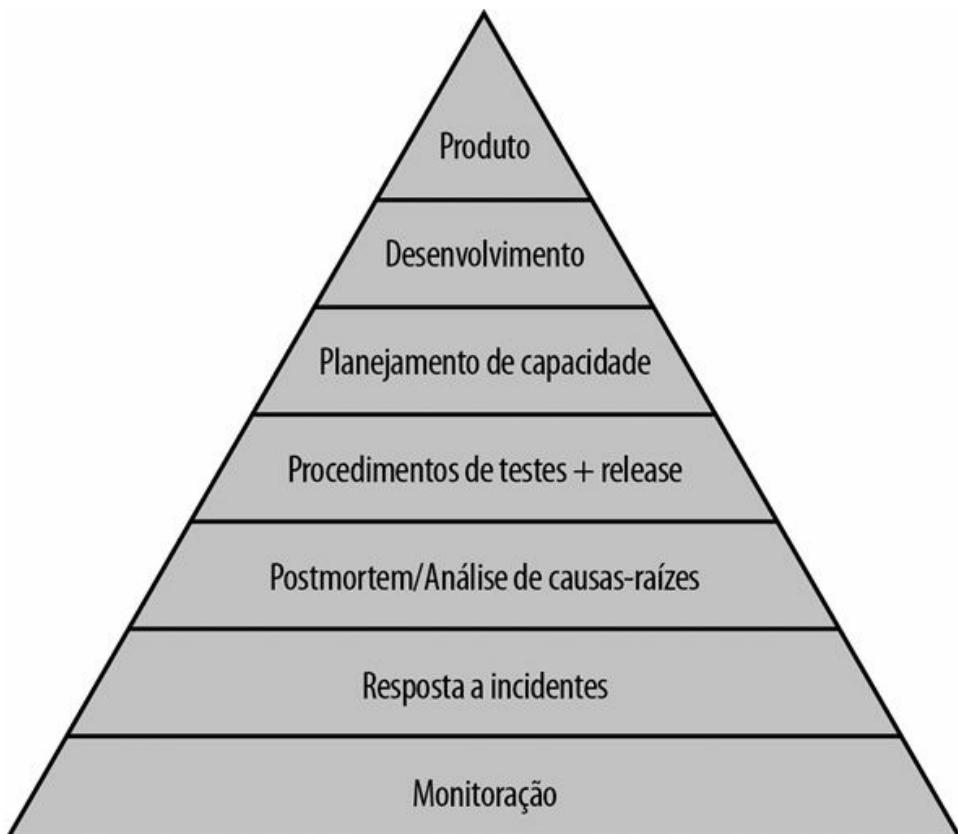


Figura III.1 – Hierarquia da Confiabilidade de Serviços.

Monitoração

Sem monitoração, você não terá nenhuma maneira de dizer se o serviço está sequer funcionando; sem uma infraestrutura de monitoração cuidadosamente planejada, você estará voando às cegas. Talvez todos que tentem usar o site obtenham um erro, talvez não – mas você vai querer estar ciente dos problemas antes que seus usuários os percebam. Discutiremos ferramentas e filosofias no Capítulo 10, *Alertas práticos a partir de dados de séries temporais*.

Resposta a incidentes

Os SREs não fazem plantão somente por fazer: em vez disso, o suporte no plantão é uma ferramenta que usamos para cumprir nossa missão mais ampla e permanecer em contato com o modo como os sistemas de processamento distribuído realmente funcionam (e falham!). Se pudéssemos encontrar um

modo de nos livrar de portar um pager, faríamos isso. No Capítulo 11, *De plantão*, explicamos como balancear os deveres de plantão com nossas outras responsabilidades.

Depois que você estiver ciente de que há um problema, como você faz para se livrar dele? Isso não significa necessariamente corrigi-lo de uma vez por todas – talvez você possa estancar o sangramento reduzindo a precisão do sistema ou desabilitando alguns recursos temporariamente, permitindo uma degradação suave ou, quem sabe, poderia direcionar o tráfego para outra instância do serviço que esteja funcionando de modo apropriado. Os detalhes da solução que você escolher implementar são necessariamente específicos ao seu serviço e à sua empresa. Responder de modo eficiente a incidentes, porém, é algo que pode ser aplicado a todas as equipes.

Descobrir o que está errado é o primeiro passo; apresentaremos uma abordagem estruturada no Capítulo 12, *Resolvendo problemas de modo eficiente*.

Durante um incidente, muitas vezes é tentador se deixar levar pela adrenalina e começar a responder *ad hoc*. Aconselhamos a não cair nessa tentação no Capítulo 13, *Resposta a emergências*, e recomendamos no Capítulo 14, *Administrando incidentes*, que administrar incidentes de modo eficiente deve reduzir seu impacto e limitar a ansiedade gerada pela interrupção do serviço.

Postmortem e análise de causas-raízes

Nosso objetivo é receber um alerta e resolver manualmente apenas problemas novos e empolgantes apresentados pelo nosso serviço; é lamentavelmente tedioso “corrigir” o mesmo problema várias vezes. De fato, essa mentalidade é um dos principais diferenciadores entre a filosofia de SRE e alguns ambientes mais tradicionais centrados em operações. Esse tema é explorado em dois capítulos.

Desenvolver uma cultura de postmortem sem acusações é o primeiro passo para entender o que deu errado (e o que deu certo!), conforme descrito no Capítulo 15, *Cultura de postmortem: aprendendo com o fracasso*.

Relacionado a essa discussão, no Capítulo 16, *Monitorando interrupções de*

serviço, descrevemos brevemente uma ferramenta interna, o monitor de interrupções, que permite às equipes de SRE manter o controle sobre os incidentes recentes em produção, suas causas e as ações tomadas em resposta a eles.

Testes

Depois que entendermos o que tende a dar errado, nosso próximo passo é tentar evitar isso, pois um grama de prevenção vale por um quilo de cura. Suítes de testes oferecem certa garantia de que o nosso software não cometerá determinadas classes de erros antes de ser entregue para produção; discutiremos a melhor maneira de usá-las no Capítulo 17, *Testes voltados à confiabilidade*.

Planejamento de capacidade

No Capítulo 18, *Engenharia de software em SRE*, apresentamos um estudo de caso de engenharia de software em SRE com o Auxon, uma ferramenta para automatizar o planejamento de capacidade.

Seguindo naturalmente o planejamento de capacidade, a distribuição de carga garante que usaremos a capacidade oferecida de forma apropriada. Discutiremos como as requisições aos nossos serviços são enviadas aos datacenters no Capítulo 19, *Distribuição de carga no frontend*. Então, prosseguimos com as discussões no Capítulo 20, *Distribuição de carga no datacenter*, e no Capítulo 21, *Tratando sobrecarga*, ambos essenciais para garantir a confiabilidade do serviço.

Por fim, no Capítulo 22, *Tratando falhas em cascata*, apresentamos conselhos para tratar falhas em cascata, tanto no design do sistema quanto no caso de seu serviço se ver em uma falha em cascata.

Desenvolvimento

Um dos aspectos essenciais da abordagem do Google à Site Reliability Engineering (Engenharia de Confiabilidade de Sites) é que fazemos design de sistemas e um trabalho de engenharia de software significativos e de larga

escala na empresa.

No Capítulo 23, *Administrando estados críticos: consenso distribuído para confiabilidade*, explicamos o consenso distribuído que (disfarçado de Paxos) está no núcleo de muitos sistemas distribuídos do Google, inclusive em nosso sistema Cron globalmente distribuído. No Capítulo 24, *Escalonamento periódico e distribuído com o Cron*, apresentamos um sistema que escala em datacenters inteiros e além, o que não é uma tarefa fácil.

O Capítulo 25, *Pipelines de processamento de dados*, discute as várias formas que os pipelines de processamento de dados podem assumir: de jobs MapReduce executados de uma só vez periodicamente a sistemas que operam quase em tempo real. Diferentes arquiteturas podem resultar em desafios surpreendentes e constraintuitivos.

Garantir que os dados que você armazenou ainda estejam presentes quando quiser lê-los está no coração da integridade de dados; no Capítulo 26, *Integridade de dados: o que você lê é o que você escreveu*, explicaremos como manter os dados seguros.

Produto

Por fim, depois de ter percorrido nosso caminho até o topo da pirâmide da confiabilidade, nos vemos no ponto em que temos um produto com o qual podemos trabalhar. No Capítulo 27, *Lançamento de produtos confiáveis em escala*, descrevemos como o Google faz lançamentos de produtos confiáveis em escala para tentar oferecer aos usuários a melhor experiência possível, desde o Dia Zero.

Outras leituras da SRE do Google

Conforme discutimos antes, testar é uma atividade sutil, e sua execução inapropriada pode ter grandes efeitos na estabilidade em geral. Em um artigo da ACM [Kri12], explicamos como o Google faz testes de resiliência *em toda a empresa* para garantir que somos capazes de enfrentar o inesperado caso um apocalipse zumbi ou outro desastre ocorra.

Embora muitas vezes seja encarado como magia negra, cheio de planilhas

místicas para adivinhar o futuro, o planejamento de capacidade, apesar de tudo, é vital, e como [Hix15a] mostra, você não *precisa* realmente de uma bola de cristal para fazê-lo corretamente.

Por fim, uma abordagem nova e interessante para a segurança de rede corporativa é detalhada em [War14]: uma iniciativa para substituir intranets com privilégios por dispositivos e credenciais de usuários. Conduzido pelos SREs no nível de infraestrutura, definitivamente essa é uma abordagem para ter em mente quando você criar sua próxima rede.

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 cujo propósito (em parte) é levar os princípios e as práticas de SRE aos sistemas de TI do governo norte-americano.

CAPÍTULO 10

Alertas práticos a partir de dados de séries temporais

Escrito por Jamie Wilkinson

Editado por Kavita Guliani

Que as consultas fluam e o pager permaneça em silêncio.

— Bênção tradicional de SRE

A monitoração, que é a camada na base da *Hierarquia das Necessidades de Produção*, é fundamental para operar um serviçoável. A monitoração permite aos proprietários de serviços tomar decisões racionais sobre o impacto de mudanças no serviço, aplicar o método científico na resposta a incidentes e, é claro, garantir sua razão de ser: avaliar o alinhamento do serviço com os objetivos de negócio (veja o Capítulo 6).

Independentemente de um serviço desfrutar de suporte de SRE, ele deve ser operado em uma relação de simbiose com sua monitoração. Porém, tendo recebido a máxima responsabilidade na Produção do Google, os SREs desenvolvem um conhecimento particularmente íntimo da infraestrutura de monitoração que dá suporte aos seus serviços.

Monitorar um sistema bem grande é desafiador por dois motivos:

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

Os sistemas de monitoração do Google não calculam apenas métricas simples, por exemplo, o tempo médio de resposta de um servidor web europeu sem carga; também precisamos entender a distribuição desses tempos de resposta entre todos os servidores web dessa região. Esse

conhecimento nos permite identificar os fatores que contribuem para a cauda (tail) na distribuição da latência.

Na escala com que nossos sistemas operam, receber alertas por falhas em uma só máquina é inaceitável, pois esses dados geram muito ruído para que sejam passíveis de ação. Em vez disso, tentamos criar sistemas que sejam robustos contra falhas nos sistemas dos quais eles dependam. No lugar de exigir um gerenciamento de vários componentes individuais, um sistema grande deve ser projetado para agregar sinais e eliminar os pontos fora da curva. Precisamos de sistemas de monitoração que nos permitam gerar alertas para objetivos de serviço de alto nível, mas que mantenham a granularidade para inspecionar componentes individuais conforme for necessário.

Os sistemas de monitoração do Google evoluíram no curso de dez anos, desde o modelo tradicional de scripts personalizados que verificam respostas e geram alertas, totalmente separados da apresentação visual das tendências, até um novo paradigma. Esse novo modelo fez com que a coleta de séries temporais tivesse um papel fundamental no sistema de monitoração e substituiu aqueles scripts de verificação por uma linguagem rica para manipular séries temporais, transformando-as em gráficos e alertas.

O surgimento do Borgmon

Logo depois que a infraestrutura de escalonamento de jobs, o Borg [Ver15], foi criada em 2003, um novo sistema de monitoração – o Borgmon – foi desenvolvido para complementá-la.

Monitoração de séries temporais fora do Google

Este capítulo descreve a arquitetura e a interface de programação de uma ferramenta de monitoração interna que teve importância fundamental para o crescimento e a confiabilidade do Google durante quase dez anos... mas como isso pode ajudar você, meu caro leitor?

Nos últimos anos, a monitoração passou por uma Explosão Cambriana: Riemann, Heka, Bosun e Prometheus surgiram como ferramentas de código aberto, muito semelhantes ao sistema de geração de alertas baseado em séries temporais do Borgmon. Em particular, o Prometheus¹

compartilha muitas semelhanças com o Borgmon, em especial se compararmos as duas linguagens de regras. Os princípios de coleta variável e avaliação de regras permanecem iguais em todas essas ferramentas e oferecem um ambiente no qual você pode fazer experimentos; esperamos que você possa lançar as ideias inspiradas por este capítulo em seu ambiente de produção.

Em vez de executar scripts personalizados para detectar falhas de sistema, o Borgmon conta com um formato comum de exposição de dados; isso permite a coleta de dados em massa com pouco overhead e evita os custos da execução de subprocessos e de configuração de conexão de rede. Chamamos isso de *monitoração caixa-branca* (veja o Capítulo 6, que apresenta uma comparação entre monitoração caixa-branca e monitoração caixa-preta).

Os dados são usados tanto para gerar gráficos quanto alertas, o que é feito por meio de uma aritmética simples. Como a coleta não está mais em um processo de vida curta, o histórico dos dados coletados também pode ser usado para o processamento desses alertas.

Esses recursos ajudam no atendimento da meta de simplicidade descrita no Capítulo 6. Eles permitem que o overhead do sistema permaneça baixo para que as pessoas operando os serviços possam ser ágeis e responder a mudanças contínuas no sistema à medida que ele crescer.

Para facilitar a coleta em massa, o formato das métricas precisou ser padronizado. Um método mais antigo de exportar o estado interno (conhecido como `varz`)² foi formalizado para permitir a coleta de todas as métricas de um único alvo em um acesso HTTP. Por exemplo, para visualizar uma página de métricas manualmente, você pode usar o comando a seguir:

```
% curl http://webserver:80/varz
http_requests 37
errors_total 12
```

Um Borgmon pode coletar dados de outros Borgmon³; desse modo, podemos construir hierarquias que acompanhem a topologia do serviço, agregando e sintetizando informações, além de descartar outros dados de forma estratégica em cada nível. Geralmente uma equipe executa um único Borgmon por cluster, e um par deles no nível global. Alguns serviços bem grandes se

fragmentam abaixo do nível de cluster em vários Borgmon *scraper* que, por sua vez, alimentam o Borgmon do nível de cluster.

Instrumentação das aplicações

O handler HTTP de /varz simplesmente lista todas as variáveis exportadas em formato texto simples, como chaves e valores separados por espaço, um em cada linha. Uma extensão feita depois adicionou uma variável de mapa, que permite a quem exportou definir vários rótulos (labels) em um nome de variável e então exportar uma tabela de valores ou um histograma. Um exemplo de variável com *valor de mapa* tem a aparência a seguir, que mostra 25 respostas HTTP 200 e 12 HTTP 500s:

```
http_responses map:code 200:25 404:0 500:12
```

O acréscimo de uma métrica a um programa exige apenas uma única declaração no código no qual a métrica é necessária.

Em retrospectiva, é evidente que essa interface textual sem esquema reduz bastante a barreira para a adição de novas instrumentações, o que é um ponto positivo tanto para a engenharia de software quanto para as equipes de SRE. No entanto, isso tem um preço na manutenção em andamento; o desacoplamento entre a definição da variável e seu uso nas regras do Borgmon exige um gerenciamento de mudanças cuidadoso. Na prática, essa negociação tem sido satisfatória, pois ferramentas para validar e gerar as regras também foram escritas.⁴

Exportando variáveis

As raízes web do Google são profundas: cada uma das principais linguagens usadas no Google tem uma implementação da interface de variáveis exportadas que se registra automaticamente junto ao servidor HTTP incluído em todo binário do Google por padrão.⁵ As instâncias da variável a ser exportada permitem que o autor do servidor realize operações óbvias como somar uma quantia ao valor atual, definir uma chave para um valor específico, e assim por diante. A biblioteca expvar⁶ de Go e seu formato de saída JSON têm uma variante dessa API.

Coleta de dados exportados

Para encontrar seus alvos, uma instância do Borgmon é configurada com uma lista deles usando um dos muitos métodos de resolução de nomes.⁷ A lista de alvos geralmente é dinâmica, portanto usar descoberta de serviços reduz o custo de mantê-la e permite que a monitoração escale.

A intervalos predefinidos, o Borgmon busca o URI /varz em cada alvo, decodifica o resultado e armazena os valores em memória. O Borgmon também distribui a coleta de cada instância da lista de alvos por todo o intervalo, de modo que a coleta de cada alvo não acabe em sintonia com a de seu pares.

O Borgmon também registra variáveis “sintéticas” para cada alvo a fim de identificar:

- Se o nome foi resolvido para um host e uma porta
- Se o alvo respondeu a uma coleta
- Se o alvo respondeu a uma verificação de saúde
- Quando a coleta terminou

Essas variáveis sintéticas facilitam escrever regras para detectar se as tarefas monitoradas estão indisponíveis.

É interessante o fato de o varz ser bem diferente do SNMP (Simple Networking Monitoring Protocol, ou Protocolo Simples de Gerenciamento de Redes), que “foi projetado [...] para ter requisitos mínimos de transporte e continuar funcionando quando a maior parte das outras aplicações de rede falharem” [Mic03]. Extrair dados de alvos por meio de HTTP parece estar em desacordo com esse princípio de design; porém a experiência mostra que isso raramente é um problema.⁸ O próprio sistema já está projetado para ser robusto contra falhas de rede e de máquinas, e o Borgmon permite que os engenheiros escrevam regras mais inteligentes de alertas usando a própria falha na coleta como um sinal.

Armazenagem na arena de séries temporais

Um serviço geralmente é composto de muitos binários executando diversas

tarefas em várias máquinas, em vários clusters. O Borgmon precisa manter todos esses dados organizados, ao mesmo tempo que permite consultas flexíveis e um fatiamento desses dados.

O Borgmon armazena todos os dados em um banco de dados em memória, cujo estado é salvo regularmente em disco. Os pontos de dados têm o formato (timestamp, valor) e são armazenados em listas cronológicas chamadas *séries temporais* (time-series); cada série temporal é nomeada de acordo com um conjunto único de *rótulos* (labels) no formato nome=valor.

Conforme mostrado na Figura 10.1, uma série temporal é, conceitualmente, uma matriz de números de uma dimensão, progredindo no tempo. À medida que você acrescentar permutações de rótulos nessa série temporal, a matriz se tornará multidimensional.

"http_requests"	:	:	:	:	:	:	:	:	:	:
	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
agora - $2\Delta t$	0	0	0	0	0	0	0	0	0	0
agora - Δt	0	0	0	0	0	0	0	0	0	0
agora	0	0	0	0	0	0	0	0	0	0
	host1	host2	host3	host4	host5

Figura 10.1 – Séries temporais para erros nomeadas de acordo com o host original do qual cada série foi coletada.

Na prática, a estrutura corresponde a um bloco de memória de tamanho fixo, conhecida como *arena de séries temporais*, com um coletor de lixo (garbage collector) que faz as entradas mais antigas expirarem quando a arena estiver cheia. O intervalo de tempo entre as entradas mais recentes e as entradas mais antigas da arena é o *horizonte*, que indica quantos dados possíveis de consulta são mantidos em RAM. Geralmente, o Borgmon do datacenter e o global são dimensionados para armazenar aproximadamente 12 horas de dados⁹ para apresentação em consoles, e muito menos tempo se forem fragmentos de coletores de mais baixo nível. O requisito de memória para um único ponto de dado é de aproximadamente 24 bytes, portanto podemos

colocar um milhão de séries temporais únicas de 12 horas a intervalos de um minuto em 17 GB de RAM.

Periodicamente, o estado em memória é arquivado em um sistema externo conhecido como TSDB (Time-Series Database, ou Banco de Dados de Séries Temporais). O Borgmon é capaz de consultar dados mais antigos no TSDB e, embora seja mais lento, o TSDB é mais barato e maior que a RAM de um Borgmon.

Rótulos e vetores

Como mostra o exemplo de série temporal na Figura 10.2, as séries temporais são armazenadas como sequências de números e timestamps, chamadas de *vetores*. Assim como os vetores em álgebra linear, esses vetores são fatias e cortes transversais da matriz multidimensional de pontos de dados da arena. Conceitualmente, os timestamps podem ser ignorados, pois os valores são inseridos no vetor a intervalos regulares de tempo – por exemplo, a intervalos de um ou dez segundos, ou de um minuto.

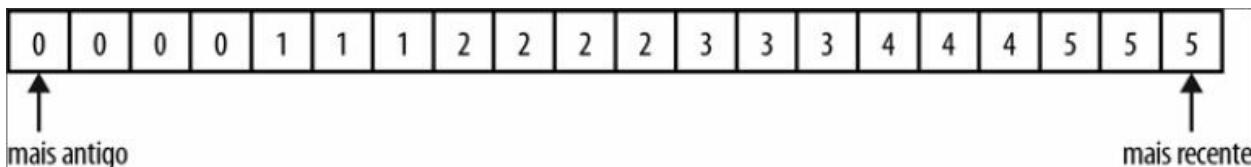


Figura 10.2 – Um exemplo de série temporal.

O nome de uma série temporal é um *labelset* (conjunto de rótulos), pois é implementado como um conjunto de rótulos expresso como pares chave=valor. Um desses rótulos é o próprio nome da variável, isto é, a chave que aparece na página varz.

Alguns nomes de rótulos são declarados como importantes. Para que a série possa ser identificada no banco de dados de séries temporais, ela deve ter, no mínimo, os seguintes rótulos:

var

O nome da variável.

job

O nome dado ao tipo de servidor monitorado.

service

Uma coleção de jobs sem definição rígida, que oferece um serviço aos usuários, sejam eles internos ou externos.

zone

Uma convenção do Google que se refere à localização (geralmente o datacenter) do Borgmon que realizou a coleta dessa variável.

Em conjunto, essas variáveis têm uma aparência semelhante a esta, chamada de *expressão da variável* (variable expression):

```
{var=http_requests,job=webserver,instance=host0:80,service=web,zone=us-west}
```

Uma consulta a uma série temporal não exige a especificação de todos esses rótulos, e uma busca de um *labelset* devolve todas as séries temporais correspondentes em um vetor. Desse modo, podemos devolver um vetor de resultados removendo o rótulo instance da query anterior, se houver mais de uma instância no cluster. Por exemplo:

```
{var=http_requests,job=webserver,service=web,zone=us-west}
```

pode ter como resultado cinco linhas em um vetor, com o valor mais recente das séries temporais 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 a partir das seguintes informações:

- O nome do alvo; por exemplo, o job e a instância.
- O próprio alvo; por exemplo, por meio de variáveis com valores de mapa.
- A configuração do Borgmon; por exemplo, anotações sobre localização ou nova atribuição de rótulo.
- As regras do Borgmon avaliadas.

Também podemos consultar as séries temporais no tempo, especificando uma duração para a expressão da variável:

```
{var=http_requests,job=webserver,service=web,zone=us-west}[10m]
```

Essa consulta devolve os últimos dez minutos de história das séries temporais que corresponderem à expressão. Se estivéssemos coletando pontos de dados uma vez por minuto, esperaríamos que dez pontos de dados fossem devolvidos em uma janela de dez 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
```

Avaliação de regras

O Borgmon, na verdade, é apenas uma calculadora programável com um pouco de açúcar sintático que lhe permite gerar alertas. A coleta de dados e os componentes de armazenagem já descritos são apenas um mal necessário para, em última instância, deixar essa calculadora programável apropriada para servir ao propósito, nesse caso, de ser um sistema de monitoração. :)



Centralizar a avaliação de regras em um sistema de monitoração, em vez de delegá-la a subprocessos separados, implica que os processamentos podem executar em paralelo para vários alvos semelhantes. Essa prática mantém a configuração relativamente pequena (por exemplo, removendo a duplicação de códigos), porém mais eficaz por meio de sua expressividade.

O código do programa Borgmon, também conhecido como *regras do Borgmon*, é constituído de expressões algébricas simples que calculam séries temporais a partir de outras séries temporais. Essas regras podem ser bem eficazes, pois podem consultar a história de uma única série temporal (isto é, o eixo do tempo), consultar subconjuntos diferentes de rótulos de várias séries temporais de uma só vez (isto é, o eixo do espaço) e aplicar muitas operações matemáticas.

As regras são executadas em um pool de threads paralelas sempre que for possível, mas dependem de ordenação quando usam regras definidas anteriormente como entrada. O tamanho dos vetores devolvidos por suas expressões de consulta também determina o tempo de execução total de uma regra. Assim, geralmente, uma pessoa pode adicionar recursos de CPU a uma

tarefa Borgmon em resposta à sua execução lenta. Para auxiliar em análises mais detalhadas, métricas internas no momento da execução das regras são exportadas para realizar a depuração e acompanhar a monitoração.

A agregação é a pedra angular da avaliação de regras em um ambiente distribuído. A agregação implica tomar a soma de um conjunto de séries temporais das tarefas de um job a fim de tratar o job como um todo. A partir dessas somas, as taxas em geral podem ser calculadas. Por exemplo, a taxa total de consultas por segundo de um job em um datacenter é a soma de todas as taxas de mudança¹¹ de todos os contadores de consultas.¹²



Um contador é qualquer variável não monotonicamente decrescente – o que equivale a dizer que os contadores só aumentam de valor. Os medidores (gauges), por outro lado, podem assumir qualquer valor desejado. Os contadores medem valores crescentes, como o número total de quilômetros dirigidos, enquanto os medidores mostram o estado atual, por exemplo, a quantidade de combustível restante ou a velocidade atual. Ao coletar dados no estilo do Borgmon, é melhor usar contadores, pois eles não perdem o significado quando ocorrem eventos entre intervalos de amostragem. Caso alguma atividade ou mudança ocorram entre os intervalos de amostragem, uma coleta de medições provavelmente deixará de perceber essa atividade.

Em um servidor web de exemplo, podemos querer gerar um alerta quando o cluster de nosso servidor web começar a servir mais erros do que achamos que seja normal, expresso como um percentual das requisições – ou, de modo mais técnico, quando a soma das taxas de códigos de retorno diferentes de HTTP 200 de todas as tarefas do cluster dividida pela soma das taxas de requisições a todas as tarefas nesse cluster for maior que algum valor.

Isso é feito da seguinte maneira:

1. Agregando as taxas de códigos de resposta de todas as tarefas, gerando um vetor de taxas nesse ponto no tempo, um para cada código.
2. Calculando a taxa total de erros como a soma desse vetor, gerando um único valor para o cluster nesse ponto no tempo. Essa taxa total de erros exclui o código 200 da soma, pois ele não é um erro.
3. Calculando a razão entre erros e requisições para o cluster todo, dividindo a taxa total de erros pela taxa de requisições que chegaram e, novamente, gerando um único valor para o cluster nesse ponto no tempo.

Cada um desses resultados em um ponto no tempo é concatenado à sua expressão de variável nomeada, o que cria uma nova série temporal. Como resultado, poderemos inspecionar a história das taxas de erro e as razões de erro em algum outro momento.

As regras para a taxa de requisições seriam escritas na linguagem de regras do Borgmon da seguinte maneira:

```
rules <<<
# Calcula a taxa de requisições para cada tarefa a partir do contador de requisições
{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;
# 'without instance' instrui o Borgmon a remover o rótulo instance
# do lado direito.
{var=dc:http_requests:rate10m,job=webserver} =
    sum without instance({var=task:http_requests:rate10m,job=webserver})
>>>
```

A função `rate()` toma a expressão entre parênteses e devolve o delta total dividido pelo tempo total entre os valores mais antigo e mais recente.

Com os dados da série temporal de exemplo da consulta anterior, o resultado para a regra `task:http_requests:rate10m` teria o seguinte aspecto: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 o resultado para a regra `dc:http_requests:rate10m` seria:

```
{var=dc:http_requests:rate10m,job=webserver,service=web,zone=us-west} 4
```

porque a segunda regra utiliza a primeira como entrada.



O rótulo `instance` está ausente na saída agora, pois foi descartado pela regra de agregação. Se ele tivesse permanecido na regra, o Borgmon não teria sido capaz de somar as cinco linhas.

Nesses exemplos, usamos uma janela de tempo, pois estamos lidando com pontos discretos nas séries temporais, em oposição a funções contínuas. Fazer isso facilita o cálculo das taxas, mas significa que, para calcular uma taxa,

precisamos selecionar um número suficiente de pontos de dados. Também devemos lidar com a possibilidade de algumas coletas recentes terem falhado. Lembre-se de que a notação da expressão de variável histórica utiliza o intervalo [10m] para evitar pontos de dados ausentes causados por erros em coleta.

O exemplo também utiliza uma convenção do Google que ajuda na legibilidade. Cada nome de variável calculada contém uma trinca separada por dois-pontos que indica o nível de agregação, o nome da variável e a operação que criou esse nome. Nesse exemplo, as variáveis do lado esquerdo são “tarefa requisições HTTP taxa em 10 minutos” e “datacenter requisições HTTP taxa em 10 minutos”.

Agora que sabemos como criar uma taxa de consultas, podemos expandir isso para calcular também uma taxa de erros e então podemos calcular a razão entre respostas e requisições a fim de entender o volume de trabalho útil do serviço. Podemos comparar a razão entre a taxa de erros e o nosso objetivo de nível de serviço (veja o Capítulo 4) e gerar um alerta se esse objetivo não estiver sendo alcançado ou se correr o risco de deixar de ser atendido:

```
rules <<<
# Calcula uma taxa por tarefa e por rótulo 'code'
{var=task:http_responses:rate10m,job=webserver} =
    rate by code({var=http_responses,job=webserver}[10m]);

# Calcula uma taxa de resposta no nível de cluster por rótulo 'code'
{var=dc:http_responses:rate10m,job=webserver} =
    sum without instance({var=task:http_responses:rate10m,job=webserver});

# Calcula uma nova taxa no nível de cluster somando todos os códigos
# diferentes de 200
{var=dc:http_errors:rate10m,job=webserver} = sum without code(
    {var=dc:http_responses:rate10m,job=webserver,code!=200});

# Calcula a razão entre a taxa de erros e a taxa de requisições
{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 mostra a convenção de colocar a operação que criou

a nova série temporal como sufixo do nome da variável que a representa. Esse resultado é lido como “datacenter erros HTTP razão entre as taxas em 10 minutos”.

A saída dessa regras pode ter a seguinte aparência: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_errors:rate10m,job=webserver,code=500, ...} 0.6

{var=dc:http_errors:rate10m,job=webserver}

{var=dc:http_errors:rate10m,job=webserver, ...} 0.6

{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 diferentes de 200. Embora os valores das expressões sejam iguais, observe que o rótulo de código é mantido em uma, mas é removido da outra.

Conforme mencionamos antes, as regras do Borgmon criam novas séries temporais, portanto os resultados dos cálculos são mantidos na arena de séries temporais e podem ser inspecionados do mesmo modo que as séries temporais originais. A capacidade de fazer isso permite consultas *ad hoc*, além de avaliações e explorações na forma de tabelas ou gráficos. Esse é um recurso útil para depuração quando se está de plantão e, se essas consultas *ad hoc* se mostrarem úteis, elas poderão se transformar em visualizações

permanentes no console de um serviço.

Geração de alertas

Quando uma regra de alerta é avaliada por um Borgmon, o resultado é verdadeiro – caso em que o alerta é acionado – ou é falso. A experiência mostra que alertas podem “oscilar” (alternar seu estado rapidamente); desse modo, as regras permitem uma duração mínima para a qual a regra de alerta deve ser verdadeira antes que o alerta seja enviado. Geralmente, essa duração é definida com pelo menos dois ciclos de avaliação da regra para garantir que nenhuma coleta que não tenha sido feita provoque um falso alerta.

O exemplo a seguir cria um alerta quando a razão de erros em dez minutos exceder 1% e o número total de erros exceder 1:

```
rules <<<
{var=dc:http_errors:ratio_rate10m,job=webserver} > 0.01
    and by job, error
{var=dc:http_errors:rate10m,job=webserver} > 1
    for 2m
=> ErrorRatioTooHigh
    details "webserver error ratio at [[trigger_value]]"
    labels {severity=page};
>>>
```

Nosso exemplo tinha uma razão de 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 nesse momento, portanto o alerta não será ativado. Depois que o número de erros exceder 1, o alerta ficará *pendente* por dois minutos para garantir que não é um estado transitório, e só então será *disparado*.

A regra de alerta contém um pequeno template para preencher uma mensagem com informações contextuais: para qual job é o alerta, o nome do alerta, o valor numérico da regra que o disparou, e assim por diante. As informações contextuais preenchidas pelo Borgmon quando o alerta é disparado são enviadas no Alert RPC.

O Borgmon está conectado a um serviço operado de forma centralizada, conhecido como Alertmanager, que recebe Alert RPCs quando a regra é inicialmente acionada e, novamente, quando o alerta é considerado “em

disparo”. O Alertmanager é responsável por encaminhar a notificação de alerta ao destino correto. Ele pode ser configurado para:

- Inibir determinados alertas quando outros estão ativos.
- Remover alertas duplicados de vários Borgmon que tenham os mesmos labelsets.
- Reunir vários alertas em um só ou separar um alerta em vários com base em seus labelsets quando vários alertas com labelsets similares são disparados.

Conforme descrito no Capítulo 6, as equipes enviam os alertas que merecem pages para o rodízio de plantão e seus alertas importantes, porém não críticos, para suas filas de tickets. Todos os demais alertas devem ser preservados como dados informativos para painéis de controle de status.

Um guia mais abrangente para design de alertas pode ser encontrado no Capítulo 4.

Fragmentando a topologia de monitoração

Um Borgmon é capaz de importar dados de séries temporais de outro Borgmon também. Embora seja possível tentar coletar dados de todas as tarefas de um serviço de modo global, fazer isso se transformaria rapidamente em um gargalo cada vez maior e introduziria um único ponto de falha no design. Em vez disso, um protocolo de streaming é usado para transmitir dados de séries temporais entre os Borgmon, economizando tempo de CPU e bytes de rede se comparado ao formato varz baseado em texto. Uma implantação típica desse tipo utiliza dois ou mais Borgmon globais para agregação de mais alto nível e um Borgmon em cada datacenter para monitorar todos os jobs executando nessa localidade. (O Google divide a rede de produção em zonas para mudanças em produção; portanto, ter duas ou mais réplicas globais oferece diversidade diante da manutenção e das interrupções de serviço para esse que, de outro modo, seria um ponto único de falha.)

Como mostra a Figura 10.3, implantações mais complicadas fragmentam o Borgmon do datacenter em uma camada puramente de extração de dados

(muitas vezes por causa de limitações de RAM e de CPU em um único Borgmon para serviços muito grandes) e uma camada de agregação de datacenter que realiza principalmente avaliação de regras para agregação. Às vezes, a camada global é separada entre avaliação de regras e painéis de controle.

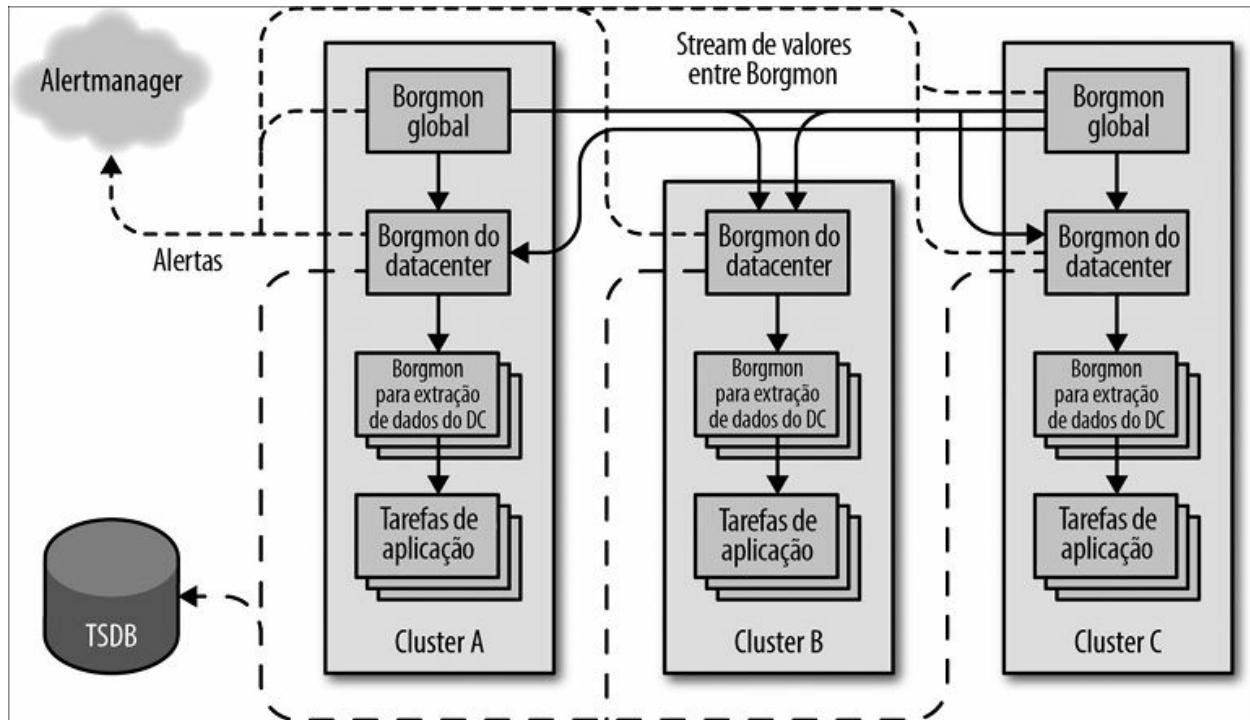


Figura 10.3 – Um modelo de fluxo de dados de uma hierarquia de Borgmon em três clusters.

O Borgmon da camada superior é capaz de filtrar os dados do Borgmon da camada inferior que devem ser transmitidos para que o Borgmon global não preencha sua arena com todas as séries temporais por tarefa das camadas inferiores. Desse modo, a hierarquia de agregação constrói caches locais de séries temporais relevantes, possíveis de ser exploradas quando for necessário.

Monitoração caixa-preta

O Borgmon é um sistema de monitoração caixa-branca – ele inspeciona o estado interno do serviço alvo, e as regras são escritas com conhecimento do funcionamento interno em mente. A natureza transparente desse modelo

oferece muita capacidade para identificar rapidamente quais componentes estão falhando, quais filas estão cheias e onde estão os gargalos, tanto ao responder a um incidente quanto ao testar a implantação de uma nova funcionalidade.

No entanto, a monitoração caixa-branca não oferece um quadro geral do sistema monitorado; contar exclusivamente com a monitoração caixa-branca significa que você não estará ciente do que os usuários veem. Você verá apenas as consultas que chegam ao alvo; as consultas que não chegam por causa de um erro de DNS serão invisíveis, enquanto as consultas perdidas por causa de uma falha de servidor não darão sinal de vida. Você só pode gerar alertas sobre as falhas esperadas.

As equipes do Google resolvem esse problema de abrangência com o Prober, que executa uma verificação de protocolo em um alvo e informa sucesso ou falha. O Prober pode enviar alertas diretamente para o Alertmanager, ou seu próprio varz pode ser coletado por um Borgmon. O Prober é capaz de validar o payload da resposta do protocolo (por exemplo, o conteúdo HTML de uma resposta HTTP) e validar se o conteúdo é esperado, e até mesmo extrair e exportar valores na forma de séries temporais. As equipes muitas vezes usam o Prober para exportar histogramas de tempos de resposta por tipo de operação e tamanho de payload para que seja possível analisar o desempenho visível aos usuários. O Prober é um híbrido entre o modelo de verificação-e-teste e uma extração de variáveis mais rica para criar séries temporais.

O Prober pode ser apontado tanto para o domínio do frontend quanto por trás do distribuidor de carga. Ao usar os dois alvos, podemos detectar falhas localizadas e remover alertas. Por exemplo, podemos monitorar tanto o www.google.com com carga distribuída quanto os servidores web em cada datacenter por trás do distribuidor de carga. Essa configuração nos permite saber se o tráfego continua sendo servido quando um datacenter falha ou isolar rapidamente uma fronteira no gráfico de fluxo de tráfego no ponto em que uma falha ocorreu.

Mantendo a configuração

A configuração do Borgmon separa a definição das regras dos alvos

monitorados. Isso significa que os mesmos conjuntos de regras podem ser aplicados a vários alvos de uma só vez, evitando escrever configurações quase idênticas de forma repetida. Essa separação de preocupações pode parecer incidental, mas reduz enormemente o custo de manutenção da monitoração ao evitar muitas repetições na descrição dos sistemas alvos.

O Borgmon também tem suporte para templates de linguagem. Esse sistema do tipo macro permite aos engenheiros construir bibliotecas de regras que podem ser reutilizadas. Novamente, essa funcionalidade diminui a repetição, reduzindo assim a probabilidade de bugs na configuração.

É claro que qualquer ambiente de programação de alto nível cria oportunidades para a complexidade; por isso, o Borgmon oferece uma maneira de criar unidades extensíveis e testes de regressão sintetizando dados de séries temporais a fim de garantir que as regras se comportem como o autor acha que elas o farão. A equipe de Monitoração de Produção opera um serviço de integração contínua que executa uma suíte desses testes, empacota a configuração e a envia a todos os Borgmon em produção, que então validam a configuração antes de aceitá-la.

Na vasta biblioteca de templates comuns que foram criados, duas classes de configuração de monitoração surgiram. A primeira classe simplesmente codifica o esquema emergente de variáveis exportadas de uma dada biblioteca de código, de modo que qualquer usuário da biblioteca possa reutilizar o template de seu varz. Esses templates existem para a biblioteca de servidor HTTP, alocação de memória, biblioteca de cliente de armazenagem e serviços genéricos de RPC, entre outros. (Embora a interface varz não declare nenhum esquema, a biblioteca de regras associada à biblioteca de código acaba declarando um.)

A segunda classe de biblioteca surgiu à medida que construímos templates para administrar a agregação de dados de uma tarefa em um único servidor nos dados do serviço global. Essas bibliotecas contêm regras genéricas de agregação para variáveis exportadas que os engenheiros podem usar para modelar a topologia de seus serviços.

Por exemplo, um serviço pode oferecer uma única API global, mas pode estar em vários datacenters. Em cada datacenter, o serviço é composto de vários

fragmentos, e cada fragmento é composto de vários jobs com números variados de tarefas. Um engenheiro é capaz de modelar essa divisão com regras do Borgmon de modo que, na depuração, subcomponentes possam ser isolados do restante do sistema. Esses agrupamentos geralmente seguem o destino compartilhado pelos componentes; por exemplo, as tarefas individuais compartilham o destino por causa dos arquivos de configuração, os jobs em um fragmento compartilham o destino porque estão no mesmo datacenter e sites físicos compartilham o destino por causa da rede.

Convenções para a atribuição de rótulos possibilitam essa divisão: um Borgmon adiciona rótulos indicando o nome da instância do alvo e o fragmento e o datacenter que ele ocupa, os quais podem ser usados para agrupar e agregar essas séries temporais.

Assim, temos vários usos para os rótulos em uma série temporal, embora todos sejam intercambiáveis:

- Rótulos que definem divisões dos próprios dados (por exemplo, nosso código de resposta HTTP na variável `http_responses`).
- Rótulos que definem a fonte dos dados (por exemplo, o nome da instância ou do job).
- Rótulos que indicam a localidade ou a agregação dos dados no serviço como um todo (por exemplo, o rótulo de zona que descreve uma localização física, um rótulo de fragmento que descreve um agrupamento lógico de tarefas).

A natureza de template dessas bibliotecas permite ter flexibilidade em seu uso. O mesmo template pode ser usado para agregar dados em cada camada.

Dez anos se passaram...

O Borgmon fez a transposição do modelo de verificação-e-alerta por alvo para uma coleta de variáveis em massa e uma avaliação de regras centralizada com base nas séries temporais para gerar alertas e fazer diagnósticos.

Esse desacoplamento permite que o tamanho do sistema monitorado escale de modo independente do tamanho das regras de alertas. Essas regras custam menos para ser mantidas, pois são abstraídas a partir de um formato comum

de séries temporais. Novas aplicações vêm prontas com exportação de métricas em todos os componentes e bibliotecas aos quais estão ligadas, além de agregações e templates de console bem maduros, reduzindo mais ainda a carga da implementação.

Garantir que o custo de manutenção escala de forma proporcionalmente menor ao tamanho do serviço é fundamental para que a monitoração (e todos os trabalhos de operação que a sustentam) possa ser mantida. Esse tema é recorrente nas atividades de todos os SREs, pois eles trabalham para escalar todos os aspectos de suas tarefas em escala global.

Porém, dez anos é bastante tempo, e é claro que o terreno da monitoração no Google evoluiu com experimentos e mudanças, além de esforços para melhorias contínuas à medida que a empresa cresce.

Embora o 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 estiver lendo este texto.

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

² O Google nasceu nos Estados Unidos, portanto o termo é pronunciado como “var-zí”.

³ O plural de Borgmon é Borgmons, como na palavra lápis.

⁴ Muitas equipes que não são de SRE usam um gerador para acabar com o código repetitivo (boilerplate) inicial e as atualizações contínuas, e acham o gerador muito mais fácil de usar (embora seja menos eficaz) do que editar diretamente as regras.

⁵ Muitas outras aplicações usam o protocolo de seu serviço para exportar seu estado interno também. O OpenLDAP o exporta por meio da subárvore cn=Monitor; o MySQL é capaz de informar o estado com uma query SHOW VARIABLES; o Apache tem seu handler mod_status.

⁶ <https://golang.org/pkg/expvar/>

⁷ O BNS (Borg Name System) está descrito no Capítulo 2.

⁸ Lembre-se da distinção entre alertas com base em sintomas e com base em causas, descrita no Capítulo 6.

⁹ Esse horizonte de 12 horas é um número mágico que visa a oferecer informações suficientes para depuração de um incidente em RAM, proporcionando consultas rápidas sem consumir RAM *demais*.

¹⁰ Os rótulos service e zone foram omitidos aqui por questões de espaço, mas estão presentes na expressão devolvida.

¹¹ Calcular a soma das taxas em vez da taxa das somas protege o resultado de reinicializações de contadores ou de dados ausentes, talvez devido a uma reinicialização de tarefa ou uma falha na coleta de dados.

12 Apesar de não ter tipos, a maior parte do varz é composta de contadores simples. A função de taxa do Borgmon trata todos os casos extremos de reinicialização de contadores.

13 Os rótulos service e zone foram omitidos por questão de espaço.

14 Os rótulos service e zone foram omitidos por questão de espaço.

CAPÍTULO 11

De plantão

Escrito por Andrea Spadaccini¹

Editado por Kavita Guliani

Estar de plantão é um encargo importante, que muitas equipes de operações e de engenharia devem assumir para manter seus serviços confiáveis e disponíveis. No entanto, há muitas armadilhas na organização de rodízios e responsabilidades de plantão que podem resultar em sérias consequências para os serviços e para as equipes se não forem evitadas. Este capítulo descreve os princípios essenciais da abordagem que os SREs (Site Reliability Engineers, ou Engenheiros de Confiabilidade de Sites) do Google desenvolveram com o passar dos anos, e explica como essa abordagem tem resultado em serviços confiáveis e cargas de trabalho sustentáveis ao longo do tempo.

Introdução

Muitas profissões exigem que os funcionários realizem algum tipo de plantão obrigatório, que implica estar disponível para chamadas durante horários comerciais ou não. No contexto de TI, historicamente, as atividades de plantão são exercidas por equipes dedicadas de Ops cuja atribuição principal é manter a boa saúde do(s) serviço(s) pelo(s) qual(is) elas são responsáveis.

Muitos serviços importantes no Google (por exemplo, Search, Ads e Gmail) têm equipes dedicadas de SREs, responsáveis pelo desempenho e pela confiabilidade desses serviços. Assim, os SREs ficam de plantão para os serviços aos quais dão suporte. As equipes de SRE são bem diferentes das equipes puramente operacionais, pois colocam bastante ênfase no uso de engenharia para abordar os problemas. Esses problemas, que geralmente se

enquadram no domínio operacional, existem em uma escala em que não seria possível tratá-los sem que houvesse soluções baseadas em engenharia de software.

Para garantir esse tipo de resolução de problemas, o Google contrata pessoas com experiências anteriores variadas em sistemas e em engenharia de software para as equipes de SRE. Limitamos a quantidade de tempo que os SREs gastam com tarefas puramente operacionais em 50%; no mínimo 50% do tempo de um SRE deve ser alocado em projetos de engenharia, que aumentam mais o impacto da equipe por meio de automação, além de melhorar o serviço.

Vida de um engenheiro de plantão

Esta seção descreve as atividades típicas de um engenheiro de plantão e oferece um pouco de informações contextuais para o restante do capítulo.

Como guardiões dos sistemas de produção, os engenheiros de plantão cuidam das operações que lhes foram atribuídas administrando interrupções de serviço que afetam a equipe e realizando e/ou analisando mudanças em produção.

Quando está de plantão, um engenheiro está disponível para realizar operações em sistemas de produção em questão de minutos, de acordo com os tempos de resposta a pages combinados entre a equipe e os proprietários de sistema do negócio. Valores típicos são cinco minutos para serviços voltados aos usuários ou que sejam altamente críticos quanto ao tempo, e trinta minutos para sistemas menos sensíveis ao tempo. A empresa disponibiliza o dispositivo para recepção de pages, que geralmente é um telefone celular. O Google tem sistemas flexíveis para entrega de alertas, capazes de enviar pages por meio de vários mecanismos (email, SMS, chamadas automáticas, aplicativos) a diversos dispositivos.

Os tempos de resposta estão relacionados à disponibilidade desejada para o serviço, conforme mostra o exemplo simplista a seguir: se um sistema voltado ao usuário precisa ter quatro noves de disponibilidade em um dado trimestre (99,99%), o downtime permitido por trimestre é de

aproximadamente 13 minutos (Apêndice A). Essa restrição implica que o tempo de reação dos engenheiros de plantão deve ser da ordem de minutos (falando estritamente, 13 minutos). Para sistemas com SLOs menos rígidos, o tempo de reação pode ser da ordem de dezenas de minutos.

Assim que um page é recebido e reconhecido, espera-se que o engenheiro de plantão faça a triagem do problema e trabalhe visando à sua solução, possivelmente envolvendo outros membros da equipe e escalando conforme for necessário.

Eventos de produção que não gerem pages, por exemplo, alertas de baixa prioridade ou releases de software, também podem ser tratados e/ou analisados pelo engenheiro de plantão durante o horário comercial. Essas atividades são menos urgentes que eventos de paging, que têm prioridade sobre quase todas as demais tarefas, incluindo trabalhos em projeto. Para mais esclarecimentos sobre interrupções e outros eventos que não geram pages, mas contribuem para a carga operacional, consulte o Capítulo 29.

Muitas equipes têm rodízios de plantão principal e secundário. A distribuição dos encargos entre o plantão principal e o plantão secundário varia de equipe para equipe. Uma equipe pode empregar o plantão secundário como um plantão de reserva para os pages que não tenham sido tratados pelo plantão principal. Outra equipe pode especificar que o plantão principal tratará somente pages, enquanto o plantão secundário tratará todas as demais atividades não urgentes de produção.

Nas equipes em que um rodízio secundário não seja estritamente necessário para a distribuição dos encargos, é comum que duas equipes relacionadas sirvam como plantão secundário uma da outra, com atribuições para servir de plantão reserva. Essa configuração elimina a necessidade de haver um rodízio de plantão secundário exclusivo.

Há muitas maneiras de organizar rodízios de plantão; para ver uma análise detalhada, consulte o capítulo “Oncall” (Plantão) de [Lim14].

Plantão equilibrado

As equipes de SRE têm restrições específicas quanto à quantidade e à

qualidade dos turnos de plantão. A quantidade de plantão pode ser calculada pelo percentual de tempo gasto pelos engenheiros com encargos de plantão. A qualidade do plantão pode ser calculada pelo número de incidentes que ocorrem durante um turno de plantão.

Os gerentes de SREs têm a responsabilidade de manter a carga de trabalho de plantão equilibrada e sustentável nesses dois eixos.

Equilíbrio quanto à quantidade

Acreditamos convictamente que o “E” em “SRE” é uma característica determinante de nossa empresa, portanto 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 outros 25% para diferentes tipos de trabalhos operacionais, não ligados a projetos.

Usando a regra de 25% de plantão, podemos calcular o número mínimo de SREs necessários para manter um rodízio de plantão 24/7. Supondo que sempre haja duas pessoas de plantão (principal e secundário, com encargos diferentes), o número mínimo de engenheiros necessário para cumprir um plantão em uma equipe em um único local (single-site) é oito: supondo turnos de uma semana de duração, cada engenheiro estará de plantão (principal ou secundário) durante uma semana a cada mês. Para equipes em duas localidades, um tamanho mínimo razoável para cada equipe é seis, tanto para cumprir a regra de 25% quanto para garantir uma massa substancial e crítica de engenheiros para a equipe.

Se um serviço implica trabalho suficiente para justificar o aumento de uma equipe em uma única localidade, preferimos criar uma equipe multissite (em várias localidades). Uma equipe multissite é vantajosa por dois motivos:

- Turnos à noite têm efeitos prejudiciais à saúde das pessoas [Dur05], e um rodízio multissite que “acompanhe o sol” permite às equipes evitar totalmente os turnos à noite.
- Limitar o número de engenheiros no rodízio de plantão garante que os engenheiros não percam o contato com os sistemas de produção (veja a seção “Um inimigo traiçoeiro: pouca carga operacional”).

Contudo, equipes multissite implicam overhead de comunicação e de coordenação. Desse modo, a decisão de ter a equipe em um só local ou em vários deve ser baseada na relação custo-benefício de cada opção, na importância do sistema e na carga de trabalho que cada sistema gera.

Equilíbrio quanto à qualidade

Para cada turno de plantão, um engenheiro deve ter tempo suficiente para lidar com quaisquer incidentes e atividades de acompanhamento (follow-up), por exemplo, escrever postmortems [Loo10]. Vamos definir um incidente como uma sequência de eventos e alertas relacionados à mesma causa-raiz e que seriam discutidos como parte do mesmo postmortem. Percebemos que, em média, lidar com as tarefas envolvidas em um incidente de plantão – análise de causas-raízes, reparação e atividades de acompanhamento, como escrever um postmortem e corrigir bugs – demora seis horas. Segue daí que o número máximo de incidentes por dia é dois por turno de 12 horas de plantão. Para permanecer dentro desse limite máximo, a distribuição de eventos de paging deve ser bem uniforme no tempo, com um valor de mediana desejável de 0: se um dado componente ou problema gerar pages todos os dias ($\text{mediana de incidentes/dia} > 1$), é provável que algo a mais apresente problema em algum ponto, causando mais incidentes do que deveria ser permitido.

Se esse limite for temporariamente ultrapassado, por exemplo, durante um trimestre, medidas corretivas devem ser implantadas para garantir que a carga operacional retorne a um estado sustentável (veja a seção “Sobrecarga operacional”, e o Capítulo 30).

Pagamentos

Um pagamento adequado deve ser considerado para suporte fora do horário comercial. Empresas diferentes tratam pagamentos por plantão de formas distintas; o Google oferece compensação de horas ou pagamentos diretos em dinheiro, limitados a alguma proporção do salário como um todo. O limite no pagamento, na prática, representa um limite na quantidade de trabalho de plantão assumido por qualquer indivíduo. Essa estrutura de pagamento

garante que haja um incentivo para se envolver com os encargos de plantão conforme exigido pela equipe, mas promove também uma distribuição equilibrada das tarefas de plantão e limita as desvantagens em potencial resultantes do trabalho de plantão excessivo, como burnout (esgotamento profissional) ou tempo inadequado para tarefas de projeto.

Sentindo-se seguro

Conforme mencionamos antes, as equipes de SRE dão suporte aos sistemas mais críticos do Google. Ser um SRE de plantão geralmente significa assumir a responsabilidade por sistemas de receita crítica, voltados a usuários, ou pela infraestrutura necessária para manter esses sistemas funcionando. A metodologia de SRE para pensar nos problemas e enfrentá-los é vital para a operação apropriada dos serviços.

Pesquisas modernas identificaram dois modos distintos de pensar, que um indivíduo pode, de forma consciente ou não, escolher quando está diante de desafios [Kah11]:

- Ação intuitiva, automática e rápida
- Funções cognitivas racionais, focadas e deliberadas

Quando alguém está lidando com interrupções de serviço relacionadas a sistemas complexos, é mais provável que a segunda opção gera resultados melhores e conduza a um tratamento de incidentes bem planejado.

Para garantir que os engenheiros tenham a mentalidade adequada para tirar proveito da última forma de pensar, é importante reduzir o estresse relacionado ao fato de estar de plantão. A importância e o impacto dos serviços e as consequências de potenciais interrupções de serviço podem gerar pressões significativas nos engenheiros de plantão, prejudicando o bem-estar de membros individuais da equipe e, possivelmente, levando os SREs a fazer escolhas incorretas que podem colocar em risco a disponibilidade do serviço. Sabe-se que os hormônios do estresse, como o cortisol e o CRH (Corticotropin-Releasing Hormone, ou Hormônio Liberador de Corticotrofina), produzem consequências comportamentais – incluindo medo – que podem prejudicar as funções cognitivas e provocar tomadas de decisões

que não sejam as ideais [Chr09].

Sob a influência desses hormônios do estresse, a abordagem cognitiva mais deliberada geralmente dá lugar a uma ação sem reflexão e consideração (porém imediata), resultando em um potencial abuso de métodos heurísticos. Usar métodos heurísticos é um comportamento muito tentador quando se está de plantão. Por exemplo, quando o mesmo alerta gera um page pela quarta vez na semana, e os três pages anteriores foram iniciados por um sistema de infraestrutura externo, é extremamente tentador ser tendencioso na confirmação, associando automaticamente essa quarta ocorrência do problema à causa anterior.

Embora intuição e reações rápidas possam parecer traços desejáveis na administração de incidentes, elas têm suas desvantagens. A intuição pode estar errada e, com frequência, é menos sustentada por dados óbvios. Desse modo, seguir a intuição pode fazer um engenheiro desperdiçar tempo acompanhando uma linha de raciocínio incorreta desde o princípio. Reações rápidas estão profundamente enraizadas no hábito, e respostas habituais não são submetidas a considerações, o que significa que podem ser desastrosas. A metodologia ideal no gerenciamento de incidentes apresenta um equilíbrio perfeito entre executar os passos no ritmo desejado quando houver dados suficientes disponíveis para tomar uma decisão razoável e, ao mesmo tempo, analisar suas pressuposições de forma crítica.

É importante que os SREs de plantão entendam que podem contar com vários recursos que deixam a experiência de estar de plantão menos assustadora do que possa parecer. Os recursos mais importantes para o plantão são:

- Caminhos claros para escalar
- Procedimentos bem definidos para gerenciamento de incidentes
- Uma cultura de postmortem sem acusações ([Loo10], [All12])

As equipes de desenvolvedores de sistemas com suporte de SREs geralmente participam do rodízio de plantão 24/7, e é sempre possível escalar para essas equipes parceiras quando for necessário. A escalação apropriada das interrupções de serviço geralmente é um princípio para reagir a interrupções sérias de serviço com dimensões desconhecidas e significativas.

Quando uma pessoa estiver tratando incidentes, se o problema for complexo o suficiente a ponto de envolver várias equipes ou se, após alguma investigação, ainda não for possível estimar um limite superior para o tempo de duração do incidente, adotar um protocolo formal de gerenciamento de incidentes pode ser útil. A SRE do Google utiliza o protocolo descrito no Capítulo 14, que oferece um conjunto de passos fáceis de seguir e bem definidos, os quais ajudam um engenheiro de plantão a procurar uma solução satisfatória a um incidente de forma racional, com toda a ajuda necessária. Esse protocolo tem suporte interno de uma ferramenta baseada em web que automatiza a maior parte das ações para gerenciamento de incidentes, como atribuir papéis e registrar e divulgar atualizações de status. Essa ferramenta permite que os gerentes de incidentes se concentrem no tratamento do incidente, em vez de gastar tempo e esforços cognitivos em ações mundanas, como formatar emails ou atualizar vários canais de comunicação de uma só vez.

Por fim, quando um incidente ocorre, é importante avaliar o que deu errado, reconhecer o que deu certo e tomar uma atitude para evitar que os mesmos erros ocorram novamente no futuro. As equipes de SRE devem escrever postmortems após incidentes significativos e criar uma linha do tempo completa e detalhada com os eventos ocorridos. Por se concentrar nos eventos e não nas pessoas, esses postmortems são muito valiosos. Em vez de colocar a culpa nos indivíduos, dados importantes são obtidos a partir da análise sistemática dos incidentes em produção. Erros acontecem e o software deve garantir que vamos cometer o mínimo possível de erros. Reconhecer oportunidades para automação é uma das melhores maneiras de evitar erros humanos [Loo10].

Evitando uma carga operacional inadequada

Conforme mencionado na seção “Plantão equilibrado” os SREs gastam no máximo 50% de seu tempo em tarefas operacionais. O que acontece se as atividades operacionais excederem esse limite?

Sobrecarga operacional

A equipe de SRE e a liderança são responsáveis por incluir objetivos concretos no planejamento das atividades trimestrais a fim de garantir que a carga de trabalho retorne a níveis sustentáveis. O empréstimo temporário de um SRE experiente a uma equipe sobre carregada, conforme discutido no Capítulo 30, pode criar espaço suficiente para respirar, de modo que a equipe possa avançar no tratamento dos problemas.

O ideal é que os sintomas da sobrecarga operacional sejam mensuráveis para que as metas possam ser quantificadas (por exemplo, número de tickets diários < 5, eventos de paging por turno < 2).

Uma monitoração com configuração incorreta é uma causa comum de sobrecarga operacional. Alertas que gerem pages devem estar alinhados com os sintomas que ameaçam os SLOs de um serviço. Deve ser possível tomar uma atitude em relação a todos os alertas que gerem pages. Alertas de baixa prioridade que incomodem o engenheiro de plantão a cada hora (ou com mais frequência) prejudicam a produtividade, e a fatiga causada por alertas desse tipo também pode fazer com que alertas sérios sejam tratados com menos atenção que o necessário. Veja o Capítulo 29 para outras discussões sobre esse assunto.

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 dar origem a vários alertas, portanto é importante controlar o número de alertas gerados, garantindo que alertas relacionados sejam agrupados pelo sistema de monitoração ou de alertas. Se, por algum motivo, alertas duplicados ou não informativos forem gerados durante um incidente, silenciá-los pode oferecer a tranquilidade necessária para que o engenheiro de plantão se concentre no incidente propriamente dito. Alertas ruidosos, que geram sistematicamente mais de um alerta por incidente, devem ser ajustados para se aproximar de uma razão de 1:1 para alertas/incidentes. Fazer isso permite ao engenheiro de plantão se concentrar no incidente, e não na triagem de alertas duplicados.

Às vezes, as alterações que causam sobrecarga operacional não estão no controle das equipes de SRE. Por exemplo, os desenvolvedores de aplicação podem introduzir mudanças que façam o sistema ser mais ruidoso, menos

confiável, ou ambos. Nesse caso, é apropriado trabalhar com os desenvolvedores da aplicação a fim de definir metas comuns para melhorar o sistema.

Em casos extremos, as equipes de SRE podem ter a opção de “devolver o pager” – o SRE pode pedir à equipe de desenvolvedores que fique exclusivamente de plantão cuidando do sistema, até que ele atinja os padrões da equipe de SRE em questão. Devolver o pager não acontece com muita frequência, pois quase sempre é possível trabalhar com a equipe de desenvolvedores a fim de reduzir a carga operacional e deixar um determinado sistema mais confiável. Em alguns casos, porém, mudanças complexas ou arquiteturais que ocorrem durante vários trimestres podem ser necessárias para deixar um sistema sustentável do ponto de vista operacional. Em casos como esses, a equipe de SRE não deve se sujeitar a uma carga operacional excessiva. Em vez disso, é apropriado negociar a reorganização das responsabilidades de plantão com a equipe de desenvolvimento, possivelmente encaminhando alguns alertas que gerem pages, ou todos, ao desenvolvedor de plantão. Uma solução como essa geralmente é uma medida temporária, durante a qual as equipes de SRE e de desenvolvedores trabalham em conjunto de modo a deixar o serviço em forma para que possa ser assumido pela equipe de SRE novamente.

A possibilidade de renegociação das responsabilidades de plantão entre as equipes de SRE e de desenvolvimento de produtos atesta o equilíbrio de forças entre as equipes.² Esse relacionamento profissional também é um exemplo de como a tensão saudável entre essas duas equipes e os valores que elas representam – confiabilidade *versus* rapidez no lançamento de novas funcionalidades – geralmente é resolvida com benefícios profundos ao serviço e, por extensão, à empresa como um todo.

Um inimigo traiçoeiro: pouca carga operacional

Estar de plantão em um sistema tranquilo é uma bênção, mas o que acontecerá se o sistema for tranquilo demais ou quando os SREs não ficarem de plantão com uma frequência suficiente? Uma falta de carga operacional é indesejável para uma equipe de SRE. Deixar de estar em contato com a

produção por longos períodos de tempo pode resultar em problemas de confiança, tanto em termos de excesso quanto de falta dela, ao mesmo tempo que lacunas no conhecimento são descobertas somente quando um incidente ocorrer.

Para combater essa eventualidade, as equipes de SRE devem ser dimensionadas de modo a permitir que todo engenheiro esteja de plantão pelo menos uma ou duas vezes em um trimestre, garantindo assim que cada membro da equipe seja suficientemente exposto ao ambiente de produção. Os exercícios de “Wheel of Misfortune” (Roda do Azar), discutidos no Capítulo 28, também são atividades úteis para a equipe, e podem ajudar a aperfeiçoar e a melhorar as habilidades de resolução de problemas e o conhecimento do serviço. O Google também tem um evento anual de recuperação de desastres que envolve a empresa toda – o DiRT (Disaster Recovery Training, ou Treinamento para Recuperação de Desastres) – e que combina treinamentos teóricos e práticos para realizar testes de vários dias em sistemas de infraestrutura e em serviços individuais; veja [Kri12].

Conclusões

A abordagem para plantão descrita neste capítulo serve como um guia para todas as equipes de SRE do Google e é essencial para promover um ambiente de trabalho sustentável e administrável. A abordagem do Google para o plantão nos permitiu usar o trabalho de engenharia como o principal meio de ter escala quanto às responsabilidades em produção e manter um alto nível de confiabilidade e de disponibilidade, apesar da complexidade e do número de sistemas e serviços cada vez maiores pelos quais os SREs são responsáveis.

Embora essa abordagem talvez não seja imediatamente aplicável a todos os contextos em que os engenheiros precisem estar de plantão em serviços de TI, acreditamos que ela representa um modelo sólido que as empresas podem adotar ao escalar a fim de atender a um volume crescente de atividades de plantão.

¹ Uma versão anterior deste capítulo foi publicada como um artigo em *login*: (outubro de 2015, vol. 40, nº 5).

² Para mais discussões sobre a tensão natural entre as equipes de SRE e de desenvolvimento de

produtos, veja o Capítulo 1.

CAPÍTULO 12

Resolvendo problemas de modo eficiente

Escrito por Chris Jones

Saiba que ser um expert é mais que entender como um sistema deve funcionar. O expertise é adquirido ao investigar por que um sistema não funciona.

— Brian Redman

Modos pelos quais algo dá certo são casos especiais dos modos pelos quais eles dão errado.

— John Allspaw

A resolução de problemas é uma habilidade crucial para qualquer pessoa que opere sistemas de processamento distribuído – em especial, para os SREs –, mas, com frequência, ela é vista como uma habilidade inata, que algumas pessoas têm e outras não. Um motivo para essa suposição é que, para aqueles que resolvem problemas com frequência, é um processo enraizado; explicar *como* resolver problemas é difícil: seria como explicar como andar de bicicleta. Todavia, acreditamos que resolver problemas é algo que pode ser aprendido e ensinado.

Os iniciantes muitas vezes tropeçam na resolução de problemas porque o ideal é que o exercício dependa de dois fatores: uma compreensão de como resolver problemas de modo geral (isto é, sem o conhecimento de qualquer sistema em particular) e de um sólido conhecimento do sistema. Embora você possa investigar um problema usando somente o processo genérico e fazendo inferências a partir de princípios básicos¹, normalmente achamos que essa abordagem é menos eficiente e menos eficaz do que entender como um sistema deve funcionar. A falta de conhecimento do sistema geralmente limita a eficiência de um novo SRE; há poucos substitutos para conhecer como o sistema está projetado e como foi construído.

Vamos analisar um modelo geral do processo de resolução de problemas. Os leitores especializados na resolução de problemas poderão ter objeções quanto às nossas definições e ao processo; se seu método for eficaz para você, não há motivos para deixar de usá-lo.

Teoria

Formalmente, podemos pensar no processo de resolução de problemas como uma aplicação do método hipotético-dedutivo:² dado um conjunto de observações sobre um sistema e uma base teórica para compreender o seu comportamento, criamos hipóteses sobre possíveis causas de falhas de forma iterativa e procuramos testar essas hipóteses.

Em um modelo idealizado como o da Figura 12.1, começariamos com um relato do problema que nos diria que algo está errado no sistema. Então, podemos observar os dados de telemetria³ do sistema e os logs a fim de entender o seu estado atual. Essas informações, em conjunto com o nosso conhecimento de como o sistema está construído, como ele deve funcionar e seus modos de falha, nos permitem identificar algumas causas possíveis.

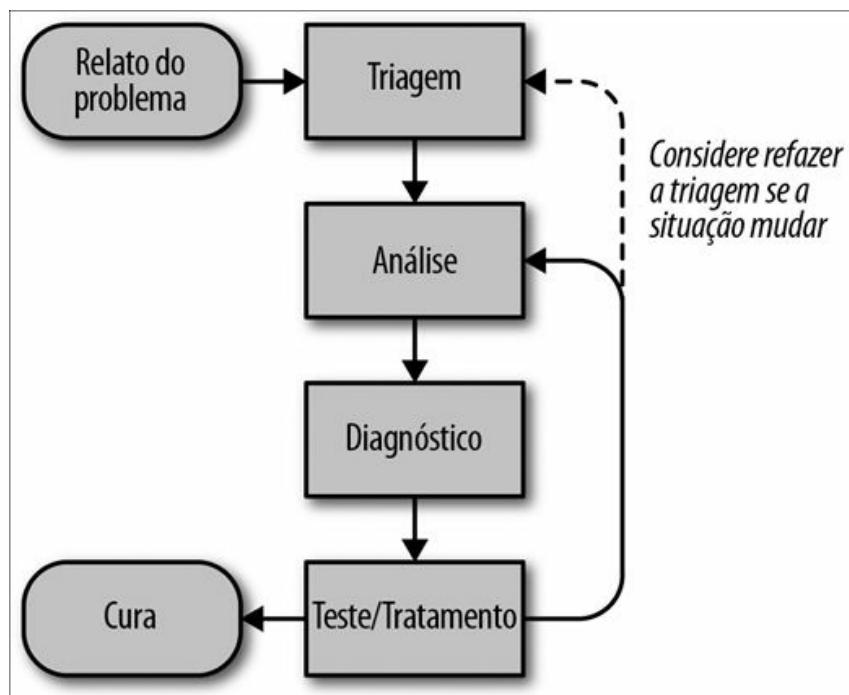


Figura 12.1 – Um processo para resolução de problemas.

Então, podemos testar nossas hipóteses por meio de uma de duas maneiras. Podemos comparar o estado observado do sistema em relação às nossas teorias a fim de encontrar evidências para confirmá-las ou não. Ou, em alguns casos, podemos “tratar”ativamente o sistema – isto é, mudá-lo de forma controlada – e observar o resultado. Essa segunda abordagem melhora nossa compreensão do estado do sistema e da(s) possível(is) causa(s) dos problemas relatados. Ao usar uma dessas estratégias, testamos repetidamente até que uma causa-raiz seja identificada; nesse ponto, podemos então realizar uma ação corretiva a fim de evitar uma nova ocorrência e escrevemos um postmortem. É claro que não é preciso esperar pela análise de causas-raízes ou pela escrita do postmortem para corrigir a(s) causa(s) imediata(s).

Armadilhas comuns

Sessões ineficientes de resolução de problemas estão sujeitas a transtornos nos passos de Triagem, Análise e Diagnóstico, muitas vezes por falta de uma compreensão profunda do sistema. A seguir, apresentamos as armadilhas comuns a serem evitadas:

- Olhar para sintomas que não sejam relevantes ou compreender erroneamente o significado das métricas do sistema. Com frequência, isso resulta em uma perseguição inútil do problema.
- Não compreender a forma de modificar o sistema, suas entradas ou seu ambiente para testar as hipóteses de forma segura e eficiente.
- Criar teorias extremamente improváveis sobre o que está errado ou ater-se a causas de problemas passados, raciocinando que, se eles ocorreram uma vez, devem estar acontecendo novamente.
- Ir atrás de correlações espúrias que, na verdade, são coincidências ou correlações com causas compartilhadas.

Corrigir a primeira e a segunda armadilha comuns é uma questão de conhecer o sistema e adquirir experiência com os padrões comuns utilizados em sistemas distribuídos. A terceira armadilha é um conjunto de falácias lógicas que pode ser evitado lembrando que nem todas as falhas são igualmente prováveis – como é ensinado aos médicos, “quando ouvir o som de cascos, pense em cavalos, e não em zebras”.⁴ Lembre-se

também de que, considerando que todas as demais variáveis sejam iguais, devemos dar preferência às explicações mais simples.⁵

Por fim, devemos lembrar que correlação não é uma relação de causa e efeito:⁶ alguns eventos correlacionados, por exemplo, perda de pacotes em um cluster e discos rígidos com falha no cluster, compartilham causas comuns – nesse caso, uma queda de energia elétrica, embora a falha de rede claramente não provoque falhas no disco rígido nem vice-versa. Pior ainda, à medida que os sistemas crescem em tamanho e complexidade, e mais métricas são monitoradas, é inevitável que haja eventos que, por acaso, estejam bem correlacionados a outros, puramente por coincidência.⁷

Entender as falhas em nosso processo de raciocínio é o primeiro passo para evitá-las e se tornar mais eficaz na resolução de problemas. Uma abordagem metódica para saber o que sabemos, o que não sabemos e o que devemos saber facilita descobrir, de forma mais simples, o que deu errado e como corrigir isso.

Na prática

Na prática, é claro, a resolução de problemas jamais é tão limpa quanto nosso modelo idealizado sugere. Há alguns passos que podem deixar o processo menos doloroso e mais produtivo, tanto para aqueles que estão vivenciando os problemas do sistema quanto para aqueles que estão respondendo a eles.

Relato do problema

Todo problema começa com um relato, que pode ser um alerta automatizado ou um de seus colegas dizendo que “o sistema está lento”. Um relato eficaz deve informar o comportamento *esperado*, o comportamento *propriamente dito* e, se for possível, como reproduzir esse comportamento.⁸ O ideal é que os relatos tenham um formato consistente e sejam armazenados em um local em que seja possível fazer pesquisas, por exemplo, em um sistema de monitoração de bugs. Em nossa empresa, nossas equipes geralmente têm formulários personalizados ou pequenas aplicações web que solicitam as

informações relevantes ao diagnóstico dos sistemas em particular para os quais eles dão suporte; então, um bug é gerado e encaminhado de modo automático. Esse também pode ser um bom ponto para oferecer ferramentas àqueles que relatam os problemas, de modo que eles possam diagnosticar ou solucionar problemas comuns por conta própria.

No Google, abrir um bug para cada problema é uma prática comum, mesmo para aqueles recebidos por email ou por mensagens instantâneas. Fazer isso cria um log de investigação e de atividades de reparação que pode ser consultado no futuro. Muitas equipes não incentivam relatar os problemas diretamente a uma pessoa por diversos motivos: essa prática introduz um passo adicional, que é transcrever o relato em um bug, produz relatórios de baixa qualidade que não são visíveis a outros membros da equipe e tende a concentrar a carga da resolução de problemas em alguns membros da equipe que, por acaso, são conhecidos daqueles que relatam os bugs, e não na pessoa responsável no momento (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: sua monitoração caixa-preta não foi capaz de encontrar os resultados de pesquisa para “the forms of things unknown” nos últimos cinco minutos. O sistema de alerta registrou um bug – com links para os resultados recentes do prober (ferramenta de sondagem) caixa-preta e para a entrada no manual de regras para esse alerta – e o atribuiu a você. É hora de entrar em ação!

Triagem

Após receber um relato de problema, o próximo passo é descobrir o que fazer a seu respeito. Os problemas podem variar quanto à severidade: um problema pode afetar apenas um usuário em circunstâncias bem específicas (e pode ser que haja uma solução para contornar a situação), ou pode implicar uma interrupção global completa de um serviço. Sua resposta deve ser apropriada ao impacto do problema: é apropriado declarar uma emergência do tipo “todos a postos” no último caso (veja o Capítulo 14), mas fazer isso no primeiro seria um exagero. Avaliar a severidade de um problema exige um

exercício de bom discernimento de engenharia e, com frequência, um grau de calma sob pressão.

Sua primeira resposta em uma interrupção de serviço importante pode ser começar a resolver o problema e tentar descobrir uma causa-raiz o mais rápido possível. Ignore esse instinto!

Em vez disso, seu curso de ação deve ser *deixar o sistema funcionar da melhor maneira possível nessas circunstâncias*. Isso pode implicar opções de emergência, como desviar o tráfego de um cluster com falhas para outros que ainda estejam funcionando, descartar o tráfego como um todo a fim de evitar uma falha em cascata ou desabilitar subsistemas para diminuir a carga. Estancar o sangramento deve ser a sua primeira prioridade; você não ajudará seus usuários se o sistema morrer enquanto você tenta identificar as causas-raízes. É claro que uma ênfase na triagem rápida não impede a execução de passos para guardar evidências do que está errado, como ativar logs, para ajudar na subsequente análise de causas-raízes.

Pilotos iniciantes aprendem que sua primeira responsabilidade em uma emergência é conduzir a aeronave [Gaw09]; a resolução de problemas é secundária em relação a fazer o avião e todos que estiverem a bordo pousarem de forma *segura*. Essa abordagem também pode ser aplicada aos sistemas de computação: por exemplo, se um bug estiver levando a uma corrupção de dados possivelmente irrecuperável, congelar o sistema a fim de evitar mais falhas pode ser melhor do que deixar que esse comportamento seja mantido.

Essa percepção muitas vezes é bastante perturbadora e contraintuitiva para novos SREs, em particular para aqueles cuja experiência anterior tenha sido em empresas de desenvolvimento de produtos.

Análise

Precisamos ser capazes de analisar o que cada componente do sistema está fazendo para entender se ele está ou não se comportando da forma correta.

O ideal é que um sistema de monitoração esteja registrando métricas para seu sistema, conforme discutido no Capítulo 10. Essas métricas são um bom lugar para começar a descobrir o que está errado. Colocar séries temporais

em gráficos e realizar operações nessas séries pode ser uma maneira eficaz de entender o comportamento de partes específicas de um sistema e identificar correlações que possam sugerir em que ponto os problemas começaram.⁹

O logging é outra ferramenta de valor inestimável. Exportar informações sobre cada operação e sobre o estado do sistema possibilita entender o que um processo estava fazendo exatamente em um dado ponto no tempo. Talvez seja necessário analisar os logs de um ou de vários processos do sistema. Rastrear requisições pela pilha toda usando ferramentas como o Dapper [Sig10] é uma maneira bem eficaz de entender como um sistema distribuído está funcionando, embora casos de uso variados impliquem designs significativamente diferentes de rastreamento [Sam14].

Logging

Logs textuais são muito convenientes para depuração reativa em tempo real, enquanto armazenar logs em um formato binário estruturado possibilita a construção de ferramentas para conduzir análises retrospectivas com muito mais informações.

É realmente útil ter vários níveis de verbosidade disponíveis, juntamente com uma maneira de aumentar esses níveis durante a execução. Essa funcionalidade permite analisar qualquer operação, ou todas elas, com um nível impressionante de detalhes, sem precisar reiniciar seu processo, ao mesmo tempo que permite reduzir os níveis de verbosidade quando seu serviço estiver operando normalmente. Conforme o volume de tráfego recebido pelo seu serviço, usar amostragens estatísticas pode ser melhor; por exemplo, você pode mostrar uma operação a cada mil.

Um próximo passo é incluir uma linguagem de seleção para que você possa dizer “mostre-me as operações que correspondam a X” para uma grande variedade de X – por exemplo, RPCs Set com um tamanho de payload menor que 1.024 bytes ou operações que demoraram mais de dez milissegundos para retornar ou chamaram `doSomethingInteresting()` em `rpc_handler.py`. Você pode até mesmo fazer o design de sua infraestrutura de logging para que seja possível ativá-la conforme for necessário, de forma rápida e seletiva.

Expor o estado atual é o terceiro truque em nossa caixa de ferramentas. Por exemplo, os servidores do Google têm endpoints que exibem uma amostra das RPCs enviadas e recebidas recentemente, portanto é possível entender de que modo um servidor específico está se comunicando com outros sem consultar um diagrama de arquitetura. Esses endpoints também mostram histogramas de taxas de erro e de latência para cada tipo de RPC, de modo que é possível dizer rapidamente o que não está saudável. Alguns sistemas têm endpoints que mostram sua configuração atual ou permitem fazer uma análise de seus dados; por exemplo, os servidores do Borgmon do Google (Capítulo 10) podem exibir as regras de monitoração que estão usando e permitem até mesmo rastrear um processamento em particular, passo a passo, até as métricas originais a partir das quais um valor foi calculado.

Por fim, talvez você precise instrumentar até mesmo um cliente para fazer experimentos e descobrir o que um componente está devolvendo em resposta às requisições.

Depurando o Shakespeare

Utilizando o link para os resultados da monitoração caixa-preta no bug, você descobre que o prober envia uma requisição HTTP GET ao endpoint `/api/search`:

```
{  
  "search_text": "the forms of things unknown"  
}
```

Ele espera receber uma resposta com um código HTTP 200 e um payload JSON que corresponda exatamente a:

```
[{  
  "work": "A Midsummer Night's Dream",  
  "act": 5,  
  "scene": 1,  
  "line": 2526,  
  "speaker": "Theseus"  
}]
```

O sistema está configurado para enviar uma sondagem a cada minuto; nos últimos dez minutos, aproximadamente metade das sondagens foi bem-sucedida, embora sem um padrão perceptível. Infelizmente, o prober não

mostra *o que* foi devolvido quando falhou; você toma nota para corrigir isso no futuro.

Usando o `curl`, você envia requisições manualmente para o endpoint de pesquisa e obtém uma resposta com falha, com código de resposta HTTP 502 (Bad Gateway) e sem payload. A resposta tem um cabeçalho HTTP `X-Request-Trace` que lista os endereços dos servidores de backend responsáveis pela resposta a essa requisição. Com essa informação, você pode agora analisar esses backends a fim de testar se eles estão respondendo de forma apropriada.

Diagnóstico

Uma compreensão completa do design do sistema definitivamente é útil para criar hipóteses plausíveis sobre o que deu errado, mas há também algumas práticas genéricas que ajudarão, mesmo sem o conhecimento do domínio.

Simplificar e reduzir

O ideal é que componentes de um sistema tenham interfaces bem definidas e façam transformações conhecidas de suas entradas para suas saídas (em nosso exemplo, dado um texto de pesquisa como entrada, um componente pode devolver uma saída contendo possíveis correspondências). Então, é possível observar as conexões *entre* os componentes – ou, de modo equivalente, os dados que fluem entre eles – a fim de determinar se um dado componente está funcionando de forma apropriada. Injetar dados conhecidos de teste para verificar se a saída resultante é a saída esperada (uma forma de teste caixa-preta) a cada passo pode ser especialmente eficaz, assim como injetar dados cujo propósito seja sondar possíveis causas de erros. Ter um caso de teste sólido e reproduzível torna a depuração mais rápida e é possível usar esse caso de teste em um ambiente que não seja de produção, para o qual técnicas mais invasivas e arriscadas estão disponíveis, e que não seriam possíveis em produção.

Dividir e conquistar é uma técnica de solução de propósito geral muito útil. Em um sistema multicamadas, em que o trabalho é feito em uma pilha de componentes, geralmente é melhor começar de modo sistemático, partindo de

uma das extremidades da pilha, e trabalhar em direção à outra extremidade, analisando um componente de cada vez. Essa estratégia também é bastante adequada para uso em pipelines de processamento de dados. Em sistemas excepcionalmente grandes, proceder de forma linear pode demorar muito; uma alternativa, a *bissecção*, separa o sistema na metade e analisa os caminhos de comunicação entre os componentes em um lado e no outro. Após determinar se uma metade parece estar funcionando de modo apropriado, repita o processo até que reste um possível componente com falha.

Pergunte “o quê”, “onde” e “por quê”

Um sistema que não funcione bem muitas vezes ainda está tentando fazer *algo*, mas simplesmente não é aquilo que você quer que ele faça. Descobrir *o que* ele está fazendo e então perguntar *por que* o sistema está fazendo isso e *onde* seus recursos estão sendo usados ou para onde sua saída está indo pode ajudar a entender como o sistema começou a dar errado.¹⁰

Detalhando as causas de um sintoma

Sintoma: Um cluster Spanner tem alta latência e as RPCs para seus servidores estão sofrendo timeout.

Por quê? As tarefas do servidor Spanner estão usando todo o seu tempo de CPU e não conseguem tratar todas as requisições enviadas pelos clientes.

Onde, no servidor, o tempo de CPU está sendo usado? Gerar o perfil do servidor mostra que ele está ordenando entradas em logs descarregados em disco.

Onde, no código de ordenação de log, a CPU está sendo usada? Ao avaliar uma expressão regular em relação a paths para arquivos de log.

Soluções: Reescrever a expressão regular para evitar backtracking. Procure padrões semelhantes na base de código. Considere o uso de RE2, que não faz backtrack e garante um aumento linear do tempo de execução de acordo com o tamanho da entrada.¹¹

Quem mexeu por último

Os sistemas têm inércia: descobrimos que um sistema de computação em funcionamento tende a permanecer em movimento até que uma força externa atue sobre ele, por exemplo, uma mudança de configuração ou no tipo de carga servida. Mudanças recentes em um sistema podem ser um local produtivo para começar a identificar o que está errado.¹²

Sistemas bem projetados devem ter um logging de produção extenso para rastrear novas implantações de versão e mudanças de configuração em todas as camadas da pilha, desde os binários do servidor tratando tráfego de usuários até os pacotes instalados em nós individuais do cluster. Correlacionar mudanças no desempenho e no comportamento de um sistema com outros eventos do sistema e do ambiente também pode ser útil na construção de painéis para monitoração; por exemplo, você pode inserir anotações contendo o horário inicial e final da implantação de uma nova versão em um gráfico das taxas de erro do sistema, como mostra a Figura 12.2.

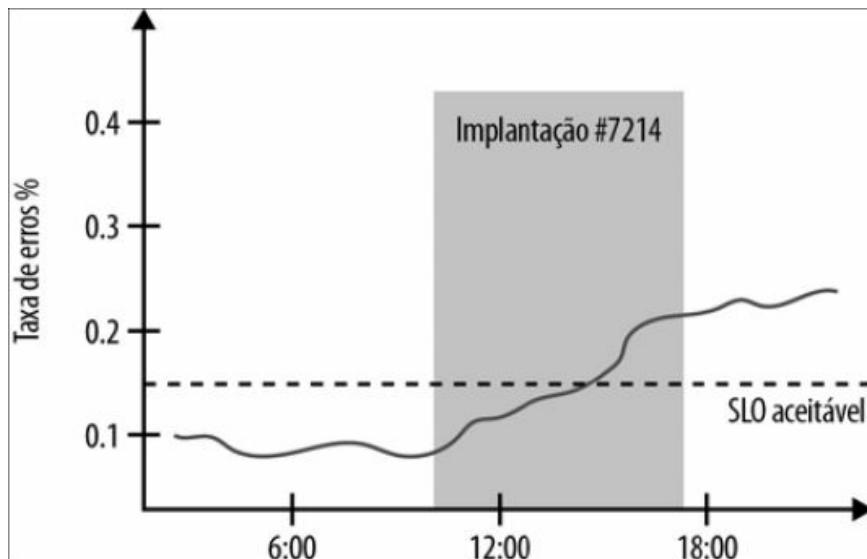


Figura 12.2 – Taxas de erros em um gráfico que mostra os horários inicial e final de implantação.

Enviar uma requisição manualmente para o endpoint /api/search (veja a caixa de texto “Depurando o Shakespeare”) e ver a falha listando os servidores de backend que trataram a resposta permite que você descarte a possibilidade de o problema ser no servidor da API de frontend e com os

distribuidores de carga: a resposta provavelmente não teria incluído essa informação se a requisição não tivesse pelo menos chegado até os backends de pesquisa e falhado ali. Agora você pode concentrar seus esforços nos backends – analisar seus logs, enviar consultas de teste para ver quais respostas são devolvidas e analisar as métricas exportadas.

Diagnósticos específicos

Apesar de as ferramentas genéricas descritas antes serem úteis em vários domínios de problemas, é provável que você ache útil construir ferramentas e sistemas para ajudar no diagnóstico de seus serviços em particular. Os SREs do Google investem boa parte de seu tempo construindo ferramentas desse tipo. Embora muitas dessas ferramentas sejam necessariamente específicas a um dado sistema, não se esqueça de procurar pontos comuns entre serviços e equipes para evitar duplicação de esforços.

Teste e tratamento

Depois que você tiver uma pequena lista de possíveis causas, é hora de tentar descobrir *qual* fator está na raiz do problema propriamente dito. Usando o método experimental, podemos tentar comprovar ou descartar nossas hipóteses. Por exemplo, suponha que achemos que um problema seja causado por uma falha de rede entre o servidor da lógica da aplicação e um servidor de banco de dados, ou pelo fato de o banco de dados estar recusando conexões. Tentar se conectar com o banco de dados usando as mesmas credenciais utilizadas pelo servidor da lógica da aplicação pode refutar a segunda hipótese, enquanto fazer um ping no servidor de banco de dados pode refutar a primeira, dependendo da topologia da rede, das regras de firewall e de outros fatores. Seguir o código e tentar imitar seu fluxo passo a passo pode apontar exatamente para o que está errado.

Há várias considerações para ter em mente ao fazer o design dos testes (que podem ser tão simples quanto enviar um ping ou tão complicados quanto remover o tráfego de um cluster e injetar requisições especialmente compostas para identificar uma condição de concorrência):

- Um teste ideal deve ter alternativas mutuamente exclusivas para que possa

aceitar um grupo de hipóteses e descartar outro conjunto. Na prática, isso pode ser difícil de conseguir.

- Considere o óbvio primeiro: realize os testes em ordem decrescente de probabilidade, considerando possíveis riscos ao sistema causados pelo teste. É provável que faça mais sentido testar problemas de conectividade de rede entre duas máquinas antes de ver se uma mudança recente de configuração removeu o acesso de um usuário à segunda máquina.
- Um experimento pode fornecer resultados enganosos devido a fatores que podem causar confusão. Por exemplo, uma regra de firewall pode permitir acesso somente a partir de um endereço IP específico, o que pode fazer o ping no banco de dados a partir de sua estação de trabalho falhar, mesmo que o ping feito a partir da máquina em que está o servidor da lógica da aplicação seja bem-sucedido.
- Testes ativos podem ter efeitos colaterais que mudam o resultado de testes futuros. Por exemplo, permitir que um processo use mais CPU pode deixar as operações mais rápidas, mas pode aumentar a probabilidade de haver concorrência de dados. De modo semelhante, ativar um logging verboso pode piorar ainda mais um problema de latência e confundir seus resultados: o problema está piorando por si só ou é por causa do logging?
- Alguns testes podem não ser conclusivos, mas apenas sugestivos. Pode ser muito difícil fazer condições de concorrência ou deadlocks ocorrerem na hora certa e de maneira reproduzível; desse modo, talvez você precise se contentar com evidências menos certeiras de que essas sejam as causas.

Tome notas claras das ideias que você teve, quais testes executou e os resultados vistos.¹³ Em particular, quando estiver lidando com casos mais complicados e incomuns, essa documentação pode ser essencial para ajudar você a se lembrar do que aconteceu exatamente e evitar a necessidade de repetir esses passos. Se você realizou testes ativos modificando um sistema – por exemplo, fornecendo mais recursos a um processo –, fazer mudanças de forma sistemática e documentada ajudará você a retornar o sistema à configuração anterior ao teste, em vez de executá-lo com uma configuração caótica e desconhecida.

Resultados negativos são mágicos

Escrito por Randall Bosetti

Editado por Joan Wendt

Um resultado “negativo” é um resultado experimental em que o efeito esperado está ausente – isto é, qualquer experimento que não funcionou conforme planejado. Isso inclui novos designs, métodos heurísticos ou processos humanos que não tiveram êxito em melhorar os sistemas que estavam substituindo.

Resultados negativos não devem ser ignorados nem descartados. Perceber que você está errado é muito importante: um resultado negativo claro é capaz de resolver algumas das questões mais difíceis de design. Com frequência, uma equipe tem dois designs aparentemente razoáveis, mas o progresso em uma direção deve abordar questões vagas e especulativas sobre o fato de a outra direção poder ser melhor.

Experimentos com resultados negativos são conclusivos. Eles nos dizem algo certo sobre a produção, o espaço de design ou os limites de desempenho de um sistema existente. Esses experimentos podem ajudar outras pessoas a determinar se vale a pena investir em seus próprios experimentos ou designs. Por exemplo, uma dada equipe de desenvolvimento pode decidir contra o uso de um servidor web em particular porque ele é capaz de tratar apenas cerca de 800 conexões das 8.000 conexões necessárias, antes de falhar por causa de contenção de locks. Quando uma equipe de desenvolvimento subsequente decide avaliar servidores web, em vez de começar do zero, ela pode usar esse resultado negativo já bem documentado como ponto de partida para decidir rapidamente se (a) precisa de menos de 800 conexões ou (b) os problemas de contenção de locks foram resolvidos.

Mesmo quando resultados negativos não se aplicam diretamente ao experimento de outra pessoa, os dados suplementares coletados podem ajudar os outros a escolher novos experimentos ou evitar armadilhas em designs anteriores. Microbenchmarks, antipadrões documentados e postmortems de projetos se enquadram nessa categoria. Você deve considerar o escopo do resultado negativo ao fazer o design de um experimento, pois um resultado

negativo amplo ou especialmente robusto ajudará mais ainda os seus colegas.

Ferramentas e métodos podem ter vida mais longa que o experimento e servir de base para trabalhos futuros. Como exemplo, ferramentas de benchmarking e geradores de carga podem resultar tanto de um experimento que descartou uma hipótese quanto de um que deu suporte a ela. Muitos webmasters já se beneficiaram do trabalho difícil e orientado a detalhes que resultou no Apache Bench – um teste de carga de servidores web –, apesar de seus primeiros resultados provavelmente terem sido decepcionantes.

Desenvolver ferramentas para experimentos repetíveis pode trazer benefícios indiretos também: embora uma aplicação que você crie possa não se beneficiar com o fato de ter seu banco de dados em SSDs ou com a criação de índices para chaves densas, a próxima aplicação poderá desfrutar desses benefícios. Escrever um script que permita testar facilmente essas mudanças de configuração garante que você não se esquecerá e nem deixará de fazer otimizações em seu próximo projeto.

Publicar resultados negativos melhora a cultura orientada a dados de nosso mercado. Levar em consideração resultados negativos e a insignificância estatística reduz o viés de nossas métricas e fornece um exemplo a outras pessoas de como aceitar a incerteza de forma madura. Ao publicar tudo, você incentiva outras pessoas a fazer o mesmo, e todos no mercado aprendem de forma coletiva e muito mais rápida. A SRE já aprendeu essa lição com postmortems de alta qualidade, que têm exercido um grande efeito positivo na estabilidade do ambiente de produção.

Publique seus resultados. Se você estiver interessado nos resultados de um experimento, há uma boa chance de outras pessoas também estarem. Ao publicar os resultados, essas pessoas não precisarão fazer o design nem executar um experimento semelhante por conta própria. É tentador e comum evitar o relato de resultados negativos, pois é fácil perceber que o experimento “falhou”. Alguns experimentos estão condenados e tendem a ser detectados na revisão. Muitos outros simplesmente não são relatados porque as pessoas acreditam erroneamente que resultados negativos não representam progresso.

Faça sua parte divulgando a todos quais foram os designs, os algoritmos e os

fluxos de trabalho de equipe que você descartou. Incentive seus colegas a reconhecer que os resultados negativos fazem parte do risco calculado assumido e que todo experimento bem projetado tem seu mérito. Seja cético em relação a qualquer documento de design, análise de desempenho ou artigo que não mencione falhas. Um documento desse tipo possivelmente está intensamente filtrado ou o(a) autor(a) não foi tão rigoroso(a) em seus métodos.

Acima de tudo, publique os resultados que você achar surpreendentes para que outras pessoas – inclusive você mesmo no futuro – não fiquem surpresas.

Cura

De modo ideal, agora você deve ter restringido o conjunto de possíveis causas a uma. Em seguida, queremos provar que essa é a verdadeira causa. Provar de forma conclusiva que um dado fator *causou* um problema – reproduzindo-o conforme desejado – pode ser difícil em sistemas de produção; com frequência, poderemos identificar apenas fatores causais *prováveis*, pelos seguintes motivos:

- *Os sistemas são complexos.* É bem provável que haja vários fatores, em que cada um, individualmente, não seja a causa, mas que, se forem tomados em conjunto, são as causas.¹⁴ Sistemas de verdade muitas vezes também são dependentes de caminhos, de modo que precisam estar em um estado específico para que uma falha ocorra.
- *Reproduzir o problema em um sistema ativo em produção pode não ser uma opção,* seja por causa da complexidade de levar o sistema a um estado em que a falha possa ser acionada ou porque um downtime adicional talvez seja inaceitável. Ter um ambiente que não seja de produção pode atenuar esses desafios, porém ao custo de ter outra cópia do sistema para executar.

Depois de ter identificado os fatores que causaram o problema, é hora de tomar notas do que deu errado com o sistema, como você rastreou o problema, como o corrigiu e de que modo evitar que ele ocorra novamente. Em outras palavras, você precisa escrever um postmortem (embora o ideal é que o sistema esteja vivo a essa altura!).

Estudo de caso

A App Engine¹⁵, parte da Cloud Platform do Google, é um produto de plataforma como serviço (platform-as-a-service) que permite aos desenvolvedores construir serviços sobre a infraestrutura do Google. Um de nossos clientes internos registrou um relatório de problema informando que, recentemente, havia visto um intenso aumento de latência, de uso de CPU e no número de processos em execução necessários para tratar o tráfego de sua aplicação: um sistema de gerenciamento de conteúdo usado para criar documentação para os desenvolvedores.¹⁶ O cliente não foi capaz de identificar nenhuma mudança recente em seu código que estivesse correlacionada ao aumento de recursos, e não havia ocorrido nenhum aumento de tráfego em sua aplicação (veja a Figura 12.3), de modo que estava se perguntando se uma mudança no serviço App Engine não seria a responsável por isso.

Nossa investigação descobriu que a latência havia realmente aumentado aproximadamente em uma ordem de magnitude (como mostra a Figura 12.4). Ao mesmo tempo, a quantidade de tempo de CPU (Figura 12.5) e o número de processos que estavam atendendo o serviço (Figura 12.6) haviam quase quadruplicado. Era evidente que algo estava errado. Era hora de iniciar o processo de resolução de problemas.

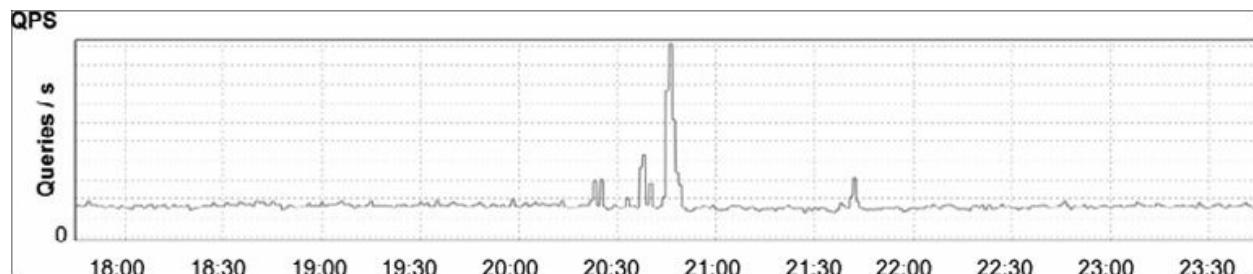


Figura 12.3 – Requisições recebidas por segundo pela aplicação, mostrando um rápido pico e uma volta à normalidade.

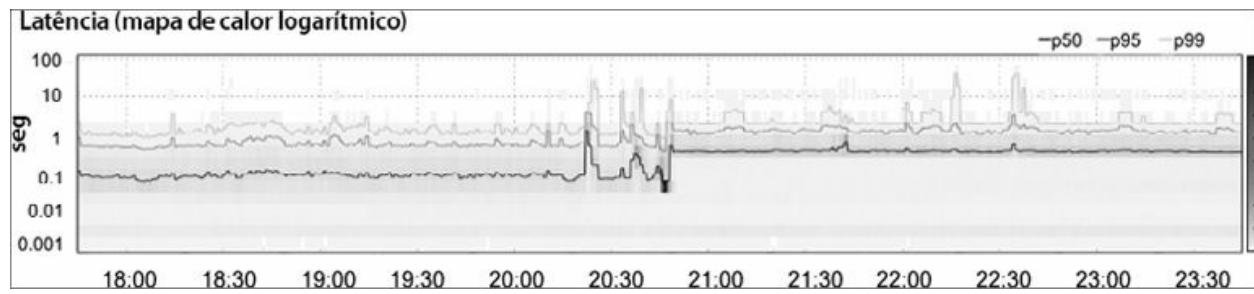


Figura 12.4 – Latência da aplicação, mostrando os 50º, 95º e 99º percentis (linhas) com um mapa de calor (heatmap) exibindo quantas requisições se enquadram em um dado conjunto de latências em qualquer ponto no tempo (sombreado).

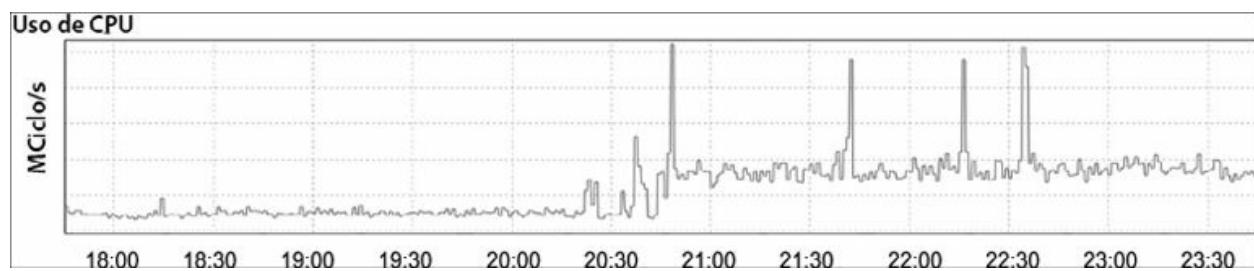


Figura 12.5 – Uso agregado de CPU para a aplicação.

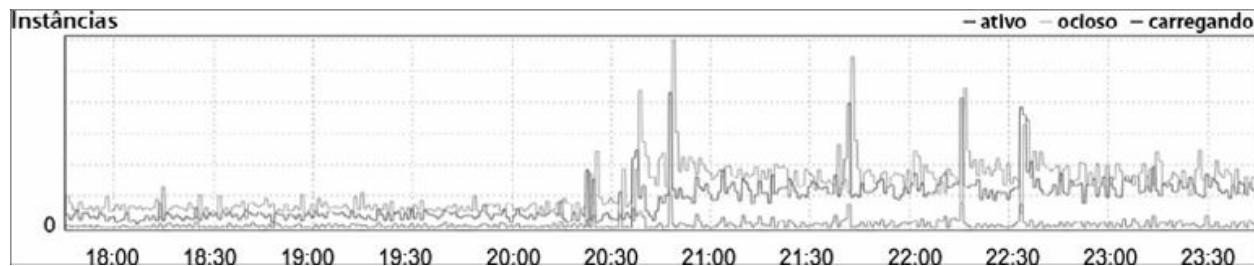


Figura 12.6 – Número de instâncias para a aplicação.

Geralmente, um aumento repentino de latência e de uso de recursos indica um aumento no tráfego enviado ao sistema ou uma mudança em sua configuração. No entanto, podíamos facilmente excluir essas possíveis causas: embora um pico no tráfego para a aplicação aproximadamente às 20:45 pudesse explicar um aumento rápido no uso de recursos, esperaríamos que o tráfego voltasse à linha de base logo depois que o volume de requisições se normalizasse. Esse pico certamente não deveria ter continuado por vários dias, começando no momento em que os desenvolvedores da aplicação registraram o relatório e, desse modo, começamos a analisar o problema. Em segundo lugar, a mudança no desempenho ocorreu em um

sábado, quando nenhuma mudança na aplicação nem no ambiente de produção havia ocorrido. As atualizações mais recentes de código e de configuração haviam sido concluídas dias antes. Além do mais, se o problema tivesse origem no serviço, esperaríamos ver efeitos semelhantes em outras aplicações que estivessem usando a mesma infraestrutura. No entanto, nenhuma outra aplicação teve efeitos semelhantes.

Enviamos o relato do problema às nossas contrapartidas, os desenvolvedores da App Engine, para investigar se o cliente estava diante de alguma idiossincrasia na infraestrutura que oferecia o serviço. Os desenvolvedores tampouco puderam encontrar algo incomum. No entanto, um desenvolvedor percebeu uma correlação entre o aumento na latência e o aumento na chamada de uma API específica para armazenagem de dados, `merge_join`, que com frequência indica uma indexação abaixo da ideal na leitura do banco de dados. Acrescentar um índice composto nas propriedades que a aplicação usa para selecionar objetos do banco de dados agilizaria essas requisições e, em princípio, aumentaria a velocidade da aplicação como um todo – mas precisaríamos descobrir *quais* propriedades precisavam de indexação. Uma olhada rápida no código da aplicação não revelou nenhum suspeito óbvio.

Era hora de lançar mão da artilharia pesada de nosso kit de ferramentas: usando o Dapper [Sig10], rastreamos os passos dados pelas requisições HTTP individuais – desde sua recepção por um proxy reverso no frontend até o ponto em que o código da aplicação devolia uma resposta – e observamos as RPCs feitas por cada servidor envolvido no tratamento dessa requisição. Fazer isso nos permitiria ver quais propriedades estavam incluídas nas requisições ao banco de dados e então criar os índices apropriados.

Enquanto investigávamos, descobrimos que as requisições para conteúdo estático, por exemplo, imagens, que não eram servidas a partir do banco de dados, também estavam muito mais lentas que o esperado. Observando os gráficos com granularidade no nível de arquivos, vimos que suas respostas alguns dias antes eram muito mais rápidas. Isso implicava que a correlação observada entre `merge_join` e o aumento de latência era espúria e que nossa teoria de indexação abaixo do ideal estava fatalmente incorreta.

Analisando as requisições inesperadamente lentas de conteúdo estático, a

maioria das RPCs enviadas a partir da aplicação era para um serviço de memcache, portanto as requisições deveriam ser muito rápidas – da ordem de alguns milissegundos. Vimos que essas requisições realmente eram bem rápidas, portanto o problema não parecia estar aí. Contudo, entre o momento em que a aplicação começava a trabalhar com uma requisição e o instante em que ela fazia as primeiras RPCs, havia um período de aproximadamente 250 ms em que a aplicação estava fazendo... bem, *alguma coisa*. Como a App Engine executa código fornecido pelos usuários, sua equipe de SRE não gera o perfil nem inspeciona códigos de aplicação, portanto não podíamos dizer o que a aplicação estava fazendo nesse intervalo; de modo semelhante, o Dapper não era capaz de ajudar a descobrir o que estava acontecendo, pois ele monitorava apenas as chamadas de RPC, e nenhuma havia sido feita naquele período.

Diante do que, a essa altura, era um grande mistério, decidimos não resolvê-lo... *por enquanto*. O cliente tinha um lançamento público agendado para a semana seguinte, e não tínhamos certeza da rapidez com que poderíamos identificar o problema e corrigi-lo. Em vez disso, recomendamos que o cliente aumentasse os recursos alocados à sua aplicação para o tipo de instância com o máximo de CPU disponível. Fazer isso reduziu a latência da aplicação a níveis aceitáveis, embora não fosse tão baixa quanto queríamos. Concluímos que a atenuação na latência era suficiente para que a equipe pudesse conduzir seu lançamento com sucesso, e então investigaríamos com mais tranquilidade.¹⁷

A essa altura, suspeitávamos que a aplicação fosse vítima de outra causa comum de aumento repentino de latência e de uso de recursos: uma mudança no tipo de trabalho. Vimos um aumento de escritas feitas pela aplicação no banco de dados, imediatamente antes de sua latência ter aumentado, mas como esse aumento não havia sido muito grande – nem havia se mantido –, nós o registramos como coincidência. No entanto, esse comportamento lembrava um padrão comum: uma instância da aplicação é inicializada lendo objetos do banco de dados e, em seguida, armazenando-os na memória da instância. Ao fazer isso, a instância evita ler configurações que são raramente alteradas no banco de dados a cada requisição; em vez disso, ela verifica os objetos em memória. Assim, o tempo que demora para tratar as requisições

geralmente escalará de acordo com a quantidade de dados de configuração.¹⁸ Não podíamos provar que esse comportamento fosse a raiz do problema, mas é um antipadrão comum.

Os desenvolvedores da aplicação acrescentaram instrumentação para entender em que ponto a aplicação estava gastando seu tempo. Eles identificaram um método chamado em todas as requisições, que verificava se um usuário estava na lista branca para acesso a um dado path. O método usava uma camada de caching que procurava minimizar os acessos tanto ao banco de dados quanto ao serviço de memcache, armazenando objetos de lista branca na memória das instâncias. Conforme um dos desenvolvedores da aplicação percebeu na investigação, “Ainda não sei onde está o fogo, mas estou cego com a fumaça que vem desse cache de lista branca”.

Um tempo depois, a causa-raiz foi encontrada: por causa de um bug de longa data no sistema de controle de acesso da aplicação, sempre que um path específico era acessado, um objeto de lista branca era criado e armazenado no banco de dados. Na corrida para o lançamento, um scanner de segurança automatizado havia testado a aplicação em busca de vulnerabilidades e, como efeito colateral, seu scan havia produzido milhares de objetos de lista branca no curso de meia hora. Esses objetos supérfluos de lista branca precisavam então ser verificados a cada requisição feita à aplicação, o que resultou em respostas patologicamente lentas – sem causar nenhuma chamada de RPC da aplicação para outros serviços. Corrigir o bug e remover esses objetos fez o desempenho da aplicação retornar aos níveis esperados.

Facilitando a resolução de problemas

Há várias maneiras de simplificar e agilizar a resolução de problemas. Talvez as mais básicas sejam:

- Incluir a capacidade de poder observar dados – tanto por meio de métricas de caixa-branca quanto por logs estruturados – em cada componente desde o princípio.
- Projetar sistemas com interfaces bem compreendidas e observáveis entre os componentes.

Garantir que as informações estejam disponíveis de modo consistente em todo o sistema – por exemplo, usando um identificador único de requisição para todas as RPCs geradas por vários componentes – reduz a necessidade de descobrir *qual* entrada de log em um componente de upstream corresponde a uma entrada de log em um componente de downstream, agilizando o tempo para diagnóstico e recuperação.

Problemas em representar corretamente o estado da realidade em uma mudança de código ou em uma mudança no ambiente muitas vezes resultam na necessidade de ativar o processo de resolução de problemas. Simplificar, controlar e fazer log dessas mudanças pode reduzir essa necessidade e facilita esse processo quando ele ocorrer.

Conclusão

Vimos alguns passos que você pode executar para deixar o processo de resolução de problemas mais claro e comprehensível aos iniciantes para que, eles também, possam se tornar eficazes na resolução de problemas. Adotar uma abordagem sistemática para a resolução de problemas – em oposição a contar com a sorte ou com a experiência – pode ajudar a limitar o tempo para a recuperação de seus serviços, resultando em uma melhor experiência para os usuários.

¹ De fato, usar apenas princípios básicos e habilidades de resolução de problemas muitas vezes é uma maneira eficiente de aprender como um sistema funciona; veja o Capítulo 28.

² Veja https://en.wikipedia.org/wiki/Hypothetico-deductive_model.

³ Por exemplo, variáveis exportadas, como descrito no Capítulo 10.

⁴ Frase atribuída a Theodore Woodward, da Escola de Medicina da Universidade de Maryland, nos anos 1940. Consulte [https://en.wikipedia.org/wiki/Zebra_\(medicine\)](https://en.wikipedia.org/wiki/Zebra_(medicine)). Isso funciona em alguns domínios, mas para outros sistemas, classes inteiras de falhas podem ser elimináveis: por exemplo, usar um sistema de arquivos de cluster bem projetado significa que um problema de latência é improvável de ocorrer por causa de um único disco morto.

⁵ Navalha de Occam, veja https://en.wikipedia.org/wiki/Occam%27s_razor. Porém, lembre-se de que ainda podemos ter o caso em que haja vários problemas; em particular, talvez seja mais provável que um sistema tenha vários problemas comuns, menos graves, que, em conjunto, expliquem todos os sintomas, em vez de haver um único problema raro que provoque todos eles. Veja https://en.wikipedia.org/wiki/Hickam%27s_dictum.

⁶ Veja, é claro, <https://xkcd.com/552>.

⁷ Pelo menos, não temos nenhuma teoria plausível para explicar por que o número de PhDs concedidos em Ciência da Computação nos Estados Unidos deveria estar 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 Talvez seja conveniente indicar a leitura de [Tat99] aos possíveis responsáveis pelo relato de bugs a fim de ajudá-los a fornecer relatos de alta qualidade dos problemas.

9 Porém, tome cuidado com correlações falsas que possam conduzi-lo por caminhos errados!

10 Em muitos aspectos, isso é semelhante à técnica dos “Cinco porquês” [Ohn88] introduzida por Taiichi Ohno para entender as causas-raízes dos erros em manufatura.

11 Em oposição ao RE2, o PCRE pode exigir um tempo exponencial para avaliar algumas expressões regulares. O RE2 está disponível em <https://github.com/google/re2>.

12 [All15] observa que essa é uma técnica heurística usada com frequência para resolver problemas de interrupção de serviço.

13 Usar um documento compartilhado ou um bate-papo de tempo real para anotações fornece um timestamp de *quando* você fez algo, o que ajuda nos postmortems. A informação também é compartilhada com outras pessoas, de modo que elas estarão cientes do estado atual do mundo e não precisarão interromper sua resolução de problemas.

14 Veja [Mea08], que explica como pensar em sistemas, e também [Coo00] e [Dek14], que expõem as limitações de encontrar uma única causa-raiz, em vez de analisar o sistema e seu ambiente em busca dos fatores causadores.

15 Veja <https://cloud.google.com/appengine>.

16 Resumimos e simplificamos esse estudo de caso para ajudar em sua compreensão.

17 Embora fazer um lançamento com um bug não identificado não seja ideal, muitas vezes é impraticável eliminar todos os bugs conhecidos. Em vez de fazer isso, às vezes temos que nos contentar com medidas que não sejam as melhores possíveis e atenuar o risco da melhor forma que pudermos usando um bom discernimento de engenharia.

18 A pesquisa no banco de dados pode utilizar um índice para agilizar a comparação, mas uma implementação frequente em memória é constituída de uma comparação em um laço for simples com todos os objetos em cache. Se houver poucos objetos, não importa que essa operação demore de forma linear – mas isso pode causar um aumento significativo de latência e de uso de recursos à medida que o número de objetos em cache aumentar.

CAPÍTULO 13

Resposta a emergências

Escrito por Corey Adam Baye

Editado por Diane Bates

As coisas quebram; a vida é assim.

Independentemente do que estiver em jogo ou do porte de uma empresa, um traço vital de sua saúde no longo prazo e que, consequentemente, a destaca das outras empresas é o modo como as pessoas envolvidas respondem a uma emergência. Poucos de nós respondem naturalmente bem em uma emergência. Uma resposta apropriada exige preparação e treinamento periódico, pertinente e prático. Definir e manter treinamentos abrangentes e processos de testes exige o suporte da diretoria e da gerência, além de uma atenção cuidadosa por parte dos funcionários. Todos esses elementos são essenciais na promoção de um ambiente em que as equipes possam gastar dinheiro, tempo, energia e, possivelmente, até mesmo uptime para garantir que os sistemas, os processos e as pessoas respondam de modo eficiente em uma emergência.

Observe que o capítulo sobre a cultura de postmortem discute as especificidades sobre como escrever postmortems de modo a garantir que incidentes que exijam respostas emergenciais também se transformem em uma oportunidade de aprendizado (veja o Capítulo 15). Este capítulo apresenta exemplos mais concretos de incidentes desse tipo.

O que fazer quando os sistemas falham

Em primeiro lugar, não entre em pânico! Você não está sozinho e o céu não está caindo. Você é um profissional e foi treinado para lidar com esse tipo de situação. Geralmente ninguém está correndo perigo fisicamente – apenas

aqueles pobres elétrons estão em perigo. No pior caso, metade da internet terá caído. Sendo assim, respire fundo... e continue firme.

Se você se sentir sobrecarregado, chame mais pessoas. Às vezes, talvez seja até mesmo necessário enviar um page para toda a empresa. Se sua empresa tiver um processo para resposta a incidentes (veja o Capítulo 14), certifique-se de que você tenha familiaridade com ele e siga esse processo.

Emergência induzida por testes

O Google adotou uma abordagem proativa para testes de desastre e emergência (veja [Kri12]). Os SREs provocam falhas em nossos sistemas, observam como eles falham e fazem alterações para melhorar a confiabilidade e evitar que as falhas ocorram novamente. Na maioria das vezes, essas falhas controladas ocorrem conforme planejado, e o sistema-alvo e os sistemas dependentes se comportam, de modo geral, da maneira esperada. Identificamos alguns pontos fracos ou dependências ocultas e documentamos ações de acompanhamento (follow-up) para corrigir as falhas descobertas. Às vezes, porém, nossas suposições e os resultados propriamente ditos são extremamente divergentes.

Eis um exemplo de um teste que revelou uma série de dependências inesperadas.

Detalhes

Queríamos que dependências ocultas fossem reveladas em um banco de dados de teste em um de nossos maiores bancos de dados MySQL distribuídos. O plano consistia em bloquear todo o acesso a apenas um banco de dados entre cem. Ninguém previu os resultados que se desdobraram.

Resposta

Minutos após o início do teste, vários serviços dependentes informaram que tanto os usuários externos quanto os usuários internos eram incapazes de acessar sistemas essenciais. Alguns sistemas estavam acessíveis de forma intermitente ou apenas parcialmente.

Supondo que o teste era responsivo, o SRE interrompeu imediatamente o exercício. Tentamos fazer rollback das mudanças em permissões, mas não fomos bem-sucedidos. Em vez de entrar em pânico, fizemos imediatamente um brainstorming para discutir como restaurar o acesso apropriado. Usando uma abordagem já testada, restauramos as permissões para as réplicas e failovers. Em um esforço paralelo, pedimos aos desenvolvedores-chaves que corrigissem a falha na biblioteca da camada de aplicação do banco de dados.

Uma hora depois da decisão original, todo o acesso havia sido completamente restaurado e todos os serviços podiam se conectar novamente. O amplo impacto desse teste motivou uma correção rápida e completa nas bibliotecas e um plano para testes periódicos a fim de evitar que uma falha importante como essa ocorresse de novo.

Descobertas

O que deu certo

Serviços dependentes afetados pelo incidente imediatamente fizeram os problemas escalarem na empresa. Supomos, corretamente, que nosso experimento controlado havia saído de controle e abortamos imediatamente o teste.

Fomos capazes de restaurar totalmente as permissões em uma hora a partir do primeiro relato de problemas e, desse momento em diante, os sistemas começaram a se comportar de forma apropriada. 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.

Ações de acompanhamento foram executadas de modo rápido e completo visando a evitar interrupções de serviço semelhantes, e instituímos testes periódicos para garantir que falhas semelhantes não ocorressem novamente.

O que aprendemos

Embora esse teste tenha sido totalmente revisado e planejado para que tivesse um bom escopo, a realidade mostrou que tínhamos uma compreensão

insuficiente dessa interação em particular entre os sistemas dependentes.

Falhamos em seguir o processo de resposta a incidentes, que havia sido implantado apenas algumas semanas antes e não havia sido totalmente disseminado. Esse processo teria garantido que todos os serviços e clientes estivessem cientes da interrupção no serviço. Para evitar cenários semelhantes no futuro, a SRE aperfeiçoa e testa continuamente nossas ferramentas e processos para resposta a incidentes, além de garantir que as atualizações em nossos procedimentos para gerenciamento de incidentes sejam claramente comunicadas a todas as partes relevantes.

Como não havíamos testado nossos procedimentos de rollback em um ambiente de teste, esses procedimentos continham falhas que fizeram a interrupção no serviço se estender. Atualmente exigimos um teste completo dos procedimentos de rollback antes que testes em larga escala como esse sejam feitos.

Emergência induzida por alterações

Como você pode imaginar, o Google tem muitas configurações – configurações complexas – e, constantemente, fazemos mudanças nessas configurações. Para evitar que nossos sistemas falhem de imediato, realizamos inúmeros testes em mudanças de configuração a fim de garantir que não resultem em comportamentos inesperados ou indesejados. Contudo, a escala e a complexidade da infraestrutura do Google fazem com que seja impossível prever todas as dependências ou interações; às vezes, mudanças de configuração não ocorrem totalmente conforme planejado.

O exemplo a seguir mostra um caso desses.

Detalhes

Uma mudança de configuração na infraestrutura que ajuda a proteger nossos serviços de abusos foi instalada globalmente em uma sexta-feira. Essa infraestrutura interage essencialmente com todos os nossos sistemas voltados ao mundo externo, e a mudança disparou um bug que provocava falhas sucessivas nesses sistemas, fazendo com que toda a frota de máquinas

começasse a entrar em um circuito de falhas sucessivas quase simultaneamente. Como a infraestrutura interna do Google também depende de nossos próprios serviços, muitas aplicações internas repentinamente se tornaram indisponíveis também.

Resposta

Em questão de segundos, alertas de monitoração começaram a ser disparados, indicando que determinadas localidades estavam inativas. Alguns engenheiros de plantão ao mesmo tempo viram o que eles acreditavam ser uma falha na rede corporativa e fizeram uma realocação para salas de segurança (quartos de pânico) dedicadas, com acesso de backup para o ambiente de produção. Outros engenheiros que estavam lutando com seus acessos corporativos se juntaram a eles.

Cinco minutos após aquela primeira atualização de configuração, o engenheiro responsável pela atualização, depois de ter tomado conhecimento da falha corporativa, mas ainda sem tomar conhecimento da interrupção mais ampla, fez outra mudança de configuração para efetuar o rollback da primeira mudança. Nesse ponto, os serviços começaram a se recuperar.

Dez minutos depois da primeira atualização, engenheiros de plantão declararam que havia um incidente e passaram a seguir os procedimentos internos de resposta a incidentes. Eles começaram a notificar o restante da empresa a respeito da situação. O engenheiro que havia feito a atualização informou os engenheiros de plantão que a interrupção no serviço provavelmente havia sido causada pela mudança que havia sido implantada e para a qual um rollback havia sido feito depois. Apesar disso, alguns serviços tiveram bugs ou erros de configurações não relacionados ao caso, que foram acionados pelo evento original, e só se recuperaram totalmente depois de uma hora.

Descobertas

O que deu certo

Havia vários fatores em cena que evitaram que esse incidente resultasse em

uma interrupção de mais longo prazo de muitos sistemas internos do Google.

Para começar, a monitoração quase imediatamente detectou e nos alertou sobre o problema. No entanto, devemos observar que, nesse caso, nossa monitoração deixou a desejar: alertas foram disparados de modo repetido e constante, sobrecarregando os engenheiros de plantão e gerando spams em canais de comunicação normais e de emergência.

Depois que o problema foi detectado, o gerenciamento de incidentes, de modo geral, se saiu bem, e as atualizações foram comunicadas de modo claro e frequente. Nossos sistemas de comunicação que utilizam uma banda diferente mantiveram todos conectados, mesmo quando algumas das pilhas de software mais complicadas estavam inutilizáveis. Essa experiência nos lembrou do motivo pelo qual a SRE mantém sistemas de backup altamente confiáveis, com pouco overhead, que usamos regularmente.

Além desses sistemas de comunicação que usam banda diferente, o Google tem ferramentas de linha de comando e métodos de acesso alternativos que nos permitem fazer atualizações e rollback de alterações mesmo quando outras interfaces estiverem inacessíveis. Essas ferramentas e métodos de acesso funcionaram bem durante a interrupção dos serviços, com a ressalva de que os engenheiros precisavam ter mais familiaridade com as ferramentas e testá-las de modo mais rotineiro.

A infraestrutura do Google oferecia também uma camada de proteção adicional em que o sistema afetado limitava a velocidade com que oferecia atualizações completas a novos clientes. Esse comportamento pode ter provocado uma contenção nas falhas sucessivas e pode ter evitado uma interrupção completa dos serviços, permitindo que os jobs permanecessem ativos o tempo suficiente para atender a algumas requisições entre as falhas.

Por fim, não podemos deixar de lado o elemento sorte na resolução rápida desse incidente: o engenheiro que fez a alteração por acaso estava seguindo os canais de comunicação em tempo real – um nível adicional de diligência que não faz parte do processo normal de release. Esse engenheiro percebeu um grande número de reclamações sobre o acesso corporativo logo depois da atualização e fez rollback da mudança quase de imediato. Se esse rollback rápido não tivesse ocorrido, a interrupção no serviço poderia ter durado bem

mais, tornando-se muito mais difícil de resolver.

O que aprendemos

Uma atualização de versão anterior da nova funcionalidade havia envolvido um canary (implantação canário) completo, mas não havia acionado o mesmo bug, pois não exercitou uma palavra-chave específica e muito rara da configuração, em conjunto com a nova funcionalidade. A mudança específica que disparou esse bug não era considerada arriscada e, desse modo, seguiu um processo menos rigoroso de canary. Quando a mudança foi implantada globalmente, ela usou a combinação não testada de palavra-chave/funcionalidade que deu início à falha.

Ironicamente, melhorias no canarying e na automação estavam designadas para ter prioridade mais alta no trimestre seguinte. Esse incidente fez subir imediatamente sua prioridade e reforçou a necessidade de um canarying completo, de modo independente do risco percebido.

Como seria de esperar, o alerta foi vocal durante esse incidente porque todas as localidades estavam essencialmente offline durante alguns minutos. Isso prejudicou o verdadeiro trabalho feito pelos engenheiros de plantão e tornou mais difícil a comunicação entre aqueles que estavam envolvidos no incidente.

No Google, dependemos de nossas próprias ferramentas. Boa parte da pilha de softwares que usamos para resolução de problemas e comunicação está atrás de jobs que estavam falhando sucessivamente. Se essa interrupção de serviço tivesse durado um pouco mais, a depuração teria sido severamente afetada.

Emergência induzida por processo

Investimos uma quantidade considerável de tempo e de energia na automação que administra nosso conjunto de máquinas. É impressionante o número de jobs que alguém pode iniciar, parar ou reconfigurar no conjunto de máquinas com muito pouco esforço. Às vezes a eficiência de nossa automação pode ser um pouco assustadora quando uma situação não ocorre exatamente conforme planejado.

O exemplo a seguir mostra um caso em que ser rápido não foi algo tão bom assim.

Detalhes

Como parte dos testes rotineiros de automação, duas requisições consecutivas de desativação para a mesma instalação de servidores foram submetidas. No caso da segunda requisição de desativação, um bug sutil na automação enviou todas as máquinas de todos esses tipos de instalação globalmente para a fila do Diskerase, para que os discos rígidos fossem limpos; veja a seção “Automação: permitindo falhas em escala”, para mais detalhes.

Resposta

Logo após a segunda requisição de desativação ter sido enviada, os engenheiros de plantão receberam um page quando a primeira instalação pequena de servidores havia sido colocada offline para ser desativada. Sua investigação havia determinado que as máquinas tinham sido transferidas para a fila do Diskerase, portanto, seguindo o procedimento normal, os engenheiros de plantão drenaram o tráfego da localidade. Como as máquinas dessa localidade haviam sido limpas, elas eram incapazes de responder às requisições. Para evitar que essas requisições falhassem totalmente, os engenheiros de plantão drenaram o tráfego dessa localidade. O tráfego foi redirecionado para localidades que pudessem responder adequadamente às requisições.

Não muito tempo depois, os pagers em todos os lugares estavam sendo acionados para todas as instalações de servidores desse tipo em todo o mundo. Em resposta, os engenheiros de plantão desabilitaram toda a automação da equipe a fim de evitar mais danos. Eles interromperam ou congelaram automações adicionais e a manutenção na produção logo depois.

Em uma hora, todo o tráfego havia sido desviado para outras localidades. Embora os usuários pudessem ter experimentado latências elevadas, suas requisições foram atendidas. A interrupção no serviço estava oficialmente superada.

Agora a parte difícil começava: a recuperação. Alguns links de rede estavam

informando um congestionamento intenso, portanto os engenheiros de rede implementaram ações para atenuação à medida que pontos de estrangulamento apareceram. Uma instalação de servidores em uma dessas localidades foi escolhida para que fosse a primeira a ressurgir das cinzas. Três horas depois da interrupção de serviço inicial, e graças à tenacidade de vários engenheiros, a instalação foi reconstruída e voltou a ficar online, aceitando requisições de usuários novamente.

As equipes americanas devolveram o controle às suas contrapartidas europeias, e a SRE desenvolveu um plano para dar prioridade às reinstalações usando um processo manual, porém organizado. A equipe foi dividida em três partes, e cada parte era responsável por um passo no processo de reinstalação manual. Em três dias, a maior parte da capacidade estava online novamente, enquanto alguns retardatários seriam recuperados nos próximos um ou dois meses.

Descobertas

O que deu certo

Proxies reversos em instalações grandes de servidores são administrados de modo muito diferente de proxies reversos em instalações pequenas, portanto as instalações grandes não sofreram impacto. Os engenheiros de plantão foram capazes de transferir rapidamente o tráfego das instalações menores para as instalações maiores. Por design, essas instalações maiores são capazes de tratar uma carga completa sem dificuldades. No entanto, alguns links de rede ficaram congestionados e, desse modo, exigiram que os engenheiros de rede desenvolvessem soluções alternativas. Para reduzir o impacto nos usuários finais, os engenheiros de plantão consideraram as redes congestionadas como o problema de mais alta prioridade.

O processo de desativação das instalações pequenas funcionou bem e de modo eficiente. Do início ao fim, demorou menos de uma hora para desativar com sucesso e limpar com segurança um grande número dessas instalações.

Embora a automação da desativação acabasse rapidamente com a monitoração nas instalações pequenas, os engenheiros de plantão puderam

reverter rapidamente essas mudanças na monitoração. Fazer isso os ajudou a avaliar a extensão dos danos.

Os engenheiros seguiram rapidamente os protocolos de resposta a incidentes, que haviam amadurecido de forma considerável naquele ano, desde a primeira interrupção de serviço descrita neste capítulo. A comunicação e a colaboração em toda a empresa e entre as equipes foram excelentes – um verdadeiro atestado da eficiência do programa de gerenciamento de incidentes e do treinamento. Todos nas respectivas equipes fizeram sua colaboração, trazendo sua vasta experiência para dar suporte.

O que aprendemos

A causa-raiz foi o fato de o servidor de automação de desativação não ter as verificações de sanidade apropriadas para os comandos enviados. Quando o servidor executou novamente em resposta à desativação inicial com falha, ele recebeu uma resposta vazia para o rack de máquinas. Em vez de filtrar a resposta, ele passou o filtro vazio para o banco de dados de máquinas, dizendo-lhe para aplicar o Diskerase em todas as máquinas envolvidas. Sim, às vezes zero significa tudo. O banco de dados de máquinas aceitou isso, de modo que o fluxo de trabalho de desativação começou a limpar as máquinas o mais rápido possível.

As reinstalações nas máquinas foram lentas e não confiáveis. Esse comportamento, em boa parte, se deveu ao uso do TFTP (Trivial File Transfer Protocol, Protocolo Trivial de Transferência de Arquivos) no QoS (Quality of Service, Qualidade de Serviço) mais baixo possível de rede, das localidades distantes. O BIOS¹ de cada máquina do sistema lidou de forma precária com as falhas. Conforme as placas de rede envolvidas, o BIOS era interrompido ou entrava em um ciclo constante de reinicializações. Eles não conseguiam transferir os arquivos de boot em cada ciclo e estavam prejudicando os instaladores. Os engenheiros de plantão foram capazes de corrigir esses problemas de reinstalação reclassificando o tráfego de instalação com uma prioridade um pouco mais alta e usando automação para reiniciar qualquer máquina que estivesse travada.

A infraestrutura de reinstalação de máquinas foi incapaz de tratar a instalação

simultânea de milhares de máquinas. Essa incapacidade se devia parcialmente a uma regressão que evitava que a infraestrutura executasse mais de duas tarefas de instalação por máquina de trabalho. A regressão também usava configurações inadequadas de QoS para transferir arquivos e tinha timeouts ajustados de modo precário. Ela forçava a reinstalação do kernel, mesmo em máquinas que ainda tivessem um kernel apropriado e nas quais o Diskerase ainda não havia ocorrido. Para reparar essa situação, os engenheiros de plantão escalaram para as partes responsáveis por essa infraestrutura, que puderam rapidamente reajustá-la a fim de dar suporte a essa carga incomum.

Todos os problemas têm soluções

O tempo e a experiência mostram que os sistemas não só falharão, como o farão de maneiras que jamais teríamos imaginado antes. Uma das maiores lições que o Google aprendeu é que há uma solução, mesmo que ela possa não parecer óbvia, em especial para a pessoa cujo pager está gritando. Se você não conseguir pensar em uma solução, jogue sua rede mais longe. Envolva outros colegas de equipe, procure ajuda, faça o que tiver que fazer, mas faça isso rapidamente. A maior prioridade é resolver o problema em mãos de modo rápido. Com frequência, a pessoa que tem mais informações sobre o estado é aquela cujas ações, de alguma forma, dispararam o evento. Utilize essa pessoa.

Mais importante ainda, depois que uma emergência foi atenuada, não se esqueça de reservar tempo para fazer uma limpeza, registrar o incidente por escrito e...

Aprenda com o passado. Não deixe que ele se repita.

Mantenha um histórico das interrupções de serviço

Não há maneira melhor de aprender do que documentar o que apresentou falhas no passado. A história diz respeito a aprender com os erros de todos. Seja completo, seja honesto, mas, acima de tudo, faça perguntas difíceis. Procure ações específicas que possam evitar que esse tipo de interrupção de

serviço ocorra novamente, não só de modo tático, mas também de forma estratégica. Garanta que todos na empresa possam aprender o que você aprendeu publicando e organizando postmortems.

Atribua a responsabilidade a você mesmo e a outras pessoas para acompanhar as ações específicas detalhadas nesses postmortems. Fazer isso evitará uma futura interrupção de serviço que seja quase idêntica e disparada por eventos quase iguais aos de uma interrupção de serviço já documentada. Depois que você tiver um registro de controle sólido para aprender com as interrupções de serviço passadas, veja o que pode fazer para evitar futuras interrupções.

Faça as perguntas importantes e até mesmo improváveis: E se...?

Não há um teste maior que a realidade. Faça a si mesmo algumas perguntas importantes, com respostas abertas. E se a energia do prédio cair...? E se os racks de equipamentos de rede ficarem submersos em meio metro de água...? E se o datacenter principal repentinamente ficar às escuras...? E se alguém comprometer seu servidor web...? O que você faria? Quem você chamaria? Quem preencherá o cheque? Você tem um plano? Você sabe como reagir? Você sabe como seus sistemas reagirão? 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, a teoria e a realidade são duas esferas totalmente diferentes. Até seu sistema ter realmente falhado, você não saberá realmente como esse sistema, seus sistemas dependentes ou seus usuários reagirão. Não conte com suposições ou com o que você não puder ou não tiver testado. Você preferiria que uma falha ocorresse às duas da manhã de sábado, quando a maior parte da empresa ainda estivesse fora em uma atividade de team-building (construção de equipes) na Floresta Negra – ou quando você tivesse seus melhores e mais brilhantes profissionais à mão, monitorando o teste que eles revisaram meticulosamente nas últimas semanas?

Conclusão

Analisamos três casos diferentes em que partes de nossos sistemas falharam. Embora todas as três emergências tenham sido disparadas de modo diferente – uma por um teste proativo, outra por uma mudança de configuração e outra ainda por automação de desativação –, as respostas tiveram muitas características em comum. As pessoas responsáveis pela resposta não entraram em pânico. Elas convocaram outras pessoas quando acharam necessário, estudaram as interrupções de serviço anteriores e aprenderam com elas. Em seguida, essas pessoas construíram seus sistemas para que respondessem melhor a esses tipos de interrupções de serviço. Sempre que novos modos de falha se apresentaram, as pessoas responsáveis por responder a eles os documentaram. Esse acompanhamento ajudou outras equipes a melhorar o processo de resolução de problemas e a fortalecer seus sistemas contra interrupções de serviço semelhantes. As pessoas responsáveis por responder aos incidentes testaram seus sistemas de modo proativo. Esses testes garantiram que as mudanças corrigissem os problemas subjacentes e identificassem outros pontos fracos antes que se tornassem interrupções de serviço.

À medida que nossos sistemas evoluem, o ciclo continua, com cada interrupção de serviço ou teste resultando em melhorias incrementais tanto para os processos quanto para os sistemas. Embora os estudos de caso deste capítulo sejam específicos do Google, essa abordagem para resposta a emergências pode ser aplicada com o tempo em qualquer empresa de qualquer porte.

¹ BIOS: Basic Input/Output System, ou Sistema Básico de Entrada/Saída. O BIOS é um software incluído em um computador, que serve para enviar instruções simples ao hardware, permitindo ter entrada e saída de dados antes de o sistema operacional ser carregado.

CAPÍTULO 14

Administrando incidentes

Escrito por Andrew Stribblehill¹

Editado por Kavita Guliani

Um gerenciamento eficiente de incidentes é fundamental para limitar a desordem causada por um incidente e fazer os negócios voltarem a operar normalmente o mais rápido possível. Se você não aperfeiçoar sua resposta a possíveis incidentes com antecedência, mesmo um gerenciamento de incidentes calcado em princípios pode escorrer pelas mãos em situações da vida real.

Este capítulo descreve um exemplo de incidente que sai do controle devido a práticas de gerenciamento de incidentes *ad hoc*, apresenta uma abordagem bem administrada ao incidente e analisa como o mesmo incidente transcorreria caso fosse tratado por um gerenciamento de incidentes com um bom funcionamento.

Incidentes não administrados

Coloque-se no lugar de Mary, a engenheira de plantão na Firma. São duas horas da tarde de uma sexta-feira e seu pager acaba de detonar. A monitoração caixa-preta informa que seu serviço parou de servir *qualquer* tráfego em todo um datacenter. Com um suspiro, você deixa seu café de lado e parte para a tarefa de corrigir o problema. Após alguns minutos nessa tarefa, outro alerta informa que um segundo datacenter parou de servir. Então, o terceiro de seus cinco datacenters falha. Para piorar a situação, há mais tráfego do que os datacenters restantes são capazes de tratar, de modo que eles começam a ficar sobrecarregados. Antes que você se dê conta, o serviço está sobrecarregado e é incapaz de servir a qualquer requisição.

Você encara os logs durante um tempo que parece ser uma eternidade. Milhares de linhas de logging sugerem que há um erro em um dos módulos atualizados recentemente, portanto você decide reverter os servidores para a versão anterior. Quando percebe que o rollback não ajudou, você liga para Josephine, que escreveu a maior parte do código do serviço que está com hemorragia no momento. Lembrando você de que são 3:30 da manhã no fuso horário dela, entorpecida pelo sono, ela concorda em fazer login e dar uma olhada. Seus colegas Sabrina e Robin começam a dar uma espiada a partir de seus próprios terminais. “Estamos só olhando”, eles dizem.

Agora um dos engravatados telefonou para o seu chefe e está colérico, exigindo saber por que ele não foi informado sobre a “total degradação desse serviço crucial aos negócios”. Além disso, os vice-presidentes estão importunando você pedindo um ETA² e perguntando repetidamente: “Como é que isso pôde ter acontecido?”. Você poderia se solidarizar, mas fazer isso exigiria um esforço cognitivo que você está guardando para o seu trabalho. Os VPs recorrem às suas experiências anteriores com engenharia e fazem comentários irrelevantes, porém difíceis de refutar, como: “Aumente o tamanho da página!”.

O tempo passa; os dois datacenters restantes falham completamente. Sem que você soubesse, Josephine, ainda bêbada de sono, ligou para Malcolm. Ele teve uma inspiração: algo sobre afinidade de CPU. Ele tinha certeza de que poderia otimizar os processos do servidor restante se pudesse simplesmente implantar essa única alteração simples no ambiente de produção, portanto ele fez isso. Em segundos, os servidores reiniciaram, captando a alteração. E então morreram.

A anatomia de um incidente não administrado

Observe que todos no cenário anterior estavam fazendo seus trabalhos, conforme eles os viam. Como tudo pôde ter dado tão errado? Algumas ameaças comuns fizeram esse incidente sair totalmente do controle.

Foco centrado no problema técnico

Temos a tendência de contratar pessoas como Mary pelo seu arrojo técnico.

Assim, não é de surpreender que ela estivesse ocupada fazendo mudanças operacionais no sistema, tentando bravamente resolver o problema. Ela não estava em uma posição para pensar no panorama geral sobre como atenuar o problema porque a tarefa técnica à mão era enorme.

Comunicação precária

Pelo mesmo motivo, Mary estava ocupada demais para se comunicar com clareza. Ninguém sabia quais ações seus colegas de trabalho estavam executando. Os líderes do negócio estavam irritados, os clientes estavam frustrados e outros engenheiros que poderiam ter dado uma mão na depuração e na correção do problema não foram usados com eficiência.

Trabalho sem coordenação

Malcolm estava fazendo alterações no sistema com a melhor das intenções. No entanto, ele não coordenou seu trabalho com o de seus colegas – nem mesmo com Mary, que tecnicamente era a responsável pela resolução do problema. Suas alterações fizeram uma situação ruim piorar mais ainda.

Elementos do processo de gerenciamento de incidentes

Habilidades e práticas de gerenciamento de incidentes existem para canalizar as energias de indivíduos entusiásticos. O sistema de gerenciamento de incidentes do Google é baseado no Incident Command System (Sistema de Controle de Incidentes)³, conhecido pela sua clareza e escalabilidade.

Um processo de gerenciamento de incidentes bem projetado tem as características a seguir.

Separação recursiva de responsabilidades

É importante garantir que todos os envolvidos no incidente conheçam seus papéis e não fiquem no caminho de outra pessoa. De forma, até certo ponto, contraintuitiva, uma separação clara de responsabilidades permite que os indivíduos tenham mais autonomia do que poderiam ter de outro modo, pois

não precisam adivinhar o que seus colegas estão fazendo.

Se a carga de trabalho de um dado participante se tornar excessiva, essa pessoa deve pedir mais funcionários ao líder de planejamento. Elas devem, então, delegar o trabalho para outras pessoas – uma tarefa que pode implicar na criação de subincidentes. De modo alternativo, o líder responsável pela atribuição de funções pode delegar componentes do sistema aos colegas, que mantêm o líder informado de modo geral.

Várias funções distintas devem ser delegadas a indivíduos em particular:

Coordenação do incidente

O coordenador do incidente conhece o estado geral do incidente. Esses coordenadores estruturam a força-tarefa para a resposta ao incidente, atribuindo responsabilidades de acordo com a necessidade e a prioridade. *De facto*, o coordenador assume todas as posições para as quais ninguém tenha sido delegado. Se for apropriado, ele pode remover obstáculos que impeçam a equipe de Ops de trabalhar de modo mais eficiente.

Tarefas operacionais

O líder de Ops trabalha junto com o coordenador do incidente para responder ao incidente aplicando ferramentas operacionais na tarefa em questão. A equipe de operações deve ser o único grupo a modificar o sistema durante um incidente.

Comunicação

Essa pessoa representa a face pública da força-tarefa de resposta ao incidente. Suas obrigações definitivamente incluem fornecer atualizações periódicas à equipe de resposta ao incidente e aos stakeholders (geralmente por email), e podem se estender a tarefas como manter a documentação do incidente precisa e atualizada.

Planejamento

A função de planejamento dá suporte à Ops ao lidar com problemas de mais longo prazo, como registrar bugs, pedir o jantar, organizar a mudança de turno e monitorar como o sistema se distanciou da norma para que possa ser restaurado quando o incidente for resolvido.

Um posto de comando reconhecido

As partes interessadas devem entender em que ponto elas podem interagir com o coordenador do incidente. Em muitas situações, colocar os membros da força-tarefa para o incidente em uma “Sala de Guerra” central pode ser apropriado. Outras equipes talvez prefiram trabalhar em suas mesas, prestando atenção às atualizações do incidente via email ou IRC.

O Google percebeu que o IRC é muito vantajoso na resposta a incidentes. O IRC é bastante confiável e pode ser usado como um log de comunicações sobre esse evento; um registro como esse tem valor inestimável para manter mudanças de estado detalhadas em mente. Também escrevemos bots que fazem log do tráfego relacionado ao incidente (o que é útil para análises postmortem), além de outros bots que fazem log de eventos, como alertas no canal. O IRC também é um meio conveniente pelo qual equipes geograficamente distribuídas podem se coordenar.

Documento vivo do estado do incidente

A principal responsabilidade do coordenador do incidente é manter um documento vivo do incidente. Esse documento pode estar em uma wiki, mas o ideal é que possa ser editado por várias pessoas ao mesmo tempo. A maioria de nossas equipes usa o Google Docs, embora a SRE do Google Docs utilize o Google Sites: afinal de contas, depender do software que você está tentando corrigir como parte de seu sistema de gerenciamento de incidentes provavelmente não é algo que acabará bem.

Veja o Apêndice C, que tem um exemplo de documento de incidente. Esse documento vivo pode ser confuso, mas deve ser funcional. Usar um template facilita a geração dessa documentação, e manter as informações mais importantes no início o deixa mais utilizável. Mantenha essa documentação para análise postmortem e, se for necessário, para meta-análises.

Passagem de responsabilidade clara e direta

É essencial que o posto de coordenador do incidente seja passado claramente no final do dia de trabalho. Se estiver passando o comando para alguém que está em outra localidade, você pode simplesmente atualizar o novo

coordenador do incidente de forma segura pelo telefone ou por meio de uma chamada de vídeo. Depois que o novo coordenador do incidente estiver totalmente informado, o coordenador que estiver de saída deve ser explícito em sua passagem de responsabilidade, afirmando de modo específico que “a partir de agora, você é o coordenador do incidente, certo?”, e não deve desligar a chamada até receber uma confirmação firme da passagem. A passagem de responsabilidade deve ser comunicada às outras pessoas que estiverem trabalhando no incidente para que sempre esteja claro quem é que está na liderança dos esforços do gerenciamento do incidente.

Um incidente administrado

Vamos agora analisar como esse incidente transcorreria se tivesse sido tratado com os princípios do gerenciamento de incidentes.

São duas horas da tarde 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 servir tráfego. Ela começa a investigar. Logo depois, outro alerta é disparado e o segundo de cinco datacenters está fora de serviço. Como esse é um problema de crescimento rápido, ela sabe que se beneficiará da estrutura de seu framework de gerenciamento de incidentes.

Mary chama Sabrina. “Você pode assumir o comando?” Balançando a cabeça em sinal de concordância, Sabrina rapidamente obtém uma descrição geral de Mary sobre o que já aconteceu até agora. Ela registra esses detalhes em um email que é então enviado a uma lista de emails previamente organizada. Sabrina reconhece que ainda não é capaz de definir o escopo do impacto do incidente, então ela pede que Mary faça uma avaliação. Mary responde: “Os usuários ainda não sofreram impactos; vamos esperar que não percamos um terceiro datacenter”. Sabrina registra a resposta de Mary em um documento vivo do incidente.

Quando o terceiro alerta dispara, Sabrina vê o alerta entre a conversa sobre depuração no IRC e rapidamente faz uma atualização na discussão por email. A discussão mantém os VPs a par do status geral, sem sobrecarregá-los com minúcias. Sabrina pede a um representante de comunicações externas que comece a compor o rascunho de uma mensagem aos usuários. Ela então

consulta Mary para ver se elas devem entrar em contato com o desenvolvedor de plantão (no momento, é Josephine). Quando recebe a aprovação de Mary, Sabrina coloca Josephine no circuito.

No momento em que Josephine faz login, Robin já havia se oferecido como voluntário para ajudar. Sabrina lembra tanto Robin quanto Josephine que eles devem dar prioridade a qualquer tarefa delegada a eles por Mary, e que devem manter Mary informada sobre quaisquer ações adicionais que executarem. Robin e Josephine rapidamente se familiarizam com a situação atual lendo o documento do incidente.

A essa altura, Mary já testou a versão antiga do binário e percebeu que isso não resolveu o problema: ela lamenta e informa isso para Robin, que atualiza o IRC a fim de dizer que essa tentativa de correção não funcionou. Sabrina copia essa atualização no documento vivo de gerenciamento do incidente.

Às cinco da tarde, Sabrina começa a encontrar funcionários substitutos para assumir o incidente, pois ela e seus colegas estão prestes a ir para casa. Ela atualiza o documento do incidente. Uma conferência rápida por telefone ocorre às 5:45 para que todos estejam cientes da situação atual. Às seis da tarde, eles passam suas responsabilidades para seus colegas no escritório irmão.

Mary volta ao trabalho na manhã seguinte para descobrir que seus colegas transatlânticos assumiram a responsabilidade pelo bug, resolveram o problema, encerraram o incidente e começaram a trabalhar no postmortem. Com o problema resolvido, ela passa um pouco de café fresco e se prepara para planejar melhorias estruturais para que problemas desse tipo não aflijam a equipe novamente.

Quando declarar que há um incidente

É melhor declarar que há um incidente com antecedência e então encontrar uma correção simples e encerrar o incidente do que ter que fazer o framework de gerenciamento de incidentes funcionar por horas com um problema que cresça rapidamente. Defina condições claras para declarar um incidente. Minha equipe segue estas diretrizes gerais – se algum dos itens a seguir for

verdadeiro, o evento será um incidente:

- Você precisa envolver uma segunda equipe na correção do problema?
- A interrupção de serviço é visível aos clientes?
- O problema não foi resolvido mesmo depois de uma hora de análise concentrada?

A proficiência no gerenciamento de incidentes se atrofia rapidamente quando não está em uso constante. Então, como os engenheiros podem manter suas habilidades no gerenciamento de incidentes atualizadas – para tratar mais incidentes? Felizmente, o framework de gerenciamento de incidentes pode ser aplicado a outras mudanças operacionais que precisem se estender por diferentes fusos horários e/ou equipes. Se você usar o framework com frequência como parte normal de seus procedimentos de gerenciamento de mudanças, poderá seguir facilmente esse framework quando um incidente real ocorrer. Se sua empresa realiza testes de recuperação de desastres (você deve, é *divertido*: veja [Kri12]), o gerenciamento de incidentes deve fazer parte desse processo de teste. Muitas vezes, fazemos uma simulação da resposta a um problema ocorrido no plantão que já tenha sido resolvido, talvez por colegas em outra localidade, a fim de termos mais familiaridade com o gerenciamento de incidentes.

Resumindo

Percebemos que, ao formular uma estratégia de gerenciamento de incidentes com antecedência, estruturar esse plano para que escale de forma suave e colocar o plano regularmente em uso, fomos capazes de reduzir nosso tempo médio de recuperação e oferecer aos funcionários um modo menos estressante de trabalhar com problemas emergenciais. Qualquer empresa preocupada com confiabilidade deve se beneficiar ao perseguir uma estratégia semelhante.

Melhores práticas no gerenciamento de incidentes

Priorize. Pare o sangramento, restaure o serviço e preserve a evidência para identificar as causas-raízes.

Prepare-se. Desenvolva e documente seus procedimentos de

gerenciamento de incidentes com antecedência, consultando pessoas que participaram de incidentes.

Confie. Dê autonomia completa na função atribuída a todos os participantes do incidente.

Faça uma introspecção. Preste atenção em seu estado emocional enquanto responde a um incidente. Se você começar a entrar em pânico ou se sentir sobrecarregado, solicite mais ajuda.

Considere as alternativas. Periodicamente, considere suas opções e reavalie se ainda faz sentido continuar o que você está fazendo ou se deveria assumir outra abordagem para a resposta ao incidente.

Exercite. Utilize o processo rotineiramente para que ele se torne natural.

Faça mudanças. Você foi o coordenador do incidente na última vez? Assuma uma função diferente desta vez. Incentive todos os membros da equipe a ter familiaridade com cada função.

¹ Uma versão anterior deste capítulo foi publicada como um artigo em ;login: (abril de 2015, vol. 40, nº 2).

² N.T.: *Estimated Time for Accomplishment*, ou Tempo Estimado para Realização.

³ Acesse <http://www.fema.gov/national-incident-management-system> para ver mais detalhes.

CAPÍTULO 15

Cultura de 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 de larga escala, complexos e distribuídos. Constantemente, aperfeiçoamos nossos serviços com novas funcionalidades e acrescentamos novos sistemas. Incidentes e interrupções de serviço são inevitáveis, considerando a nossa escala e a velocidade das mudanças. Quando um incidente ocorre, corrigimos o problema subjacente e os serviços retornam às suas condições normais de operação. A menos que tenhamos um processo formalizado de aprendizado com esses incidentes, eles poderão ocorrer novamente *ad infinitum*. Se deixados sem verificação, os incidentes podem se multiplicar em complexidade ou até mesmo em cascata, sobrecarregando um sistema e seus operadores e, em última instância, impactando nossos usuários. Desse modo, os postmortems são uma ferramenta essencial para a SRE.

O conceito de postmortem é bem conhecido no mercado de tecnologia [All12]. Um postmortem é um registro por escrito de um incidente, seu impacto, as ações executadas para atenuá-lo ou resolvê-lo, a(s) causa(s)-raiz(es) e as ações de acompanhamento (follow-up) para evitar que o incidente ocorra novamente. Este capítulo descreve os critérios para decidir quando conduzir postmortems, algumas melhores práticas em torno deles e conselhos sobre como nutrir uma cultura de postmortem baseada na

experiência que adquirirmos ao longo dos anos.

A filosofia de postmortem no Google

As principais metas ao escrever um postmortem são garantir que o incidente seja documentado, que todas as causas-raízes (ou a causa-raiz) que tenham colaborado tenham sido bem compreendidas e, em especial, se ações preventivas eficazes foram tomadas para reduzir a probabilidade e/ou o impacto de uma nova ocorrência. Uma pesquisa detalhada das técnicas de análise de causas-raízes está além do escopo deste capítulo (para isso, consulte [Roo04]); contudo, artigos, melhores práticas e ferramentas são abundantes no domínio da qualidade de sistemas. Nossas equipes utilizam técnicas variadas para análise de causa-raiz e escolhem a técnica que seja mais adequada aos seus serviços. Espera-se que os postmortems sejam escritos após qualquer evento indesejável significativo. Escrever um postmortem não é uma punição – é uma oportunidade de aprendizado para toda a empresa. O processo de postmortem tem um custo inerente em termos de tempo ou de esforço, portanto somos criteriosos para decidir quando devemos escrever um. As equipes têm certa flexibilidade interna, mas eventos comuns que acionam um postmortem incluem:

- Downtime visível ao usuário ou degradação além de determinado limite
- Perda de dados de qualquer tipo
- Intervenção do engenheiro de plantão (rollback de versão, desvio de tráfego etc.)
- Tempo de resolução acima de certo limite
- Uma falha de monitoração (que normalmente implica uma descoberta manual de um incidente)

É importante definir critérios de postmortem antes que um incidente ocorra para que todos saibam quando um postmortem é necessário. Além desses eventos objetivos que acionam um postmortem, qualquer stakeholder pode solicitar um para um evento.

Postmortems sem acusações são um princípio da cultura de SRE. Para que um postmortem realmente não contenha acusações, ele deve se concentrar na

identificação das causas que contribuíram para o incidente, sem acusar nenhum indivíduo ou equipe por um comportamento ruim ou inadequado. Um postmortem sem acusações supõe que todas as pessoas envolvidas em um incidente tinham boas intenções e fizeram o que era certo com as informações que possuíam. Se uma cultura de apontar culpados e constranger indivíduos ou equipes por fazerem algo “errado” prevalecer, as pessoas não trarão os problemas à tona por medo de punição.

A cultura de não acusar teve origem nos mercados de saúde e de aviação, em que erros podem ser fatais. Esses mercados cultivam um ambiente em que todo “erro” é visto como uma oportunidade para fortalecer o sistema. Quando os postmortems passam de identificar culpados para investigar os motivos sistemáticos pelos quais um indivíduo ou uma equipe tinham informações incompletas ou incorretas, planos eficazes de prevenção podem ser colocados em prática. Você não pode “corrigir” pessoas, mas pode corrigir sistemas e processos para dar melhor suporte às pessoas, de modo que elas façam as escolhas corretas ao fazer o design e a manutenção de sistemas complexos.

Quando uma interrupção de serviço ocorre, um postmortem não é escrito como uma formalidade para ser esquecido. Em vez disso, o postmortem é visto pelos engenheiros como uma oportunidade não só para corrigir um ponto fraco, mas também para deixar o Google mais resiliente como um todo. Um postmortem sem acusações não serve apenas para evitar a frustração gerada por apontar culpados, mas *deve* apresentar em que pontos e como os serviços podem ser melhorados. A seguir, apresentamos dois exemplos:

Apontando culpados

“Precisamos reescrever todo o sistema complicado de backend! Ele vem falhando toda semana nos últimos três trimestres e tenho certeza que estamos todos cansados de corrigir problemas aqui e ali. É sério, se eu receber mais um page, eu mesmo vou reescrevê-lo...”

Sem acusações

“Uma ação para reescrever todo o sistema de backend, na verdade, pode evitar que esses pages irritantes continuem acontecendo, e o manual de manutenção dessa versão é muito longo e realmente dificulta termos um

treinamento completo sobre ele. Tenho certeza que os futuros engenheiros de plantão nos agradecerão!”

Melhor prática: evite acusações e seja construtivo

Postmortems sem acusações podem ser desafiadores para escrever, pois o formato do postmortem identifica claramente as ações que resultaram no incidente. Remover as acusações de um postmortem dá confiança às pessoas para escalar os problemas sem medo. Também é importante não estigmatizar uma produção frequente de postmortems por uma pessoa ou uma equipe. Uma atmosfera de acusações gera o risco de criar uma cultura em que incidentes e problemas são varridos para baixo do tapete, resultando em mais riscos para a empresa [Boy13].

Colabore e compartilhe conhecimentos

Valorizamos a colaboração, e o processo de postmortem não é uma exceção. O fluxo de trabalho do postmortem inclui colaboração e compartilhamento de conhecimentos em todas as etapas.

Nossos documentos de postmortem são feitos no Google Docs, com um template interno (veja o Apêndice D). Independentemente da ferramenta específica que você usar, observe as seguintes características fundamentais:

Colaboração em tempo real

Permite a coleta rápida de dados e ideias. É essencial no início da criação de um postmortem.

Um sistema aberto para comentários/anotações

Facilita soluções de crowdsourcing⁴ e melhora a abrangência.

Notificações por email

Podem ser direcionadas aos que colaboram com o documento ou usadas para incluir outras pessoas para que forneçam informações.

Escrever um postmortem também envolve uma revisão formal e a publicação. Na prática, as equipes compartilham a primeira versão preliminar do postmortem internamente e pedem a um grupo de engenheiros mais

experientes que avaliem se ela está completa. Os critérios de revisão podem incluir o seguinte:

- Dados essenciais do incidente foram coletados para a posteridade?
- As avaliações de impacto estão completas?
- A causa-raiz foi suficientemente profunda?
- O plano de ação é apropriado e está resultando em correções de bug com a prioridade adequada?
- Compartilhamos o resultado com stakeholders relevantes?

Depois que a revisão inicial estiver concluída, o postmortem é compartilhado de forma mais abrangente, em geral com a equipe mais ampla de engenharia ou uma lista de emails interna. Nossa meta é compartilhar postmortems com o público-alvo mais amplo possível que possa se beneficiar com o conhecimento ou com as lições compartilhadas. O Google tem regras rigorosas para o acesso a qualquer informação que possa identificar um usuário², e até mesmo documentos internos como postmortems jamais incluem informações desse tipo.

Melhor prática: nenhum postmortem deve ficar sem revisão

Um postmortem não revisado poderia muito bem nem ter existido. Para garantir que cada versão preliminar concluída seja revisada, incentivamos sessões regulares de revisão para postmortems. Nessas reuniões, é importante encerrar qualquer discussão e comentários em andamento, registrar ideias e finalizar o estado.

Depois que as pessoas envolvidas estiverem satisfeitas com o documento e sua lista de ações, o postmortem é adicionado a um repositório de incidentes passados de uma equipe ou da empresa.³ Um compartilhamento transparente faz com que seja mais fácil às outras pessoas encontrar o postmortem e aprender com ele.

Introduzindo uma cultura de postmortem

É mais fácil falar da introdução de uma cultura de postmortem em sua empresa do que fazê-lo; uma empreitada desse tipo exige cultivo e reforço

contínuos. Incentivamos uma cultura de postmortem colaborativa por meio da participação ativa da gerência sênior nos processos de revisão e de colaboração. A gerência pode incentivar essa cultura, mas o ideal é que os postmortems sem acusações sejam o produto da própria motivação dos engenheiros. No espírito de nutrir uma cultura de postmortem, os SREs, de forma proativa, criam atividades que disseminam o que aprendemos sobre a infraestrutura dos sistemas. Alguns exemplo de atividades incluem:

Postmortem do mês

Em um boletim mensal, um postmortem interessante e bem escrito é compartilhado com toda a empresa.

Grupo de postmortem no Google+

Esse grupo compartilha e discute postmortems internos e externos, melhores práticas e comentários sobre postmortems.

Clubes de leitura de postmortems

As equipes mantêm clubes regulares de leitura de postmortems, em que um postmortem interessante ou que teve impacto significativo é colocado na mesa (juntamente com alguns salgadinhos) para um diálogo aberto com participantes, não participantes e novos Googlers sobre o que aconteceu, quais lições foram aprendidas com o incidente e os efeitos após a sua ocorrência. Com frequência, o postmortem analisado tem meses ou anos!

Wheel of Misfortune (Roda do Azar)

Novos SREs muitas vezes são contemplados com uma participação no Wheel of Fortune (veja a seção “Interpretando papéis em situações de desastre”), em que um postmortem anterior é representado com um elenco de engenheiros desempenhando papéis conforme apresentados no postmortem. O coordenador do incidente original participa para ajudar a deixar a experiência o mais “real” possível.

Um dos maiores desafios ao introduzir postmortems em uma empresa é que algumas pessoas podem questionar seu valor, considerando o custo de sua preparação. As estratégias a seguir podem ajudar a encarar esse desafio:

- Introduza os postmortems no fluxo de trabalho de forma suave. Um

período de experiência com vários postmortems completos e bem-sucedidos pode ajudar a provar que eles são importantes, além de ajudar a identificar os critérios que devem dar início a um postmortem.

- Certifique-se de que escrever postmortems eficientes seja uma prática recompensada e celebrada, tanto publicamente, por meio dos métodos sociais mencionados antes, quanto pelo gerenciamento de desempenho individual e da equipe.
- Incentive o reconhecimento e a participação da liderança sênior. Até mesmo Larry Page fala da grande importância dos postmortems!

Melhor prática: recompense visivelmente as pessoas por fazerem o que é certo

Larry Page e Sergey Brin, fundadores do Google, apresentam o TGIF, uma reunião geral semanal ao vivo em nossa matriz em Mountain View, na Califórnia, transmitida para os escritórios do Google no mundo todo. Um TGIF de 2014 focou a “Arte do postmortem”, que apresentou uma discussão de SREs sobre incidentes de alto impacto. Um SRE falou sobre uma recente atualização de versão; apesar de testes completos terem sido feitos, uma interação inesperada inadvertidamente derrubou um serviço crítico por quatro minutos. O incidente só durou quatro minutos porque o SRE teve a presença de espírito de fazer o rollback da alteração imediatamente, evitando uma interrupção muito mais longa e de maior escala no serviço. Esse engenheiro não só recebeu dois bônus de seus colegas (peer bonus)⁴ logo depois, em reconhecimento pelo seu tratamento rápido e sereno do incidente, como também ganhou muitos aplausos do público do TGIF, que incluía os fundadores da empresa, além de Googlers na casa dos milhares. Além de um fórum visível como esse, o Google tem um conjunto de redes sociais internas em que colegas de trabalho podem elogiar postmortems bem escritos e tratamentos excepcionais a incidentes. Esse é um exemplo entre muitos em que o reconhecimento por essas contribuições é proveniente de colegas de trabalho, CEOs e de todos entre eles.⁵

Melhor prática: peça feedback sobre a eficiência do postmortem

No Google, nós nos esforçamos para abordar os problemas à medida que eles surgem e compartilhamos inovações internamente. Com regularidade, fazemos pesquisas junto a nossas equipes sobre como o processo de postmortem está dando suporte às suas metas e como o processo pode ser melhorado. Fazemos perguntas como: A cultura tem dado suporte ao seu trabalho? Escrever um postmortem implica muitas tarefas penosas (toil work – veja o Capítulo 5)? Quais são as melhores práticas que sua equipe recomenda para outras equipes? Que tipos de ferramentas você gostaria que fossem desenvolvidas? Os resultados da pesquisa dão aos SREs nas trincheiras a oportunidade de solicitar melhorias que aumentarão a eficiência da cultura de postmortem.

Além dos aspectos operacionais do gerenciamento e acompanhamento de incidentes, a prática de postmortem tem se entrelaçado na cultura do Google: atualmente, o fato de qualquer incidente significativo ser seguido de um postmortem abrangente é uma norma cultural.

Conclusão e melhorias contínuas

Podemos dizer com confiança que, graças ao nosso investimento contínuo em nutrir uma cultura de postmortem, o Google enfrenta menos interrupções de serviços e promove uma melhor experiência aos usuários. Nossa grupo de trabalho “Postmortems no Google” é um exemplo de nosso comprometimento com a cultura de postmortems sem acusações. Esse grupo coordena esforços voltados a postmortems em toda a empresa: reunindo templates para postmortems, automatizando a criação de postmortems com dados de ferramentas usadas durante um incidente e ajudando na automatização da extração de dados de postmortems para que possamos fazer análises de tendências. Temos sido capazes de contribuir com melhores práticas provenientes de produtos tão distintos quanto YouTube, Google Fiber, Gmail, Google Cloud, AdWords e Google Maps. Embora esses produtos sejam muito diferentes, todos eles conduzem postmortems com a meta universal de aprender com nossas horas mais escuras.

Com um grande número de postmortems gerados a cada mês no Google,

ferramentas para agregá-los estão se tornando cada vez mais úteis. Essas ferramentas nos ajudam a identificar temas comuns e áreas de melhoria que cruzem as fronteiras dos produtos. Para facilitar a compreensão e uma análise automatizada, recentemente aperfeiçoamos nosso template de postmortem (veja o Apêndice D) com campos adicionais para metadados. Futuros trabalhos nesse domínio incluem aprendizagem automática (machine learning) para ajudar a prever nossos pontos fracos, facilitar a investigação de incidentes em tempo real e reduzir incidentes duplicados.

¹ N.T.: *Crowdsourcing* é “...o processo de obtenção de serviços, ideias ou conteúdo mediante a solicitação de contribuições de um grande grupo de pessoas e, especialmente, de uma comunidade online em vez de usar fornecedores tradicionais ou uma equipe de empregados”. (Fonte: <https://pt.wikipedia.org/wiki/Crowdsourcing>)

² Veja <http://www.google.com/policies/privacy/>.

³ Se quiser dar início ao seu próprio repositório, a Etsy disponibilizou o *Morgue* (<https://github.com/etsy/morgue>), que é uma ferramenta para gerenciamento de postmortems.

⁴ O programa Peer Bonus (Bônus de colegas) do Google é uma maneira de os Googlers reconhecerem seus colegas por esforços excepcionais e envolve uma recompensa em dinheiro.

⁵ Para ver mais discussões sobre esse incidente em particular, veja o Capítulo 13.

CAPÍTULO 16

Monitorando interrupções de serviço

Escrito por Gabe Krabbe

Editado por Lisa Carey

Melhorar a confiabilidade com o tempo só é possível se você começar com uma base de referência conhecida e puder monitorar o seu progresso. O “Outalator” – nosso monitor de interrupções de serviço – é uma das ferramentas que usamos para fazer exatamente isso. O Outalator é um sistema que recebe passivamente todos os alertas enviados pelos nossos sistemas de monitoração e nos permite fazer anotações nesses dados, agrupá-los e analisá-los.

Aprender de forma sistemática com problemas passados é essencial para um gerenciamento de serviços eficiente. Os postmortems (veja o Capítulo 15) oferecem informações detalhadas sobre interrupções individuais de serviço, mas são apenas uma parte da resposta. Eles são escritos apenas no caso de incidentes com impacto significativo, portanto problemas que tenham individualmente um impacto pequeno, mas que são frequentes e estejam espalhados, não se enquadram em seu escopo. De modo semelhante, os postmortems tendem a oferecer insights úteis para melhorar um único serviço ou um conjunto de serviços, mas podem deixar passar oportunidades que teriam um pequeno efeito em casos individuais, ou oportunidades que tenham uma razão custo/benefício ruim, mas que teriam impactos horizontais significativos.¹

Também podemos obter informações úteis a partir de perguntas como: “Quantos alertas por turno de plantão essa equipe recebe?”, “Qual é a razão entre alertas passíveis de ação ou não no último trimestre?”, ou até mesmo de uma pergunta simples como “Quais são os serviços administrados por essa equipe que geram mais tarefas penosas (toil)?”.

Escalator

No Google, todas as notificações de alerta para a SRE compartilham um sistema central replicado que monitora se um ser humano confirmou a recepção de uma notificação. Se nenhuma confirmação for recebida após um período de tempo configurado, o sistema escalará para o(s) próximo(s) destino(s) configurado(s) – por exemplo, do plantão principal para o plantão secundário. Esse sistema, chamado de “The Escalator”, foi inicialmente projetado como uma ferramenta bem transparente que recebia cópias dos emails enviados para aliases de plantão. Essa funcionalidade permitia que o Escalator se integrasse facilmente nos fluxos de trabalho existentes, sem exigir qualquer mudança no comportamento dos usuários (ou, naquela época, no comportamento do sistema de monitoração).

Outalator

Seguindo o exemplo do Escalator, em que adicionamos funcionalidades úteis na infraestrutura existente, criamos um sistema que lidaria não só com notificações individuais para escalar, mas com a próxima camada de abstração: as interrupções de serviço.

O Outalator permite que os usuários visualizem uma lista de notificações organizadas no tempo, para várias filas simultaneamente, em vez de exigir que um usuário alterne entre as filas de forma manual. A Figura 16.1 mostra várias filas conforme elas aparecem na apresentação das filas do Outalator. Essa funcionalidade é prática, pois, com frequência, uma única equipe de SRE é o ponto de contato principal para serviços cujos alvos secundários para escalar – normalmente, são as equipes de desenvolvedores – são distintos.

The screenshot shows the Outalator web application interface. At the top, there's a navigation bar with links for 'Settings', 'About', 'Feedback', and 'Sign out'. Below the navigation is a search bar with placeholder text 'e.g. has:bug tag:cause:human-error -summary:"global"'. To the left, there's a sidebar titled 'Teams/Alert queues:' with two items: 'doodle-serving' and 'shakespeare'. The main content area is titled 'Tickets / Outages' and contains a table with the following data:

Team, From	Summary	Date
▶ Shakespeare (3), agoogler (7)	probe ShakespeareBlackboxProbe_SearchFailure bug:94043	2015-07-24 11:32:59 PDT
▶ Shakespeare, agoogler	frontend TaskFlapping bug:90210	2015-07-22 13:15:09 PDT
▶ Shakespeare, jrandom	frontend ManyHttp500s cl:8675309 bug:89191	2015-07-22 04:19:44 PDT
▶ Shakespeare, agoogler (2)	storage AnnotationConsistencyTooEventual	2015-07-21 19:31:12 PDT
▶ Shakespeare, jrandom	frontend HighSearchLatency cause:alert-tuning bug:89109 action:silence	2015-07-20 03:35:43 PDT
5 Outalations		

Figura 16.1 – Apresentação das filas no Outalator.

O Outalator armazena uma cópia da notificação original e permite inserir anotações nos incidentes. Por questões de conveniência, ele também recebe e salva silenciosamente uma cópia de qualquer resposta a emails. Como algumas ações de acompanhamento (follow-ups) são menos úteis do que outras (por exemplo, uma resposta enviada a todos com o único propósito de acrescentar mais receptores na lista cc), as anotações podem ser marcadas como “importante”. Se uma anotação for importante, outras partes da mensagem são contraídas na interface para reduzir o excesso de informações. Em conjunto, isso oferece mais contexto para se referenciar a um incidente do que um encadeamento de emails possivelmente fragmentado.

Várias notificações para escalar (“alertas”) podem ser combinadas em uma única entidade (“incidente”) no Outalator. Essas notificações podem estar relacionadas ao mesmo incidente, podem ser eventos não relacionados e desinteressantes usados para auditoria (por exemplo, acessos privilegiados a bancos de dados), ou podem ser falhas espúrias de monitoração. Essa funcionalidade de agrupamento, como mostra a Figura 16.2, elimina o excesso de informações da visão geral exibida e permite uma análise separada de “incidentes por dia” versus “alertas por dia”.

The screenshot shows the Outalator web interface for managing incidents. At the top, there's a navigation bar with links for 'Settings', 'About', 'Feedback', and 'Sign out'. Below the header, a banner says 'Hit ? to see keyboard shortcuts. See help, and what could be better.' On the left, a sidebar lists 'Teams/Alert queues' including 'doodle-serving' and 'shakespeare'. The main content area displays a ticket summary for 'Ticket m23bca7d408000005'. The ticket details include:

- Summary:** frontend ManyHttp500s
- Team:** shakespeare
- Escalation Level:** 1
- Status:** closed
- Parent Outage:** None
- Short Link:** <http://o/e/m23bca7d408000005>
- Escalator Link:** <http://escalator/2575117616058204165>
- Tags:** bug:89191, cl:8675309

Below the summary, there are sections for 'Content' and 'Followups'.

Content: Condition 'ManyHttp500s' was triggered.
 Job: shakespeare.frontend
 Zone: europe
 Dashboard: <http://console/shakespeare/frontend>
 Playbook: http://playbooks/shakespeare/ManyHttp500s_frontend

Followups:

Time	From
2015-07-22 04:20:01 PDT	agoogler
2015-07-22 05:10:52 PDT	jrandom

Frontend task dropping requests into the bit bucket, fixed by [cl/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

At the bottom right of the interface, there are three small decorative icons: a blue sphere, a grey sphere, and a white sphere.

Figura 16.2 – Apresentação de um incidente no Outalator.

Construindo seu próprio Outalator

Muitas empresas usam sistemas de mensagens como Slack, Hipchat ou até mesmo o IRC para comunicação interna e/ou atualização de painéis de status. Esses sistemas são ótimos lugares para conectar um sistema como o Outalator.

Agregação

Um único evento pode acionar vários alertas, e muitas vezes o fará. Por exemplo, falhas de rede provocam timeouts e serviços de backend inacessíveis para todos, portanto todas as equipes afetadas receberão seus próprios alertas, incluindo os proprietários dos serviços de backend; enquanto isso, o centro de operações de rede terá seus próprios alarmes disparando.

Entretanto, até mesmo problemas menores que afetem um único serviço podem disparar vários alertas devido ao fato de diversas condições de erro serem diagnosticadas. Embora valha a pena tentar minimizar o número de alertas disparado por um único evento, acionar vários alertas é inevitável na maioria dos cálculos de custo/benefício entre falso-positivos e falso-negativos.

A capacidade de agrupar vários alertas em um único *incidente* é fundamental para lidar com essa duplicação. Enviar um email informando que “isso é igual àquilo; são sintomas comuns do mesmo incidente” funciona para um dado alerta: pode evitar duplicação de depuração ou pânico. Porém, enviar um email para cada alerta não é uma solução prática nem escalável para tratar alertas duplicados em uma equipe, muito menos entre equipes ou por períodos de tempo mais longos.

Atribuição de rótulos

É claro que nem todo evento de alerta é um incidente. Alertas falso-positivos ocorrem, assim como eventos de teste ou emails com destino incorreto gerados por seres humanos. O próprio Outalator não faz a distinção entre esses eventos, mas permite uma *atribuição de rótulos* (tagging) de propósito geral para adicionar metadados às notificações em qualquer nível. Os rótulos, na maioria das vezes, são “palavras” únicas de formato livre. Dois-pontos, porém, são interpretados como separadores semânticos, que sutilmente promovem o uso de namespaces hierárquicos e permitem algum tratamento automático. Esse namespacing conta com o suporte de prefixos formados por rótulos sugeridos, principalmente “cause” (causa) e “action” (ação), mas a lista é específica para cada equipe e é gerada com base no uso histórico. Por exemplo, “cause:network” pode conter informações suficientes para algumas equipes, enquanto outras podem optar por rótulos mais específicos, como “cause:network:switch” versus “cause:network:cable”. Algumas equipes podem usar rótulos no estilo “customer:132456” com frequência, portanto “customer” seria uma sugestão para essas equipes, mas não para outras.

Os rótulos podem ser interpretados e transformados em um link conveniente (“bug:76543” tem um link para o sistema de monitoração de bugs). Outros

rótulos são compostos de apenas uma única palavra (“bogus” [falso] é amplamente usado para falso-positivos). É claro que alguns rótulos contêm erros de digitação (“cause:netwrok”), enquanto outros não são particularmente úteis (“problem-went-away”), mas evitar uma lista predeterminada e permitir que as equipes definam suas próprias preferências e padrões resulta em uma ferramenta mais útil e em dados melhores. De modo geral, os rótulos têm sido uma ferramenta notadamente eficaz para as equipes obterem e fornecerem uma visão geral dos pontos problemáticos de um dado serviço, sem muita ou nem mesmo nenhuma análise formal. Embora pareça trivial, a atribuição de rótulos provavelmente é uma das funcionalidades individuais mais úteis do Outalator.

Análise

É claro que a SRE faz muito mais do que reagir a incidentes. Dados históricos são úteis quando alguém está respondendo a um incidente – a pergunta “o que fizemos da última vez?” é sempre um bom ponto de partida. Porém, informações históricas são muito mais úteis quando dizem respeito a problemas sistemáticos, periódicos ou mais amplos. Possibilitar uma análise desse tipo é uma das funções mais importantes de uma ferramenta de monitoração de interrupções de serviço.

A camada mais baixa da análise inclui contadores e estatísticas básicas agregadas 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 oferecer: comparação entre equipes/serviços e tempo adicional para identificar os primeiros padrões e tendências. Essa camada permite que as equipes determinem se uma dada carga de alertas é “normal” em relação aos seus próprios registros e aos registros de outros serviços. “Essa é a terceira vez que isso ocorre nesta semana” pode ser bom ou ruim, mas saber se “isso” costumava ocorrer cinco vezes por dia ou cinco vezes por mês permite fazer uma interpretação.

O próximo passo na análise de dados consiste em descobrir problemas mais amplos, que não sejam apenas contadores brutos, mas exijam um pouco de

análise semântica. Por exemplo, a capacidade de identificar o componente da infraestrutura que está causando a maior parte dos incidentes e, desse modo, conhecer a vantagem em potencial resultante do aumento da estabilidade ou do desempenho desse componente² pressupõe que haja uma maneira simples de fornecer essas informações juntamente com os registros de incidentes. Como um exemplo simples, equipes diferentes têm condições de alerta específicas do serviço, como “dados vencidos” ou “alta latência”. As duas condições podem ter sido provocadas por congestionamento de rede, que resultou em atrasos na replicação de bancos de dados e exige intervenção. Ou elas poderiam estar dentro do objetivo de nível de serviço nominal, mas não estão atendendo às expectativas mais altas dos usuários. Analisar essas informações em várias equipes nos permite identificar problemas sistêmicos e escolher a solução correta, especialmente se a solução puder ser a introdução de falhas mais artificiais para acabar com um desempenho exagerado.

Informando e comunicando

De uso mais imediato para os SREs na linha de frente está a capacidade de selecionar zero ou mais outalations e incluir seus assuntos, rótulos e anotações “importantes” em um email para o próximo engenheiro de plantão (e para uma lista qualquer de cc) a fim de informar o estado mais recente na troca de turnos. Para revisões periódicas dos serviços de produção (que ocorrem semanalmente para a maioria das equipes), o Outalator também tem suporte para um “modo de relatório”, em que as anotações importantes são expandidas na lista principal a fim de oferecer uma visão geral rápida das informações de menor destaque.

Benefícios inesperados

Ser capaz de identificar se um alerta, ou uma enxurrada deles, coincide com uma outra interrupção de serviço específica tem vantagens evidentes: aumenta a velocidade do diagnóstico e reduz a carga de outras equipes ao reconhecer que há, na verdade, um incidente. Existem outras vantagens adicionais não óbvias. Para usar o Bigtable como exemplo, se um serviço tiver um distúrbio por causa de um incidente aparente no Bigtable, mas você puder perceber que a equipe de SRE do Bigtable ainda não foi alertada, é

provável que alertá-la manualmente seja uma boa ideia. Uma visibilidade melhor entre as equipes pode e faz uma grande diferença na resolução de incidentes ou, no mínimo, em sua atenuação.

Algumas equipes na empresa foram mais longe, a ponto de escalar configurações dummy no Escalator: nenhum ser humano recebe as notificações enviadas ali, mas elas aparecem no Outalator e podem receber rótulos e anotações e ser analisadas. Um exemplo de uso desse “sistema de registros” é fazer log e auditar a utilização de contas privilegiadas ou associadas a funções (embora devemos observar que essa funcionalidade é básica e é utilizada para auditorias técnicas, e não legais). Outro uso é registrar e fazer anotações automáticas de execuções de jobs periódicos que possam não ser idempotentes – por exemplo, aplicação automática de mudanças de esquema do sistema de controle de versões para os sistemas de banco de dados.

¹ Por exemplo, talvez uma alteração em particular no Bigtable que tenha apenas um pequeno efeito atenuador em uma interrupção de serviço exija um significativo esforço de engenharia. No entanto, se essa mesma atenuação estiver disponível para muitos eventos, o esforço de engenharia pode realmente valer a pena.

² Por um lado, “causando a maior parte dos incidentes” é um bom ponto de partida para reduzir o número de alertas disparados e melhorar o sistema em geral. Por outro, essa métrica pode ser simplesmente um artefato de uma monitoração sensível demais ou de um pequeno conjunto de sistemas clientes se comportando mal ou executando fora do nível de serviço combinado. Além disso, o número de incidentes por si só não oferece nenhuma indicação da dificuldade de correção nem da severidade do impacto.

CAPÍTULO 17

Testes voltados à confiabilidade

Escrito por Alex Perry e Max Luebbe

Editado por Diane Bates

Se você não testou, pressuponha que tem falha.

— Desconhecido

Uma responsabilidade fundamental dos Site Reliability Engineers (Engenheiros de Confiabilidade de Sites) é quantificar a confiança nos sistemas aos quais eles dão manutenção. Os SREs realizam essa tarefa adaptando técnicas clássicas de testes de software a sistemas em escala.¹ A confiança pode ser medida tanto pela confiabilidade passada quanto pela confiabilidade futura. A primeira é capturada por meio de análise de dados fornecidos pela monitoração do comportamento histórico do sistema, enquanto a última é quantificada com previsões feitas a partir de dados sobre o comportamento do sistema no passado. Para que essas previsões sejam robustas o suficiente a ponto de serem úteis, uma das condições a seguir deve ser verdadeira:

- O site permanece totalmente inalterado com o tempo, sem novas versões de software nem mudanças na frota de servidores, o que significa que comportamentos futuros serão semelhantes aos comportamentos passados.
- Você é capaz de descrever todas as alterações no site com confiança, de modo que as análises permitam avaliar as incertezas resultantes de cada uma dessas mudanças.

Os testes são o mecanismo que você usa para demonstrar áreas específicas de equivalência quando mudanças ocorrem.² Cada teste que passe tanto antes quanto depois de uma mudança reduz a incerteza que a análise deve permitir. Testes abrangentes nos ajudam a prever a futura confiabilidade de um dado

site, com detalhes suficientes para que sejam úteis do ponto de vista prático.

A quantidade de testes que você deve realizar depende dos requisitos de confiabilidade de seu sistema. À medida que o percentual de sua base de código coberta pelos testes aumentar, você reduzirá a incerteza e o potencial decréscimo de confiabilidade associado a cada mudança. Uma cobertura de testes adequada significa que você pode fazer mais mudanças antes que a confiabilidade diminua e atinja um nível abaixo do aceitável. Se você fizer mudanças demais de modo muito rápido, a confiabilidade prevista se aproximará do limite aceitável. A essa altura, talvez você queira parar de fazer alterações, enquanto novos dados de monitoração são acumulados. Os dados acumulados suplementam a cobertura dos testes, o que valida a confiabilidade garantida pelos caminhos de execução analisados. Supondo que os clientes servidos estejam aleatoriamente distribuídos [Woo96], amostragens estatísticas podem ser extrapoladas com base em métricas monitoradas se o comportamento agregado estiver fazendo uso de novos caminhos. Essas estatísticas identificam as áreas que precisam de testes melhores ou de outras adaptações.

Relações entre testes e tempo médio de reparação

Passar em um teste ou em uma série de testes não prova, necessariamente, que haja confiabilidade. No entanto, testes que falham em geral provam a ausência dela.

Um sistema de monitoração pode revelar bugs, mas apenas com a velocidade com que o processo que os informa é capaz de reagir. O *MTTR* (*Mean Time To Repair*, ou Tempo Médio Para Reparação) calcula quanto tempo demora para a equipe de operações corrigir o bug, seja por meio de um rollback, seja com outra ação.

É possível que um sistema de testes identifique um bug com MTTR zero. Um MTTR zero ocorre quando um teste de nível de sistema é aplicado a um subsistema, e esse teste detecta exatamente o mesmo problema que a monitoração detectaria. Um teste como esse permite que a atualização de software seja bloqueada de modo que o bug jamais alcance a produção (embora ainda precise ser corrigido no código-fonte). Corrigir bugs com

MTTR zero fazendo o bloqueio de uma atualização de versão é uma solução rápida e conveniente. Quanto mais bugs você puder encontrar com MTTR zero, maior será o *MTBF* (*Mean Time Between Failures*, ou Tempo Médio Entre Falhas) experimentado pelos seus usuários.

À medida que o MTBF aumentar em resposta a testes mais adequados, os desenvolvedores serão incentivados a entregar funcionalidades de modo mais rápido. Algumas dessas funcionalidades, é claro, conterão bugs. Novos bugs resultam em um ajuste oposto na velocidade da entrega de novas versões à medida que forem encontrados e corrigidos.

Os autores que escrevem sobre testes de software em geral concordam sobre a abrangência necessária. A maioria dos conflitos de opinião se origina de terminologia conflitante, ênfases diferentes quanto ao impacto dos testes em cada uma das fases do ciclo de vida do software ou de particularidades dos sistemas em que os testes são realizados. Para ver uma discussão geral sobre testes no Google, consulte [Whi12]. As próximas seções especificam como a terminologia relacionada a testes de software é usada neste capítulo.

Tipos de testes de software

Os testes de software, de modo amplo, se classificam em duas categorias: tradicionais e de produção. Os testes tradicionais são mais comuns em desenvolvimento de software para avaliar se o software offline está correto durante o desenvolvimento. Os testes de produção são realizados em um serviço web ativo para avaliar se um sistema de software implantado está funcionando corretamente.

Testes tradicionais

Como mostra a Figura 17.1, testes tradicionais de software começam com os testes de unidade (unit tests). Testes de funcionalidades mais complexas estão em camadas acima dos testes de unidade.

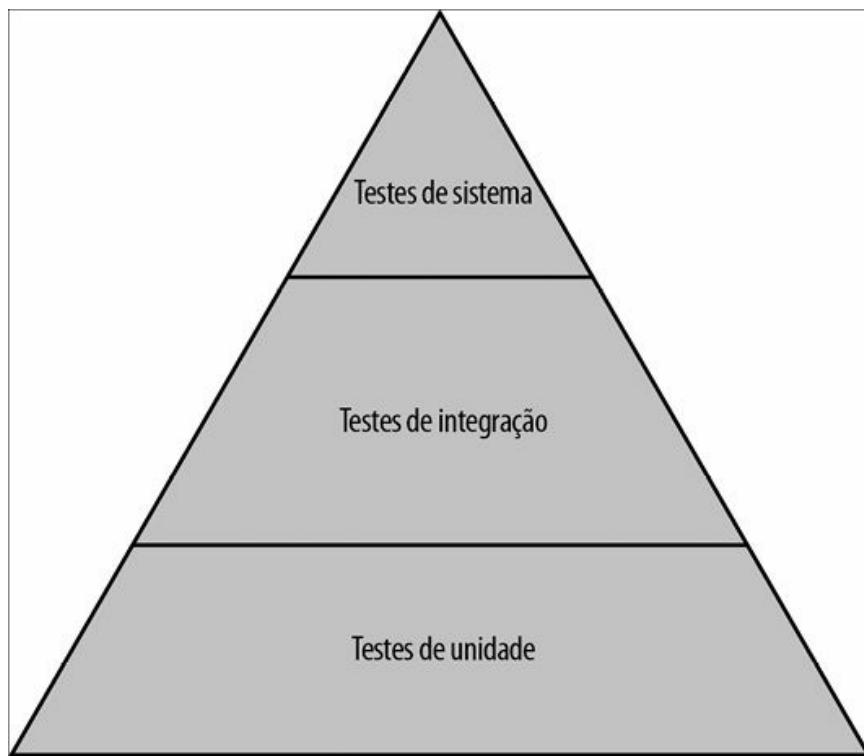


Figura 17.1 – A hierarquia dos testes tradicionais.

Testes de unidade

Um *teste de unidade* ou *teste unitário* (unit test) é a menor e mais simples forma de teste de software. Esses testes são empregados para avaliar uma unidade de software separável, por exemplo, uma classe ou uma função, para ver se está correta, independentemente do sistema de software maior que contém essa unidade. Os testes de unidade também são empregados como uma forma de especificação para garantir que uma função ou um módulo tenham exatamente o comportamento necessário ao sistema. Os testes de unidade são comumente utilizados para introduzir conceitos de desenvolvimento orientado a testes.

Testes de integração

Os componentes de software que passam pelos testes de unidade individuais são reunidos em componentes maiores. Os engenheiros então executam um *teste de integração* (integration test) em um componente composto dessa maneira para verificar se ele funciona corretamente. A injeção de dependências, realizada com ferramentas como o Dagger³, é uma técnica

extremamente eficaz para criar simulações (mocks) de dependências complexas para que um engenheiro possa testar um componente de modo limpo. Um exemplo comum de uma injeção de dependências consiste em substituir um banco de dados stateful (com estado) por um simulador leve que tenha exatamente o comportamento especificado.

Testes de sistema

Um *teste de sistema* ou *teste sistêmico* (system test) é o teste de maior escala que os engenheiros podem realizar em um sistema ainda não implantado. Todos os módulos que pertencem a um componente específico, por exemplo, um servidor que tenha passado pelos testes de integração, são reunidos para compor o sistema. Então o engenheiro testa a funcionalidade fim a fim do sistema. Os testes de sistema têm muitas variantes diferentes:

Testes de fumaça

Os *testes de fumaça* (smoke tests), em que os engenheiros testam comportamentos muito simples, porém cruciais, estão entre os tipos mais simples de testes de sistema. Os testes de fumaça também são conhecidos como *testes de sanidade* (sanity testing), e servem para barrar testes adicionais e mais custosos.

Testes de desempenho

Depois que um nível básico de correção é definido por meio de um teste de fumaça, um próximo passo comum é escrever outra variante de um teste de sistema para garantir que o desempenho do sistema permaneça aceitável durante seu ciclo de vida. Como os tempos de resposta das dependências ou os requisitos para recursos podem mudar drasticamente durante o curso do desenvolvimento, um sistema deve ser testado para garantir que não se torne lento de forma incremental, sem que ninguém perceba (antes de ser disponibilizado aos usuários). Por exemplo, um dado programa pode evoluir de modo a precisar de 32 GB de memória quando antes precisava somente 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 degradará nem se tornará custoso demais.

Testes de regressão

Outro tipo de teste de sistema envolve evitar que bugs se insinuem novamente na base de código. Podemos fazer uma analogia entre os testes de regressão com uma galeria de bugs ardilosos que, historicamente, fizeram o sistema falhar ou gerar resultados incorretos. Ao documentar esses bugs como testes no nível de sistema ou de integração, os engenheiros que fazem refatoração da base de código podem ter certeza de que não introduzirão accidentalmente bugs aos quais já investiram tempo e esforço para eliminar.

É importante observar que os testes têm um custo, tanto em termos de tempo quanto de recursos computacionais. Em uma extremidade, os testes de unidade têm custo muito baixo nas duas dimensões, pois geralmente podem ser concluídos em milissegundos com os recursos disponíveis em um notebook. Na outra extremidade, configurar um servidor completo com as dependências necessárias (ou com simuladores equivalentes) para executar testes relacionados pode exigir muito mais tempo – de vários minutos a muitas horas – e, possivelmente, até mesmo recursos computacionais dedicados. Estar ciente desses custos é essencial para a produtividade dos desenvolvedores, além de incentivar um uso mais eficiente dos recursos de testes.

Testes de produção

Os testes de produção interagem com um sistema ativo em produção, em oposição a um sistema em um ambiente de testes hermético. Esses testes, em muitos aspectos, são semelhantes à monitoração caixa-preta (veja o Capítulo 6) e, desse modo, às vezes são chamados de *testes caixa-preta* (black-box testing). Os testes de produção são essenciais para executar um serviço confiável no ambiente de produção.

Rollouts complicam os testes

Com frequência, dizemos que testes são (ou deveriam ser) realizados em um ambiente hermético [Nar12]. Essa afirmação deixa subentendido que a produção não é hermética. É claro que a produção geralmente não é hermética, pois a cadênci a de rollouts provoca alterações ao vivo no

ambiente de produção em porções pequenas e bem compreendidas.

Para administrar a incerteza e ocultar o risco dos usuários, as alterações podem ser feitas ao vivo na mesma ordem em que foram adicionadas no sistema de controle de versões. Os rollouts muitas vezes ocorrem em etapas, usando mecanismos que misturam gradualmente os usuários, além de monitorar cada etapa para garantir que o novo ambiente não esteja enfrentando problemas previstos, porém inesperados. Como resultado, o ambiente de produção como um todo não é intencionalmente representativo de nenhuma dada versão de um binário que esteja no sistema de controle de versões.

É possível que o sistema de controle de versões tenha mais de uma versão de um binário e de seus arquivos de configuração associados esperando para se tornarem ativos em produção. Esse cenário pode causar problemas quando testes são conduzidos em um ambiente ao vivo. Por exemplo, o teste pode usar a versão mais recente de um arquivo de configuração que esteja no sistema de controle de versões, juntamente com uma versão mais antiga do binário ativo. Ou o teste pode utilizar uma versão mais antiga do arquivo de configuração e encontrar um bug que foi corrigido em uma versão mais recente do arquivo.

De modo semelhante, um teste de sistema pode usar os arquivos de configuração para compor seus módulos antes de executar o teste. Se o teste passar, mas o arquivo tem uma versão para a qual o teste de configuração (discutido na próxima seção) falha, o resultado do teste será válido do ponto de vista hermético, mas não do ponto de vista operacional. Um resultado como esse é inconveniente.

Teste de configuração

No Google, as configurações dos web services são descritas em arquivos armazenados em nosso sistema de controle de versões. Para cada arquivo de configuração, um *teste de configuração* (configuration test) separado analisa a produção para ver como um binário em particular está realmente configurado e informa as discrepâncias em relação a esse arquivo. Esses

testes são inherentemente não herméticos, pois operam fora da sandbox da infraestrutura de testes.

Os testes de configuração são criados e realizados para uma versão específica do arquivo de configuração registrado no sistema de controle de versões. Comparar qual versão do teste está passando em relação à versão-alvo para automação indica implicitamente o quanto distante está a produção em relação ao trabalho de engenharia no momento.

Esses testes de configuração não herméticos tendem a ser especialmente valiosos como parte de uma solução de monitoração distribuída, pois o padrão de passar/falhar na produção pode identificar caminhos pela pilha do serviço que não tenham combinações razoáveis das configurações locais. As regras da solução de monitoração tentam combinar os caminhos das requisições dos usuários (a partir de logs de trases) em relação a esse conjunto de caminhos indesejáveis. Qualquer correspondência encontrada pelas regras se transforma em alertas de que versões e/ou atualizações em andamento não estão se comportando de forma segura, e ações de reparação são necessárias.

Os testes de configuração podem ser muito simples quando a versão implantada em produção utiliza o conteúdo propriamente dito do arquivo e oferece uma consulta em tempo real para obter uma cópia do conteúdo. Nesse caso, o código de teste simplesmente executa essa consulta e verifica a diferença entre a resposta e o arquivo. Os testes se tornam mais complexos quando um dos seguintes itens ocorre com a configuração:

- Ela implicitamente incorpora defaults incluídos no binário (o que significa que os testes recebem números diferentes de versão, como resultado).
- Passa por um pré-processador, como o bash, usando flags de linha de comando (resultando em testes sujeitos a regras de expansão).
- Especifica um contexto comportamental para um runtime compartilhado (fazendo os testes dependerem do cronograma de entrega de versões desse runtime).

Teste de estresse

Para operar um sistema de forma segura, os SREs precisam entender os

limites tanto do sistema quanto de seus componentes. Em muitos casos, componentes individuais não degradam de forma suave após ultrapassar determinado ponto – em vez disso, eles falham de modo catastrófico. Os engenheiros usam *testes de estresse* (stress tests) para descobrir os limites de um web service. Os testes de estresse respondem a perguntas como:

- Quão cheio um banco de dados pode estar até que as escritas comecem a falhar?
- Quantas consultas por segundo podem ser enviadas a um servidor de aplicações antes que ele fique sobrecarregado, fazendo as requisições falharem?

Teste canário

O *teste canário* (canary test) está notadamente ausente nessa lista de testes de produção. O termo *canário* (canary) é proveniente da expressão “canário em uma mina de carvão”, e faz referência à prática de usar uma ave viva para detectar gases tóxicos antes que os seres humanos sejam envenenados.

Para conduzir um teste canário, um subconjunto de servidores é atualizado com uma nova versão ou configuração e, em seguida, é mantido por um período de incubação. Se nenhuma variação inesperada ocorrer, a versão é mantida e o restante dos servidores será atualizado de forma progressiva.⁴ Se algo der errado, o único servidor modificado poderá ser rapidamente revertido a um bom estado conhecido. É comum nos referirmos ao período de incubação do servidor atualizado como “cozinhando o binário” (baking the binary).

Um teste canário não é realmente um teste, mas uma aceitação estruturada do usuário. Enquanto os testes de configuração e de estresse confirmam a existência de uma condição específica sobre um software determinístico, um teste canário é mais *ad hoc*. Ele simplesmente expõe o código em teste a um tráfego ativo menos previsível em produção e, desse modo, não é perfeito e nem sempre detecta novas falhas introduzidas.

Para fornecer um exemplo concreto de como um teste canário pode ocorrer: considere uma dada falha subjacente que impacte o tráfego de usuários de forma relativamente rara, e que seja implantada com um rollout exponencial

de atualização de versão. Esperamos um número cumulativo crescente de variações informadas de $CU = RK$, em que R é a taxa desses relatos, U é a ordem da falha (definida depois) e K é o período em que o tráfego aumenta de um fator de e , ou 172%.⁵

Para evitar impactos nos usuários, um rollout que acione variações indesejáveis precisa sofrer rollback rapidamente para a configuração anterior. No curto intervalo de tempo que demora para a automação observar as variações e responder, é provável que vários relatos adicionais sejam gerados. Depois que a poeira baixar, esses relatos podem estimar tanto o número cumulativo C quanto a taxa R .

Dividir e corrigir por K fornece uma estimativa de U , a ordem da falha subjacente.⁶ Alguns exemplos:

- $U=1$: a requisição do usuário se deparou com um código que está simplesmente com falha.
- $U=2$: essa requisição de usuário aleatoriamente danifica dados que uma futura requisição de usuário poderá ver.
- $U=3$: os dados danificados aleatoriamente também são um identificador válido para uma requisição anterior.

A maioria dos bugs é de ordem um: eles escalam linearmente com o volume de tráfego de usuários [Per07]. Em geral, você pode rastrear esses bugs convertendo os logs de todas as requisições com respostas incomuns em novos testes de regressão. Essa estratégia não funciona para bugs de ordem mais alta; uma requisição que falhe repetidamente se todas as requisições anteriores forem tentadas em sequência passará repentinamente se algumas requisições forem omitidas. É importante capturar esses bugs de alta ordem durante a implantação de uma nova versão, pois, do contrário, a carga de trabalho operacional poderá aumentar rapidamente.

Ao manter a dinâmica de bugs de alta ordem *versus* bugs de baixa ordem em mente, quando você usar uma estratégia de rollout exponencial, não será necessário tentar alcançar uma equidade entre frações de tráfego de usuários. Desde que cada método para determinar uma fração utilize o mesmo intervalo K , a estimativa de U será válida, mesmo que você ainda não possa

determinar qual método foi fundamental para revelar a falha. Usar vários métodos sequencialmente, ao mesmo tempo que um pouco de sobreposição é permitido, mantém o valor de K baixo. Essa estratégia minimiza o número total de variações C visíveis ao usuário, ao mesmo tempo que permite uma estimativa de U com antecedência (esperando que seja 1, é claro).

Criando um ambiente de teste e de build

Embora seja maravilhoso pensar nesses tipos de testes e cenários de falhas no primeiro dia de um projeto, com frequência os SREs se juntam a uma equipe de desenvolvedores quando um projeto já está bem adiantado – depois que o projeto da equipe validar seu modelo de pesquisa, sua biblioteca provar que o algoritmo subjacente do projeto é escalável ou, quem sabe, quando todos os simuladores (mocks) da interface de usuário finalmente estiverem aceitáveis. A base de código da equipe ainda é um protótipo e testes abrangentes ainda não foram projetados nem implantados. Em situações como essa, em que ponto seus esforços de teste devem começar? Conduzir testes de unidade para toda função e classe importante é uma perspectiva totalmente desanimadora se a cobertura atual do teste for baixa ou inexistente. Em vez disso, comece com testes que proporcionem o maior impacto com o menor esforço.

Você pode dar início à sua abordagem fazendo as seguintes perguntas:

- Você pode atribuir prioridades à base de código de alguma maneira? Para emprestar uma técnica do desenvolvimento de funcionalidades e do gerenciamento de projetos, se todas as tarefas tiverem alta prioridade, nenhuma delas terá alta prioridade. Você é capaz de classificar os componentes do sistema que está testando de acordo com algum critério de importância?
- Há alguma função ou classe em particular que sejam absolutamente cruciais para a missão ou para o negócio? Por exemplo, um código que envolva faturamento é comumente crucial aos negócios. Além disso, o código de faturamento muitas vezes é claramente separável de outras partes do sistema.
- Quais APIs outras equipes estão integrando? Mesmo o tipo de falha que

jamais passe pelos testes de versão e chegue até um usuário pode ser extremamente prejudicial se causar confusão para outra equipe de desenvolvedores, fazendo-a escrever clientes incorretos (ou que simplesmente não sejam ideais) para a sua API.

Entregar software que esteja obviamente com falha está entre os pecados mais graves de um desenvolvedor. Não é preciso muito esforço para criar uma série de testes de fumaça a ser executada para cada versão entregue. Esse tipo de primeiro passo de pouco esforço e alto impacto pode resultar em um software altamente testado e confiável.

Uma forma de estabelecer uma cultura de testes robusta⁷ é começar a documentar todos os bugs informados como casos de teste. Se todo bug for convertido em um teste, cada teste deve inicialmente falhar, pois o bug ainda não foi corrigido. À medida que os engenheiros corrigirem os bugs, o software passará nos testes e você estará no caminho certo para desenvolver uma suíte de testes de regressão abrangente.

Outra tarefa essencial para gerar um software bem testado é configurar uma infraestrutura de testes. A base para uma infraestrutura de testes robusta é um sistema de controle de versões de códigos-fontes que controle todas as alterações na base de código.

Depois que um sistema de controle de versões estiver implantado, você poderá acrescentar um sistema de builds contínuas que construa o software e execute testes sempre que um código for submetido. Achamos que o ideal é que o sistema de build notifique os engenheiros no momento em que uma mudança cause falhas em um projeto de software. Correndo o risco de soar óbvio, é essencial que a versão mais recente de um projeto de software no sistema de controle de versões esteja funcionando de forma completa. Quando o sistema de build notifica os engenheiros sobre um código com falha, eles devem deixar de lado todas as demais tarefas e dar prioridade à correção do problema. É apropriado tratar os defeitos com esse nível de seriedade por alguns motivos:

- Geralmente é mais difícil corrigir o que está com falha se houver mudanças na base de código depois que o defeito foi introduzido.
- Um software com falha deixa a equipe lenta, pois ela precisará trabalhar

para contorná-la.

- Cadências de geração de novas versões, por exemplo, builds noturnas ou semanais, perdem seu valor.
- A capacidade da equipe de responder a uma solicitação de uma versão para emergência (por exemplo, em resposta a uma divulgação de vulnerabilidade de segurança) se torna muito mais complexa e difícil.

Tradicionalmente, há uma tensão entre os conceitos de estabilidade e de agilidade no mundo da SRE. O último item apresentado mostra um caso interessante em que a estabilidade, na verdade, conduz à agilidade. Quando a build é previsivelmente sólida e confiável, os desenvolvedores podem fazer iterações de modo mais rápido!

Alguns sistemas de build, como o Bazel⁸, têm funcionalidades valiosas que permitem um controle mais preciso sobre os testes. Por exemplo, o Bazel cria gráficos de dependência para projetos de software. Quando uma mudança é feita em um arquivo, o Bazel reconstrói somente a parte do software que depende desse arquivo. Sistemas como esse permitem builds reproduzíveis. Em vez de executar todos os testes a cada submissão, eles são executados apenas para os códigos modificados. Como resultado, os testes são executados com um custo menor e de modo mais rápido.

Existem várias ferramentas para ajudar a quantificar e visualizar o nível de cobertura de testes necessário [Cra10]. Utilize essas ferramentas para moldar o foco de seus testes: aborde a perspectiva de criar um código altamente testado como um projeto de engenharia, e não como um exercício mental filosófico. Em vez de repetir o refrão ambíguo “Precisamos fazer mais testes”, defina metas e prazos explícitos.

Lembre-se de que nem todos os softwares são criados iguais. Sistemas dos quais dependam vidas ou que sejam cruciais para a receita da empresa exigem níveis substancialmente mais altos de qualidade e cobertura de testes do que um script com um prazo de validade curto que não vá ser usado em produção.

Testando em escala

Agora que já discutimos o básico sobre os testes, vamos analisar como a SRE adota um ponto de vista de sistema para os testes a fim de ter confiabilidade em escala.

Um teste de unidade simples pode ter uma lista pequena de dependências: um arquivo-fonte, a biblioteca de testes, as bibliotecas em tempo de execução, o compilador e o hardware local executando os testes. Um ambiente de testes robusto determina que cada uma dessas dependências tenha sua própria cobertura de teste, com testes que especificamente abordem casos de uso que outras partes do ambiente esperam. Se a implementação desse teste de unidade depender de um caminho no código de uma biblioteca usada em tempo de execução que não tenha cobertura de teste, uma mudança não relacionada no ambiente⁹ pode levar o teste de unidade a passar de forma consistente, independentemente das falhas do código em teste.

Em oposição, um teste de nova versão pode depender de tantas partes a ponto de ter uma dependência transitiva de todos os objetos no repositório de código. Se o teste depender de uma cópia limpa do ambiente de produção, em princípio, toda correção pequena exigirá realizar uma iteração completa de recuperação de desastre. Ambientes de teste práticos tentam selecionar pontos no branch entre as versões e merges. Fazer isso resolve a quantidade máxima de incertezas de dependências com um número mínimo de iterações. É claro que, quando uma área de incerteza é resolvida como uma falha, você precisará selecionar pontos de branch adicionais.

Testando ferramentas escaláveis

Por serem softwares, as ferramentas de SRE também precisam ser testadas.¹⁰ As ferramentas desenvolvidas pela SRE podem realizar tarefas como:

- Obter e propagar métricas de desempenho de bancos de dados
- Prever métricas de uso para planejar riscos à capacidade
- Refatorar dados em uma réplica de serviço não acessível aos usuários
- Alterar arquivos em um servidor

As ferramentas de SRE compartilham duas características:

- Seus efeitos colaterais permanecem dentro da API convencional testada.

- São isoladas da produção voltada aos usuários por uma barreira existente de validação e de versão.

Barreiras de defesa contra softwares arriscados

Softwares que ignorem a API usual intensamente testada (mesmo que o façam por uma boa causa) poderiam causar uma enorme confusão em um serviço ativo. Por exemplo, a implementação de uma engine de banco de dados pode permitir que os administradores temporariamente desativem as transações para diminuir as janelas de manutenção. Se a implementação for usada por softwares de atualização em batch, o isolamento de sistemas voltados ao usuário pode ser perdido se esse utilitário for accidentalmente executado em uma réplica voltada aos usuários. Evite esse risco de confusão por meio de design:

1. Utilize uma ferramenta separada para colocar uma barreira na configuração de replicação para que a réplica não possa passar pela sua verificação de sanidade. Como resultado, a réplica não será disponibilizada aos usuários.
2. Configure o software arriscado para verificar a barreira na inicialização. Permita que o software arriscado acesse apenas réplicas não saudáveis.
3. Utilize a ferramenta de validação de saúde da réplica usada para monitoração caixa-preta para remover a barreira.

As ferramentas de automação também são softwares. Como sua área de risco aparece fora do espectro para uma camada diferente de serviço, suas necessidades de teste são mais sutis. As ferramentas de automação realizam tarefas como:

- Seleção de índice de bancos de dados
- Distribuição de carga entre datacenters
- Organização de logs de relay para remastering rápido

As ferramentas de automação compartilham duas características:

- A operação propriamente dita é feita em uma API robusta, previsível e bem testada.

- O propósito da operação é o efeito colateral que é uma descontinuidade invisível para outro cliente da API.

Testes podem demonstrar o comportamento desejado de outra camada do serviço, tanto antes quanto depois da mudança. Com frequência, é possível testar se o estado interno, conforme visto por meio da API, é constante durante a operação. Por exemplo, os bancos de dados procuram respostas corretas, mesmo que um índice adequado não esteja disponível para a query. Por outro lado, algumas invariantes documentadas da API (por exemplo, um cache de DNS mantido até o TTL) podem não se manter durante a operação. Por exemplo, se uma mudança no runlevel (nível de execução) substituir um nome de servidor local por um proxy de caching, as duas escolhas podem prometer reter as buscas concluídas por vários segundos. É improvável que o estado do cache seja passado de um para o outro.

Considerando que as ferramentas de automação implicam testes adicionais em versões para que outros binários tratem transientes ambientais, como você define o ambiente em que essas ferramentas de automação executam? Afinal de contas, é provável que a automação para organizar contêineres a fim de melhorar o uso tente reorganizar a si mesma em algum momento se ela também executar em um contêiner. Seria constrangedor se uma nova versão de seu algoritmo interno produzisse páginas de memória sujas tão rapidamente que a banda de rede do espelhamento associado acabasse impedindo o código de finalizar a migração ao vivo. Mesmo que houvesse um teste de integração para o qual o binário intencionalmente reorganizasse a si mesmo, é provável que o teste não vá utilizar um modelo do tamanho da produção da frota de contêineres. É quase certo que ele não terá permissão para usar uma banda intercontinental escassa de alta latência para testar concorrências como essa.

Mais impressionante ainda é que uma ferramenta de automação pode mudar o ambiente em que outra ferramenta de automação executa. Ou as duas ferramentas podem alterar o ambiente da outra ferramenta de automação simultaneamente! Por exemplo, uma ferramenta de atualização de frota provavelmente consome o máximo de recursos quando está implantando atualizações de versões. Como resultado, o redistribuidor de contêineres

ficaria tentado a mover a ferramenta. Por sua vez, a ferramenta de redistribuição de contêineres ocasionalmente precisa de atualização. Essa dependência circular não apresenta problema se as APIs associadas tiverem semânticas de reinicialização, alguém se lembrou de implementar uma cobertura de teste para essas semânticas e a saúde do ponto de verificação (checkpoint) for garantida de modo independente.

Testando para desastres

Muitas ferramentas para recuperação de desastres podem ser cuidadosamente projetadas para operar *offline*. Ferramentas como essas fazem o seguinte:

- Calculam um estado de *ponto de verificação* (checkpoint) que seja equivalente a interromper o serviço de forma limpa.
- Implantam o estado do ponto de verificação para que seja *carregável* por ferramentas de validação existentes, que não sejam para desastre.
- Oferecem suporte às ferramentas usuais de *barreira* de versão, que acionam o procedimento de *início limpo*.

Em muitos casos, você pode implementar essas fases para que os testes associados sejam fáceis de escrever e ofereçam uma cobertura excelente. Se qualquer restrição (*offline*, ponto de verificação, carregável, barreira ou *início limpo*) tiver falha, será muito mais difícil mostrar confiança de que a implementação da ferramenta associada funcionará a qualquer momento sem aviso prévio.

Ferramentas de reparação online operam inherentemente fora da API convencional e, desse modo, são mais interessantes para testar. Um desafio com que você se deparará em um sistema distribuído é determinar se o comportamento normal, que pode ter consistência eventual (*eventually consistent*) por natureza, interagirá de forma ruim com a reparação. Por exemplo, considere uma condição de concorrência (*race condition*) que você possa tentar analisar usando as ferramentas *offline*. Uma ferramenta *offline* geralmente é escrita para esperar consistência instantânea, em oposição a uma consistência eventual, pois uma consistência instantânea é menos desafiadora para testar. Essa situação se torna complicada porque o binário de reparação em geral é construído de modo separado do binário servindo em produção em

relação ao qual há concorrência. Consequentemente, talvez você precise construir um binário instrumentado e unificado para executar com esses 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. Simplesmente executar de novo esses testes após uma mudança de código não prova de modo conclusivo que a falha observada tenha sido corrigida.¹³ No entanto, essas técnicas podem ser úteis:

- Podem oferecer um log de todas as ações selecionadas aleatoriamente que tiveram lugar em uma dada execução – às vezes apenas fazendo logging da semente do gerador de números aleatórios.
- Se esse log for imediatamente refatorado como um teste de versão, executá-lo algumas vezes antes de iniciar o relatório de bug muitas vezes será útil. A taxa de ausência de falhas na repetição informará a dificuldade de confirmar depois que a falha foi corrigida.
- As variações no modo como a falha é expressa ajudam a identificar áreas suspeitas no código.
- Algumas dessas últimas execuções podem demonstrar situações de falha que sejam mais severas do que aquelas da execução original. Em resposta, talvez você queira escalar a severidade e o impacto do bug.

A necessidade de ser rápido

Para toda versão (correção) no repositório de código, todo teste definido oferece uma indicação para informar se ele passou ou falhou. Essa indicação pode mudar para execuções aparentemente idênticas e repetidas. Você pode estimar a probabilidade de um teste passar ou falhar tirando uma média dessas várias execuções e calculando a incerteza estatística dessa probabilidade. No entanto, fazer esse cálculo para cada teste em todas as versões, do ponto de vista computacional, é impraticável.

Em vez disso, você deve formular hipóteses sobre os diversos cenários de interesse e executar o número apropriado de repetições para cada teste e versão de modo a permitir uma inferência razoável. Alguns desses cenários

são benignos (no sentido da qualidade do código), enquanto outros são passíveis de ação. Esses cenários afetam todas as tentativas de teste em graus variados e, como são acoplados, obter uma lista de hipóteses passíveis de ação de forma confiável e rápida (isto é, componentes que estão realmente com falha) significa estimar todos os cenários ao mesmo tempo.

Os engenheiros que usam a infraestrutura de testes querem saber se seus códigos – geralmente, uma minúscula fração de todo o código-fonte por trás de uma dada execução de teste – estão com falha. Com frequência, não estar com falha implica que qualquer falha observada pode ser atribuída ao código de outra pessoa. Em outras palavras, o engenheiro quer saber se seu código tem uma condição de concorrência não prevista que deixe o teste frágil (ou mais frágil do que o teste já era devido a outros fatores).

Prazos para testes

A maioria dos testes é simples, pois executam como um binário hermético e autocontido que cabe em um pequeno contêiner computacional durante alguns segundos. Esses testes dão aos engenheiros um feedback interativo sobre os erros antes que eles mudem de contexto para o próximo bug ou tarefa.

Testes que exijam coordenação entre vários 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 feedbacks interativos, portanto podem ser classificados como testes em lote. Em vez de dizer “não feche a aba do editor” ao engenheiro, essas falhas de teste dizem “este código não está pronto para revisão” ao revisor do código.

O prazo informal para o teste é o ponto em que o engenheiro faz a próxima mudança de contexto. Os resultados dos testes são mais bem fornecidos ao engenheiro antes que ele mude de contexto, pois, do contrário, o próximo contexto poderá envolver uma compilação XKCD.¹⁴

Suponha que um engenheiro esteja trabalhando em um serviço com mais de 21 mil testes simples e, ocasionalmente, proponha uma correção na base de código do serviço. Para testar a correção, você deve comparar o vetor de resultados indicando se o teste passou/falhou na base de código antes da

correção com o vetor de resultados da base de código depois dela. Uma comparação favorável desses dois vetores qualifica provisoriamente a base de código como adequada para nova versão. Essa qualificação cria um incentivo para executar os vários testes de versão e de integração, assim como outros testes distribuídos de binários que analisem a escala do sistema (no caso de a correção usar mais recursos locais de processamento de forma significativa) e a complexidade (no caso de a correção criar uma carga de trabalho superlinear em outro lugar).

Com que taxa você pode sinalizar a correção de um usuário de forma incorreta como prejudicial ao calcular erroneamente a fragilidade do ambiente? Parece provável que os usuários reclamariam veementemente se 1 em cada 10 correções fosse rejeitada. Porém, uma rejeição de 1 correção entre 100 correções perfeitas poderia passar sem comentários.

Isso significa que você está interessado na raiz 42.000^{a} (uma para cada teste definido antes da correção, e outra para cada teste definido após a correção) de 0,99 (a fração de correções que pode ser rejeitada). Este cálculo:

$$0.99^{\frac{1}{2 \times 21000}}$$

sugere que esses testes individuais devem executar corretamente em mais de 99,9999% do tempo. Humm...

Atualizando versões em produção

Embora o gerenciamento de configuração em produção seja comumente mantido em um repositório de controle de códigos-fontes, a configuração geralmente fica separada dos códigos-fontes dos desenvolvedores. De modo semelhante, a infraestrutura de testes de software com frequência não enxerga a configuração de produção. Mesmo que as duas estivessem no mesmo repositório, mudanças no gerenciamento de configuração são feitas em branches e/ou em uma árvore de diretório separada, historicamente ignorados pela automação de testes.

Em um ambiente corporativo legado, em que engenheiros de software desenvolvem binários e os lançam por cima do muro para os administradores que atualizam os servidores, a segregação entre a infraestrutura de testes e

configuração de produção, no melhor caso, é irritante e, no pior, pode prejudicar a confiabilidade e a agilidade. Uma segregação desse tipo também pode resultar na duplicação de ferramentas. Em um ambiente de Ops nominalmente integrado, essa segregação provoca degradação na resiliência, pois cria inconsistências sutis no comportamento dos dois conjuntos de ferramentas. Essa segregação também limita a velocidade dos projetos, por causa de concorrência de commits nos sistemas de controle de versões.

No modelo de SRE, o impacto de segregar a infraestrutura de testes da configuração de produção é consideravelmente pior, pois impede relacionar o modelo que descreve a produção ao modelo que descreve o comportamento da aplicação. Essa discrepância tem impacto nos engenheiros que querem encontrar inconsistências estatísticas em expectativas durante o desenvolvimento. No entanto, essa segregação não deixa o desenvolvimento mais lento nem impede que a arquitetura do sistema mude, pois não há nenhuma maneira de eliminar riscos à migração.

Considere um cenário de sistemas de controle de versões e testes unificados, de modo que a metodologia de SRE seja aplicável. Qual seria o impacto da falha de uma migração em arquitetura distribuída? Um volume considerável de testes provavelmente ocorrerá. Até agora, supomos que um engenheiro de software provavelmente aceitaria o teste de sistema com uma resposta incorreta a cada 10 ou algo semelhante. Quais riscos você está disposto a correr com a migração se souber que os testes podem informar um falso-negativo e a situação poderia ficar realmente emocionante, de forma muito rápida? É claro que algumas áreas de cobertura de testes exigem um nível maior de paranoia do que outras. Essa distinção pode ser generalizada: algumas falhas de teste são indicativas de um risco de mais impacto do que outras falhas de teste.

Falha esperada em testes

Não muito tempo atrás, um produto de software seria lançado uma vez por ano. Seus binários eram gerados por uma cadeia de ferramentas de compilação durante muitas horas ou dias, e a maioria dos testes era realizada por seres humanos executando instruções escritas manualmente. Esse

processo de entrega de versão era ineficiente, mas havia pouca necessidade de automatizá-lo. O esforço para gerar uma nova versão era dominado pela documentação, migração de dados, novos treinamentos aos usuários e outros fatores. O MTBF (Mean Time Between Failure, ou Tempo Médio Entre Falhas) para essas versões era de um ano, independentemente do volume de testes feito. Tantas mudanças ocorriam em cada versão que algumas falhas visíveis ao usuário estavam fadadas a permanecer ocultas no software. Na prática, dados de confiabilidade da versão anterior eram irrelevantes para a próxima versão.

Ferramentas eficientes de gerenciamento de API/ABI e linguagens interpretadas que escalam para grandes quantidades de código atualmente oferecem suporte para construir e executar uma nova versão de software a intervalos de alguns minutos. Em princípio, um batalhão suficientemente grande de pessoas¹⁵ poderia completar os testes em cada nova versão usando os métodos descritos anteriormente e alcançar o mesmo nível de qualidade para cada versão incremental. Apesar de, em última instância, apenas os mesmos testes serem aplicados ao mesmo código, essa versão final do software terá mais qualidade como a versão resultante lançada anualmente. Isso ocorre porque, além das versões anuais, as versões intermediárias de código também são testadas. Ao usar versões intermediárias, você pode mapear os problemas encontrados durante os testes de forma não ambígua às suas causas subjacentes e estar confiante de que o problema todo, e não apenas o sintoma limitado que havia sido exposto, estará corrigido. Esse princípio de um ciclo mais rápido de feedback é igualmente eficiente quando aplicado à cobertura de testes automatizada.

Se você deixar que os usuários testem mais versões de software durante o ano, o MTBF sofrerá porque haverá mais oportunidades para falhas visíveis ao usuário. No entanto, você também poderá descobrir áreas que se beneficiarão da cobertura adicional de testes. Se esses testes forem implementados, cada melhoria fará a proteção contra outra falha no futuro. Um gerenciamento cuidadoso da confiabilidade combina os limites na incerteza devido à cobertura de testes com os limites nas falhas visíveis aos usuários a fim de ajustar a cadência do lançamento de novas versões. Essa combinação maximiza o conhecimento obtido com base em operações e a

partir dos usuários finais. Esses ganhos orientam a cobertura dos testes e, por sua vez, a velocidade de entrega do produto.

Se um SRE modificar um arquivo de configuração ou ajustar a estratégia de uma ferramenta de automação (em oposição a implementar uma funcionalidade para o usuário), o trabalho de engenharia terá o mesmo modelo conceitual. Ao definir uma cadência de entrega de versões com base na confiabilidade, muitas vezes fará sentido segmentar a provisão de confiabilidade por funcionalidades ou (de modo mais conveniente) por equipe. Em um cenário como esse, a equipe de engenharia responsável por novas funcionalidades visa a alcançar um determinado limite de incerteza que afete a sua meta de cadência de novas versões. A equipe de SRE tem uma provisão separada com sua própria incerteza associada e, desse modo, um limite superior para sua taxa de novas versões.

Para se manter confiável e evitar escalar o número de SREs que dão suporte a um serviço de forma linear, o ambiente de produção deve executar de modo independente na maior parte do tempo. Para ser independente, o ambiente deve ser resiliente a falhas menores. Quando um evento importante, que exija intervenção manual de um SRE, ocorre, as ferramentas usadas pela SRE devem estar apropriadamente testadas. Caso contrário, essa intervenção diminuirá a confiança de que dados históricos possam ser aplicáveis no futuro próximo. A redução na confiança exige esperar uma análise dos dados de monitoração a fim de eliminar a incerteza gerada. Enquanto a discussão anterior na seção “Testando ferramentas escaláveis”, tinha como foco de que modo usar a oportunidade de cobertura de teste para uma ferramenta de SRE, nesta seção você verá que os testes determinam a frequência apropriada para usar essa ferramenta em produção.

Os arquivos de configuração em geral existem porque mudar a configuração é mais rápido do que reconstruir uma ferramenta. Essa baixa latência muitas vezes é um fator para manter o MTTR baixo. Entretanto, esses mesmos arquivos também são alterados com frequência por motivos que não exigem essa latência reduzida. Quando vistos do ponto de vista da confiabilidade:

- Um arquivo de configuração que exista para manter o MTTR baixo, e seja modificado somente quando houver uma falha, tem uma cadência de

entrega de novas versões mais lenta que o MTBF. Pode haver um nível razoável de incerteza em relação ao fato de uma dada edição manual ser realmente ideal, sem que ela tenha impactos na confiabilidade do site como um todo.

- Um arquivo de configuração que mude mais de uma vez a cada nova versão de uma aplicação voltada ao usuário (por exemplo, porque armazena o estado da nova versão) pode representar um grande risco se essas alterações não forem tratadas do mesmo modo que as entregas de novas versões da aplicação. Se a cobertura dos testes e da monitoração desse arquivo de configuração não for consideravelmente melhor do que a da aplicação do usuário, esse arquivo dominará a confiabilidade do site de forma negativa.

Um método para tratar arquivos de configuração é garantir que todo arquivo de configuração seja classificado de acordo com apenas uma das opções da lista anterior de itens, e de algum modo garantir o cumprimento dessa regra. Caso você adote a última estratégia, garanta que:

- todo arquivo de configuração tenha cobertura suficiente de testes para aceitar uma mudança regular rotineira;
- antes da entrega de novas versões, as alterações no arquivo, de certo modo, são postergadas enquanto se espera pelos testes da versão;
- um sistema de break-glass esteja disponível para atualização do arquivo ao vivo antes que os testes sejam concluídos. Como o uso do mecanismo de break-glass compromete a confiabilidade, em geral é uma boa ideia usá-lo gerando bastante ruído (por exemplo), registrando um bug que exija uma solução mais robusta da próxima vez.

Break-glass e testes

Você pode implementar um sistema de break-glass a fim de desabilitar os testes de versão. Fazer isso significa que quem fizer uma alteração manual às pressas não saberá de nenhum erro até que o verdadeiro impacto no usuário seja informado pela monitoração. É melhor deixar os testes executando, associar o evento prévio de atualização de versão com o evento de teste pendente e (assim que for possível) fazer anotações de

forma retroativa na atualização com qualquer teste que tenha falhado. Dessa maneira, uma atualização manual com problemas pode ser rapidamente seguida de outra atualização manual (espera-se que essa tenha menos falhas). O ideal é que esse sistema de break-glass aumente automaticamente a prioridade desses testes de versão para que possam tomar o lugar da validação incremental de rotina e da carga de trabalho de cobertura que a infraestrutura de teste já está processando.

Integração

Além de realizar testes de unidade em um arquivo de configuração para atenuar seus riscos à confiabilidade, também é importante considerar os testes de integração em arquivos de configuração. O conteúdo do arquivo de configuração é (do ponto de vista de testes), potencialmente, um conteúdo hostil ao interpretador que lê a configuração. Linguagens interpretadas, como Python, são comumente utilizadas para arquivos de configuração, pois seus interpretadores podem ser incluídos, e uma sandboxing simples está disponível para proteção contra erros não maliciosos de programação.

Escrever seus arquivos de configuração em uma linguagem interpretada é arriscado, pois essa abordagem é repleta de falhas latentes, difíceis de serem tratadas de forma definitiva. Como a carga de conteúdo na verdade consiste na execução de um programa, não há um limite superior inerente para o nível de ineficiência da carga. Além de qualquer outro teste, você deve combinar esse tipo de teste de integração com verificações cuidadosas de prazos em todos os métodos de testes de integração a fim de identificar como falhas os testes que não executaram até o fim em um intervalo de tempo razoável.

Por outro lado, se a configuração for escrita como texto em uma sintaxe personalizada, toda categoria de testes precisará de uma cobertura separada desde o princípio. Utilizar uma sintaxe existente, como YAML, em conjunto com um parser altamente testado, como o `safe_load` de Python, remove parte da tarefa penosa (`toil`) imposta pelo arquivo de configuração. Uma escolha cuidadosa da sintaxe e do parser pode garantir que haja um limite superior rígido para o tempo que a operação de carga pode demorar. No entanto, quem faz a implementação deve tratar 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 ter testes de unidade robustos.

A vantagem de usar buffers de protocolo¹⁶ é que o esquema é definido com antecedência e verificado automaticamente em tempo de carga, eliminando mais tarefas penosas, ao mesmo tempo que oferece o tempo de execução limitado.

A função da SRE, de modo geral, inclui escrever ferramentas de engenharia de sistemas¹⁷ (se não houver mais ninguém que já as esteja escrevendo) e acrescentar uma validação robusta com cobertura de testes. Todas as ferramentas podem se comportar de forma inesperada por causa de bugs não capturados pelos testes, portanto uma defesa robusta é aconselhável. Quando uma ferramenta se comporta de modo inesperado, os engenheiros devem ter o máximo de confiança possível de que a maior parte das outras ferramentas esteja funcionando corretamente e, desse modo, atenuar ou resolver os efeitos colaterais desse mau comportamento. Um elemento essencial para proporcionar confiabilidade de sites é identificar cada forma de mau comportamento com antecedência e garantir que algum teste (ou o validador de entradas testado de outra ferramenta) informe esse mau comportamento. A ferramenta que encontrar o problema poderá não ser capaz de corrigi-lo ou nem mesmo interrompê-lo, mas deve, pelo menos, informar o problema antes que uma interrupção de serviço catastrófica ocorra.

Por exemplo, considere a lista configurada de todos os usuários (como */etc/passwd* em uma máquina de estilo Unix que não esteja em rede) e suponha que uma alteração, de forma não intencional, faça o parser parar após ter feito o parse de apenas metade do arquivo. Como usuários criados recentemente não foram carregados, a máquina provavelmente continuará a executar sem problemas, e muitos usuários talvez nem percebam a falha. A ferramenta que mantém os diretórios home pode facilmente perceber a discrepância entre os diretórios presentes e aqueles sugeridos pela lista (parcial) de usuários e informar urgentemente essa diferença. A importância dessa ferramenta está em informar o problema, e ela deve evitar tentar remediar-lo por conta própria (apagando vários dados de usuários).

Sondas na produção

Considerando que os testes especificam comportamentos aceitáveis face a dados conhecidos, enquanto a monitoração confirma comportamentos aceitáveis diante de dados desconhecidos de usuários, pode parecer que fontes importantes de riscos – tanto os conhecidos quanto os desconhecidos – estejam cobertas pela combinação de testes e monitoração. Infelizmente, os riscos reais são mais complicados.

Requisições boas conhecidas devem funcionar, enquanto requisições ruins conhecidas devem causar erros. Implementar os dois tipos de cobertura como um teste de integração em geral é uma boa ideia. Você pode reproduzir o mesmo conjunto de requisições de teste como um teste de versão. Separar as requisições boas conhecidas entre aquelas que podem ser reproduzidas em produção e aquelas que não resulta em três conjuntos de requisições:

- Requisições ruins conhecidas
- Requisições boas conhecidas que podem ser reproduzidas em produção
- Requisições boas conhecidas que não podem ser reproduzidas em produção

Você pode usar cada conjunto tanto como testes de integração quanto de versão. A maioria desses testes também pode ser usada como sondas (probes) para monitoração.

Pode parecer supérfluo e, em princípio, sem sentido implantar uma monitoração desse tipo, pois essas mesmas requisições já foram testadas de duas outras maneiras. No entanto, essas duas maneiras eram diferentes por alguns motivos:

- O teste de versão provavelmente encapsulou o servidor integrado com um frontend e um backend simulado.
- O teste de sondagem provavelmente encapsulou o binário da versão com um frontend de distribuição de carga e um backend persistente separado e escalável.
- Frontends e backends provavelmente têm ciclos independentes de novas versões. É provável que os cronogramas para esses ciclos tenham ritmos diferentes (por causa da cadência de versões adaptativa).

Desse modo, a sonda para monitoração executando em produção é uma configuração que não foi testada antes.

Essas sondas jamais devem falhar, mas, se falharem, o que isso significa? A API do frontend (do distribuidor de carga) ou a API do backend (para a área de armazenagem persistente) não são equivalentes nos ambientes de produção e de nova versão. A menos que você já saiba por que os ambientes de produção e de nova versão não são equivalentes, é provável que o site tenha falhas.

O mesmo atualizador de produção que, gradualmente, substitui a aplicação também substitui gradualmente as sondas para que todas as quatro combinações de sondas antigas-ou-novas enviendo requisições para aplicações antigas-ou-novas sejam continuamente geradas. Esse atualizador é capaz de detectar quando uma das quatro configurações gera erros e faz rollback para o último bom estado conhecido. Geralmente, o atualizador espera que cada instância da aplicação recém-iniciada não esteja saudável durante um breve período de tempo, enquanto ela se prepara para começar a receber bastante tráfego de usuário. Se as sondas já estiverem inspecionadas como parte do processo para verificar se as aplicações estão prontas, a atualização falhará indefinidamente de forma segura, e nenhum tráfego de usuário será encaminhado para a nova versão. A atualização permanecerá em pausa até que os engenheiros tenham tempo e inclinação para diagnosticar a condição em falha e então estimular o atualizador de produção a fazer um rollback de forma limpa.

Esse teste de produção por sonda oferece proteção ao site, além de proporcionar feedbacks claros aos engenheiros. Quanto mais cedo esses feedbacks forem dados a eles, mais úteis serão. 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 sendo substituída, e a versão mais recente sendo instalada (agora ou muito em breve). A versão mais recente pode conversar com um par de versão antiga, o que a força a usar a API obsoleta. Ou a versão mais antiga pode conversar com uma versão mais recente de seu par, usando a API que (na época em que

a versão mais antiga foi disponibilizada) não funcionava de forma apropriada ainda. Porém, ela funciona agora, é sério! Melhor esperar que esses testes para compatibilidade futura (que estão executando como sondas para monitoração) tenham uma boa cobertura de API.

Versões simuladas de backend

Ao implementar testes de versão, o backend simulado com frequência é mantido pela equipe de engenharia do serviço que é seu par e é simplesmente referenciado como uma dependência de build. O teste hermético executado pela infraestrutura de testes sempre combina o backend simulado e o frontend de teste no mesmo ponto de build no histórico de controle de versões.

Essa dependência de build pode fornecer um binário executável hermético; o ideal é que a equipe de engenharia que o mantém gere uma versão desse binário simulado de backend ao mesmo tempo que gera sua aplicação principal de backend e suas sondas. Se essa versão de backend estiver disponível, talvez valha a pena incluir testes de versão de frontend herméticos (sem o binário simulado de backend) no pacote de versão do frontend.

Sua monitoração deve estar ciente de todas as versões nos dois lados de uma dada interface de serviço entre dois pares. Essa configuração garante que recuperar todas as combinações entre as duas versões e determinar se o teste continua passando não exija muita configuração extra. Essa monitoração não precisa ocorrer continuamente – você só precisa executar novas combinações que sejam o resultado da geração de uma nova versão em cada equipe. Problemas como esse não devem bloquear essa nova versão.

Por outro lado, o ideal é que a automação de rollout bloqueie o rollout de produção associado até que as combinações problemáticas não sejam mais possíveis. De modo semelhante, a automação da equipe correspondente pode considerar a drenagem (e a atualização) de réplicas que ainda não tenham resolvido uma combinação problemática.

Conclusão

Testar é um dos investimentos mais rentáveis que os engenheiros podem fazer para melhorar a confiabilidade de seu produto. Testar não é uma atividade que ocorra uma ou duas vezes no ciclo de vida de um projeto; é uma atividade contínua. O volume de esforços exigido para escrever bons testes é substancial, assim como o esforço para construir e manter a infraestrutura que promova uma cultura de testes robusta. Você não será capaz de corrigir um problema até entendê-lo e, em engenharia, só é possível entender um problema avaliando-o. As metodologias e técnicas deste capítulo oferecem uma base sólida para avaliar falhas e incertezas em um sistema de software e ajudam os engenheiros a pensar na confiabilidade do software conforme ela está escrita e disponibilizada aos usuários.

¹ Este capítulo explica como maximizar o valor resultante do investimento de esforços de engenharia em testes. Depois que um engenheiro definir testes adequados (a um dado sistema) de forma generalizada, o restante do trabalho será comum a todas as equipes de SRE e, desse modo, pode ser considerado como infraestrutura compartilhada. Essa infraestrutura é constituída de um escalonador (para compartilhar recursos a serem provisionados para projetos que, de outro modo, não estão relacionados) e executores (que fazem sandbox de binários de teste a fim de evitar que sejam considerados confiáveis). Cada um desses dois componentes da infraestrutura pode ser considerado como um serviço comum com suporte de SRE (muito semelhante à armazenagem em escala de cluster) e, desse modo, nenhum será discutido com detalhes aqui.

² Para outras leituras sobre equivalência, acesse <http://stackoverflow.com/questions/1909280/equivalence-class-testing-vs-boundary-value-testing>.

³ Veja <https://google.github.io/dagger/>.

⁴ Uma regra geral padrão é iniciar fazendo a nova versão impactar 0,1% do tráfego de usuários e então escalar em ordens de magnitude a cada 24 horas, ao mesmo tempo que variamos a localização geográfica dos servidores atualizados (então, no dia 2: 1%, no dia 3: 10%, no dia 4: 100%).

⁵ Por exemplo, supondo um intervalo de 24 horas de crescimento exponencial contínuo entre 1% e 10%, $K = 86400/\ln 0,1/0,01 = 37523$ segundos, ou aproximadamente 10 horas e 25 minutos.

⁶ Estamos usando ordem aqui no sentido de ordem de complexidade “notação O grande” (Big O notation). Para mais contexto, acesse https://en.wikipedia.org/wiki/Big_O_notation.

⁷ Para saber mais sobre esse assunto, recomendamos fortemente a leitura de [Bla14], de nosso ex-colega de trabalho e ex-Googler Mike Bland.

⁸ Veja <https://github.com/google/bazel>.

⁹ Por exemplo, um código em teste que encapsule uma API não trivial para oferecer uma abstração mais simples, compatível com versões anteriores. A API que costumava ser síncrona passa a devolver uma future. Erros nos argumentos de chamada continuam resultando em uma exceção, mas não até que a future seja avaliada. O código em teste passa o resultado da API diretamente a quem chamou. Muitos casos de uso incorreto de argumentos podem não ser capturados.

¹⁰ Esta seção discute especificamente as ferramentas usadas pela SRE que precisam ser escaláveis. No entanto, a SRE também desenvolve e utiliza ferramentas que não precisam necessariamente ser

escaláveis. Ferramentas desse tipo também devem ser testadas, mas estão fora do escopo desta seção e, desse modo, não serão discutidas aqui. Como a área sujeita a riscos é semelhante ao caso de aplicações voltadas ao usuário, estratégias semelhantes de testes são aplicáveis às ferramentas escaláveis desenvolvidas pela SRE.

11 Veja <https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey>.

12 Veja <https://github.com/aphyr/jepsen>.

13 Mesmo que a execução do teste seja repetida com a mesma semente aleatória para que as tarefas sejam mortas na mesma ordem, não há serialização entre as mortes e o tráfego de usuários simulado. Desse modo, não há garantias de que o caminho de código observado anteriormente seja exercitado de novo.

14 Veja <http://xkcd.com/303/>.

15 Adquiridos, quem sabe, por meio de *Mechanical Turk* ou outro serviço semelhante.

16 Veja <https://github.com/google/protobuf>.

17 Não que os engenheiros de software não devam escrevê-las. Ferramentas que cruzam verticais de tecnologia e se estendem por camadas de abstração tendem a ter associações mais fracas com muitas equipes de software e uma associação um pouco mais forte com equipes de sistemas.

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 nomear um esforço de engenharia de software no Google, e essa pessoa provavelmente mencionará um produto voltado a clientes como o Gmail ou o Maps; outros podem até mesmo mencionar uma infraestrutura subjacente, como o Bigtable ou o Colossus. Porém, na verdade, há uma quantidade enorme de engenharia de software por baixo dos panos, que os clientes jamais veem. Vários desses produtos são desenvolvidos pela SRE.

O ambiente de produção do Google é – segundo certos critérios – uma das máquinas mais complexas já criadas pela humanidade. Os SREs têm experiência em primeira mão com as complexidades do ambiente de produção, tornando-os, de forma única, bastante adequados ao desenvolvimento de ferramentas apropriadas para resolver problemas internos e casos de uso relacionados à manutenção da operação em produção. A maioria dessas ferramentas está relacionada à diretriz geral de preservar o uptime e manter a latência baixa, mas isso assume diversas formas: exemplos incluem sistemas para rollout de binários, monitoração ou um ambiente de desenvolvimento construído com base na composição dinâmica de servidores. De modo geral, essas ferramentas desenvolvidas pela SRE são projetos completos de engenharia de software, diferentes de soluções usadas uma só vez ou de hacks rápidos, e os SREs que as desenvolvem têm adotado uma mentalidade orientada a produto que leva em consideração tanto os clientes internos quanto um cronograma geral para planos futuros.

Por que a engenharia de software na SRE é importante?

Em muitos aspectos, a vasta escala de produção do Google tem exigido desenvolvimento de softwares internos, pois poucas ferramentas de terceiros estão projetadas para uma escala suficiente às necessidades do Google. A história de projetos de software bem-sucedidos da empresa nos levou a apreciar as vantagens de desenvolver diretamente no interior da SRE.

Os SREs estão em uma posição única para desenvolver softwares internos de modo eficiente por uma série de motivos:

- A amplitude e a profundidade do conhecimento do ambiente de produção específico do Google na organização SRE permitem que seus engenheiros projetem e criem softwares com as considerações apropriadas para dimensões como escalabilidade, degradação suave durante falhas e capacidade de fazer interface facilmente com outras infraestruturas ou ferramentas.
- Por estarem inseridos no assunto em questão, os SREs compreendem facilmente as necessidades e os requisitos das ferramentas desenvolvidas.
- Um relacionamento direto com o usuário visado – SREs que são seus colegas de trabalho – resulta em um feedback de usuário franco e de boa qualidade. Entregar uma ferramenta a um público-alvo interno com grande familiaridade com o domínio do problema significa que uma equipe de desenvolvimento pode lançar versões e iterar de modo mais rápido. Usuários internos geralmente são mais compreensivos quando se trata de uma UI mínima e outros problemas de produtos alfa.

De um ponto de vista puramente pragmático, o Google claramente se beneficia em ter engenheiros com experiência de SRE desenvolvendo software. Por meio de um design criterioso, a taxa de crescimento dos serviços com suporte de SRE excede a taxa de crescimento da organização SRE; um dos princípios orientadores da SRE é que “o tamanho da equipe não deve escalar diretamente com o crescimento do serviço”. Alcançar um crescimento linear da equipe diante do crescimento exponencial do serviço exige um trabalho contínuo de automação e esforços para organizar

ferramentas, processos e outros aspectos de um serviço que introduzem ineficiência na operação cotidiana da produção. Ter pessoas com experiência direta na operação de sistemas de produção desenvolvendo as ferramentas que, em última instância, colaborarão com o uptime e as metas de latência faz bastante sentido.

Por outro lado, os SREs individuais, assim como a organização SRE mais ampla, também se beneficiam com o desenvolvimento de software orientado por SREs.

Projetos completos de desenvolvimento de software na SRE oferecem oportunidades de desenvolvimento na carreira dos SREs, assim como uma oportunidade para os engenheiros que não querem que suas habilidades de programação enferrujem. Trabalhos em projetos de longo prazo oferecem o equilíbrio bastante necessário com interrupções e trabalhos de plantão, e podem proporcionar satisfação no emprego aos engenheiros que queiram manter um equilíbrio entre engenharia de software e engenharia de sistemas em suas carreiras.

Além do design de ferramentas de automação e outros esforços para reduzir a carga de trabalho dos engenheiros em SRE, os projetos de desenvolvimento de software podem beneficiar mais a organização SRE atraindo e ajudando a reter engenheiros com uma grande variedade de habilidades. O desejo de ter uma equipe diversificada é duplamente verdadeiro para a SRE, em que uma variedade de experiências anteriores e abordagens para resolução de problemas podem ajudar a evitar pontos cegos. Para isso, o Google sempre se esforça para ter equipes de SRE com uma mistura de engenheiros com experiência em desenvolvimento de software tradicional e engenheiros com experiência em engenharia de sistemas.

Estudo do caso Auxon: contexto do projeto e domínio do problema

Este estudo de caso analisa o Auxon – uma ferramenta eficaz desenvolvida na SRE para automatizar o planejamento de capacidade para serviços que executam no ambiente de produção do Google. Para entender melhor como o

Auxon foi concebido e os problemas que ele trata, inicialmente analisaremos o domínio do problema associado ao planejamento de capacidade, e as dificuldades que abordagens tradicionais a essa tarefa apresentam para os serviços no Google, e no mercado como um todo. Para mais informações sobre o contexto em que o Google utiliza os termos *serviço* e *cluster*, consulte o Capítulo 2.

Planejamento de capacidade tradicional

Há uma variedade de táticas para planejamento de capacidade de recursos computacionais (veja [Hix15a]), mas a maior parte dessas abordagens se reduz a um *ciclo* para o qual a seguinte aproximação pode ser feita:

1) *Coletar previsões de demanda.*

Quantos recursos são necessários? Quando e onde esses recursos são necessários?

- Utiliza os melhores dados que temos disponíveis hoje para planejar o futuro.
- Geralmente abrange qualquer período, de vários trimestres a anos.

2) *Criar planos de construção e alocação.*

Considerando essa previsão, qual é a melhor maneira de atender a essa demanda com um fornecimento adicional de recursos? Quantos recursos adicionais e em que localidades?

3) *Revisar e assinar um plano.*

A previsão é razoável? O plano está alinhado com orçamentos, considerações no nível de produto e considerações técnicas?

4) *Implantar e configurar recursos.*

Depois que os recursos chegam (possivelmente em fases no decorrer de um período de tempo definido), quais serviços poderão usá-los? Como faço para deixar os recursos tipicamente de mais baixo nível (CPU, disco etc.) utilizáveis aos serviços?

Vale a pena enfatizar que o planejamento de capacidade é um *ciclo* que

jamais termina: suposições mudam, implantações escorregam no prazo e orçamentos são cortados, resultando em revisão em cima de revisão no Plano. Além disso, cada revisão tem efeitos multiplicadores 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 utiliza a demanda como um fator orientador essencial, e molda manualmente o fornecimento de recursos para que esteja de acordo com a demanda em resposta a cada mudança.

Frágil por natureza

O planejamento de capacidade tradicional gera um plano de alocação de recursos que pode ser desestruturado por qualquer mudança aparentemente pequena. Por exemplo:

- Um serviço passa por uma redução de eficiência e precisa de mais recursos do que o esperado para atender à mesma demanda.
- As taxas de adoção dos clientes aumentam, resultando em um aumento na demanda prevista.
- A data de entrega de um novo cluster de recursos computacionais escorrega.
- Uma decisão de produto sobre uma meta de desempenho muda o aspecto da implantação do serviço exigida (a extensão do serviço) e a quantidade de recursos necessários.

Alterações pequenas exigem verificação cruzada em todo o plano de alocação para garantir que ele continue viável; alterações maiores (como atraso na entrega de recursos ou mudanças na estratégia do produto) podem, em potencial, exigir a reconstrução total do plano. Um atraso na entrega de um único cluster pode impactar os requisitos de redundância ou de latência de vários serviços: alocações de recursos em outros clusters devem ser aumentadas para compensar o atraso, e essa e qualquer outra alteração deverão se propagar por todo o plano.

Além disso, considere que o plano de capacidade para um dado trimestre (ou outro intervalo de tempo) é baseado no resultado esperado dos planos de capacidade de trimestres anteriores, o que significa que uma mudança em

qualquer trimestre resulta em trabalho para atualizar os trimestres subsequentes.

Laborioso e impreciso

Para muitas equipes, o processo de coletar os dados necessários para gerar previsões de demanda é lento e suscetível a erros. Além disso, quando for o momento de descobrir a capacidade para atender a essa demanda futura, nem todos os recursos serão igualmente adequados. Por exemplo, se os requisitos de latência significarem que um serviço deva se comprometer a servir à demanda no mesmo continente em que estão os usuários, obter recursos adicionais na América do Norte não aliviará a falta de capacidade na Ásia. Toda previsão tem *restrições*, ou parâmetros em torno de como ela pode ser atendida; as restrições estão fundamentalmente relacionadas à intenção, que será discutida na próxima seção.

Mapear solicitações de recursos com restrições às suas alocações a partir da capacidade disponível é igualmente um processo lento: é complexo e tedioso fazer o bin packing¹ das requisições em um espaço limitado de forma manual ou encontrar soluções que se enquadrem em um orçamento limitado.

Esse processo já pode pintar um quadro obscuro, mas, para piorar mais ainda a situação, as ferramentas que ele exige geralmente não são confiáveis ou são inconvenientes para usar. Planilhas sofrem severamente de problemas de escalabilidade e têm capacidades limitadas para verificação de erros. Os dados se tornam ultrapassados e é difícil controlar as mudanças. Com frequência, as equipes são forçadas a fazer suposições simplificadoras e reduzir a complexidade de seus requisitos, apenas para deixar que a manutenção de uma capacidade adequada seja um problema administrável.

Quando os donos dos serviços encaram os desafios de adequar uma série de requisições de capacidade para diversos serviços em recursos disponíveis a eles de modo a atender às várias restrições que um serviço possa ter, há uma imprecisão adicional como consequência. O bin packing é um problema NP-difícil², complexo para seres humanos calcularem manualmente. Além disso, a requisição de capacidade para um serviço, em geral, é um conjunto inflexível de requisitos de demanda: X cores (núcleos de CPU) no cluster Y .

Os motivos pelos quais X cores ou Y clusters são necessários, e qualquer grau de liberdade em torno desses parâmetros, há muito tempo se perderam no momento em que a requisição chega até um ser humano tentando adequar uma lista de demandas aos recursos disponíveis.

O resultado líquido é um gasto enorme de esforço humano para chegar, no melhor caso, próximo a um bin packing. O processo é frágil a mudanças e não há nenhum caminho direto para uma solução ideal.

Nossa solução: planejamento de capacidade baseado em intenção

Especifique os requisitos, não a implementação.

No Google, muitas equipes passaram a adotar uma abordagem que chamamos de *Planejamento de Capacidade Baseado em Intenção* (*Intent-based Capacity Planning*). A premissa básica dessa abordagem é codificar as dependências e parâmetros (*intenção*) das necessidades de um serviço por meio de programação e usar essa codificação para gerar automaticamente um plano de alocação que detalhe quais recursos serão destinados a quais serviços, em qual cluster. Se a demanda, os recursos disponíveis ou os requisitos do serviço mudarem, um novo plano pode ser simplesmente gerado de modo automático em resposta aos parâmetros alterados, que a partir de agora será a nova melhor distribuição para os recursos.

Com os verdadeiros requisitos e a flexibilidade de um serviço capturados, o plano de capacidade agora passa a ser muito mais dinâmico face a mudanças, e podemos alcançar uma solução ótima que atenda ao máximo possível de parâmetros. Com a delegação do bin packing aos computadores, as tarefas penosas (toil) dos seres humanos se reduzem drasticamente, e os donos dos serviços podem se concentrar em prioridades de alta ordem como SLOs, dependências em produção e requisitos de infraestrutura do serviço, em oposição a brigar por recursos de baixo nível.

Como vantagem adicional, usar uma otimização computacional para mapear intenção à implementação proporciona uma precisão muito maior, resultando, em última instância, em economias de custo para a empresa. O bin packing ainda está longe de ser um problema resolvido, pois determinados tipos ainda

são considerados NP-difícil; entretanto, os algoritmos atuais podem ser resolvidos para uma solução ótima conhecida.

Planejamento de capacidade baseado em intenção

Intenção é a racionalização de como um proprietário de serviço deseja executar o seu serviço. Passar de demandas concretas de recursos para razões motivacionais para chegar na verdadeira intenção do planejamento de capacidade com frequência exige várias camadas de abstração. Considere a seguinte cadeia de abstração:

- 1) “Quero 50 cores nos clusters X, Y e Z para o serviço Foo.”

Essa é uma requisição de recurso explícita. Mas... *por que precisamos dessa quantidade de recursos especificamente nesses clusters em particular?*

- 2) “Quero usar 50 cores em quaisquer 3 clusters na região geográfica YYY para o serviço Foo.”

Essa requisição introduz um grau maior de liberdade e é possivelmente mais fácil de ser atendida, embora não explique o raciocínio por trás de seus requisitos. Mas... *por que precisamos dessa quantidade de recursos, e por que em 3 clusters?*

- 3) “Quero atender à demanda do serviço Foo em cada região geográfica, e ter uma redundância de $N + 2$.”

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 níveis de confiabilidade.”

Esse é um requisito mais abstrato, e a ramificação, se o requisito não for atendido, se torna clara: a confiabilidade sofrerá. Além disso, temos uma flexibilidade maior ainda nesse caso: talvez executar a $N + 2$, na verdade, não seja suficiente nem ideal para esse serviço, e algum outro plano de implantação poderia ser mais adequado.

Então, que nível de intenção deveria ser usado pelo planejamento de capacidade orientado a intenção? O ideal é que todos os níveis de intenção sejam tratados em conjunto, com os serviços se beneficiando quanto mais se deslocarem das intenções especificadas para a implementação. Na experiência do Google, os serviços tendem a alcançar as melhores vitórias quando cruzam o passo 3: bons graus de flexibilidade estão disponíveis, e as ramificações dessa requisição estão em alto nível e em termos mais compreensíveis. Serviços particularmente sofisticados podem visar ao passo 4.

Precursors da intenção

De quais informações precisamos para capturar a intenção de um serviço? Forneça dependências, métricas de desempenho e prioridades.

Dependências

Os serviços no Google dependem de vários outros serviços de infraestrutura e serviços voltados aos usuários, e essas dependências influenciam bastante o local em que um serviço pode ser colocado. Por exemplo, suponha que haja um serviço Foo voltado a usuários que dependa de Bar, um serviço de infraestrutura para armazenagem. Foo expressa um requisito de que Bar deva estar localizado em um raio de 30 milissegundos de latência de rede de Foo. Esse requisito tem repercussões importantes para o lugar em que Foo e Bar serão colocados, e o planejamento de capacidade orientado a intenção deve levar essas restrições em consideração.

Além do mais, as dependências em produção estão aninhadas: para desenvolver o exemplo anterior, suponha que o serviço Bar tenha suas próprias dependências de Baz, um serviço distribuído de baixo nível para armazenagem, e de Qux, um serviço de gerenciamento de aplicações. Desse modo, o lugar em que agora podemos colocar Foo depende do lugar em que podemos colocar Bar, Baz e Qux. Um dado conjunto de dependências de produção pode ser compartilhado, possivelmente com diferentes determinações em torno da intenção.

Métricas de desempenho

A demanda para um serviço se propaga e resulta em demanda para um ou mais serviços diferentes. Entender a cadeia de dependências ajuda a formular o escopo geral do problema de bin packing, mas ainda precisamos de mais informações sobre o uso esperado dos recursos. De quantos recursos computacionais o serviço Foo precisa para tratar N consultas de usuários? Para cada N consultas ao serviço Foo, quantos Mbps de dados esperamos para o serviço Bar?

As métricas de desempenho servem de cola entre as dependências. Elas fazem a conversão entre um ou mais tipos de recursos de alto nível para um ou mais tipos de recursos de baixo nível. Calcular as métricas de desempenho apropriadas para um serviço pode envolver testes de carga e monitoração do uso de recursos.

Prioridades

Inevitavelmente, limitações em recursos resultam em negociações e decisões difíceis: dos vários requisitos que todos os serviços têm, quais deles devem ser sacrificados diante de uma capacidade insuficiente?

Talvez uma redundância de $N + 2$ para o serviço Foo seja mais importante que uma redundância de $N + 1$ para o serviço Bar. Ou, quem sabe, o lançamento da funcionalidade X seja menos importante que uma redundância de $N + 0$ para o serviço Baz.

O planejamento orientado a intenção força que essas decisões sejam feitas de modo transparente, aberto e consistente. Limitações de recursos implicam nas mesmas negociações, mas com muita frequência a atribuição de prioridades pode ser *ad hoc* e opaca aos proprietários de serviços. O planejamento baseado em intenção permite que a atribuição de prioridades seja tão granular ou genérica conforme a necessidade.

Introdução ao Auxon

O Auxon é a implementação do Google de um planejamento de capacidade baseado em intenção e uma solução para alocação de recursos; é um excelente exemplo de um produto de engenharia de software projetado e desenvolvido pela SRE: foi criado por um pequeno grupo de engenheiros de

software e um gerente de programa técnico na SRE ao longo de dois anos. O Auxon é um estudo de caso perfeito para demonstrar como o desenvolvimento de software pode ser promovido dentro da SRE.

O Auxon é utilizado ativamente para planejar o uso de muitos milhões de dólares em recursos de máquina no Google. Ele se tornou um componente crucial no planejamento de capacidade para várias divisões importantes na empresa.

Como produto, o Auxon oferece os meios para coletar descrições baseadas em intenção dos requisitos de recursos e dependências de um serviço. Essas intenções dos usuários são expressas como requisitos para o modo como o proprietário gostaria que o provisionamento do serviço fosse feito. Os requisitos podem ser especificados como requisições do tipo: “Meu serviço deve ser $N + 2$ por continente” ou “Os servidores de frontend não devem estar a mais de 50 ms de distância dos servidores de backend”. O Auxon coleta essas informações por meio de uma linguagem de configuração de usuário ou uma API de programação, traduzindo, assim, as intenções humanas em restrições interpretáveis por máquinas. Os requisitos podem receber prioridades – uma funcionalidade útil caso os recursos sejam insuficientes para atender a todos os requisitos e, desse modo, negociações se tornem necessárias. Esses requisitos – a intenção –, em última instância, são representados internamente por meio de uma combinação de programação linear ou inteira gigante. O Auxon resolve a programação linear e utiliza a solução de bin packing resultante para formular um plano de alocação para os recursos.

A Figura 18.1 e as explicações na sequência apresentam os principais componentes do Auxon.

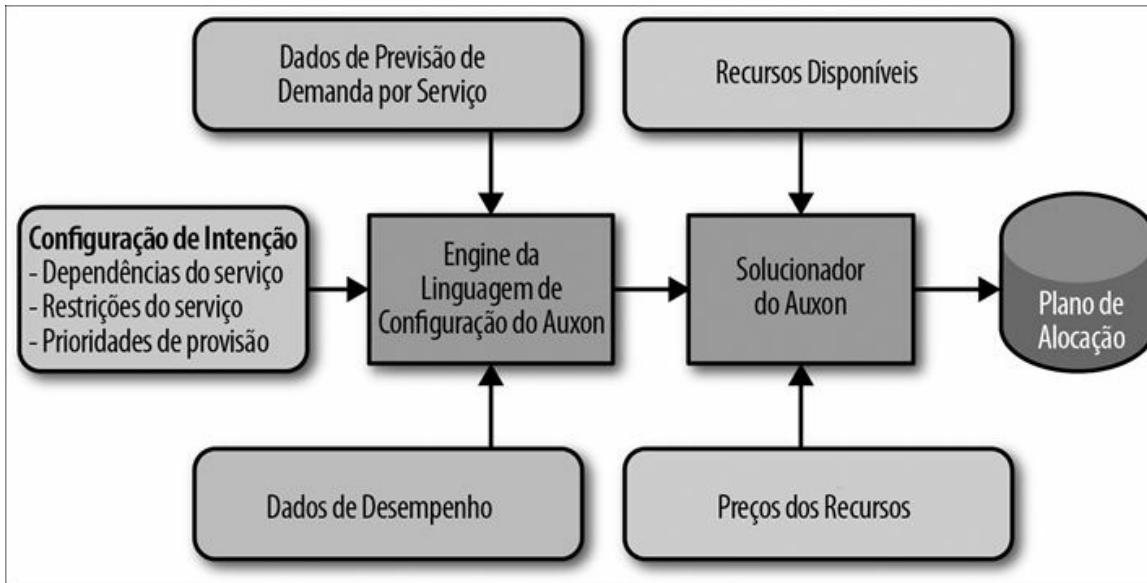


Figura 18.1 – Os principais componentes do Auxon.

Dados de Desempenho descreve como um serviço escala: para cada unidade de demanda X no cluster Y , quantas unidades de dependência Z são usadas? Esses dados para escalar podem ser calculados de várias maneiras, de acordo com a maturidade do serviço em questão. Alguns serviços passam por testes de carga, enquanto outros inferem sua escala com base no desempenho passado.

Dados de Previsão de Demanda por Serviço descreve a tendência de uso para os sinais de demanda previstos. Alguns serviços calculam seu uso futuro a partir de previsões de demanda – uma previsão de consultas por segundo, separada por continente. Nem todos os serviços têm uma previsão de demanda: alguns (por exemplo, um serviço de armazenagem como o Colossus) calculam sua demanda exclusivamente a partir de serviços que dependam deles.

Recursos Disponíveis oferece dados sobre a disponibilidade de recursos básicos e fundamentais: por exemplo, o número de máquinas que se espera estarem disponíveis para uso em um determinado momento no futuro. Na terminologia de programação linear, os recursos disponíveis atuam como um *limite superior* que restringe como os serviços podem crescer e em que lugares podem ser colocados. Em última instância, queremos fazer o melhor uso desses recursos disponíveis, conforme a descrição baseada em intenção do grupo combinado de serviços permite.

Preços dos Recursos oferece dados sobre o custo dos recursos básicos e fundamentais. Por exemplo, o custo das máquinas pode variar globalmente de acordo com o preço do espaço/energia elétrica de uma dada instalação. Na terminologia de programação linear, os preços informam os custos gerais calculados, que atuam como o *objetivo* que queremos minimizar.

A *Configuração de Intenção* é essencial para o modo como as informações baseadas em intenção são passadas para o Auxon. Ela define o que constitui um serviço e como os serviços se relacionam uns com os outros. Em última análise, a configuração atua como uma camada que permite que todos os demais componentes sejam conectados. É projetada para ser legível e configurável por seres humanos.

A *Engine da Linguagem de Configuração do Auxon* atua com base nas informações recebidas da Configuração de Intenção. Esse componente formula uma requisição legível por máquina (um buffer de protocolo) possível de ser compreendida pelo Solucionador do Auxon. Ele aplica uma verificação de sanidade leve na configuração e foi projetado para atuar como gateway entre a definição de intenção configurável por seres humanos e a requisição de otimização interpretável pelas máquinas.

O *Solucionador do Auxon* é o cérebro da ferramenta. Ele formula a combinação de programação linear ou inteira gigante com base na requisição de otimização recebida da Engine da Linguagem de Configuração. Foi projetado para ser bem escalável, o que permite que o solucionador execute em paralelo com centenas ou até mesmo milhares de máquinas executando nos clusters do Google. Além de kits de ferramentas para programação linear e inteira combinadas, há também componentes no Solucionador do Auxon que tratam tarefas como escalonamento, gerenciamento de pool de workers e árvores de decisão descendentes.

O *Plano de Alocação* é a saída do Solucionador do Auxon. Ele prescreve quais recursos devem ser alocados a quais serviços em quais localidades. São os detalhes de implementação calculados a partir da definição baseada em intenção dos requisitos para o problema de planejamento de capacidade. O Plano de Alocação também inclui informações sobre qualquer requisito que possa não ter sido satisfeito – por exemplo, se um requisito não pôde ser

atendido por falta de recursos ou requisitos concorrentes excessivamente rígidos.

Requisitos e implementação: sucessos e lições aprendidas

Inicialmente, o Auxon foi imaginado por um SRE e um gerente de programa técnico que, separadamente, haviam recebido de suas respectivas equipes a tarefa de efetuar o planejamento de capacidade de porções grandes da infraestrutura do Google. Por terem feito planejamentos de capacidade manualmente em planilhas, eles estavam em uma boa posição para entender as ineficárias e as oportunidades de melhoria por meio de automação, e as funcionalidades que uma ferramenta desse tipo poderia exigir.

Durante o desenvolvimento do Auxon, a equipe de SRE por trás do produto continuou profundamente envolvida no mundo da produção. Ela mantinha uma função nos rodízios de plantão para vários serviços do Google e participava das discussões de design e da liderança técnica desses serviços. Por meio dessas interações contínuas, a equipe foi capaz de permanecer em sintonia com o mundo da produção: atuava tanto como o cliente quanto como o desenvolvedor de seu próprio produto. Quando o produto falhava, a equipe sofria impactos diretos. As solicitações de funcionalidades eram baseadas nas próprias experiências em primeira mão da equipe. A experiência em primeira mão no domínio do problema não só criou um enorme senso de responsabilidade pelo sucesso do produto como também ajudou a dar credibilidade e legitimidade ao produto dentro da SRE.

Aproximação

Não enfoque na perfeição nem na pureza de uma solução, especialmente se as fronteiras do problema não forem bem conhecidas. Lance e itere.

Qualquer esforço de engenharia de software suficientemente complexo está fadado a se deparar com incertezas quanto ao modo como um componente deve ser projetado ou um problema deve ser enfrentado. O Auxon se deparou com essas incertezas no início de seu desenvolvimento porque o mundo da programação linear era um território inexplorado para os membros da equipe. As limitações da programação linear, que parecia ser uma parte central de

como o produto provavelmente funcionaria, não eram bem compreendidas. Para acabar com a consternação da equipe em relação a essa dependência de compreensão insuficiente, optamos por, inicialmente, construir uma engine solucionadora simplificada (o chamado “Solucionador Estúpido”), que aplicava um pouco de heurística simples para o modo como os serviços deveriam ser organizados com base nos requisitos especificados pelo usuário. Embora o Solucionador Estúpido jamais produzisse uma solução realmente ótima, ele deu à equipe uma noção de que o que visávamos para o Auxon era factível, mesmo que não criássemos algo perfeito no início.

Ao implantar uma aproximação para ajudar a agilizar o desenvolvimento, é importante conduzir a tarefa de modo que seja possível à equipe fazer melhorias e rever essa aproximação no futuro. No caso do Solucionador Estúpido, toda a interface do solucionador foi abstraída no Auxon de modo que o funcionamento interno do solucionador pudesse ser trocado depois. Em algum momento no futuro, à medida que adquiríamos mais confiança em um modelo unificado de programação linear, trocar o Solucionador Estúpido por algo mais inteligente foi uma operação simples.

Os requisitos de produto do Auxon também tinham alguns aspectos desconhecidos. Construir um software com requisitos difusos pode ser um desafio frustrante, mas certo grau de incerteza não precisa ser paralisante. Utilize essa incerteza como incentivo para garantir que o software seja projetado para ser tanto genérico quanto modular. Por exemplo, um dos objetivos do projeto Auxon era se integrar aos sistemas de automação do Google a fim de permitir que um Plano de Alocação pudesse ser encenado diretamente na produção (atribuindo recursos e ativando/desativando/redimensionando os serviços conforme fosse apropriado). No entanto, naquela época, o mundo dos sistemas de automação apresentava um fluxo intenso, pois uma enorme variedade de abordagens estava em uso. Em vez de tentar fazer o design de soluções exclusivas para permitir que o Auxon trabalhasse com cada ferramenta em particular, moldamos o Plano de Alocação para que fosse universalmente conveniente, de modo que esses sistemas de automação pudessem trabalhar em seus próprios pontos de integração. Essa abordagem “agnóstica” se tornou essencial ao processo do Auxon para conquistar novos clientes, pois permitia

que eles começassem a usar o Auxon sem mudar para uma ferramenta de automação de ativação, de previsão ou de dados de desempenho em particular.

Também tiramos proveito de designs modulares para lidar com requisitos difusos ao construir um modelo de desempenho de máquinas no Auxon. Dados sobre o desempenho de futuras plataformas de máquinas (por exemplo, CPU) eram escassos, mas nossos usuários queriam ter uma maneira de modelar vários cenários de capacidade de máquina. Abstraímos os dados de máquina por trás de uma única interface, permitindo ao usuário alternar entre diferentes modelos de desempenho futuro das máquinas. Mais tarde, estendemos ainda mais essa modularidade, com base em requisitos cada vez mais definidos, a fim de fornecer uma biblioteca simples de modelagem de desempenho de máquinas que funcionasse nessa interface.

Se há uma lição aprendida com o nosso estudo de caso do Auxon é que o antigo lema “lance e itere” é particularmente relevante em projetos de desenvolvimento de software da SRE. Não espere pelo design perfeito; em vez disso, mantenha a visão geral em mente enquanto prossegue com o design e o desenvolvimento. Quando se deparar com áreas de incerteza, faça o design do software de modo que ele seja flexível o suficiente para que, se o processo ou a estratégia mudarem em um nível mais alto, você não incorra em um custo enorme de retrabalho. Ao mesmo tempo, porém, mantenha os pés no chão ao garantir que as soluções genéricas tenham uma implementação específica no mundo real, que demonstre a utilidade do design.

Aumentando a divulgação e levando à adoção

Como ocorre com qualquer produto, um software desenvolvido pela SRE deve ser projetado com o conhecimento de seus usuários e dos requisitos. Ele deve conduzir à adoção por meio de utilidade, desempenho e capacidade demonstrada, tanto para colaborar com as metas de confiabilidade do ambiente de produção do Google quanto para melhorar as vidas dos SREs. O processo de socialização de um produto e a conquista de usuários em toda a empresa é fundamental para o sucesso do projeto.

Não subestime os esforços necessários para aumentar a divulgação e o interesse em seu produto de software – uma única apresentação ou um anúncio via email não são suficientes. A socialização de ferramentas de software internas para um grande público-alvo exige tudo o que está listado a seguir:

- Uma abordagem consistente e coerente
- Defesa por parte dos usuários
- Patrocínio dos engenheiros seniores e da gerência, a quem você deverá demonstrar a utilidade de seu produto

É importante considerar o ponto de vista do cliente ao deixar seu produto utilizável. Um engenheiro pode não ter tempo nem inclinação para mergulhar no código-fonte a fim de descobrir como usar uma ferramenta. Embora os clientes internos, em geral, sejam mais tolerantes que os clientes externos à presença de arestas e a versões alfa preliminares, continua sendo necessário fornecer uma documentação. Os SREs são ocupados e se sua solução for muito difícil ou confusa, eles escreverão suas próprias soluções.

Defina expectativas

Quando um engenheiro com anos de familiaridade com um domínio de problema dá início ao design de um produto, é fácil imaginar um estado final utópico para o trabalho. No entanto, é importante diferenciar metas do produto que sejam uma aspiração dos critérios mínimos de sucesso (ou o Minimum Viable Product, isto é, o Produto Mínimo Viável). Os projetos podem perder a credibilidade e falhar ao fazer promessas demais, muito cedo; ao mesmo tempo, se um produto não prometer um resultado suficientemente gratificante, poderá ser difícil proporcionar a energia de ativação necessária para convencer as equipes internas a experimentarem algo novo. Demonstrar um progresso contínuo e incremental por meio de versões rápidas aumenta a confiança do usuário na capacidade de sua equipe de entregar um software útil.

No caso do Auxon, atingimos um equilíbrio ao planejar um cronograma de longo prazo e fazer correções de curto prazo. Prometemos às equipes que:

- Qualquer esforço de adoção e de configuração proporcionaria a vantagem

imediata de aliviar o sofrimento de fazer o bin packing das requisições de recursos no curto prazo manualmente.

- À medida que funcionalidades adicionais fossem desenvolvidas no Auxon, os mesmos arquivos de configuração poderiam ser portados e proporcionariam economias de custo e outras vantagens muito mais amplas e de longo prazo. O cronograma de longo prazo do projeto permitia que os serviços determinassem rapidamente se seus casos de uso ou as funcionalidades exigidas estariam ou não implementados nas primeiras versões. Enquanto isso, a abordagem de desenvolvimento iterativo do Auxon alimentava as prioridades de desenvolvimento e garantiam novos marcos (milestones) para o cronograma.

Identifique os clientes apropriados

A equipe que estava desenvolvendo o Auxon percebeu que uma solução única não seria adequada a todos; muitas equipes maiores já tinham soluções internas para planejamento de capacidade que funcionavam bem o suficiente para serem aceitáveis. Embora suas ferramentas personalizadas não fossem perfeitas, essas equipes não sofriam muito com o processo de planejamento de capacidade a ponto de experimentar uma nova ferramenta, em especial uma versão alfa com arestas a serem aparadas.

As versões iniciais do Auxon tinham intencionalmente como alvos as equipes que não possuíam nenhum processo de planejamento de capacidade implantado. Como essas equipes teriam que investir esforços de configuração independentemente de adotarem uma ferramenta existente ou a nossa nova abordagem, elas estavam interessadas em adotar a ferramenta mais nova. Os primeiros sucessos que o Auxon alcançou junto a essas equipes demonstraram a utilidade do projeto e transformaram os próprios clientes em defensores da ferramenta. Quantificar a utilidade do produto se mostrou mais benéfico ainda; quando conquistamos uma das Áreas de Negócio do Google, a equipe escreveu um estudo de caso detalhando o processo e comparando os resultados antes e depois. As economias de tempo e a redução de tarefas penosas aos seres humanos por si sós constituíram um enorme incentivo para outras equipes experimentarem o Auxon.

Serviço aos clientes

Apesar de o software desenvolvido na SRE ter como público-alvo TPMs (Technical Program Managers, ou Gerentes Técnicos de Programa) e engenheiros com alto grau de proficiência técnica, qualquer software suficientemente inovador apresenta uma curva de aprendizado aos novos usuários. Não tenha medo de oferecer um suporte de primeira classe aos primeiros a adotarem a ferramenta para ajudá-los nesse processo. Às vezes, a automação também implica uma série de questões emocionais, como o medo de que o cargo de alguém seja substituído por um shell script. Ao trabalhar individualmente com os primeiros usuários, você poderá abordar esses medos pessoalmente e mostrar que, em vez de precisar executar tarefas penosas enfadonhas de forma manual, a equipe será dona das configurações, dos processos e dos resultados finais de seu trabalho técnico. Aqueles que adotarem a ferramenta mais tarde serão convencidos pelos exemplos felizes dos primeiros que a adotaram.

Além do mais, como as equipes de SRE do Google estão distribuídas pelo mundo, é vantajoso ter os primeiros a adotar um projeto como seus defensores, pois eles podem servir como experts locais para outras equipes interessadas em experimentar o projeto.

Design no nível correto

Uma ideia à qual demos o nome de *agnosticismo* – escrever o software para que seja genérico a fim de permitir uma variedade de fontes de dados como entrada – foi um princípio essencial no design do Auxon. Agnosticismo significa que os clientes não precisariam se comprometer com nenhuma ferramenta exclusiva para usar o framework do Auxon. Essa abordagem possibilitou que o Auxon tivesse uma utilidade suficientemente genérica, mesmo quando equipes com casos de uso divergentes passaram a usá-lo. Abordávamos usuários em potencial com a mensagem “venha como está; trabalharemos com o que você tem”. Ao evitar um excesso de personalização para um ou dois usuários mais importantes, conseguimos uma adoção mais ampla pela empresa e reduzimos a barreira de entrada em novos serviços.

Também nos esforçamos de forma consciente para evitar a armadilha de

definir o sucesso como 100% de adoção pela empresa. Em muitos casos, há cada vez menos retornos por percorrer a última milha a fim de possibilitar um conjunto de funcionalidades que seja suficiente para todos os serviços na grande diversidade do Google.

Dinâmica das equipes

Ao selecionar engenheiros para trabalhar no desenvolvimento de um produto de software na SRE, percebemos que há uma grande vantagem em criar uma equipe inicial que combine pessoas com habilidades genéricas, capazes de se preparar rapidamente em relação a um novo assunto, com engenheiros que possuam uma variedade de conhecimentos e experiência. Uma diversidade de experiências cobre pontos cegos assim como evita as armadilhas de supor que os casos de uso de todas as equipes sejam iguais aos seus.

É essencial que sua equipe estabeleça um relacionamento profissional com os especialistas necessários, e que seus engenheiros se sintam confortáveis em trabalhar em um novo domínio de problema. Para as equipes de SRE na maioria das empresas, aventurar-se nesse novo domínio de problema exige tarefas de terceirização ou trabalhar com consultores, mas as equipes de SRE em empresas maiores podem fazer parcerias com experts internos. Durante as fases iniciais de conceito e design do Auxon, apresentamos nosso documento de design às equipes internas do Google especializadas em Operations Research (Pesquisas Operacionais) e Quantitative Analysis (Análises Quantitativas) para que tirássemos proveito de seu expertise no campo e também para dar um impulso inicial no conhecimento da equipe do Auxon sobre planejamento de capacidade.

À medida que o desenvolvimento do projeto continuava e o conjunto de funcionalidades do Auxon se tornou mais amplo e complexo, a equipe contratou membros com experiência anterior em estatística e em otimização matemática – o que, em uma empresa menor, seria correspondente a trazer um consultor externo para dentro da empresa. Esses novos membros da equipe puderam identificar áreas de melhoria quando as funcionalidades básicas do projeto estavam concluídas, e acrescentar refinamentos passou a ser a nossa maior prioridade.

O momento certo para engajar especialistas, é claro, variará de projeto para projeto. Como diretriz geral, o projeto deve estar em andamento, com um sucesso demonstrável, para que as habilidades da equipe atual sejam significativamente impulsionadas pelo expertise adicional.

Promovendo a engenharia de software na SRE

O que faz de um projeto um bom candidato para passar de uma ferramenta utilizada uma só vez para um esforço de engenharia de software completo? Sinais positivos claros incluem engenheiros com experiência em primeira mão no domínio relativo, interessados em trabalhar no projeto, e uma base de usuários como alvo que seja altamente técnica (e, desse modo, capaz de fornecer relatórios de bug claros durante as fases iniciais do desenvolvimento). O projeto deve oferecer vantagens perceptíveis, como redução de tarefas penosas para os SREs, melhoria de uma parte da infraestrutura ou a organização de um processo complexo.

É importante que o projeto esteja alinhado com o conjunto geral de objetivos da empresa para que os líderes de engenharia possam avaliar seu impacto em potencial e, subsequentemente, defender seu projeto, tanto junto às equipes que respondem a eles quanto junto a outras equipes que possam ter interface com suas equipes. Uma socialização em toda a empresa e revisões ajudam a evitar esforços discrepantes ou que se sobreponham, e é mais fácil contratar funcionários e apoiar um produto que seja fácil de demonstrar que colabore com o objetivo geral de um departamento.

O que faz de um projeto um candidato ruim? Muitos dos mesmos sinais vermelhos que você pode instintivamente identificar em qualquer projeto de software, por exemplo, um software que entre em contato com muitas partes ao mesmo tempo ou um design de software que exija uma abordagem do tipo tudo ou nada e impeça um desenvolvimento iterativo. Como as equipes de SRE do Google estão atualmente organizadas em torno dos serviços que operam, projetos desenvolvidos pela SRE, em particular, correm o risco de ser um trabalho demasiadamente específico, que beneficie apenas uma pequena porcentagem da empresa. Como os bônus para as equipes estão alinhados principalmente com o oferecimento de uma ótima experiência aos

usuários de um serviço em particular, muitas vezes os projetos falham em fazer generalizações para um caso de uso mais amplo, pois a padronização entre equipes de SRE vem em segundo lugar. Na extremidade oposta do espectro, frameworks genéricos demais podem ser igualmente problemáticos; se uma ferramenta se esforçar para ser flexível e universal demais, ela correrá o risco de não se adequar bem a nenhum caso de uso e, desse modo, ter um valor insuficiente por si só. Projetos com escopo amplo e metas abstratas geralmente exigem um escopo significativo de desenvolvimento, mas não tratam os casos de uso concretos necessários para proporcionar vantagens ao usuário final em um período de tempo razoável.

Como exemplo de um caso de uso amplo: um distribuidor de carga de camada 3 desenvolvido pelos SREs do Google se mostrou tão bem-sucedido ao longo dos anos que foi adaptado como um produto voltado a clientes oferecido como Google Cloud Load Balancer [Eis16].

Construindo uma cultura de engenharia de software com sucesso em SRE: composição da equipe e tempo de desenvolvimento

Os SREs muitas vezes são generalistas, pois o desejo de aprender de forma diversificada, e não em profundidade, é bem adequado para entender o panorama geral (e há poucos panoramas maiores que o funcionamento interno complexo de uma infraestrutura técnica moderna). Esses engenheiros com frequência têm boas habilidades de programação e desenvolvimento de software, mas podem não ter a experiência tradicional em engenharia de software de fazer parte de uma equipe de produto ou ter que pensar em requisições de funcionalidades feitas pelos clientes. Uma citação de um engenheiro em um projeto inicial de desenvolvimento de software na SRE sintetiza a abordagem convencional da SRE ao software: “Tenho um documento de design; por que precisamos de requisitos?”. Parcerias com engenheiros, TPMs ou PMs que tenham familiaridade com o desenvolvimento de software voltado a usuários podem ajudar a construir uma cultura de desenvolvimento de software em equipe que reúna o melhor do desenvolvimento de produto de software e a experiência prática em produção.

Tempo dedicado e sem interrupções para trabalhar em projetos é essencial para qualquer esforço de desenvolvimento de software. Um tempo dedicado a projetos é necessário para possibilitar progressos, pois é quase impossível escrever código – muito menos se concentrar em projetos maiores, de mais impacto – quando você estiver alternando entre várias tarefas no curso de uma hora. Desse modo, a capacidade de trabalhar em um projeto de software sem interrupções muitas vezes é um motivo atraente para os engenheiros começarem a trabalhar em um projeto de desenvolvimento. Esse tempo deve ser ferozmente defendido.

A maioria dos produtos de software desenvolvidos na SRE começa como projetos secundários, cuja utilidade os leva a crescer e a serem formalizados. A essa altura, um produto pode se ramificar em uma de diversas direções possíveis:

- Permanecer como um esforço local desenvolvido no tempo livre dos engenheiros.
- Estabelecer-se como um projeto formal por meio de processos estruturados (veja a seção “Chegando lá”).
- Obter patrocínio executivo da liderança de SRE para expandir e se transformar em um esforço de desenvolvimento de software com uma equipe completa.

No entanto, em qualquer um desses cenários – e esse é um ponto que vale a pena enfatizar – é essencial que os SREs envolvidos em qualquer esforço de desenvolvimento continuem a trabalhar como SREs, em vez de se tornarem desenvolvedores em tempo integral dentro da organização SRE. A imersão no mundo da produção confere aos SREs realizando trabalhos de desenvolvimento uma perspectiva de valor inestimável, pois eles são tanto os criadores quanto os clientes de qualquer produto.

Chegando lá

Se você gosta da ideia de um desenvolvimento de software organizado em SRE, é provável que esteja se perguntando como introduzirá um modelo de desenvolvimento de software em uma organização SRE cujo foco está no suporte à produção.

Inicialmente, reconheça que essa meta é tanto uma mudança organizacional quanto um desafio técnico. Os SREs estão acostumados a trabalhar em proximidade com seus colegas de equipe, analisando e reagindo aos problemas rapidamente. Desse modo, você estará trabalhando contra o instinto natural de um SRE para escrever um código de forma rápida a fim de atender às suas necessidades imediatas. Se sua equipe de SRE for pequena, essa abordagem talvez não seja problemática. Entretanto, à medida que sua empresa crescer, essa abordagem *ad hoc* não escalará, resultando em soluções de software amplamente funcionais, porém restritas ou de propósito único, que não poderão ser compartilhadas, o que, inevitavelmente, levará a esforços duplicados e desperdício de tempo.

Em seguida, pense no que você quer realizar ao desenvolver software na SRE. Você quer apenas promover práticas melhores de desenvolvimento de software em sua equipe ou está interessado em um desenvolvimento de software que produza resultados utilizáveis por outras equipes, possivelmente como um padrão para a empresa? Em empresas maiores, já consolidadas, a última mudança exigirá tempo, possivelmente se estendendo por vários anos. Uma mudança como essa deve ser enfrentada em várias frentes, mas proporciona um retorno maior. A seguir, apresentamos algumas diretrizes oriundas da experiência do Google:

Crie e transmita uma mensagem clara

É importante definir e comunicar sua estratégia, os planos e – acima de tudo – as vantagens que a SRE terá como resultado desse esforço. Os SREs são muito céticos (de fato, o ceticismo é um traço que levamos especificamente em consideração na contratação); a resposta inicial de um SRE a um esforço desse tipo provavelmente será “parece que há overhead demais” ou “não vai funcionar nunca”. Comece definindo uma argumentação convincente sobre como essa estratégia ajudará a SRE; por exemplo:

- Soluções de software consistentes e com suporte agilizam a adaptação de novos SREs.
- Reduzir o número de maneiras de realizar a mesma tarefa permite que todo o departamento se beneficie das habilidades que qualquer equipe tenha desenvolvido, fazendo com que esforços e conhecimentos sejam

portáveis entre as equipes.

Quando os SREs começarem a fazer perguntas sobre *como* sua estratégia funcionará, em vez de perguntar *se* ela deverá ser perseguida, você saberá que passou pelo primeiro obstáculo.

Avalie as capacidades de sua empresa

Os SREs têm muitas habilidades, mas é relativamente comum um SRE ter pouca experiência como parte de uma equipe que tenha criado e lançado um produto a um conjunto de usuários. Para desenvolver um software útil, você deverá efetivamente criar uma equipe de produto. Essa equipe inclui funções e habilidades necessárias, das quais sua organização de SRE talvez não tenha precisado antes. Alguém desempenhará a função de gerente de produto, atuando como defensor dos clientes? Seu líder técnico ou gerente de projeto tem as habilidades e/ou a experiência para conduzir um processo de desenvolvimento ágil?

Comece a preencher essas lacunas tirando proveito das habilidades já presentes em sua empresa. Peça à sua equipe de desenvolvimento de produtos que ajude você a definir práticas ágeis por meio de treinamento e coaching. Solicite tempo de consultoria de um gerente de produto para ajudar você a definir os requisitos do produto e priorizar o trabalho nas funcionalidades. Considerando que a oportunidade de desenvolvimento de software seja suficientemente grande, talvez seja o caso de contratar pessoas dedicadas a essas funções. Defender a contratação de pessoas para essas funções será mais fácil depois que você tiver alguns resultados positivos com o experimento.

Lance e itere

Quando iniciar um programa de desenvolvimento de software na SRE, seus esforços serão acompanhados por muitos olhares observadores. É importante estabelecer credibilidade entregando algum produto de valor em um período de tempo razoável. Sua primeira rodada de produtos deve visar a alvos relativamente simples e alcançáveis – aqueles sem soluções controversas ou existentes. Também tivemos sucesso ao combinar essa abordagem com um ritmo de atualização de versão do produto a cada seis

meses, oferecendo funcionalidades adicionais úteis. Esse ciclo de novas versões permitiu às equipes se concentrar na identificação do conjunto correto de funcionalidades a serem desenvolvidas e, então, implementá-las, ao mesmo tempo que aprendiam a ser uma equipe de desenvolvimento de software produtiva. Após o lançamento inicial, algumas equipes do Google passaram para um modelo push-on-green a fim de fazer entregas e ter feedback de modo mais rápido ainda.

Não abaixe seus padrões

Quando começar a desenvolver software, você se sentirá tentado a tomar atalhos. Resista a esse impulso atendo-se aos mesmos padrões utilizados pelas suas equipes de desenvolvimento de produtos. Por exemplo:

- Pergunte a si mesmo: se esse produto fosse criado por uma equipe diferente de desenvolvimento, você o adotaria?
- Se sua solução tiver uma adoção ampla, realizar suas tarefas com sucesso pode se tornar crucial para os SREs. Desse modo, a confiabilidade é da máxima importância. Você tem práticas de revisão de código apropriadas definidas? Você tem testes fim a fim ou de integração? Faça outra equipe de SRE revisar o produto para saber se ele está pronto para produção, como fariam se estivessem assumindo a responsabilidade por qualquer outro serviço.

É preciso bastante tempo para construir credibilidade para seus esforços de desenvolvimento de software, mas basta um período de tempo muito curto para perder a credibilidade devido a um passo errado.

Conclusões

Os projetos de engenharia de software na SRE do Google floresceram à medida que a empresa cresceu e, em muitos casos, as lições aprendidas com projetos anteriores de desenvolvimento de software e sua execução bem-sucedida prepararam o caminho para empreendimentos subsequentes. A experiência prática ímpar em produção que os SREs trazem para o desenvolvimento de ferramentas pode levar a abordagens inovadoras para problemas antigos, conforme vimos no desenvolvimento do Auxon para

tratar o complexo problema do planejamento de capacidade. Projetos de software conduzidos pela SRE também são notadamente benéficos à empresa no desenvolvimento de um modelo sustentável para suporte a serviços em escala. Como os SREs muitas vezes desenvolvem softwares para organizar processos ineficientes ou automatizar tarefas comuns, esses projetos implicam que a equipe de SRE não precisa escalar linearmente de acordo com o tamanho do serviço ao qual ela dá suporte. Em última instância, as vantagens de ter SREs dedicando parte de seu tempo ao desenvolvimento de software são desfrutadas pela empresa, pela organização SRE e pelos próprios SREs.

¹ N.T.: De modo geral, no problema de *bin packing* (empacotamento), objetos de volumes diferentes devem ser empacotados em um número finito de receptáculos (bins) ou contêineres, cada um com um volume V , de modo a minimizar o número de contêineres usados (baseado em https://en.wikipedia.org/wiki/Bin_packing_problem). Nesse caso, trata-se de alocar os recursos conforme as requisições, da melhor maneira possível.

² N.T.: Veja <https://pt.wikipedia.org/wiki/NP-difícil>.

CAPÍTULO 19

Distribuição de carga no frontend

Escrito por Piotr Lewandowski

Editado por Sarah Chavis

Servimos a muitos milhões de requisições por segundo e, como você já deve ter adivinhado, usamos mais do que um único computador para lidar com essa demanda. Contudo, *mesmo* que tivéssemos um supercomputador que, de algum modo, fosse capaz de tratar todas essas requisições (pense na conectividade de rede que uma configuração como essa exigiria!), ainda não empregariámos uma estratégia que dependesse de um único ponto de falha; quando lidar com sistemas de larga escala, colocar todos os seus ovos em um único cesto é receita para um desastre.

Este capítulo tem como foco a distribuição de carga em alto nível – como distribuímos o tráfego de usuários *entre* os datacenters. O capítulo seguinte explora em detalhes o modo como implementamos a distribuição de carga *dentro* de um datacenter.

Capacidade não é a resposta

Por questão de argumentação, vamos supor que tenhamos uma máquina inacreditavelmente poderosa e uma rede que nunca falhe. *Essa* configuração seria suficiente para atender às necessidades do Google? Não. Mesmo essa configuração ainda estaria limitada pelas restrições físicas associadas à nossa infraestrutura de rede. Por exemplo, a velocidade da luz é um fator limitante nas velocidades de comunicação em cabos de fibra óptica, e cria um limite superior para a velocidade com que podemos servir dados de acordo com a distância que eles precisam trafegar. Mesmo em um mundo ideal, depender de uma infraestrutura com um único ponto de falha é uma péssima ideia.

Na verdade, o Google tem milhares de máquinas e mais usuários ainda, muitos dos quais geram várias requisições ao mesmo tempo. A *distribuição de carga de tráfego* é o modo como decidimos quais das muitas e muitas máquinas de nossos datacenters servirão a uma requisição em particular. O ideal é que o tráfego seja distribuído entre vários links de rede, datacenters e máquinas de uma maneira “ótima”. Todavia, o que “ótima” significa nesse contexto? Na realidade, não há uma resposta única, pois a solução ótima depende intensamente de uma variedade de 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 lidamos

Vamos começar analisando dois cenários comuns de tráfego: uma requisição básica de pesquisa e uma requisição de upload de vídeo. Os usuários querem obter os resultados de suas consultas rapidamente, portanto a variável mais importante para a requisição de pesquisa é a latência. Por outro lado, os usuários esperam que os uploads de vídeo demorem um período de tempo não desprezível, mas também querem que essas requisições sejam bem-sucedidas na primeira vez, portanto a variável mais importante para upload de vídeo é o throughput. As diferentes necessidades das duas requisições desempenham um papel no modo de determinar a distribuição ótima para cada requisição no nível *global*:

- A requisição de pesquisa é enviada para o datacenter mais próximo disponível – conforme medido em RTT (Round-Trip Time, ou Tempo de Ida e Volta) –, pois queremos minimizar a latência da requisição.
- O stream de upload de vídeo é roteado para um caminho diferente – talvez para um link que, no momento, esteja subutilizado – a fim de maximizar o throughput em detrimento da latência.

Porém, no nível *local*, dentro de um datacenter em particular, muitas vezes supomos que todas as máquinas no prédio estão igualmente distantes do usuário e conectadas à mesma rede. Desse modo, uma distribuição ótima de carga tem como foco uma utilização ótima de recursos e a proteção de um único servidor contra sobrecarga.

É claro que esse exemplo apresenta um quadro extremamente simplificado. Na verdade, muitas outras considerações influenciam uma distribuição de carga ótima: algumas requisições podem ser direcionadas a um datacenter um pouco mais distante a fim de manter os caches aquecidos, ou um tráfego não interativo pode ser encaminhado para uma região totalmente diferente para evitar congestionamento de rede. A distribuição de carga, especialmente em sistemas de grande porte, é tudo, menos simples e estática. No Google, abordamos o problema fazendo a distribuição de carga em vários níveis, dois dos quais serão descritos nas próximas seções. Com o intuito de apresentar uma discussão concreta, consideraremos requisições HTTP enviadas sobre TCP. A distribuição de carga de serviços stateless (sem estado) – como DNS sobre UDP – difere um pouco, mas a maior parte dos mecanismos descritos aqui deve ser aplicável a serviços stateless também.

Distribuição de carga usando DNS

Antes que um cliente sequer possa enviar uma requisição HTTP, com frequência ele deve buscar um endereço IP usando DNS. Isso oferece a oportunidade perfeita para introduzir nossa primeira camada de distribuição de carga: *distribuição de carga no DNS*. A solução mais simples é devolver vários registros A ou AAAA na resposta do DNS e deixar o cliente escolher um endereço IP de forma arbitrária. Embora seja conceitualmente simples e trivial de implementar, essa solução apresenta vários desafios.

O primeiro problema é que ela proporciona muito pouco controle sobre o comportamento do cliente: os registros são selecionados aleatoriamente, e cada um atrairá uma quantidade aproximadamente igual de tráfego. Podemos atenuar esse problema? Teoricamente, poderíamos usar registros SRV para especificar pesos e prioridades aos registros, mas registros SRV ainda não foram adotados para HTTP.

Outro problema em potencial tem origem no fato de que, em geral, o cliente não é capaz de determinar o endereço mais próximo. *Podemos* atenuar o problema desse cenário usando um endereço anycast para servidores de nomes autoritativos e tirar proveito de as consultas DNS fluírem para o endereço mais próximo. Em sua resposta, o servidor pode devolver endereços

roteados ao datacenter mais próximo. Uma melhoria adicional criaria um mapa de todas as redes e suas localizações físicas aproximadas e serviria respostas DNS com base nesse mapeamento. Entretanto, essa solução tem o custo de ter uma implementação de servidor DNS muito mais complexa e a necessidade de manter um processo que deixe o mapa de localizações atualizado.

É claro que nenhuma dessas soluções é trivial, devido a uma característica fundamental do DNS: usuários finais raramente conversam de forma direta com servidores de nomes autoritativos. Em vez disso, um servidor DNS recursivo geralmente está em algum ponto entre os usuários finais e os servidores de nomes. Esse servidor faz proxy das consultas entre um usuário e um servidor e, com frequência, oferece uma camada de caching. O DNS intermediário tem três implicações muito importantes no gerenciamento de tráfego:

- Resolução recursiva de endereços IP
- Caminhos de resposta não determinísticos
- Complicações adicionais de caching

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 do resolver (resolutor) recursivo. Essa é uma limitação séria, pois permite otimização de respostas apenas para a menor distância entre o resolver 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 resolver recursivo. Dessa maneira, um servidor de nomes autoritativo devolve uma resposta que é ótima do ponto de vista do usuário, e não do ponto de vista do resolver. Embora esse ainda não seja o padrão oficial, suas vantagens evidentes levaram os maiores resolvers de DNS (como OpenDNS e Google¹) a darem suporte a ele.

Não só é difícil encontrar um endereço IP ótimo para devolver ao servidor de nomes para uma dada requisição de usuário, como também esse servidor de nomes pode ser responsável por servir a milhares ou a milhões de usuários, em regiões que variam de um único escritório a todo um continente. Por exemplo, um ISP nacional de grande porte pode executar servidores de

nomes para toda a sua rede a partir de um datacenter, mas tem interconexões de rede em todas as áreas metropolitanas. Os servidores de nomes do ISP então retornariam uma resposta com o endereço IP mais adequado ao seu datacenter, apesar de haver caminhos de rede melhores para todos os usuários!

Por fim, resolvers recursivos geralmente fazem cache das respostas e as encaminham dentro dos limites indicados pelo campo TTL (Time-To-Live, ou Tempo de Vida) do registro de DNS. O resultado final é que estimar o impacto de uma dada resposta é difícil: uma única resposta autoritativa pode alcançar um único usuário ou muitos milhares deles. Resolvemos esse problema de duas maneiras:

- Analisamos mudanças de tráfego e atualizamos continuamente nossa lista de resolvers de DNS conhecidos com o tamanho aproximado da base de usuários por trás de um dado resolver, o que nos permite monitorar o possível impacto de qualquer resolver.
- Estimamos a distribuição geográfica dos usuários por trás de cada resolver monitorado para aumentar as chances de direcionarmos esses usuários para o melhor local.

Estimar a distribuição geográfica é particularmente complicado se a base de usuários estiver distribuída em regiões grandes. Em casos como esse, fazemos negociações para selecionar o melhor local e otimizar a experiência para a maioria dos usuários.

Porém, o que “melhor local” realmente significa no contexto de distribuição de carga no DNS? A resposta mais óbvia é o local mais próximo ao usuário. Entretanto (como se determinar as localizações dos usuários não fosse, por si só, difícil), há critérios adicionais. O distribuidor de carga no DNS precisa garantir que o datacenter que ele selecionar tenha capacidade suficiente para servir às requisições dos usuários que provavelmente receberão a sua resposta. Ele também precisa saber que o datacenter selecionado e sua conectividade de rede estão em bom estado, pois direcionar requisições de usuários a um datacenter com problemas de energia ou de rede não é ideal. Felizmente, podemos integrar o servidor DNS autoritativo com nossos sistemas de controle global que monitoram tráfego, capacidade e o estado de

nossa infraestrutura.

A terceira implicação do DNS intermediário está relacionada a caching. Considerando que os servidores de nomes autoritativos não podem limpar os caches dos resolvers, os registros de DNS precisam de um TTL relativamente baixo. Isso define de modo eficiente um limite inferior para a velocidade com que mudanças de DNS podem ser propagadas até os usuários.² Lamentavelmente, não há muito que possamos fazer, além de ter isso em mente quando tomarmos decisões referentes à distribuição de carga.

Apesar de todos esses problemas, o DNS ainda é a maneira mais simples e mais eficiente de distribuir a carga antes que a conexão do usuário sequer tenha início. Por outro lado, devemos esclarecer que apenas a distribuição de carga com DNS não é suficiente. Tenha em mente que todas as respostas de DNS servidas devem estar dentro do limite de 512 bytes³ definido pela RFC 1035 [Moc87]. Essa restrição define um limite superior para o número de endereços que podemos inserir em uma única resposta DNS, e é quase certo que esse valor é menor que o nosso número de servidores.

Para *realmente* resolver o problema de distribuição de carga no frontend, esse nível inicial de distribuição de carga no DNS deve ser seguido de um nível que tire proveito dos endereços IP virtuais.

Distribuição de carga no endereço IP virtual

Os endereços IP virtuais (VIPs) não são atribuídos a nenhuma interface de rede em particular. Em vez disso, eles são geralmente compartilhados entre vários dispositivos. No entanto, do ponto de vista do usuário, o VIP continua sendo um único endereço IP comum. Teoricamente, essa prática nos permite ocultar detalhes de implementação (como o número de máquinas por trás de um VIP em particular) e facilita a manutenção, pois podemos agendar upgrades ou adicionar mais máquinas ao pool sem que o usuário saiba.

Na prática, a parte mais importante da implementação de VIP é um dispositivo chamado *distribuidor de carga de rede*. O distribuidor recebe pacotes e os encaminha para uma das máquinas por trás do VIP. Esses backends podem então processar a requisição.

Há várias abordagens possíveis que o distribuidor pode adotar ao decidir qual backend deve receber a requisição. A primeira abordagem (e talvez a mais intuitiva) é sempre dar preferência ao backend menos carregado. Teoricamente, essa abordagem deve resultar na melhor experiência ao usuário final, pois as requisições serão sempre encaminhadas à máquina menos ocupada. Infelizmente, essa lógica se desfaz rapidamente no caso de protocolos stateful (com estado), que devem usar o mesmo backend pela duração de uma requisição. Esse requisito implica que o distribuidor deve manter o controle de 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 esse ID para selecionar um backend. Por exemplo, o ID de conexão poderia ser expresso como:

$$\text{id(packet)} \bmod N$$

em que id é a função que recebe packet como entrada e gera um ID de conexão, e N é o número de backends configurados.

Isso evita a armazenagem do estado, e todos os pacotes que pertençam a uma única conexão serão sempre encaminhados ao mesmo backend. Sucesso? Ainda não, exatamente. O que aconteceria se um backend falhar e precisar ser removido da lista de backends? Repentinamente, N se torna $N-1$, e então $\text{id(packet)} \bmod N$ se torna $\text{id(packet)} \bmod N-1$. De repente, quase todos os pacotes serão mapeados para um backend diferente! Se os backends não compartilharem nenhum estado entre si, esse remapeamento forçará uma reinicialização de quase todas as conexões existentes. Esse cenário, definitivamente, *não* representa a melhor experiência ao usuário, mesmo que esses eventos não sejam frequentes.

Felizmente, há uma solução alternativa que não exige manter o estado de todas as conexões em memória, mas não forçará a reinicialização de todas as conexões quando uma única máquina cair: *hashing consistente*. Proposto em 1997, o hashing consistente [Kar97] descreve uma maneira de fornecer um algoritmo de mapeamento que permaneça relativamente estável, mesmo quando novos backends são adicionados ou removidos da lista. Essa

abordagem minimiza a desordem nas conexões existentes quando o pool de backends mudar. Como resultado, geralmente podemos utilizar uma monitoração simples de conexões, mas lançar mão de um hashing consistente quando o sistema estiver sob pressão (por exemplo, durante um ataque de negação de serviço, ou denial of service).

Voltando à pergunta mais ampla: como, exatamente, um distribuidor de carga de rede deve encaminhar pacotes a um backend selecionado com VIP? Uma solução é fazer uma Network Address Translation (Tradução de Endereços de Rede). No entanto, isso exige manter uma entrada para cada conexão na tabela de monitoração, o que impossibilita ter um sistema secundário (fallback) totalmente sem estados (stateless).

Outra solução é modificar informações na camada de enlace de dados (camada 2 do modelo de rede OSI). Ao mudar o endereço MAC de destino de um pacote encaminhado, o distribuidor de carga pode deixar todas as informações das camadas superiores intactas, de modo que o backend receberá os endereços IP de origem e de destino originais. O backend pode então enviar uma resposta direta a quem fez o envio originalmente – uma técnica conhecida como *DSR* (*Direct Server Response*, ou Resposta Direta do Servidor). Se as requisições do usuário forem pequenas e as respostas forem grandes (por exemplo, a maioria das requisições HTTP), o DSR proporciona uma economia enorme, pois apenas uma pequena fração do tráfego precisará passar pelo distribuidor de carga. Melhor ainda, o DSR não exige que mantenhamos os estados no dispositivo distribuidor de carga. Infelizmente, usar a camada 2 para distribuição de carga interna *implica* em sérias desvantagens quando implantado em escala: todas as máquinas (isto é, todos os distribuidores de carga e todos os seus backends) devem ser capazes de acessar umas às outras na camada de enlace de dados. Isso não será um problema se essa conectividade puder ser tratada pela rede e o número de máquinas não crescer excessivamente, pois todas as máquinas precisarão estar em um único domínio de broadcast. Como você pode imaginar, o Google passou do ponto em que essa solução era viável há um bom tempo, e teve que encontrar uma abordagem alternativa.

Nossa solução atual de distribuição de carga com VIP [Eis16] utiliza

encapsulamento de pacotes. Um distribuidor de carga de rede coloca o pacote encaminhado em outro pacote IP com GRE (Generic Routing Encapsulation, ou Encapsulamento para Roteamento Genérico) [Han94], e utiliza o endereço de um backend como destino. Um backend que receba o pacote remove a camada IP+GRE mais externa e processa o pacote IP interno como se tivesse sido diretamente entregue para sua interface de rede. O distribuidor de carga de rede e o backend não precisam mais estar no mesmo domínio de broadcast; eles podem estar até mesmo em continentes diferentes, desde que haja uma rota entre eles.

O encapsulamento de pacotes é um sistema eficaz, que oferece grande flexibilidade para o modo como nossas redes são projetadas e evoluem. Infelizmente, o encapsulamento também tem um preço: um tamanho de pacote maior. O encapsulamento introduz overhead (24 bytes no caso de IPv4+GRE, para ser exato), o que pode fazer o pacote exceder o tamanho disponível de MTU (Maximum Transmission Unit, ou Unidade Máxima de Transmissão) e exigir fragmentação.

Depois que o pacote alcança o datacenter, a fragmentação pode ser evitada usando um MTU maior dentro do datacenter; no entanto, essa abordagem exige uma rede que tenha suporte para PDUs (Protocol Data Units, ou Unidades de Dados de Protocolo) maiores. Como ocorre com muitos sistemas em escala, a distribuição de carga parece simples à primeira vista – faça a distribuição de carga com antecedência e frequentemente –, mas a dificuldade está nos detalhes, tanto na distribuição de carga no frontend quanto no tratamento de pacotes depois que esses alcançam o datacenter.

¹ Veja <https://groups.google.com/forum/#topic/public-dns-announce/67oxFjSLeUM>.

² Infelizmente, nem todos os resolvers de DNS respeitam o valor de TTL definido pelos servidores de nomes autoritativos.

³ Caso contrário, os usuários deverão estabelecer uma conexão TCP apenas para obter uma lista de endereços IP.

CAPÍTULO 20

Distribuição de carga no datacenter

Escrito por Alejandro Forero Cuervo

Editado por Sarah Chavis

Este capítulo enfoca a distribuição de carga no interior do datacenter. Especificamente, serão discutidos os algoritmos para distribuir a tarefa de tratar um fluxo de consultas em um determinado datacenter. Discutiremos políticas no nível de aplicação para encaminhamento de requisições a servidores individuais que possam processá-las. Princípios de rede de mais baixo nível (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 – esse fluxo poderia vir do próprio datacenter, de datacenters remotos ou ser uma mistura de ambos – a uma taxa que não exceda os recursos que o datacenter tenha para processá-lo (ou que exceda apenas durante pequenos intervalos de tempo). Suponha também que haja *serviços* no datacenter, nos quais essas consultas operam. Esses serviços estão implementados na forma de vários processos servidores homogêneos e intercambiáveis, em sua maioria executando em máquinas distintas. Os serviços menores geralmente têm pelo menos três desses processos (usar menos processos significa perder 50% ou mais de sua capacidade em caso de perda de uma única máquina) e o maior pode ter mais de 10.000 processos (conforme o tamanho do datacenter). No caso típico, os serviços são compostos de 100 a 1.000 processos. Chamamos esses processos de *tarefas de backend* (ou apenas *backends*). Outras tarefas, conhecidas como *tarefas clientes*, mantêm conexões com as tarefas de backend. Para cada consulta de entrada, uma tarefa cliente deve decidir qual tarefa de backend deve tratar essa consulta. Os clientes se comunicam com os backends usando um protocolo implementado sobre uma combinação de TCP

e UDP.

Devemos observar que os datacenters do Google hospedam um conjunto extremamente diversificado de serviços que implementam diferentes combinações das políticas discutidas neste capítulo. Nossa exemplo de trabalho, conforme acabamos de descrevê-lo, não se enquadra em nenhum serviço diretamente. É um cenário genérico, que nos permite discutir as várias técnicas que achamos serem úteis para diversos serviços. Algumas dessas técnicas podem ser mais (ou menos) aplicáveis a casos de uso específicos, porém elas foram projetadas e implementadas por vários engenheiros do Google no decorrer de muitos anos.

Essas técnicas são aplicadas em muitas partes de nossa pilha. Por exemplo, a maior parte das requisições HTTP externas alcança o GFE (Google Frontend, ou Frontend do Google), que é o nosso sistema de proxying HTTP reverso. O GFE utiliza esses algoritmos, juntamente com aqueles descritos no Capítulo 19, para encaminhar os payloads das requisições e os metadados aos processos individuais que executam as aplicações capazes de processar essas informações. Isso é baseado em uma configuração que mapeia diversos padrões de URL a aplicações individuais sob o controle de diferentes equipes. Para gerar os payloads das respostas (devolvidas ao GFE, para que sejam enviadas de volta aos navegadores), essas aplicações, por sua vez, muitas vezes usam esses mesmos algoritmos para se comunicar com a infraestrutura ou com serviços complementares dos quais elas dependem. Às vezes, a pilha de dependências pode se tornar relativamente profunda: uma única requisição HTTP de entrada pode disparar uma longa cadeia transitiva de requisições dependentes para vários sistemas, possivelmente com muitas saídas em vários pontos.

O caso ideal

Em um caso ideal, a carga de um dado serviço estará perfeitamente distribuída entre todas as suas tarefas de backend e, em qualquer instante no tempo, as tarefas mais e menos carregadas consumirão exatamente a mesma quantidade de CPU.

Podemos enviar tráfego a um datacenter somente até o ponto em que a tarefa

mais carregada alcance 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 distribuição de carga entre datacenters deve evitar o envio de qualquer tráfego adicional ao datacenter, pois fazer isso provocaria o risco de algumas tarefas ficarem sobrecarregadas.

Como mostra o gráfico à esquerda na Figura 20.2, um volume significativo de capacidade está sendo desperdiçado: isso corresponde à capacidade ociosa de todas as tarefas, exceto da tarefa mais carregada.

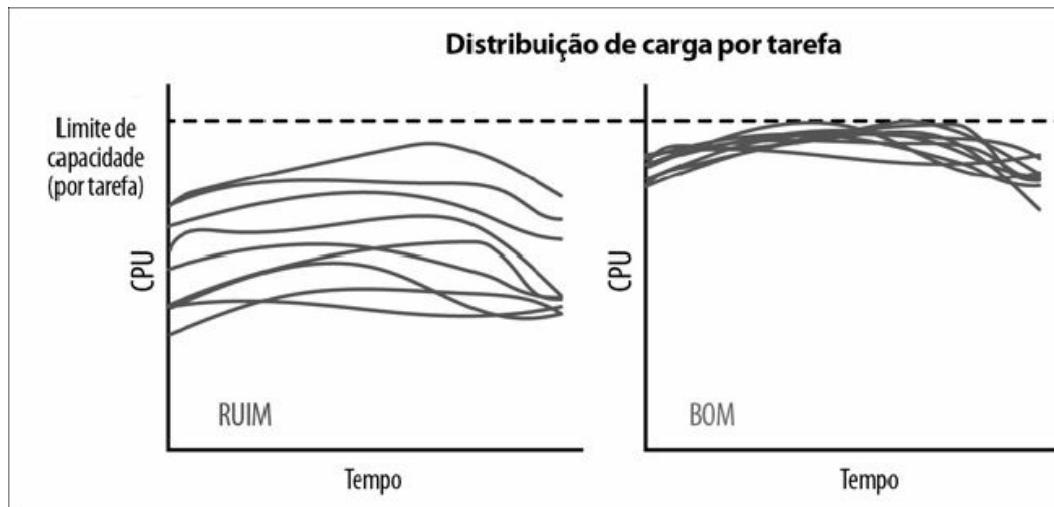


Figura 20.1 – Dois cenários de distribuição de carga por tarefa no tempo.

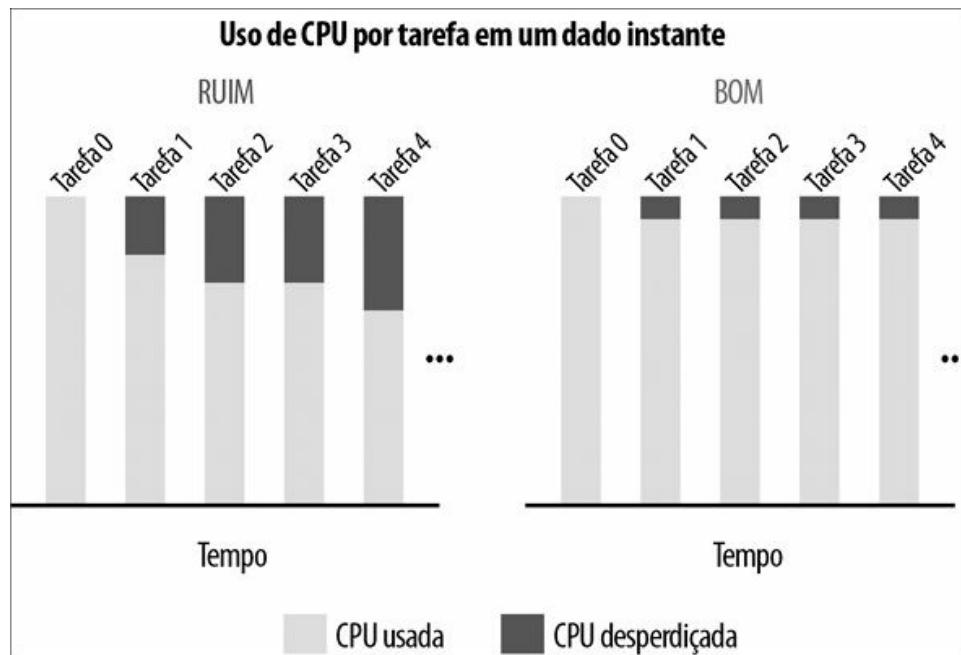


Figura 20.2 – Histograma de CPU usada e desperdiçada em dois cenários.

De modo mais formal, suponha que $CPU[i]$ seja a taxa de CPU consumida pela tarefa i em um dado instante no tempo, e que a tarefa 0 seja a tarefa mais carregada. Então, no caso de uma distribuição ampla, desperdiçamos a soma das diferenças entre o uso de CPU de qualquer tarefa e $CPU[0]$, ou seja, a soma de $(CPU[0] - CPU[i])$ para todas as tarefas i será desperdiçada. Nesse caso, “desperdiçada” significa reservada, porém não usada.

Esse exemplo mostra como práticas de distribuição de carga ruins no interior de um datacenter limitam artificialmente a disponibilidade de recursos: você pode reservar 1.000 CPUs para o seu serviço em um determinado datacenter, mas ser incapaz de realmente usar mais do que, digamos, 700 CPUs.

Identificando tarefas ruins: controle de fluxo e estado de incapacidade

Antes de podermos decidir qual tarefa de backend deve receber uma requisição de cliente, devemos identificar – e evitar – tarefas não saudáveis em nosso pool de backends.

Uma abordagem simples para tarefas não saudáveis: controle de fluxo

Suponha que nossas tarefas clientes monitorem o número de requisições ativas que enviaram a cada conexão com uma tarefa de backend. Quando esse contador de requisições ativas alcançar um limite configurado, o cliente tratará o backend como não saudável e não lhe enviará novas requisições. Para a maioria dos backends, 100 é um limite razoável; no caso médio, as requisições tendem a terminar de modo suficientemente rápido, e é muito raro que o número de requisições ativas de um dado cliente alcance esse limite em condições normais de operação. Essa forma (muito básica!) de controle de fluxo também funciona como um modo simplista de fazer distribuição de carga: se uma dada tarefa de backend ficar sobrecarregada e as requisições começarem a acumular, os clientes evitarão esse backend e a carga de trabalho será distribuída organicamente entre as demais tarefas de backend.

Infelizmente, essa abordagem bem simplista protege as tarefas de backend somente contra formas bem extremas de sobrecarga, e é muito fácil que os backends fiquem sobrecarregados muito antes de esse limite ser alcançado. O inverso também é verdadeiro: em alguns casos, os clientes poderão alcançar esse limite quando seus backends ainda tiverem muitos recursos disponíveis. Por exemplo, alguns backends podem ter requisições de duração bem longa que impeçam respostas rápidas. Vimos casos em que esse limite default saiu pela culatra, fazendo todas as tarefas de backend se tornarem inacessíveis, com requisições bloqueadas nos clientes até sofrerem timeout e falhar. Elevar o limite de requisições ativas pode evitar essa situação, mas não soluciona o problema subjacente de saber se uma tarefa está realmente não saudável ou se está apenas demorando para responder.

Uma abordagem robusta para tarefas não saudáveis: estado de incapacidade

Do ponto de vista do cliente, uma dada tarefa de backend pode estar em qualquer um dos seguintes estados:

Saudável

A tarefa de backend inicializou corretamente e está processamento requisições.

Recusando conexões

A tarefa de backend não está respondendo. Isso pode ocorrer porque a tarefa está iniciando ou finalizando, ou porque o backend está em um estado anormal (embora seja raro que um backend pare de ouvir sua porta se não estiver em processo de finalização).

Estado de incapacidade (lame duck)

A tarefa de backend está ouvindo sua porta e é capaz de servir, mas está explicitamente pedindo aos clientes que parem de enviar requisições.

Quando uma tarefa entra no estado de incapacidade, ela faz broadcast desse fato a todos os clientes ativos. Mas e quanto aos clientes inativos? Com a

implementação de RPC do Google, clientes inativos (isto é, clientes sem conexões TCP ativas) continuam enviando verificações de sanidade periódicas via UDP. O resultado é que essa informação de incapacidade é rapidamente propagada para todos os clientes – geralmente em 1 ou 2 RTT – independentemente de seu estado no momento.

A principal vantagem de permitir que uma tarefa exista em um estado de aparente incapacidade operacional é que isso simplifica um processo limpo de finalização, evitando servir erros a todas as requisições infelizes que, por acaso, estavam ativas nas tarefas de backend em finalização. Finalizar uma tarefa de backend que tenha requisições ativas sem servir nenhum erro facilita a implantação de novas versões de código, atividades de manutenção ou falhas de máquinas que possam exigir a reinicialização de todas as tarefas relacionadas. Uma finalização desse tipo seguiria os seguintes passos gerais:

1. O escalonador de jobs envia um sinal SIGTERM para a tarefa de backend.
2. A tarefa de backend entra em estado de incapacidade e pede que seus clientes enviem novas requisições a outras tarefas de backend. Isso é feito por meio de uma chamada de API na implementação de RPC que é explicitamente chamada no handler de SIGTERM.
3. Qualquer requisição em andamento iniciada antes de a tarefa de backend ter entrado no estado de incapacidade (ou após ela ter entrado nesse estado, mas antes de um cliente perceber) é executada normalmente.
4. À medida que as respostas fluem de volta aos clientes, o número de requisições ativas para o backend diminui gradualmente até atingir zero.
5. Após um intervalo configurado, a tarefa de backend sai de forma limpa ou o escalonador de jobs a encerra. O intervalo deve ser definido com um valor alto o suficiente para que todas as requisições típicas tenham tempo de terminar. Esse valor depende do serviço, mas uma boa regra geral é entre 10 e 150 segundos, conforme a complexidade do cliente.

Essa estratégia também permite que um cliente estabeleça conexões com as tarefas de backend enquanto elas realizam procedimentos de inicialização possivelmente de longa duração (não estando, portanto, prontas para começar

a servir). As tarefas de backend, por outro lado, poderiam começar a ouvir conexões somente quando estivessem prontas para servir, mas fazer isso atrasaria a negociação das conexões de forma desnecessária. Assim que a tarefa de backend estiver pronta para começar a servir, ela sinaliza isso explicitamente aos clientes.

Limitando o pool de conexões com a criação de subconjuntos

Além do gerenciamento da sanidade, outra consideração para a distribuição de carga é a *criação de subconjuntos*: limitar o pool de tarefas de backend em potencial com o qual uma tarefa cliente interage.

Cada cliente em nosso sistema de RPC mantém um pool de conexões de longa duração com seus backends, usado para enviar novas requisições. Essas conexões geralmente são estabelecidas com antecedência, quando o cliente está iniciando, e em geral permanecem abertas, com requisições fluindo por elas, até que o cliente seja encerrado. Um modelo alternativo seria estabelecer e desfazer uma conexão para cada requisição, mas esse modelo tem custos significativos de recursos e latência. No caso extremo de uma conexão que permaneça ociosa durante um longo período de tempo, nossa implementação de RPC tem uma otimização que alterna a conexão para um modo “inativo” de baixo custo, em que, por exemplo, a frequência de verificações de sanidade é reduzida e a conexão TCP subjacente é substituída em favor de UDP.

Toda conexão exige um pouco de memória e de CPU (por causa das verificações periódicas de sanidade) nas duas extremidades. Embora, na teoria, esse overhead seja baixo, ele pode se tornar rapidamente significativo quando está presente em várias máquinas. A criação de subconjuntos evita a situação em que um único cliente se conecta a um número muito grande de tarefas de backend ou uma única tarefa de backend recebe conexões de um número bem grande de tarefas clientes. Nos dois casos, você poderá desperdiçar uma quantidade enorme de recursos em troca de um ganho bem pequeno.

Escolhendo o subconjunto correto

Escolher o subconjunto correto se reduz a escolher a quantas tarefas de backend cada cliente se conecta – o tamanho do subconjunto – e o algoritmo de seleção. Geralmente, utilizamos um tamanho de subconjunto de 20 a 100 tarefas de backend, mas o tamanho “correto” do subconjunto em um sistema depende bastante do comportamento típico de seu serviço. Por exemplo, talvez você queira usar um tamanho maior de subconjunto se:

- O número de clientes for significativamente menor que o número de backends. Nesse caso, você vai querer que o número de backends por cliente seja grande o suficiente para que você não acabe com tarefas de backend que jamais recebam qualquer tráfego.
- Há um desequilíbrio frequente de carga nos jobs dos clientes (isto é, uma tarefa cliente envia mais requisições do que outras). Esse cenário é típico em situações em que os clientes ocasionalmente enviam bursts (rajadas) de requisições. Nesse caso, os próprios clientes recebem requisições de outros clientes que, ocasionalmente, geram muitas saídas (por exemplo, “leia todas as informações de todos os seguidores de um dado usuário”). Como um burst de requisições estará concentrado no subconjunto atribuído ao cliente, você precisará de um tamanho maior de subconjunto para garantir que a carga seja uniformemente distribuída no conjunto maior de tarefas de backend disponíveis.

Depois que o tamanho do subconjunto estiver determinado, precisamos de um algoritmo para definir o subconjunto de tarefas de backend que cada tarefa cliente usará. Isso pode parecer simples, mas se torna rapidamente complexo quando trabalhamos com sistemas de larga escala, em que um provisionamento eficiente é fundamental e é certo que reinicializações de sistema ocorrerão.

O algoritmo de seleção dos clientes deve atribuir backends de modo uniforme a fim de otimizar o provisionamento de recursos. Por exemplo, se o subconjunto sobrecarregar um backend em 10%, o conjunto todo de backends precisará ter um aumento de provisionamento de 10%. O algoritmo também deve tratar reinicializações e falhas de forma elegante e robusta, continuando a carregar os backends da maneira mais uniforme possível, ao mesmo tempo

que minimiza a bagunça. Nesse caso, a “bagunça” está relacionada à seleção do backend substituto. Por exemplo, quando uma tarefa de backend se torna indisponível, seus clientes talvez precisem escolher temporariamente um backend substituto. Quando um backend substituto é selecionado, os clientes devem criar novas conexões TCP (e, provavelmente, realizar uma negociação no nível de aplicação), o que cria um overhead adicional. De modo semelhante, quando uma tarefa cliente reinicia, ela precisa reestabelecer as conexões com todos os seus backends.

O algoritmo também deve tratar redimensionamentos no número de clientes e/ou no número de backends, com o mínimo de bagunça nas conexões e sem conhecer esses números com antecedência. Essa funcionalidade é particularmente importante (e complicada) quando o conjunto todo de tarefas clientes ou de backends é reiniciado, um de cada vez (por exemplo, para implantação de uma nova versão). À medida que os backends são atualizados, queremos que os clientes continuem servindo, de modo transparente, com o mínimo possível de bagunça nas conexões.

Um algoritmo de seleção de subconjunto: criação aleatória de subconjuntos

Uma implementação ingênuia de um algoritmo de seleção de subconjunto pode fazer cada cliente embaralhar aleatoriamente a lista de backends uma vez e preencher seu subconjunto selecionando backends resolvíveis/saudáveis da lista. Embaralhar uma vez e então escolher os backends do início da lista trata reinicializações e falhas de forma robusta (por exemplo, relativamente com pouca bagunça), pois os limita explicitamente de serem considerados. No entanto, percebemos que essa estratégia, na verdade, funciona de forma muito precária na maior parte dos cenários práticos, pois distribui a carga de maneira muito desigual.

Durante os primeiros trabalhos em distribuição de carga, implementamos uma criação de subconjuntos aleatória e calculamos a carga esperada em vários casos. Como exemplo, considere:

- 300 clientes
- 300 backends

- Um tamanho de subconjunto igual a 30% (cada cliente se conecta a 90 backends)

Como mostra a Figura 20.3, o backend menos carregado tem apenas 63% da carga média (57 conexões, em que 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 iríamos querer na prática. A distribuição de carga calculada muda sempre que executamos a simulação, enquanto o padrão geral se mantém.

Infelizmente, tamanhos menores de subconjuntos resultam em desequilíbrios ainda maiores. Por exemplo, a Figura 20.4 representa o resultado se o tamanho do subconjunto fosse reduzido para 10% (30 backends por cliente). Nesse caso, o backend menos carregado recebe 50% da carga média (15 conexões) e o mais carregado recebe 150% (45 conexões).

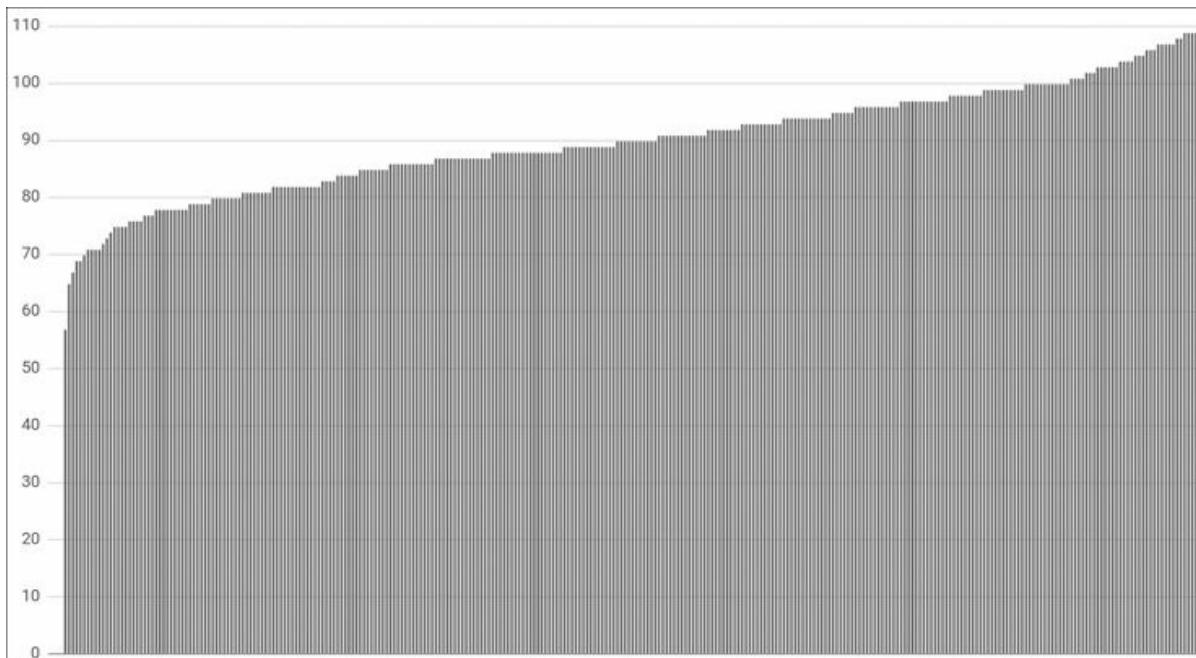


Figura 20.3 – Distribuição das conexões com 300 clientes, 300 backends e um tamanho de subconjunto de 30%.

Concluímos que, para que uma criação de subconjuntos aleatória distribua a carga de maneira relativamente uniforme entre todas as tarefas disponíveis, precisaríamos de tamanhos de subconjuntos tão grandes quanto 75%. Um subconjunto desse tamanho é simplesmente impraticável; a variação no número de clientes se conectando a uma tarefa é simplesmente grande demais

para considerarmos que uma criação de subconjuntos aleatória seja uma boa política de seleção de subconjuntos em escala.

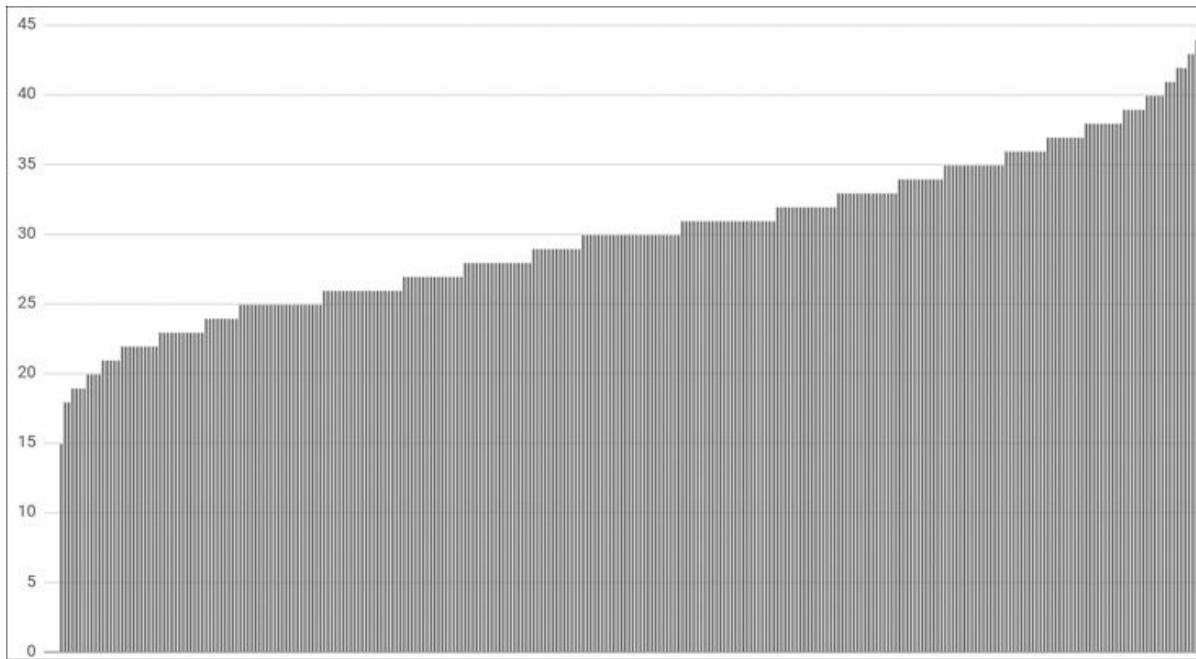


Figura 20.4 – Distribuição das conexões com 300 clientes, 300 backends e um tamanho de subconjunto de 10%.

Um algoritmo de seleção de subconjuntos: criação determinística de subconjuntos

A solução do Google para as limitações da criação aleatória de subconjuntos é a criação *determinística*. O código a seguir implementa esse algoritmo, descrito em detalhes na sequência:

```
def Subset(backends, client_id, subset_size):
    subset_count = len(backends) / subset_size

    # Agrupa clientes em rodadas; cada rodada utiliza a mesma lista embaralhada:
    round = 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 *clientes* em “rodadas” (rounds), em que round i é constituído de subset_count tarefas clientes consecutivas, começando na tarefa $\text{subset_count} \times i$, e subset_count é o número de subconjuntos (isto é, o número de tarefas de backend dividido pelo tamanho do subconjunto desejado). Em cada rodada, um backend é atribuído exatamente a um cliente (exceto, possivelmente, na 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 ($\text{subset_count} = 12/3$). Se tivéssemos 10 clientes, o algoritmo anterior produziria 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 principal a ser observado é que cada rodada atribui apenas um backend da lista toda a um cliente (exceto na última, em que ficamos sem clientes). Nesse exemplo, todo backend é atribuído a exatamente dois ou três clientes.

A lista deve ser embaralhada; caso contrário, os clientes serão atribuídos a um grupo de tarefas de backend consecutivas, que podem ficar todas temporariamente indisponíveis (por exemplo, porque o job de backend está sendo gradualmente atualizado em sequência, da primeira à última tarefa). Rodadas diferentes utilizam uma semente (seed) diferente para embaralhar. Se não o fizerem, quando um backend falhar, a carga que ele estava recebendo será distribuída apenas entre os backends restantes *em seu subconjunto*. Se backends adicionais no subconjunto falharem, o efeito composto e a situação poderão piorar rapidamente de modo significativo: se N backends em um subconjunto caírem, sua carga correspondente será distribuída pelos $(\text{subset_size} - N)$ backends restantes. Uma abordagem muito melhor é distribuir essa carga por todos os backends restantes usando um método diferente para embaralhar em cada rodada.

Quando usamos um método diferente para embaralhar em cada rodada, os

clientes na mesma rodada começarão com a mesma lista embaralhada, porém os clientes em rodadas diferentes terão listas embaralhadas diferentes. A partir daí, o algoritmo cria *definições* de subconjuntos com base na lista embaralhada de backends e no tamanho de subconjunto desejado. Por exemplo:

- $\text{Subset}[0] = \text{shuffled_backends}[0]$ a $\text{shuffled_backends}[2]$
- $\text{Subset}[1] = \text{shuffled_backends}[3]$ a $\text{shuffled_backends}[5]$
- $\text{Subset}[2] = \text{shuffled_backends}[6]$ a $\text{shuffled_backends}[8]$
- $\text{Subset}[3] = \text{shuffled_backends}[9]$ a $\text{shuffled_backends}[11]$

em que `shuffled_backend` é a lista embaralhada que cada cliente cria. Para atribuir um subconjunto a uma tarefa cliente, simplesmente tomamos o subconjunto que corresponda à sua posição em sua rodada (por exemplo, $(i \% 4)$ para `client[i]` com quatro subconjuntos):

- `client[0], client[4], client[8]` usarão `subset[0]`
- `client[1], client[5], client[9]` usarão `subset[1]`
- `client[2], client[6], client[10]` usarão `subset[2]`
- `client[3], client[7], client[11]` usarão `subset[3]`

Como os clientes em diferentes rodadas usarão um valor diferente para `shuffled_backends` (e, desse modo, para `subset`) e os clientes nas mesmas rodadas utilizam subconjuntos diferentes, a carga de conexões será distribuída uniformemente. Nos casos em que o número total de backends não é divisível pelo tamanho do subconjunto desejado, permitimos que alguns subconjuntos sejam um pouco maiores do que outros, mas, na maioria das vezes, o número de clientes atribuído a um backend diferirá no máximo em 1.

Como mostra a Figura 20.5, a distribuição do exemplo anterior com 300 clientes, cada um se conectando a 10 de 300 backends, produz resultados muito bons: cada backend recebe exatamente o mesmo número de conexões.

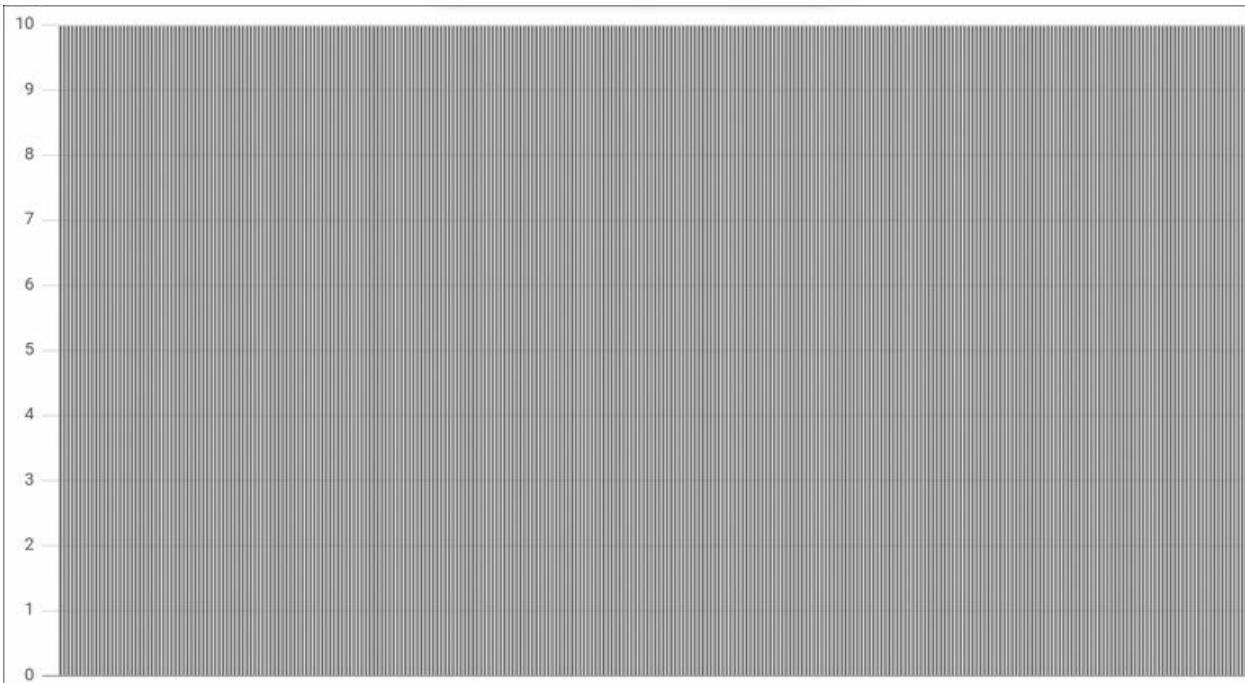


Figura 20.5 – Distribuição das conexões com 300 clientes e criação determinística de subconjuntos com 10 em 300 backends.

Políticas de distribuição de carga

Agora que definimos o trabalho de base para o modo como uma dada tarefa cliente mantém um conjunto de conexões que sabemos serem saudáveis, vamos analisar as *políticas de distribuição de carga*. São os mecanismos utilizados pelas tarefas clientes para selecionar qual tarefa de backend em seu subconjunto receberá uma requisição de cliente. Muitas das complexidades nas políticas de distribuição de carga nascem da natureza distribuída do processo de tomada de decisão, em que os clientes precisam decidir em tempo real (e com informações apenas parciais e/ou ultrapassadas dos estados dos backends) qual backend deve ser usado para cada requisição.

As políticas de distribuição de carga podem ser muito simples e não levar em consideração nenhuma informação sobre o estado dos backends (por exemplo, *Round Robin*), ou podem atuar com mais informações sobre os backends (por exemplo, *Least-Loaded Round Robin* ou *Weighted Round Robin*).

Round Robin Simples

Segundo uma abordagem bem simples para a distribuição de carga, cada cliente envia requisições pelo método round-robin a cada tarefa de backend em seu subconjunto, com a qual ele pode se conectar com sucesso e que não esteja em um estado de incapacidade (lame duck). Durante muitos anos, essa foi a nossa abordagem mais comum, e continua sendo usada por muitos serviços.

Infelizmente, embora o Round Robin tenha a vantagem de ser bem simples e apresente um desempenho significativamente melhor do que apenas selecionar as tarefas de backend de forma aleatória, os resultados dessa política podem ser muito ruins. Embora os números propriamente ditos dependam de vários fatores (por exemplo, um custo de consulta variado e diversidade de máquinas), percebemos que o Round Robin pode resultar em uma distribuição com diferenças de até 2x no consumo de CPU, da tarefa menos carregada para a mais carregada. Uma distribuição como essa causa imensos desperdícios e ocorre por uma série de motivos, incluindo:

- Criação de subconjuntos pequenos
- Custos variados de consultas
- Diversidade de máquinas
- Fatores de desempenho imprevisíveis

Criação de subconjuntos pequenos

Um dos motivos mais simples pelos quais o Round Robin distribui a carga de forma precária é que todos os seus clientes podem não fazer requisições na mesma velocidade. Taxas diferentes de requisições entre clientes são mais prováveis, em especial quando processos extremamente diferentes compartilham os mesmos backends. Nesse caso, em particular se você estiver usando tamanhos de subconjuntos relativamente menores, os backends nos subconjuntos dos clientes que geram a maior parte do tráfego terão uma tendência natural a ficarem mais carregados.

Custos variados de consultas

Muitos serviços tratam requisições que exigem quantidades extremamente

diferentes de recursos para processamento. Na prática, percebemos que a semântica de muitos serviços do Google é tal que as requisições mais custosas consomem 1000x mais CPU (ou mais) que as requisições de custo mais baixo. A distribuição de carga com Round Robin é mais difícil ainda quando o custo da consulta não pode ser previsto com antecedência. Por exemplo, uma consulta como “devolva todos os emails recebidos pelo usuário XYZ no último dia” poderia ter um custo muito baixo (se o usuário recebeu poucos emails no curso do dia) ou ser extremamente custosa.

A distribuição de carga em um sistema com grandes discrepâncias no potencial custo da consulta é muito problemática. Talvez seja necessário ajustar as interfaces do serviço para limitar funcionalmente a quantidade de trabalho feita por requisição. Por exemplo, no caso da consulta de emails descrita anteriormente, você poderia introduzir uma interface de paginação e alterar a semântica da requisição para “devolva os cem emails (ou menos) mais recentes recebidos pelo usuário XYZ no último dia”. Infelizmente, muitas vezes é difícil introduzir alterações de semântica desse tipo. Isso não só exige mudanças em todo o código do cliente como também implica considerações adicionais de consistência. Por exemplo, o usuário pode estar recebendo novos emails ou apagando-os à medida que o cliente busca os emails página a página. Para esse caso de uso, um cliente que itere ingenuamente pelos resultados e concatene as respostas (em vez de fazer a paginação com base em uma visualização fixa dos dados) provavelmente gerará uma visualização inconsistente, repetindo algumas mensagens e/ou ignorando outras.

Para manter as interfaces (e suas implementações) simples, os serviços em geral são definidos para permitir que as requisições mais custosas consumam 100, 1.000 ou até mesmo 10.000 vezes mais recursos que as requisições de menor custo. No entanto, requisitos de recursos variados por requisição naturalmente significam que algumas tarefas de backend não terão sorte e, ocasionalmente, receberão requisições mais custosas que outras. A extensão com que essa situação afeta a distribuição de carga depende do quão custosas são as requisições mais custosas. Por exemplo, para um de nossos backends Java, as consultas consomem aproximadamente 15 ms de CPU em média, mas algumas consultas podem facilmente exigir até 10 segundos. Cada tarefa

nesse backend reserva vários cores (núcleos) de CPU, o que reduz a latência ao permitir que alguns processamentos ocorram em paralelo. Porém, apesar desses cores reservados, quando um backend recebe uma dessas consultas custosas, sua carga aumenta significativamente durante alguns segundos. Uma tarefa com um comportamento ruim pode ficar sem memória ou até mesmo parar totalmente de responder (por exemplo, devido a thrashing de memória¹), mas mesmo no caso normal (isto é, quando o backend tem recursos suficientes e sua carga se normaliza depois que a query custosa termina) a latência de outras requisições sofre por causa da concorrência de recursos com a requisição custosa.

Diversidade de máquinas

Outro desafio ao Round Robin Simples é o fato de nem todas as máquinas no mesmo datacenter serem necessariamente iguais. Um determinado datacenter pode ter máquinas com CPUs cujos desempenhos variam e, desse modo, a mesma requisição pode representar uma quantidade de trabalho significativamente diferente para máquinas distintas.

Lidar com a diversidade das máquinas – *sem exigir uma homogeneidade rigorosa* – foi um desafio por muitos anos no Google. Teoricamente, a solução para trabalhar com capacidade heterogênea de recursos em uma frota de máquinas é simples: escala as reservas de CPU de acordo com o tipo de processador/máquina. Entretanto, na prática, implantar essa solução exigiu um esforço significativo, pois foi necessário que nosso escalonador de jobs levasse em conta equivalências de recursos baseadas no desempenho médio das máquinas em uma amostragem dos serviços. Por exemplo, duas unidades de CPU na máquina X (uma máquina “lenta”) são equivalentes a 0,8 unidade de CPU na máquina Y (uma máquina “rápida”). Com essa informação, o escalonador de jobs é então solicitado a ajustar reservas de CPU para um processo com base no fator de equivalência e no tipo de máquina em que o processo foi escalonado. Em uma tentativa de reduzir essa complexidade, criamos uma unidade virtual para taxa de CPU chamada “GCU” (Google Compute Units, ou Unidades de Processamento do Google). As GCUs se tornaram padrões para modelar taxas de CPU, e foram usadas para manter um mapeamento entre cada arquitetura de CPU em nossos datacenters à GCU

correspondente baseada em seu desempenho.

Fatores de desempenho imprevisíveis

Talvez o maior fator complicador para o Round Robin Simples seja o fato de as máquinas – ou, de modo mais exato, o desempenho das tarefas de backend – poderem diferir enormemente devido a vários aspectos *imprevisíveis* que não podem ser considerados de forma estática.

Dois dos vários fatores imprevisíveis que contribuem para o desempenho incluem:

Vizinhos antagônicos

Outros processos (muitas vezes, totalmente não relacionados e executados por equipes diferentes) podem causar um impacto significativo no desempenho de seus processos. Já vimos diferenças de desempenho dessa natureza de até 20%. Em sua maior parte, essa diferença tem origem na competição por recursos compartilhados, como espaço em caches de memória ou largura de banda, de formas que talvez não sejam diretamente óbvias. Por exemplo, se a latência de requisições de saída de uma tarefa de backend aumentar (por causa de competição por recursos de rede com um vizinho antagônico), o número de requisições ativas também aumentará, o que poderá acionar mais coleta de lixo (garbage collection).

Reinicializações de tarefas

Quando uma tarefa é reiniciada, com frequência, exige mais recursos de forma significativa durante alguns minutos. Apenas para citar um exemplo, já vimos essa condição afetar plataformas como Java que otimizam códigos dinamicamente, mais do que outras. Em resposta, fizemos acréscimos na lógica do código de alguns servidores – nós os mantemos em estado de incapacidade (*lame duck*) e os pré-aquecemos (disparando essas otimizações) por um período de tempo depois que inicializam, até que seu desempenho seja o esperado. O efeito das reinicializações de tarefas pode se transformar em um grande problema se considerarmos que atualizamos muitos servidores (por exemplo, com implantação de novas versões, que exigem a reinicialização dessas tarefas) todos os dias.

Se sua política de distribuição de carga não puder se adaptar a limitações de desempenho imprevistas, você acabará inherentemente com uma distribuição de carga abaixo da ideal ao trabalhar em escala.

Least-Loaded Round Robin

Uma abordagem alternativa ao Round Robin Simples é fazer com que cada tarefa cliente mantenha o controle do número de requisições ativas que tem para cada tarefa de backend em seu subconjunto e utilize o Round Robin *entre o conjunto de tarefas com um número mínimo de requisições ativas*.

Por exemplo, suponha que um cliente utilize um subconjunto de tarefas de backend de $t0$ a $t9$ e, no momento, tenha o seguinte número de requisições ativas para cada backend:

	$t0$	$t1$	$t2$	$t3$	$t4$	$t5$	$t6$	$t7$	$t8$	$t9$
	2	1	0	0	1	0	2	0	0	1

Para uma nova requisição, o cliente filtraria a lista de tarefas de backend em potencial para ficar apenas com as tarefas com o menor número de conexões ($t2$, $t3$, $t5$, $t7$ e $t8$), e escolheria um backend dessa lista. Vamos supor que o cliente escolha $t2$. A tabela de estados de conexões do cliente agora passa a ter o seguinte aspecto:

	$t0$	$t1$	$t2$	$t3$	$t4$	$t5$	$t6$	$t7$	$t8$	$t9$
	2	1	1	0	1	0	2	0	0	1

Supondo que nenhuma das requisições atuais tenha sido concluída, na próxima requisição o pool de backends candidatos conterá $t3$, $t5$, $t7$ e $t8$.

Vamos avançar rapidamente até termos gerado quatro novas requisições. Ainda supondo que nenhuma requisição tenha terminado nesse meio-tempo, a tabela de estados de conexões terá o aspecto a seguir:

	$t0$	$t1$	$t2$	$t3$	$t4$	$t5$	$t6$	$t7$	$t8$	$t9$
	2	1	1	1	1	1	2	1	1	1

A essa altura, o conjunto de backends candidatos será formado por todas as tarefas, exceto $t0$ e $t6$. No entanto, se a requisição para a tarefa $t4$ terminar,

seu estado atual se tornará “0 requisições ativas” e uma nova requisição será atribuída a $t4$.

Essa implementação, na realidade, utiliza Round Robin, mas ele é aplicado no conjunto de tarefas com o mínimo de requisições ativas. Sem um filtro desse tipo, a política poderia não ser capaz de distribuir as requisições tão bem a ponto de evitar uma situação em que parte das tarefas de backend disponíveis não seja utilizada. A ideia por trás da política de least-loaded (menos carregados) é que as tarefas carregadas tenderão a ter latências mais altas do que aquelas com capacidade remanescente, e essa estratégia afastará naturalmente a carga dessas tarefas.

Apesar de tudo que foi dito, tomamos ciência (da maneira difícil!) de uma armadilha muito perigosa na abordagem com Least-Loaded Round Robin: se uma tarefa estiver seriamente não saudável, poderá começar a servir 100% de erros. Conforme a natureza desses erros, eles poderão ter uma latência bem baixa; com frequência, é consideravelmente mais rápido devolver apenas um erro “Não estou saudável!” do que realmente processar uma requisição. Como resultado, os clientes podem começar a enviar um volume bem grande de tráfego para a tarefa não saudável, achando, erroneamente, que ela está disponível, em oposição a estar enviando falhas rapidamente! Dizemos que a tarefa não saudável agora está fazendo o tráfego *escoar pelo ralo* (sinkholing). Felizmente, esse problema pode ser resolvido de forma relativamente simples ao modificar a política de modo a contar erros recentes como se fossem requisições ativas. Dessa maneira, se uma tarefa de backend deixar de ser saudável, a política de distribuição de carga começará a desviar a carga dela do mesmo modo que o faria se a tarefa estivesse sobrecarregada.

O Least-Loaded Round Robin tem duas limitações importantes:

O contador de requisições ativas pode não ser uma aproximação muito boa para a capacidade de um dado backend

Muitas requisições gastam uma parte significativa de sua vida simplesmente esperando uma resposta da rede (isto é, esperando respostas às requisições iniciadas a outros backends) e bem pouco tempo em processamento propriamente dito. Por exemplo, uma tarefa de backend pode ser capaz de processar o dobro de requisições de outra (por exemplo,

porque executa em uma máquina com uma CPU duas vezes mais rápida que as demais), porém a latência de suas requisições ainda pode ser aproximadamente igual à latência das requisições na outra tarefa (porque as requisições gastam a maior parte de sua vida simplesmente esperando a rede responder). Nesse caso, como ficar bloqueado em E/S geralmente não consome CPU nem banda, além de consumir bem pouca RAM, ainda vamos querer enviar o dobro de requisições ao backend mais rápido. Entretanto, o Least-Loaded Round Robin considerará que as cargas nas duas tarefas de backend são iguais.

O contador de requisições ativas em cada cliente não inclui requisições de outros clientes aos mesmos backends

Isso significa que cada tarefa cliente tem apenas uma visão bem limitada do estado de suas tarefas de backend: a visão de suas próprias requisições.

Na prática, descobrimos que serviços de grande porte que usam o Least-Loaded Round Robin verão suas tarefas de backend mais carregadas usarem o dobro de CPU que as tarefas menos carregadas, resultando em um desempenho tão ruim quanto o do Round Robin.

Weighted Round Robin

O Weighted Round Robin é uma política de distribuição de carga importante, que oferece melhorias em relação ao Round Robin Simples e ao Least-Loaded Round Robin ao incorporar informações fornecidas pelos backends no processo de decisão.

O Weighted Round Robin, em princípio, é bem simples: cada tarefa cliente mantém uma pontuação para a “capacidade” de cada backend em seu subconjunto. As requisições são distribuídas em Round-Robin, porém os clientes dão pesos às distribuições das requisições aos backends de forma proporcional. Em cada resposta (incluindo respostas a verificações de sanidade), os backends incluem as taxas de consultas e erros por segundo, além da utilização (geralmente, uso de CPU) observada no momento. Os clientes ajustam periodicamente as pontuações para a capacidade a fim de escolher as tarefas de backend de acordo com o número atual de requisições tratadas de forma bem-sucedida e o custo de utilização; requisições com falha

resultam em uma penalidade que afetará decisões futuras.

Na prática, o Weighted Round Robin tem funcionado muito bem e reduziu de forma significativa a diferença entre as tarefas mais utilizadas e as menos utilizadas. A Figura 20.6 mostra as taxas de CPU para um subconjunto aleatório de tarefas de backend, aproximadamente no momento em que seus clientes passaram do Least-Loaded para o Weighted Round Robin. A diferença entre as tarefas menos carregadas para as mais carregadas diminuiu drasticamente.

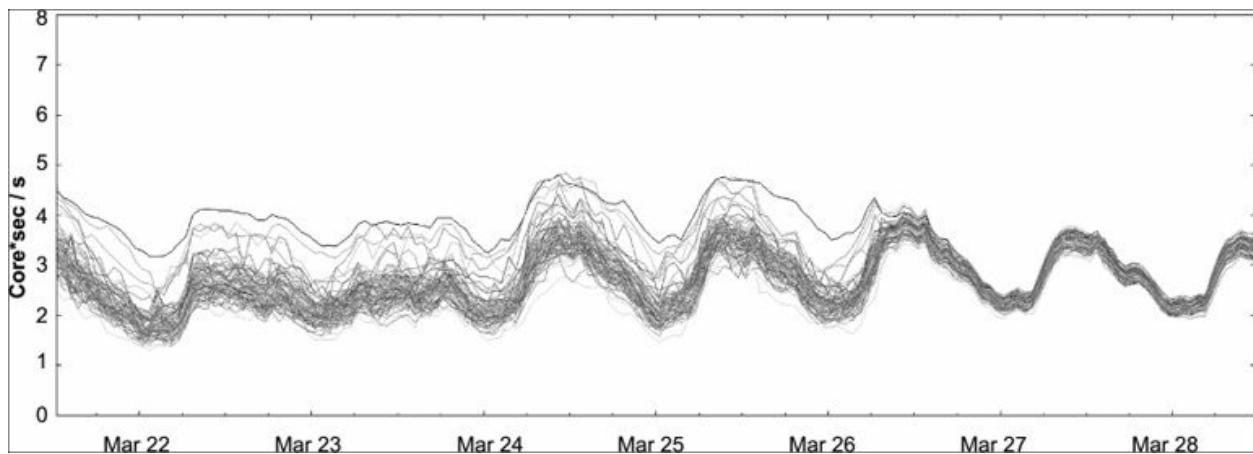


Figura 20.6 – Distribuição de CPU antes e depois de ativar o Weighted Round Robin.

¹ N.T.: O thrashing ocorre quando o subsistema de memória virtual de um computador está em constante estado de paginação, trocando rapidamente dados em memória por dados em disco, impedindo a maior parte do processamento no nível de aplicação. (Fonte: [https://en.wikipedia.org/wiki/Thrashing_\(computer_science\)](https://en.wikipedia.org/wiki/Thrashing_(computer_science)))

CAPÍTULO 21

Tratando sobrecarga

Escrito por Alejandro Forero Cuervo

Editado por Sarah Chavis

Evitar sobrecarga é uma meta das políticas de distribuição de carga. Porém, independentemente da eficiência de sua política de distribuição de carga, em *algum momento*, uma parte de seu sistema ficará sobrecarregada. Tratar condições de sobrecarga de modo elegante é fundamental para executar um sistema servidor confiável.

Uma opção para tratar a sobrecarga é servir respostas degradadas: respostas que não sejam tão exatas ou que contenham menos dados do que as respostas normais, mas são mais fáceis de processar. Por exemplo:

- Em vez de pesquisar um corpus completo a fim de oferecer os melhores resultados disponíveis a uma pesquisa, busque apenas em uma pequena porcentagem do conjunto de candidatos.
- Conte com uma cópia local dos resultados, que pode não estar totalmente atualizada, mas será menos custosa para usar do que acessar a área de armazenagem canônica.

Entretanto, em situação de sobrecarga extrema, o serviço poderá nem mesmo ser capaz de processar e servir respostas degradadas. A essa altura, talvez não haja nenhuma opção imediata a não ser servir erros. Uma maneira de atenuar esse cenário é distribuir o tráfego entre datacenters de modo que nenhum deles receba mais tráfego do que é capaz de processar. Por exemplo, se um datacenter executa 100 tarefas de backend e cada tarefa é capaz de processar até 500 requisições por segundo, o algoritmo de distribuição de carga não permitirá que mais de 50.000 consultas por segundo sejam enviadas a esse datacenter. No entanto, mesmo essa restrição pode se mostrar insuficiente

para evitar a sobrecarga quando você estiver operando em escala. No final das contas, é melhor implementar clientes e backends que tratem restrições de recursos de modo elegante: redirecione quando possível, sirva resultados degradados quando necessário e trate erros de recursos de forma transparente quando tudo o mais falhar.

As armadilhas das “consultas por segundo”

Diferentes consultas podem ter requisitos extremamente distintos para recursos. O custo de uma consulta pode variar de acordo com fatores arbitrários, como o código do cliente que a gera (para serviços que têm muitos clientes diferentes), ou até mesmo conforme o horário do dia (por exemplo, usuários domésticos *versus* usuários profissionais, ou tráfego interativo de usuários finais *versus* tráfego em lote).

Aprendemos essa lição do modo difícil: modelar a capacidade como “consultas por segundo” ou usar características estáticas das requisições que acreditamos serem uma forma de avaliar os recursos consumidos (por exemplo, “quantas chaves as requisições estão lendo”) com frequência resultam em uma métrica ruim. Mesmo que essas métricas tenham um desempenho adequado em algum ponto no tempo, as razões calculadas podem mudar. Às vezes, a mudança é gradual, porém, outras vezes, é drástica (por exemplo, uma nova versão do software fez com que, repentinamente, algumas funcionalidades de determinadas requisições exigissem menos recursos de forma significativa). Um alvo móvel resulta em uma métrica ruim para design e implementação de distribuição de carga.

Uma solução melhor consiste em mensurar a capacidade diretamente na forma de recursos disponíveis. Por exemplo, você pode ter um total de 500 cores (núcleos) de CPU e 1 TB de memória reservados a um dado serviço em um datacenter em particular. Naturalmente, usar esses números diretamente para modelar a capacidade de um datacenter funcionará muito melhor. Com frequência, falamos sobre o *custo* de uma requisição para nos referirmos a uma medida normalizada do tempo de CPU consumido (em arquiteturas diferentes de CPU, considerando as diferenças de desempenho).

Na maioria dos casos (embora, certamente, não em todos), percebemos que

simplesmente usar o consumo de CPU como sinal de provisionamento funciona bem pelos seguintes motivos:

- Em plataformas com coleta de lixo (garbage collection), a pressão por memória naturalmente se traduz em um aumento no consumo de CPU.
- Em outras plataformas, é possível provisionar os recursos restantes de modo que seja bem improvável que eles se esgotem antes da CPU.

Em casos em que um provisionamento exagerado de recursos que não a CPU seja custoso a ponto de ser proibitivo, levamos cada recurso do sistema em consideração de forma separada ao considerarmos o consumo dos recursos.

Limites por cliente

Um aspecto ao lidar com a sobrecarga diz respeito a decidir o que fazer em caso de uma sobrecarga *global*. Em um mundo perfeito, em que as equipes coordenam seus lançamentos cuidadosamente com os proprietários das dependências de seu backend, uma sobrecarga global jamais aconteceria e os serviços de backend sempre teriam capacidade suficiente para servir aos seus clientes. Infelizmente, não vivemos em um mundo perfeito. Em nosso mundo real, uma sobrecarga global ocorre com muita frequência (especialmente em serviços internos, que tendem a ter muitos clientes com várias equipes responsáveis pela sua operação).

Quando uma sobrecarga global *realmente* ocorre, é vital que o serviço entregue respostas com erro somente aos clientes com mau comportamento, ao passo que outros clientes não são afetados. Para conseguir esse resultado, os proprietários dos serviços configuram sua capacidade de acordo com o uso negociado com seus clientes e definem cotas por cliente conforme esses acordos.

Por exemplo, se um serviço de backend tiver 10.000 CPUs alocadas pelo mundo (em vários datacenters), seus limites por cliente podem ter um aspecto como este:

- O Gmail tem permissão para consumir até 4.000 segundos de CPU por segundo.
- O Calendar tem permissão para consumir até 4.000 segundos de CPU por

segundo.

- O Android tem permissão para consumir até 3.000 segundos de CPU por segundo.
- O Google+ tem permissão para consumir até 2.000 segundos de CPU por segundo.
- Todos os demais usuários têm permissão para consumir até 500 segundos de CPU por segundo.

Observe que esses números podem totalizar mais de 10.000 CPUs alocadas ao serviço de backend. O proprietário do serviço conta com o fato de que é improvável que *todos* os seus clientes atinjam seus limites de recursos simultaneamente.

Agregamos as informações de uso global de todas as tarefas de backend em tempo real e utilizamos esses dados para forçar os limites efetivos às tarefas de backend individuais. Uma observação mais detalhada do sistema que implementa essa lógica está fora do escopo desta discussão, mas escrevemos uma quantidade significativa de código para implementar isso em nossas tarefas de backend. Uma parte interessante do quebra-cabeça é calcular a quantidade de recursos em tempo real – especificamente de CPU – consumida por uma requisição. Esse cálculo é particularmente intrincado para os servidores que não implementam um modelo de thread-por-requisição, em que um pool de threads simplesmente executa diferentes partes de todas as requisições à medida que elas chegam, utilizando APIs não bloqueantes.

Throttling do lado cliente

Quando um cliente esgotar sua cota, uma tarefa de backend deve rejeitar rapidamente as requisições na expectativa de que devolver um erro de “cliente esgotou a cota” consumia significativamente menos recursos do que processar a requisição e servir uma resposta correta. No entanto, essa lógica não é válida para todos os serviços. Por exemplo, é quase igualmente custoso rejeitar uma requisição que exija uma busca simples em RAM (em que o overhead do tratamento do protocolo requisição/resposta é significativamente maior que o overhead de gerar a resposta) quanto aceitar e executar essa

requisição. Mesmo no caso em que rejeitar requisições economize recursos significativos, essas requisições *ainda* consomem alguns recursos. Se a quantidade de requisições rejeitadas for significativa, esses números aumentam rapidamente. Em casos como esse, o backend pode ficar sobrecarregado, mesmo que a maior parte de sua CPU seja consumida apenas rejeitando requisições!

Um throttling (estrangulamento) no lado cliente trata esse problema.¹ Quando um cliente detecta que uma porção significativa de suas requisições recentes foi rejeitada devido a erros de “cota esgotada”, ele começa a se autorregular e limitar a quantidade de tráfego de saída que gera. As requisições acima do limite falham localmente, sem sequer alcançar a rede.

Implementamos o throttling do lado cliente por meio de uma técnica que chamamos de *throttling adaptativo*. Especificamente, cada tarefa cliente mantém as seguintes informações dos últimos dois minutos de seu histórico:

requests

O número de requisições que a camada de aplicação tentou fazer (no cliente, acima do sistema de throttling adaptativo).

accepts

O número de requisições aceitas pelo backend.

Em condições normais, os dois valores são iguais. Quando o backend começa a rejeitar tráfego, o número de accepts passa a ser menor que o número de requests. Os clientes podem continuar a enviar requisições ao backend até requests ser K vezes maior que accepts. Depois que esse limite é alcançado, o cliente começa a se autorregular, e novas requisições são rejeitadas localmente (isto é, no cliente), com a probabilidade calculada pela Equação 21.1.

Equação 21.1 – Probabilidade de rejeição de requisições no cliente

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

À medida que o próprio cliente começa a rejeitar requisições, *requests* continuará excedendo *accepts*. Embora possa parecer contraintuitivo,

considerando que as requisições rejeitadas localmente não são realmente propagadas até o backend, esse é o comportamento preferível. Conforme a taxa com que a aplicação tenta fazer requisições ao cliente aumenta (em relação à taxa com que o backend as aceita), queremos elevar a probabilidade de descartar novas requisições.

Percebemos que o throttling adaptativo funciona bem na prática, resultando em taxas estáveis de requisições em geral. Mesmo em situações de bastante sobrecarga, os backends acabam rejeitando uma requisição para cada requisição que eles realmente processam. Uma grande vantagem dessa abordagem é que a decisão é tomada pela tarefa cliente, totalmente com base em informações locais, e uma implementação relativamente simples é usada: não há dependências adicionais nem penalidades em latência.

Para serviços em que o custo de processar uma requisição é muito próximo do custo de rejeitá-la, permitir que aproximadamente metade dos recursos de backend seja consumida pelas requisições rejeitadas pode ser inaceitável. Nesse caso, a solução é simples: modificar o multiplicador K de aceitação (por exemplo, 2) na probabilidade de rejeição de requisições no cliente (Equação 21.1). Desse modo:

- Reduzir o multiplicador fará o throttling adaptativo se comportar de modo mais agressivo.
- Aumentar o multiplicador fará o throttling adaptativo se comportar de modo menos agressivo.

Por exemplo, em vez de fazer o cliente se autorregular quando $\text{requests} = 2 * \text{accepts}$, faça-o se autorregular quando $\text{requests} = 1.1 * \text{accepts}$. Reduzir o modificador para 1,1 significa que apenas uma requisição será rejeitada pelo backend a cada 10 requisições aceitas.

Em geral, preferimos o multiplicador 2x. Ao permitir que mais requisições do que realmente se espera que sejam permitidas alcancem o backend, desperdiçamos mais recursos no backend, mas também agilizamos a propagação do estado, do backend para os clientes. Por exemplo, se o backend decidir parar de rejeitar tráfego das tarefas clientes, o período de tempo até que todas as tarefas clientes detectem essa mudança de estado será menor.

Uma consideração adicional é que o throttling do lado cliente pode não funcionar bem com clientes que enviem requisições aos seus backends somente de forma bem esporádica. Nesse caso, a visão que cada cliente tem do estado do backend será drasticamente reduzida, e as abordagens para aumentar essa visibilidade tendem a ser custosas.

Criticidade

A *criticidade* é outra noção que percebemos ser muito útil no contexto de cotas globais e throttling. Uma requisição feita a um backend é associada a um de quatro possíveis valores de criticidade, de acordo com quão crítica consideramos que seja essa requisição:

CRITICAL_PLUS

Reservado para as requisições mais críticas, isto é, aquelas que resultarão em sérios impactos, visíveis aos usuários, se falharem.

CRITICAL

O valor default das requisições enviadas pelos jobs de produção. Essas requisições resultarão em um impacto visível ao usuário, mas esse impacto poderá ser menos severo do que aqueles de CRITICAL_PLUS. Espera-se que os serviços façam o provisionamento de capacidade suficiente para todo o tráfego esperado de CRITICAL e CRITICAL_PLUS.

SCHEDDABLE_PLUS

Tráfego para o qual uma indisponibilidade parcial é esperada. É o default para jobs em batch, que podem fazer retentativas de requisições alguns minutos ou até mesmo algumas horas depois.

SCHEDDABLE

Tráfego para o qual uma indisponibilidade parcial frequente e uma indisponibilidade ocasional completa são esperadas.

Percebemos que quatro valores eram suficientemente robustos para modelar quase todos os serviços. Discutimos várias propostas para acrescentar mais valores, pois fazer isso nos permitiria classificar as requisições de modo mais

específico. Entretanto, definir valores adicionais exigiria mais recursos para operar vários sistemas que levem a criticidade em consideração.

Fizemos da criticidade uma noção de primeira classe em nosso sistema RPC e trabalhamos com afinc para integrá-la em muitos de nossos sistemas de controle para que ela pudesse ser levada em consideração quando reagíssemos a situações de sobrecarga. Por exemplo:

- Quando um cliente esgotar a cota global, uma tarefa de backend apenas rejeitará as requisições de uma dada criticidade se já estiver rejeitando todas as requisições de criticidades mais baixas (de fato, os limites por cliente que nosso sistema aceita, descritos anteriormente, podem ser definidos por criticidade).
- Quando uma tarefa estiver sobrecarregada, ela rejeitará primeiro as requisições de criticidades mais baixas.
- O sistema de throttling adaptativo também mantém dados estatísticos separados para cada criticidade.

A criticidade de uma requisição é ortogonal aos seus requisitos de latência e, desse modo, à qualidade de serviço (QoS) de rede subjacente usada. Por exemplo, quando um sistema exibe resultados de pesquisa ou sugestões enquanto o usuário digita uma consulta para uma pesquisa, as requisições subjacentes são altamente descartáveis (se o sistema estiver sobrecarregado, é aceitável não exibir esses resultados), porém elas tendem a ter requisitos rigorosos de latência.

Também estendemos significativamente o nosso sistema de RPC para propagar a criticidade de modo automático. Se um backend receber uma requisição *A* e, como parte da execução dessa requisição, gerar uma requisição *B* e uma requisição *C* de saída para outros backends, as requisições *B* e *C* utilizarão a mesma criticidade da requisição *A*, por padrão.

No passado, em muitos sistemas do Google, houve uma evolução *ad hoc* das noções de criticidade que, muitas vezes, eram incompatíveis entre serviços diferentes. Ao padronizar e propagar a criticidade como parte de nosso sistema de RPC, atualmente somos capazes de definir a criticidade em pontos específicos, de forma consistente. Isso significa que podemos estar confiantes

de que dependências sobrecarregadas obedecerão à criticidade de alto nível desejada à medida que rejeitarem tráfego, independentemente do nível de profundidade em que elas estiverem na pilha de RPC. Assim, nossa prática consiste em definir a criticidade do modo mais próximo possível dos navegadores ou de clientes móveis – geralmente, nos frontends HTTP que geram o HTML a ser devolvido – e ignorar a criticidade somente em casos específicos, quando isso fizer sentido em alguns pontos da pilha.

Sinais de utilização

Nossa implementação de proteção contra sobrecarga no nível de tarefas é baseada na noção de *utilização*. Em muitos casos, a utilização é apenas uma medida da taxa de CPU (isto é, a taxa de CPU atual dividida pelo total de CPUs reservado para a tarefa), mas, em algumas circunstâncias, também incluímos medidas como a porção de memória reservada utilizada no momento. À medida que a utilização se aproximar de limites configurados, começamos a rejeitar requisições com base em sua criticidade (limites mais altos para criticidades mais altas).

Os sinais de utilização que usamos são baseados no estado local à tarefa (pois o objetivo dos sinais é proteger a tarefa) e temos implementações para vários sinais. O sinal genérico mais útil é baseado na “carga” do processo, determinada por meio de um sistema que chamamos de *média de carga do executor* (executor load average).

Para descobrir a média de carga do executor, contamos o número de threads ativas no processo. Nesse caso, “ativas” se refere às threads que estão executando no momento ou que estão prontas para executar e esperando um processador livre. Suavizamos esse valor com um declínio exponencial e começamos a rejeitar requisições à medida que o número de threads ativas ultrapassa o número de processadores disponíveis para a tarefa. Isso significa que uma requisição de entrada que gere muitas saídas (isto é, uma que escalone um burst de várias operações de curta duração) fará a carga ter um pico muito rápido, porém a suavização absorverá a maior parte desse pico. No entanto, se as operações não tiverem curta duração (isto é, se a carga aumentar e permanecer alta por um período de tempo significativo), a tarefa

começará a rejeitar requisições.

Embora a média de carga do executor tenha se provado ser um sinal muito útil, nosso sistema pode tratar quaisquer sinais de utilização que um backend em particular possa precisar. Por exemplo, podemos usar pressão de memória – que indica se o uso de memória em uma tarefa de backend ultrapassou os parâmetros operacionais normais – como outro possível sinal de utilização. O sistema também pode ser configurado para combinar vários sinais e rejeitar requisições que excederem os limites-alvo de utilização combinados (ou individuais).

Tratando erros de sobrecarga

Além de tratar a carga de modo elegante, pensamos muito bem no modo como nossos clientes devem reagir quando receberem uma resposta de erro relacionada à carga. No caso de erros de sobrecarga, fazemos a distinção entre duas possíveis situações.

Um grande subconjunto de tarefas de backend no datacenter está sobrecarregado

Se o sistema de distribuição de carga entre datacenters estiver funcionando perfeitamente (isto é, ele é capaz de propagar estados e reagir instantaneamente a mudanças no tráfego), essa condição não ocorrerá.

Um pequeno subconjunto de tarefas de backend no datacenter está sobrecarregado

Essa situação geralmente é causada por imperfeições na distribuição de carga no interior do datacenter. Por exemplo, uma tarefa pode, bem recentemente, ter recebido uma requisição muito custosa. Nesse caso, é bem provável que o datacenter tenha capacidade remanescente em outras tarefas para tratar a solicitação.

Se um subconjunto grande de tarefas de backend no datacenter estiver sobrecarregado, não deve haver retentativas de requisições, e os erros devem ser propagados até quem fez a chamada (por exemplo, devolvendo um erro ao usuário final). É muito mais comum que apenas uma pequena porção das tarefas fique sobrecarregada, caso em que a resposta preferível é fazer uma

nova tentativa da requisição imediatamente. Em geral, nosso sistema de distribuição de carga entre datacenters tenta direcionar o tráfego dos clientes para os datacenters de backends mais próximos possíveis. Em alguns poucos casos, o datacenter mais próximo está distante (por exemplo, um cliente pode ter seu backend mais próximo disponível em um continente diferente), mas, geralmente, conseguimos deixar os clientes próximos aos seus backends. Dessa maneira, a latência adicional para uma nova tentativa da requisição – algumas viagens de ida e volta na rede – tende a ser desprezível.

Do ponto de vista de nossas políticas de distribuição de carga, as retentativas de requisições não podem ser diferenciadas de novas requisições. Isto é, não usamos nenhuma lógica explícita para garantir que uma retentativa, na verdade, vá para uma tarefa de backend diferente; simplesmente contamos com a alta probabilidade de que a retentativa será enviada para uma tarefa de backend diferente, apenas como consequência do número de backends participantes no subconjunto. Garantir que todas as retentativas sejam realmente encaminhadas a uma tarefa diferente exigiria uma complexidade adicional em nossas APIs que não valeria a pena incluir.

Mesmo que um backend esteja apenas levemente sobrecarregado, uma requisição de cliente muitas vezes será melhor servida se o backend rejeitar as retentativas e as novas requisições igualmente, de modo rápido. Essas requisições podem então ser reenviadas imediatamente para uma tarefa de backend diferente, que possa ter recursos sobrando. A consequência de tratar retentativas e novas requisições do mesmo modo no backend é que reenviar requisições para tarefas diferentes passa a ser uma forma de distribuição de carga orgânica: a carga é redirecionada para tarefas que possam ser mais adequadas a essas requisições.

Decidindo fazer uma nova tentativa

Quando um cliente recebe uma resposta de erro “tarefa sobrecarregada”, ele deve decidir se deve fazer uma nova tentativa da requisição. Temos alguns mecanismos implantados para evitar retentativas quando uma porção significativa das tarefas em um cluster estiver sobrecarregada.

Em primeiro lugar, implementamos uma *provisão de retentativas por*

requisição de até três tentativas. Se uma requisição já tiver falhado três vezes, deixamos a falha se propagar até quem fez a chamada. O raciocínio é que, se uma requisição já foi enviada para tarefas sobrecarregadas três vezes, é relativamente improvável que tentar novamente vá ajudar, pois há uma grande chance de todo o datacenter estar sobrecarregado.

Em segundo lugar, implementamos uma *provisão de retentativas por cliente*. Cada cliente mantém o controle da proporção das requisições que corresponde às retentativas. Uma requisição será feita novamente desde que essa proporção seja inferior a 10%. O raciocínio é que, se apenas um pequeno subconjunto das tarefas estiver sobrecarregado, haverá relativamente pouca necessidade de refazer uma tentativa.

Como exemplo concreto (do cenário de pior caso), vamos supor que um datacenter esteja aceitando uma pequena quantidade de requisições e rejeitando uma porção grande delas. Suponha que X seja a taxa total de requisições que se tentou fazer ao datacenter, de acordo com a lógica do lado cliente. Por causa do número de retentativas que ocorrerão, a quantidade de requisições aumentará significativamente para um valor muito próximo de $3X$. Embora tenhamos efetivamente limitado o aumento causado pelas retentativas, um aumento triplo nas requisições é significativo, em especial se o custo de rejeição *versus* de processamento de uma requisição for considerável. No entanto, se considerarmos a provisão de retentativas por cliente (uma razão de retentativas de 10%), o crescimento será reduzido para apenas 1,1x no caso geral – uma melhoria significativa.

Segundo uma terceira abordagem, os clientes incluem um contador de quantas tentativas já houve para a requisição nos metadados dessa requisição. Por exemplo, o contador começa em 0 na primeira tentativa e é incrementado a cada retentativa, até atingir 2, ponto em que a provisão por requisição faz com que não haja novas tentativas. Os backends mantêm histogramas desses valores no histórico recente. Quando um backend precisa rejeitar uma requisição, ele consulta esses histogramas a fim de determinar a probabilidade de outras tarefas de backend também estarem sobrecarregadas. Se esses histogramas revelarem uma quantidade significativa de retentativas (indicando que outras tarefas de backend provavelmente também estão

sobrecarregadas), uma resposta de erro “sobrecarregado; não faça retentativas” será devolvida no lugar do erro-padrão “tarefa sobrecarregada” que dispara as retentativas.

A Figura 21.1 mostra o número de tentativas em cada requisição recebida por uma dada tarefa de backend em várias situações de exemplo, em uma janela móvel (correspondendo a 1.000 requisições iniciais, sem contar as retentativas). Por questões de simplicidade, a provisão de retentativas por cliente foi ignorada (isto é, esses números pressupõem que o único limite para as retentativas é a provisão de retentativas igual a três tentativas por requisição), e a criação de subconjuntos poderia, de certo modo, alterar esses números.

Nossos maiores serviços tendem a ser pilhas profundas de sistemas que, por sua vez, dependem uns dos outros. Nessa arquitetura, só deve haver retentativas para as requisições na camada imediatamente acima daquela que as está rejeitando. Quando decidimos que uma dada requisição não pode ser atendida e que não deve haver uma retentativa, utilizamos um erro “sobrecarregado; não faça retentativas” e, desse modo, evitamos uma explosão combinatória de retentativas.

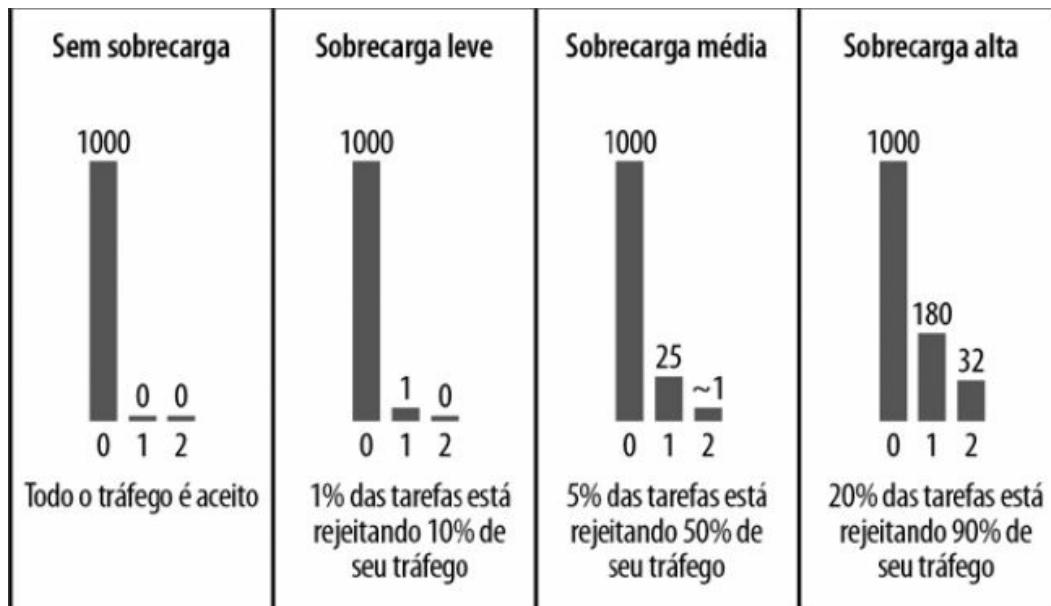


Figura 21.1 – Histogramas de tentativas em várias condições.

Considere o exemplo na Figura 21.2 (na prática, nossas pilhas, com frequência, são significativamente mais complexas). Suponha que o Frontend

de BD esteja sobrecarregado no momento e rejeite uma requisição. Nesse caso:

- O Backend B tentará fazer a requisição novamente, de acordo com as diretrizes anteriores.
- No entanto, depois que o Backend B determina que a requisição ao Frontend de BD não pode ser servida (por exemplo, porque já houve retentativa da requisição e ela foi rejeitada três vezes), o Backend B deve devolver um erro “sobrecarregado; não faça retentativas” ou uma resposta degradada ao Backend A (supondo que ele possa gerar uma resposta moderadamente útil, mesmo quando sua requisição ao Frontend de BD tenha falhado).
- O Backend A tem exatamente as mesmas opções para a requisição recebida do Frontend, e prossegue de acordo com isso.

O ponto principal é que somente o Backend B deve fazer uma retentativa da requisição com falha do Frontend de BD, pois o Backend B é a camada imediatamente acima. Se várias camadas refizerem as tentativas, teríamos uma explosão combinatória.

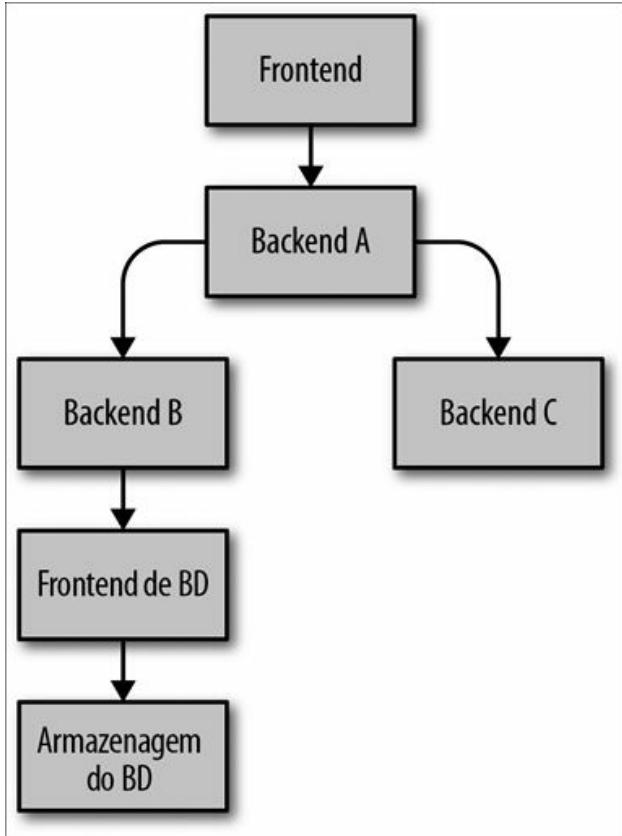


Figura 21.2 – Uma pilha de dependências.

Carga de conexões

A carga associada às conexões é um último fator que vale a pena ser mencionado. Às vezes, só levamos em consideração a carga nos backends causada diretamente pelas requisições que eles recebem (que é um dos problemas com abordagens que modelam a carga com base em consultas por segundo). No entanto, isso faz com que os custos de CPU e de memória para manter um pool grande de conexões ou o custo de uma taxa rápida de reestabelecimento de conexões sejam menosprezados. Problemas como esses são desprezíveis em sistemas pequenos, mas podem se tornar rapidamente problemáticos quando operamos sistemas de RPC de larga escala.

Conforme mencionamos antes, nosso protocolo de RPC exige que clientes inativos realizem verificações periódicas de sanidade. Depois que uma conexão estiver ociosa durante um período de tempo configurável, o cliente derruba sua conexão TCP e passa para UDP a fim de fazer verificação de

sanidade. Infelizmente, esse comportamento é problemático quando houver um número muito grande de tarefas clientes que gerem uma taxa bem baixa de requisições: as verificações de sanidade nas conexões podem exigir mais recursos do que propriamente servir às requisições. Abordagens como ajustar cuidadosamente os parâmetros das conexões (por exemplo, diminuir significativamente a frequência das verificações de sanidade) ou até mesmo criar e destruir as conexões de forma dinâmica podem melhorar essa situação de forma considerável.

Tratar bursts de novas requisições de conexão é um segundo problema (porém, está relacionado ao caso anterior). Já vimos bursts desse tipo ocorrerem no caso de jobs muito grandes em batch, que criam um número muito grande de tarefas worker clientes, todas de uma só vez. A necessidade de negociar e manter um número excessivo de novas conexões ao mesmo tempo pode facilmente sobrecarregar um grupo de backends. De acordo com nossa experiência, há duas estratégias que podem ajudar a atenuar essa carga:

- Exponha a carga ao algoritmo de distribuição de carga entre datacenters (por exemplo, baseie a distribuição de carga na utilização do cluster, e não apenas no número de requisições). Nesse caso, a carga proveniente das requisições é redistribuída de modo eficiente para outros datacenters que tenham capacidade excedente.
- Obrigue os jobs clientes em batch a utilizar um conjunto separado de tarefas de backend que sejam um *proxy de batch*, as quais não fazem nada além de encaminhar as requisições aos backends subjacentes e passar suas respostas de volta aos clientes de forma controlada. Desse modo, em vez de “cliente batch → backend”, você terá “cliente batch → proxy de batch → backend”. Nesse caso, quando os jobs bem grandes começarem, somente o job de proxy de batch sofrerá, protegendo os backends propriamente ditos (e os clientes com prioridades mais altas). De modo eficiente, o proxy de batch atua como um fusível. Outra vantagem de usar o proxy é que, geralmente, ele reduz o número de conexões com o backend, o que pode melhorar a sua distribuição de carga (por exemplo, as tarefas de proxy podem usar subconjuntos maiores e, provavelmente, ter uma melhor visão do estado das tarefas de backend).

Conclusões

Este capítulo e o Capítulo 20 discutiram como várias técnicas (criação determinística de subconjuntos, Weighted Round Robin, throttling do lado cliente, cotas para clientes etc.) podem ajudar a distribuir a carga pelas tarefas em um datacenter de forma relativamente uniforme. No entanto, esses mecanismos dependem da propagação de estados em um sistema distribuído. Embora tenham um desempenho razoavelmente bom no caso geral, aplicá-los no mundo real resultou em um pequeno número de situações em que eles não funcionaram de modo perfeito.

Como resultado, consideramos ser crucial garantir que tarefas individuais sejam protegidas contra sobrecarga. Para dizer isso de forma simples, uma tarefa de backend configurada para tratar uma determinada taxa de tráfego deve continuar a servir o tráfego nessa taxa, sem qualquer impacto significativo na latência, independentemente do volume de tráfego em excesso lançado sobre ela. Como corolário, a tarefa de backend não deve se desestruturar e falhar sob carga. Essas afirmações devem se manter válidas até uma determinada taxa de tráfego – até algum ponto acima de $2x$ ou de até $10x$ o volume para o qual a tarefa está configurada para processar. Aceitamos que possa haver um certo ponto a partir do qual um sistema começa a se desestruturar, e elevar o limite em que essa desestruturação ocorre torna-se relativamente difícil.

O segredo é levar a sério essas condições de degradação. Se forem ignoradas, muitos sistemas exibirão um comportamento horrível. À medida que as atividades se acumularem e as tarefas, em algum momento, ficarem sem memória e falharem (ou acabarem consumindo quase toda a sua CPU em thrashing de memória), a latência sofrerá, enquanto o tráfego será descartado e as tarefas competirão por recursos. Se não for verificada, a falha em um subconjunto de um sistema (por exemplo, em uma tarefa individual de backend) poderá disparar falhas nos componentes de outro sistema, possivelmente fazendo todo o sistema (ou um subconjunto considerável) falhar. O impacto desse tipo de falha em cascata pode ser tão severo que é muito importante que qualquer sistema operando em escala se proteja contra ela; veja o Capítulo 22.

Um erro comum é supor que um backend sobrecarregado deve recusar e parar de aceitar qualquer tráfego. No entanto, essa suposição, na verdade, vai contra a meta de uma distribuição de carga robusta. Na verdade, queremos que o backend continue aceitando o máximo possível de tráfego, mas que aceite essa carga somente à medida que houver capacidade disponível. Um backend bem comportado, com o apoio de políticas robustas de distribuição de carga, deve aceitar apenas as requisições que ele pode processar, e rejeitar as demais de forma elegante.

Embora tenhamos uma grande variedade de ferramentas para implementar uma boa distribuição de carga e proteções contra sobrecarga, não existem soluções mágicas: com frequência, a distribuição de carga exige um profundo conhecimento do sistema e da semântica de suas requisições. As técnicas descritas neste capítulo evoluíram juntamente com as necessidades de muitos sistemas no Google, e provavelmente permanecerão evoluindo à medida que a natureza de nossos sistemas continua a mudar.

¹ Por exemplo, veja o Doorman (<https://github.com/youtube/doorman>), que oferece um sistema de throttling cooperativo e distribuído do lado cliente.

CAPÍTULO 22

Tratando falhas em cascata

Escrito por Mike Ulrich

Se você não for inicialmente bem-sucedido, faça back off exponencialmente.

— Dan Sandler, Engenheiro de Software do Google

Por que as pessoas sempre se esquecem de que é preciso adicionar um pouco de jitter?

— Ade Oshineye, Google Developer Advocate

Uma falha em cascata é uma falha que aumenta com o tempo, como resultado de um feedback positivo.¹ Ela pode ocorrer quando parte de um sistema como um todo falha, elevando a probabilidade de outras partes falharem. Por exemplo, uma única réplica de um serviço pode falhar devido à sobrecarga, aumentando a carga das réplicas remanescentes e sua probabilidade de falhar, provocando um efeito dominó que derrubará todas as réplicas de um serviço.

Utilizaremos o serviço de pesquisa Shakespeare discutido na seção “Shakespeare: um exemplo de serviço”, como exemplo ao longo deste capítulo. Sua configuração de produção pode ter o aspecto mostrado na Figura 22.1.

Causas de falhas em cascata e design para evitá-las

Um design bem planejado de sistema deve levar em consideração alguns cenários típicos que incluem a maior parte das falhas em cascata.

Sobrecarga de servidores

A causa mais comum de falhas em cascata é a sobrecarga. A maioria das falhas em cascata descritas aqui se deve diretamente à sobrecarga de servidores ou a extensões ou variações desse cenário.

Suponha que o frontend no cluster A trate 1.000 requisições por segundo (QPS), como mostra a Figura 22.2.

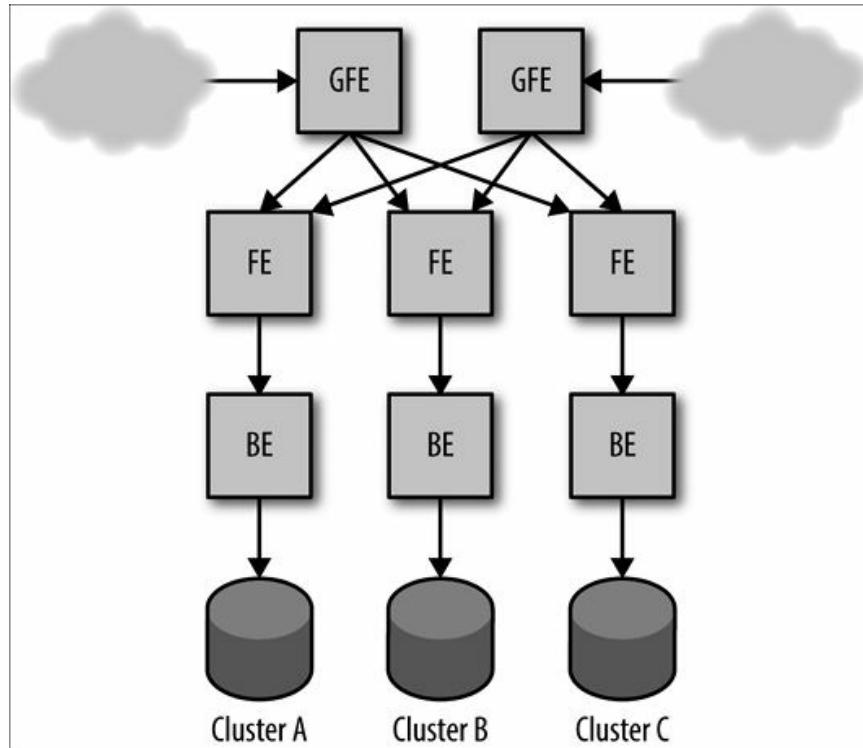


Figura 22.1 – Exemplo de configuração de produção para o serviço de pesquisa Shakespeare.

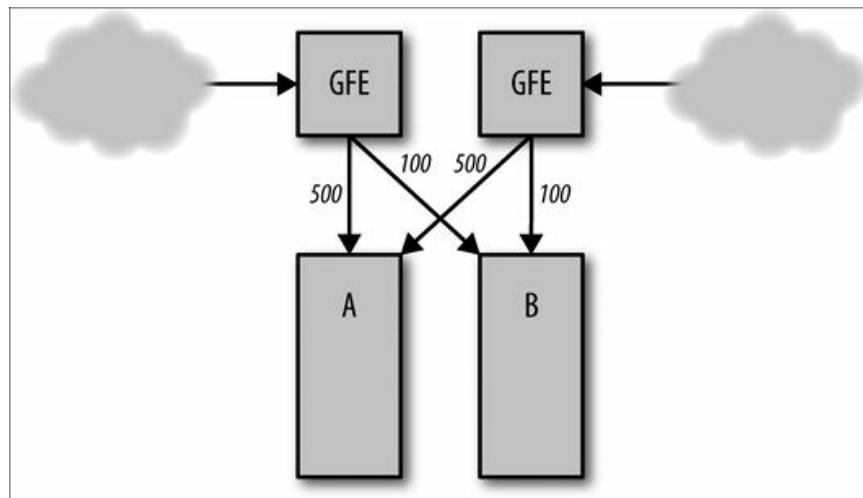


Figura 22.2 – Distribuição da carga normal de servidores entre os clusters A e B.

Se o cluster B falhar (Figura 22.3), as requisições ao cluster A aumentam para 1.200 QPS. Os frontends em A não são capazes de tratar requisições a

1.200 QPS e, desse modo, começam a ficar sem recursos, o que os faz falhar, exceder tempos de espera ou ter um mau comportamento. Como resultado, a taxa de requisições tratadas com sucesso em A se reduz para um valor bem abaixo de 1.000 QPS.

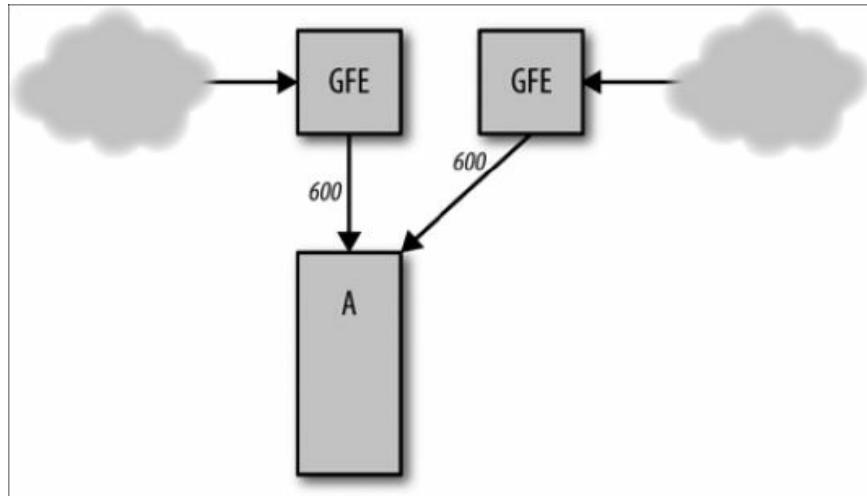


Figura 22.3 – O cluster B falha, enviando todo o tráfego ao cluster A.

Essa redução na taxa de tarefas úteis efetuadas pode atingir outros domínios de falhas, podendo se espalhar globalmente. Por exemplo, uma sobrecarga local em um cluster pode fazer seus servidores falharem; em resposta, o controlador de distribuição de carga envia requisições a outros clusters, sobrecarregando seus servidores, o que resulta em uma falha de sobrecarga em todo o serviço. Talvez não demore muito para esses eventos se tornarem aparentes (por exemplo, da ordem de alguns minutos), pois os sistemas de distribuição de carga e de escalonamento de tarefas envolvidos podem atuar bem rapidamente.

Esgotamento de recursos

O esgotamento de recursos pode resultar em uma latência mais alta, em taxas de erro elevadas ou na substituição por resultados de mais baixa qualidade. Na verdade, esses são efeitos desejados do esgotamento de recursos: em algum momento, é preciso abrir mão de algo se a carga ultrapassar o volume que um servidor é capaz de tratar.

Conforme o recurso que se esgotar em um servidor e de como o servidor está configurado, o esgotamento de recursos pode deixar o servidor menos

eficiente ou levá-lo a falhar, fazendo o distribuidor de carga distribuir os problemas de recurso para outros servidores. Quando isso acontecer, a taxa de requisições tratada com sucesso pode cair e, possivelmente, provocar uma falha em cascata no cluster ou em todo um serviço.

Tipos diferentes de recursos podem se esgotar, resultando em efeitos variados nos servidores.

CPU

Se não houver CPU suficiente para tratar a carga de requisições, geralmente todas elas ficarão mais lentas. Esse cenário pode resultar em vários efeitos secundários, incluindo estes:

Número maior de requisições em andamento

Como as requisições demoram mais para serem atendidas, mais requisições são tratadas de forma concorrente (até uma possível capacidade máxima, quando um enfileiramento poderá ocorrer). Isso afeta quase todos os recursos, incluindo memória, número de threads ativas (em um modelo de servidor com threads por requisição), número de descritores de arquivos e recursos de backend (que, por sua vez, podem provocar outros efeitos).

Tamanhos de fila excessivamente longos

Se não houver capacidade suficiente para tratar todas as requisições continuamente, as filas do servidor ficarão saturadas. Isso significa que a latência aumentará (as requisições serão enfileiradas por períodos de tempo mais longos) e a fila utilizará mais memória. Veja a seção “Gerenciamento de filas”, que apresenta uma discussão sobre estratégias de atenuação.

Starvation de thread

Quando uma thread não consegue fazer progressos por estar esperando um lock, as verificações de sanidade podem falhar se o endpoint da verificação não puder ser servido a tempo.

Starvation de CPU ou de requisições

Watchdogs² internos no servidor detectam que ele não está fazendo progressos, fazendo os servidores falharem devido à starvation de CPU, ou

por starvation de requisições, caso eventos de watchdog sejam disparados remotamente e processados como parte da fila de requisições.

Tempos de espera de RPC esgotados

À medida que um servidor se torna sobrecarregado, suas respostas a RPCs de seus clientes chegam atrasadas, o que pode fazer qualquer tempo de espera definido por esses clientes se esgotar. O trabalho feito pelo servidor para responder é então desperdiçado, e os clientes podem tentar as RPCs novamente, resultando em mais sobrecarga.

Vantagens reduzidas de caching de CPU

À medida que mais CPU é usada, a chance de distribuição em mais cores (núcleos) de CPU aumenta, resultando em um uso reduzido de caches locais e na diminuição da eficiência da CPU.

Memória

Além dos demais problemas, mais requisições em andamento consomem mais RAM por causa da alocação de objetos para a requisição, a resposta e a RPC. O esgotamento de memória pode causar os seguintes efeitos:

Tarefas que morrem

Por exemplo, uma tarefa pode ser expulsa pelo gerenciador de contêiner (VM ou algo diferente) por exceder os limites de recursos disponíveis, ou falhas específicas de aplicação podem fazer as tarefas morrerem.

Aumento na taxa de GC (garbage collection, ou coleta de lixo) em Java, resultando em mais uso de CPU

Um círculo vicioso pode ocorrer nesse cenário: menos CPU está disponível, resultando em requisições mais lentas, o que leva a um aumento no uso de RAM; isso resulta em mais GC, o que leva a menos disponibilidade ainda de CPU. Isso é conhecido coloquialmente como “espiral da morte de GC”.

Redução na taxa de acertos de cache (cache hits)³

A redução na RAM disponível pode reduzir as taxas de acertos de cache no nível de aplicação, resultando em mais RPCs aos backends, o que pode, possivelmente, fazer os backends ficarem mais sobrecarregados.

Threads

A starvation de thread pode causar erros diretamente ou levar a falhas na verificação de sanidade. Se o servidor adicionar threads conforme for necessário, o overhead das threads pode utilizar RAM demais. Em casos extremos, a starvation de threads também pode fazer com que você fique sem IDs de processo.

Descritores de arquivo

Ficar sem descritores de arquivo pode levar à incapacidade de inicializar conexões de rede, o que, por sua vez, pode fazer as verificações de sanidade falharem.

Dependências entre recursos

Observe que muitos desses cenários de esgotamento de recursos alimentam uns aos outros – um serviço passando por sobrecarga muitas vezes tem vários sintomas secundários que podem parecer a causa-raiz, dificultando a depuração.

Por exemplo, imagine o cenário a seguir:

1. Um frontend Java tem parâmetros de GC (garbage collection) ajustados de forma precária.
2. Diante de uma carga alta (porém esperada), o frontend fica sem CPU por causa de GC.
3. O esgotamento de CPU faz com que o tratamento das requisições fique lento.
4. O aumento no número de requisições em progresso faz com que mais RAM seja usada para processar as requisições.
5. A pressão por memória por causa das requisições, em conjunto com uma alocação de memória fixa para o processo de frontend como um todo, deixa menos RAM disponível para caching.
6. O tamanho reduzido de cache implica menos entradas nele, além de uma taxa menor de acertos de cache.
7. O aumento de “erros” de cache (cache misses)⁴ significa que mais

requisições recorrem ao backend para serem servidas.

8. O backend, por sua vez, fica sem CPU ou threads.

9. Por fim, a falta de CPU faz com que as verificações básicas de sanidade falhem, dando início a uma falha em cascata.

Em situações tão complexas quanto o cenário anterior, é improvável que a cadeia de causas seja totalmente diagnosticada durante uma interrupção de serviço. Pode ser muito difícil determinar que a falha no backend foi causada por uma diminuição na taxa de cache no frontend, particularmente se os componentes de frontend e backend tiverem proprietários diferentes.

Indisponibilidade do serviço

O esgotamento de recursos pode fazer os servidores falharem; por exemplo, os servidores podem falhar quando muita RAM for alocada a um contêiner. Depois que alguns servidores falharem devido à sobrecarga, a carga nos servidores remanescentes poderá aumentar, fazendo-os falhar também. O problema tende a virar uma bola de neve e, em breve, todos os servidores começam a entrar em um ciclo de falhas. Com frequência, é difícil escapar desse cenário, pois, assim que os servidores voltam a ficar online, são bombardeados com uma taxa extremamente alta de requisições e falham quase que de imediato.

Por exemplo, se um serviço estava saudável a 10.000 QPS, mas iniciou uma falha em cascata por causa de falhas a 11.000 QPS, é quase certo que reduzir a carga para 9.000 QPS não interromperá as falhas. Isso ocorre porque o serviço estará tratando uma demanda maior com capacidade reduzida; somente uma pequena fração dos servidores geralmente estará suficientemente saudável para tratar as requisições. A fração dos servidores que não estará saudável depende de alguns fatores: da 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 que acabou de ser iniciada consegue sobreviver à carga. Nesse exemplo, se 10% dos servidores estiverem suficientemente saudáveis para tratar requisições, a taxa de requisições precisaria cair para cerca de 1.000 QPS para que o sistema se estabilize e se recupere.

De modo semelhante, os servidores podem parecer não saudáveis para a camada de distribuição de carga, resultando em uma capacidade reduzida na distribuição: os servidores podem ir para um estado de “incapacidade” (lame duck – veja a seção “Uma abordagem robusta para tarefas não saudáveis: estado de incapacidade”) ou não responder devidamente às verificações de sanidade sem falhar. O efeito pode ser bem semelhante a falhar: mais servidores parecerão não saudáveis, os servidores saudáveis tenderão a aceitar requisições por um período de tempo bem curto antes de se tornarem não saudáveis e menos servidores participarão do tratamento das requisições.

Políticas de distribuição de carga que evitam servidores que forneceram erros podem exacerbar mais ainda os problemas – alguns backends servem alguns erros, portanto não contribuem para a capacidade disponível ao serviço. Isso aumenta a carga nos servidores remanescentes, iniciando o efeito bola de neve.

Evitando a sobrecarga nos servidores

A lista a seguir apresenta estratégias para evitar a sobrecarga nos servidores em uma ordem geral de prioridade:

Faça testes de carga nos limites de capacidade do servidor e teste o modo de falha para sobrecarga

Esse é o exercício mais importante que você pode realizar para evitar sobrecarga de servidores. A menos que você teste em um ambiente realista, é muito difícil prever exatamente quais recursos se esgotarão e como esse esgotamento de recursos se manifestará. Para ver detalhes, consulte a seção “Testes para falhas em cascata”.

Sirva resultados degradados

Sirva resultados de menor qualidade, mais fáceis de processar, ao usuário. Sua estratégia, nesse caso, será específica do serviço. Veja a seção “Rejeição de carga e degradação elegante”.

Instrumente o servidor para rejeitar requisições quando estiver sobrecarregado

Os servidores devem se proteger contra situações de sobrecarga e falhas. Quando houver sobrecarga nas camadas de frontend ou de backend, falhe logo e com baixo custo. Para ver detalhes, consulte a seção “Rejeição de carga e degradação elegante”.

Instrumente sistemas de mais alto nível para rejeitar requisições em vez de sobrecarregar os servidores

Observe que, como a limitação de taxa geralmente não leva a saúde do serviço como um todo em consideração, talvez não seja possível interromper uma falha que já tenha começado. É provável que implementações simples de limitação de taxa também deixem uma capacidade subutilizada. A limitação de taxa pode ser implementada em vários lugares:

- *Nos proxies reversos*, limitando o volume de requisições por meio de critérios como endereço IP para atenuar tentativas de ataques de negação de serviço e clientes abusivos.
- *Nos distribuidores de carga*, descartando requisições quando o serviço entrar em um estado de sobrecarga global. Conforme a natureza e a complexidade do serviço, essa limitação de taxa pode ser indiscriminada (“descarte todo o tráfego acima de X requisições por segundo”) ou mais seletiva (“descarte requisições que não sejam de usuários que tenham interagido recentemente com o serviço” ou “descarte requisições de operações de baixa prioridade, como sincronização em segundo plano, mas continue servindo a sessões interativas de usuários”).
- *Em tarefas individuais*, para evitar que flutuações aleatórias na distribuição de carga sobrecarreguem o servidor.

Faça planejamento de capacidade

Um bom planejamento de capacidade pode reduzir a probabilidade de uma falha em cascata ocorrer. O planejamento de capacidade deve ser combinado com testes de desempenho para determinar a carga com que o serviço falhará. Por exemplo, se o ponto de ruptura de cada cluster for de 5.000 QPS, a carga está uniformemente distribuída entre os clusters⁵ e a carga de pico do serviço será de 19.000 QPS, então aproximadamente seis

clusters serão necessários para operar o serviço a $N + 2$.

O planejamento de capacidade reduz a probabilidade de disparar uma falha em cascata, mas não é suficiente para proteger o serviço contra esse tipo de falha. Se você perder partes importantes de sua infraestrutura durante um evento planejado ou não, nenhum planejamento de capacidade será suficiente para evitar falhas em cascata. Problemas de distribuição de carga, partições de rede ou aumentos inesperados de tráfego podem criar intervalos de carga alta que excedam o que foi planejado. Alguns sistemas podem aumentar o número de tarefas para seu serviço por demanda, o que possibilita evitar a sobrecarga; contudo, um planejamento de capacidade apropriado continua sendo necessário.

Gerenciamento de filas

A maioria dos servidores que usa thread por requisição utiliza uma fila na frente de um pool de threads para tratar as requisições. As requisições chegam, esperam em uma fila e, então, as threads as retiram da fila e realizam o trabalho propriamente dito (quaisquer ações exigidas pelo servidor). Geralmente, se a fila estiver cheia, o servidor rejeitará novas requisições.

Se a taxa de requisições e a latência de uma dada tarefa forem constantes, não há motivos para enfileirar as requisições: um número constante de threads deverá ser ocupado. Nesse cenário idealizado, as requisições só serão enfileiradas se a taxa contínua de requisições de entrada exceder a taxa com que o servidor é capaz de processar as requisições, o que resulta em saturação tanto do pool de threads quanto da fila.

As requisições em fila consomem memória e aumentam a latência. Por exemplo, se o tamanho da fila for dez vezes o número de threads, o tempo para tratar a requisição em uma thread será de cem milissegundos. Se a fila estiver cheia, uma requisição demorará 1,1 segundo para ser tratada, e a maior parte do tempo será gasta na fila.

Para um sistema com um tráfego razoavelmente constante no tempo, geralmente é melhor ter tamanhos pequenos de fila em relação ao tamanho do pool de threads (por exemplo, 50% ou menos), o que resulta no servidor rejeitando requisições mais cedo quando não for capaz de sustentar a taxa de

requisições de entrada. Por exemplo, o Gmail com frequência utiliza servidores sem fila, contando com failover para tarefas de outros servidores quando as threads estiverem ocupadas. Na outra extremidade do espectro, sistemas com cargas “em burst”, 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, no tempo de processamento para cada requisição e no tamanho e frequência dos bursts.

Rejeição de carga e degradação elegante

A *rejeição de carga* (load shedding) rejeita parte da carga ao descartar 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 sanidade, sirva com latência extremamente alta ou tenha qualquer um dos demais sintomas associados à sobrecarga, enquanto continua fazendo o máximo de trabalho útil que puder.

Uma maneira simples de rejeitar a carga é fazer um throttling (estrangulamento) por tarefa, baseado em CPU, memória ou tamanho da fila; limitar o tamanho da fila, conforme discutido na seção “Gerenciamento de filas”, é uma forma que essa estratégia pode assumir. Por exemplo, uma abordagem eficiente consiste em devolver um HTTP 503 (serviço indisponível) para qualquer requisição de entrada quando houver mais que um determinado número de requisições de cliente em andamento.

Mudar o método de enfileiramento do *FIFO* padrão (first-in, first-out, ou o primeiro que entra é o primeiro que sai) para *LIFO* (last-in, first-out, ou o último que entra é o primeiro que sai) ou usar o algoritmo *CoDel* (Controlled Delay, ou Atraso Controlado) [Nic12] ou abordagens semelhantes pode reduzir a carga ao remover requisições que provavelmente não valham a pena ser processadas [Mau15]. Se uma pesquisa web de um usuário estiver lenta porque uma RPC foi enfileirada por dez segundos, há uma boa chance de o usuário ter desistido e atualizado o seu navegador, gerando outra requisição: não faz sentido responder à primeira requisição, pois ela será ignorada! Essa estratégia funciona bem quando combinada com propagação de tempos de espera de RPC pela pilha, conforme descrito na seção “Latência e tempos de

espera”.

Abordagens mais sofisticadas incluem identificar clientes para ser mais seletivo quanto ao tipo de trabalho descartado, ou escolher requisições que sejam mais importantes e priorizá-las. É mais provável que estratégias como essas sejam necessárias para serviços compartilhados.

A *degradação elegante* leva o conceito de rejeição de carga um passo além ao reduzir a quantidade de trabalho que deve ser realizada. Em algumas aplicações, é possível diminuir significativamente a quantidade de trabalho ou o tempo necessário reduzindo a qualidade das respostas. Por exemplo, uma aplicação de pesquisa pode fazer buscas somente em um subconjunto dos dados armazenados em um cache em memória, em vez de pesquisar o banco de dados completo em disco, ou pode utilizar um algoritmo de classificação menos exato (porém mais rápido) quando estiver sobrecarregado.

Ao avaliar as opções de rejeição de carga ou de degradação elegante para o seu serviço, considere os seguintes aspectos:

- Quais métricas você deve usar para determinar quando a rejeição de carga ou a degradação elegante devem entrar em ação (por exemplo, uso de CPU, latência, tamanho de fila, número de threads usado, se seu serviço entra automaticamente em modo de degradação ou se uma intervenção manual é necessária)?
- Quais ações devem ser executadas quando o servidor está em modo de degradação?
- Em qual camada a rejeição de carga e a degradação elegante devem ser implementadas? Faz sentido implementar essas estratégias em todas as camadas da pilha, ou é suficiente ter um ponto de estrangulamento de alto nível?

À medida que avaliar as opções e implantá-las, tenha em mente o seguinte:

- Uma degradação elegante não deve ser disparada com muita frequência – em geral, em casos de falha no planejamento de capacidade ou uma mudança inesperada de carga. Mantenha o sistema simples e compreensível, particularmente se ele não for usado com frequência.

- Lembre-se de que o caminho de código que você nunca usa é o caminho que (frequentemente) não funciona. Em uma operação estável, o modo de degradação elegante não será usado, o que implica que você terá muito menos experiência operacional com esse modo e com qualquer uma de suas idiossincrasias, o que *eleva* o nível de risco. Você pode garantir que a degradação elegante continue funcionando ao executar regularmente um pequeno subconjunto de servidores próximos da sobrecarga a fim de exercitar esse caminho de código.
- Monitore e alerte quando muitos servidores entrarem nesses modos.
- Rejeição de carga e degradação elegante complexas podem causar problemas a si mesmas – uma complexidade excessiva pode fazer o servidor entrar em um modo de degradação quando não se quer ou entrar em ciclos de realimentação em momentos indesejados. Projete uma maneira de desativar rapidamente uma degradação elegante complexa ou ajustar parâmetros se for necessário. Armazenar essa configuração em um sistema consistente, que cada servidor possa observar para ver se houve mudanças (por exemplo, no Chubby), pode agilizar a implantação, mas também introduz seus próprios riscos de falhas sincronizadas.

Retentativas

Suponha que o código no frontend que conversa com o backend implemente retentativas de forma ingênua. Ele refaz uma tentativa após se deparar com uma falha e limita o número de RPCs ao backend por requisição lógica em dez. Considere o código de frontend a seguir, que utiliza gRPC em Go:

```
func exampleRpcCall(client pb.ExampleClient, request pb.Request) *pb.Response {
    // Define o timeout de RPC para 5 segundos.
    opts := grpc.WithTimeout(5 * time.Second)

    // Tenta fazer a chamada de RPC até 20 vezes.
    attempts := 20
    for attempts > 0 {
        conn, err := grpc.Dial(*serverAddr, opts...)
        if err != nil {
            // Algo deu errado no estabelecimento da conexão. Tenta novamente.
            attempts--
        }
        // Faz a chamada de RPC...
    }
}
```

```

        continue
    }
    defer conn.Close()

    // Cria um stub de cliente e faz a chamada de RPC.
    client := pb.NewBackendClient(conn)
    response, err := client.MakeRequest(context.Background, request)
    if err != nil {
        // Algo deu errado ao fazer a chamada. Tenta novamente.
        attempts--
        continue
    }

    return response
}

grpclog.Fatalf("ran out of attempts")
}

```

Esse sistema pode iniciar uma falha em cascata da seguinte maneira:

1. Suponha que o nosso backend tenha um limite conhecido de 10.000 QPS por tarefa, após o qual todas as demais requisições são rejeitadas em uma tentativa de degradação elegante.
2. O frontend chama MakeRequest a uma taxa constante de 10.100 QPS e sobrecarrega o backend em 100 QPS, rejeitadas pelo backend.
3. Novas tentativas são feitas para essas 100 QPS com falha em MakeRequest a cada 1.000 ms e, provavelmente, são bem-sucedidas. No entanto, as próprias retentativas se somam às requisições enviadas ao backend, que agora recebe 10.200 QPS – 200 QPS falham devido à sobrecarga.
4. O volume de retentativas aumenta: 100 QPS de retentativas no primeiro segundo levam a 200 QPS, depois a 300 QPS, e assim sucessivamente. Cada vez menos requisições são bem-sucedidas em sua primeira tentativa, portanto menos trabalho útil será feito como uma fração das requisições ao backend.
5. Se a tarefa de backend for incapaz de tratar o aumento na carga – que consome descritores de arquivo, memória e tempo de CPU no backend –, ela pode se desintegrar e falhar com a própria carga de requisições e

retentativas. Essa falha, então, faz com que as requisições que estavam sendo recebidas sejam redistribuídas pelas tarefas de backend restantes, o que, por sua vez, sobrecarrega essas tarefas.

Algumas suposições simplificadoras foram feitas aqui para ilustrar esse cenário⁶, mas a questão é que as retentativas podem desestabilizar um sistema. Observe que tanto picos de carga temporários quanto aumentos lentos no uso podem provocar esse efeito.

Mesmo que a taxa de chamadas para MakeRequest diminuisse a níveis de pré-desintegração (9.000 QPS, por exemplo), dependendo do custo para o backend devolver uma falha, o problema talvez não desapareça. Dois fatores estão em cena nesse caso:

- Se o backend gastar uma quantidade significativa de recursos processando requisições que, em última instância, falharão devido à sobrecarga, então as próprias retentativas podem estar mantendo o backend em um estado de sobrecarga.
- Os próprios servidores de backend podem não estar estáveis. As retentativas podem amplificar os efeitos vistos na seção “Sobrecarga de servidores”.

Se uma dessas condições for verdadeira, para sair dessa interrupção de serviço, você deve reduzir drasticamente ou eliminar a carga nos frontends até que as retentativas parem e os backends se estabilizem.

Esse padrão contribuiu para várias falhas em cascata, independentemente de os frontends e os backends se comunicarem por meio de mensagens de RPC, o “frontend” ser um código de cliente JavaScript gerando chamadas XMLHttpRequest para um endpoint e fazendo retentativas em caso de falha ou se as retentativas são provenientes de um protocolo de sincronização offline que faz retentativas de modo agressivo quando se depara com uma falha.

Ao fazer retentativas automáticas, tenha em mente as seguintes considerações:

- A maior parte das estratégias de proteção de backends descrita na seção “Evitando a sobrecarga nos servidores”, se aplica. Em particular, testar o sistema pode revelar quais são os problemas, e uma degradação elegante

pode reduzir o efeito das retentativas no backend.

- Sempre utilize um backoff exponencial aleatório ao agendar retentativas. Veja também a seção “Backoff exponencial e jitter” (<https://www.awsarchitectureblog.com/2015/03/backoff.html>) no AWS Architecture Blog [Bro15]. Se as retentativas não forem aleatoriamente distribuídas na janela de retentativas, uma pequena perturbação (por exemplo, um ruído na rede) pode fazer com que ondas de retentativas sejam agendadas para o mesmo instante, o que pode amplificá-las [Flo94].
- Limite as retentativas por requisição. Não faça retentativas de uma requisição indefinidamente.
- Considere ter uma provisão de retentativas para todo o servidor. Por exemplo, permita apenas 60 retentativas por minuto em um processo e, se a provisão de retentativas se esgotar, não faça retentativas; simplesmente faça a requisição falhar. Essa estratégia pode conter o efeito das retentativas e ser a diferença entre uma falha de planejamento de capacidade que leve a algumas consultas descartadas e uma falha em cascata global.
- Pense no serviço de forma holística e decida se você realmente precisa fazer retentativas em um dado nível. Em particular, evite amplificar as retentativas gerando-as em vários níveis: uma única requisição na camada mais alta pode gerar um número de tentativas tão grande quanto o *produto* do número de tentativas em cada camada, até a camada mais baixa. Se o banco de dados não puder servir às requisições por estar sobrecarregado, e as camadas de backend, frontend e JavaScript fizerem 3 retentativas (4 tentativas), então uma única ação de usuário poderá criar 64 tentativas (4^3) no banco de dados. Esse comportamento é indesejável se o banco de dados devolver esses erros por estar sobrecarregado.
- Utilize códigos de resposta claros e considere como modos diferentes de falha devem ser tratados. Por exemplo, separe as condições de erro para as quais pode ou não haver retentativas. Não faça retentativas para erros permanentes nem para requisições malformadas em um cliente, pois nenhuma delas terá sucesso. Devolva um status específico quando houver

sobrecarga para que os clientes e outras camadas desistam e não façam novas tentativas.

Em uma emergência, talvez não seja óbvio que uma interrupção de serviço se deva a um comportamento ruim de retentativas. Os gráficos de taxas de retentativas podem apresentar indicações de um mau comportamento das retentativas, o que pode ser confundido com um sintoma, em vez de uma causa multiplicadora. Em termos de atenuação, esse é um caso especial do problema de capacidade insuficiente, com a ressalva adicional de que você deve corrigir o comportamento das retentativas (o que, geralmente, exige uma atualização de código), reduzir significativamente a carga ou descartar totalmente as requisições.

Latência e tempos de espera

Quando um frontend envia uma RPC a um servidor de backend, o frontend consome recursos à espera de uma resposta. Tempos de espera de RPC definem quanto tempo uma requisição pode esperar até o frontend desistir, limitando o tempo em que o backend pode consumir os recursos do frontend.

Definindo um tempo de espera

Geralmente, definir um tempo de espera é uma atitude sábia. Não definir um tempo de espera ou definir um que seja extremamente alto podem fazer com que problemas de curto prazo que ocorreram há muito tempo continuem a consumir recursos do servidor, até que ele seja reiniciado.

Tempos de espera altos podem resultar em consumo de recursos em níveis mais elevados da pilha quando os níveis mais baixos dela estiverem com problemas. Tempos de espera baixos podem fazer com que requisições um pouco mais custosas falhem de forma consistente. Conseguir um equilíbrio entre essas restrições para definir um bom tempo de espera pode ser uma espécie de arte.

Tempos de espera esgotados

Um tema comum em muitas interrupções de serviço provocadas por falhas em cascata é que os servidores consomem recursos tratando requisições cujos

tempos de espera se esgotarão no cliente. Como resultado, os recursos são consumidos enquanto nenhum progresso é feito: você não ganhará pontos se fizer a lição de casa com atraso no caso das RPCs.

Suponha que uma RPC tenha um tempo de espera de dez segundos, conforme definido pelo cliente. O servidor está bem sobrecarregado e, como resultado, demora 11 segundos para passar uma requisição de uma fila para um pool de threads. A essa altura, o cliente já desistiu da requisição. Na maioria das circunstâncias, não seria inteligente para o servidor tentar tratar essa requisição, pois ele faria um trabalho para o qual não será dado crédito algum – o cliente não se importará com o trabalho feito pelo servidor depois que o tempo de espera se esgotar, pois ele já terá desistido da requisição.

Se o tratamento de uma requisição for feito em várias etapas (por exemplo, há algumas callbacks e chamadas de RPC), o servidor deverá verificar o tempo de espera restante em cada fase, antes de tentar fazer qualquer trabalho adicional na requisição. Por exemplo, se uma requisição estiver dividida nas fases de parsing, requisição ao backend e processamento, talvez faça sentido verificar se há tempo de espera suficiente para tratar a requisição antes de cada etapa.

Propagação do tempo de espera

Em vez de inventar um tempo de espera ao enviar RPCs para os backends, os servidores devem empregar a propagação de tempos de espera e a propagação do cancelamento.

Com a propagação do tempo de espera, um tempo de espera é definido no topo da pilha (por exemplo, no frontend). Toda a árvore de RPCs que emana de uma requisição inicial terá o mesmo tempo de espera absoluto. Por exemplo, se o servidor *A* selecionar um tempo de espera de 30 segundos e processar a requisição durante 7 segundos antes de enviar uma RPC ao servidor *B*, a RPC de *A* para *B* terá um tempo de espera de 23 segundos. Se o servidor *B* demorar 4 segundos para tratar a requisição e enviar uma RPC ao servidor *C*, a RPC de *B* para *C* terá um tempo de espera de 19 segundos, e assim por diante. O ideal é que cada servidor na árvore de requisição implemente a propagação do tempo de espera.

Sem a propagação do tempo de espera, o cenário a seguir poderia ocorrer:

1. O servidor *A* envia uma RPC ao servidor *B* com um tempo de espera de 10 segundos.
2. O servidor *B* demora 8 segundos para começar a processar a requisição e, então, envia uma RPC ao servidor *C*.
3. Se o servidor *B* usar a propagação de tempo de espera, ele deverá definir um tempo de espera de 2 segundos, mas suponha que ele utilize um tempo de espera fixo de 20 segundos para a RPC feita ao servidor *C*.
4. O servidor *C* retira a requisição de sua fila após 5 segundos.

Se o servidor *B* tivesse usado a propagação de tempo de espera, o servidor *C* poderia ter desistido de imediato da requisição, pois o tempo de espera de 2 segundos teria se esgotado. No entanto, nesse cenário, o servidor *C* processa a requisição achando que tem 15 segundos para gastar, mas não estará fazendo nenhum trabalho útil, pois a requisição do servidor *A* para o servidor *B* já teve seu tempo de espera esgotado.

Você pode querer reduzir um pouco o tempo de espera de saída (por exemplo, em algumas centenas de milissegundos) para levar em consideração o tempo de trânsito em rede e o pós-processamento no cliente.

Considere também definir um limite superior para os tempos de espera de saída. Você pode limitar o tempo que o servidor espera pelas RPCs de saída para backends não críticos, ou RPCs para backends que geralmente concluem sua tarefa em um prazo curto. Entretanto, certifique-se de que você entenda sua variedade de tráfego, pois, do contrário, poderá inadvertidamente fazer determinados tipos de requisições falharem o tempo todo (por exemplo, requisições com payloads grandes ou que exijam responder a uma grande quantidade de processamento).

Há algumas exceções para as quais os servidores podem querer continuar processando uma requisição após o tempo de espera ter expirado. Por exemplo, se um servidor receber uma requisição que envolva executar alguma operação de sincronização custosa e, periodicamente, gere pontos de verificação (checkpoints) dessa sincronização, seria uma boa ideia verificar o tempo de espera apenas depois de escrever o ponto de verificação, em vez de

fazer isso após a operação custosa.

Propagar cancelamentos evita um possível vazamento de RPC (RPC leakage) que ocorre se uma RPC inicial tiver um tempo de espera longo, mas as RPCs entre as camadas mais profundas da pilha tiverem tempos de espera curtos e sofrerem timeout. Ao usar uma propagação de tempo de espera simples, a RPC inicial continuará a usar recursos do servidor até que, em algum momento, sofrerá timeout, apesar de ser incapaz de fazer progressos.

Latência bimodal

Suponha que o frontend do exemplo anterior seja constituído de 10 servidores, cada um com 100 threads workers. Isso significa que o frontend tem um total de 1.000 threads de capacidade. Durante a operação normal, os frontends tratam 1.000 QPS e as requisições são concluídas em 100 ms. Isso significa que os frontends geralmente têm 100 threads workers ocupadas entre as 1.000 threads configuradas ($1.000 \text{ QPS} * 0,1 \text{ segundo}$).

Suponha que um evento faça com que 5% das requisições jamais sejam concluídas. Isso poderia ser o resultado de uma indisponibilidade de alguns intervalos de linhas do Bigtable, o que impediria que as requisições correspondentes a esse espaço de chaves (keyspace) do Bigtable fossem servidas. Como resultado, 5% das requisições atingiriam o tempo de espera, enquanto as restantes 95% demorariam os 100 ms normais.

Com um tempo de espera de 100 segundos, 5% das requisições consumiriam 5.000 threads ($50 \text{ QPS} * 100 \text{ segundos}$), mas o frontend não tem tantas threads disponíveis assim. Supondo que não haja nenhum outro efeito secundário, o frontend será capaz de tratar apenas 19,6% das requisições ($1.000 \text{ threads disponíveis} / (5.000 + 95) \text{ threads que trabalham}$), resultando em uma taxa de erros de 80,4%.

Desse modo, em vez de apenas 5% das requisições receberem um erro (aqueles que não foram concluídas por causa da indisponibilidade do espaço de chaves), a maior parte delas o receberá.

As diretrizes a seguir podem ajudar a tratar essa classe de problemas:

- Detectar esse problema pode ser muito difícil. Em particular, pode não ser claro que a latência bimodal seja a causa de uma interrupção de serviço

quando você estiver observando a latência *média*. Ao ver um aumento de latência, procure observar a *distribuição* das latências, além de ver as médias.

- Esse problema pode ser evitado se as requisições que não forem concluídas devolverem logo um erro, em vez de aguardar o tempo de espera todo se esgotar. Por exemplo, se um backend estiver indisponível, geralmente é melhor devolver um erro de imediato para esse backend em vez de consumir recursos até que ele esteja disponível. Se sua camada de RPC tiver suporte para uma opção de falha rápida, utilize-a.
- Ter tempos de espera maiores em várias ordens de magnitude do que a latência média das requisições, em geral, é ruim. No exemplo anterior, um número baixo de requisições inicialmente atingiu o tempo de espera, mas esse tempo era maior em três ordens de magnitude que a latência média normal, levando ao esgotamento de threads.
- Ao usar recursos compartilhados que possam se esgotar devido a algum espaço de chaves, considere limitar as requisições em andamento de acordo com esse espaço ou usar outros tipos de monitoração de abuso. Suponha que seu backend processe requisições para diferentes clientes que tenham desempenhos e características de requisições extremamente diferentes. Você pode considerar permitir que apenas 25% de suas threads sejam ocupadas por qualquer cliente único para ser justo diante de uma carga intensa causada por um único cliente que esteja se comportando mal.

Inicialização lenta e caching frio

Com frequência, os processos são mais lentos para responder às requisições logo depois de uma inicialização do que em um estado estável. Essa lentidão pode ser causada por um dos seguintes fatores, ou ambos:

Inicialização necessária

Estabelecer conexões ao receber a primeira conexão que exija um dado backend.

Melhorias no desempenho do runtime em algumas linguagens, particularmente em Java

Compilação Just-In-Time, otimização de hotspot e carga de classes diferida. De modo semelhante, alguns binários são menos eficientes quando os caches não estão preenchidos. Por exemplo, no caso de alguns dos serviços do Google, a maioria das requisições é servida a partir de caches, de modo que as requisições que não acham os dados em cache são significativamente mais custosas. Em operação estável, com um cache aquecido, somente alguns “erros” de cache (cache misses) ocorrem, mas quando o cache está totalmente vazio, 100% das requisições terão custo alto. Outros serviços podem utilizar caches para manter o estado de um usuário em RAM. Isso pode ser feito por meio de uma associação forte ou fraca entre proxies reversos e frontends de serviços.

Se o serviço não estiver provisionado para tratar requisições com um cache frio, correrá um risco maior de interrupções de serviço e deverá executar alguns passos para evitá-las.

Os cenários a seguir podem levar a um cache frio:

Ativação de um novo cluster

Um cluster recentemente adicionado terá um cache vazio.

Fazer um cluster voltar ao serviço após uma manutenção

O cache poderá estar desatualizado.

Reinicializações

Se uma tarefa com um cache reiniciou recentemente, preencher seu cache demorará um pouco. Talvez valha a pena passar o caching de um servidor para um binário separado como o memcache, que também permite compartilhamento de cache entre vários servidores, apesar do custo de introduzir outra RPC e um pouco de latência adicional.

Se o caching tiver um efeito significativo no serviço⁷, você poderá usar uma ou algumas das estratégias a seguir:

- Superprovisionamento do serviço. É importante observar a distinção entre um cache de latência *versus* um cache de capacidade: quando um cache

de latência é empregado, o serviço é capaz de sustentar a carga esperada com um cache vazio, mas um serviço que utilize um cache de capacidade não poderá sustentar a carga esperada com um cache vazio. Os proprietários dos serviços devem estar vigilantes em relação à adição de caches em seus serviços e garantir que quaisquer caches novos sejam de latência ou estejam suficientemente bem planejados para funcionar como caches de capacidade, de modo seguro. Às vezes, caches são adicionados a um serviço para melhorar o desempenho, porém, na verdade, acabam se transformando em dependências fortes.

- Empregue técnicas gerais de prevenção contra falhas em cascata. Em particular, os servidores devem rejeitar requisições quando estiverem sobrecarregados ou entrar em modo de degradação, e testes devem ser feitos para ver como o serviço se comportará após eventos como uma reinicialização grande.
- Ao adicionar carga a um cluster, aumente-a aos poucos. A taxa inicialmente baixa de requisições aquecerá o cache; depois que ele estiver aquecido, mais tráfego poderá ser adicionado. É uma boa ideia garantir que todos os clusters tratem uma carga nominal e que os caches sejam mantidos aquecidos.

Sempre desça na pilha

No serviço Shakespeare usado como exemplo, o frontend conversa com um backend que, por sua vez, conversa com a camada de armazenagem. Um problema que se manifeste na camada de armazenagem pode provocar problemas aos servidores com os quais ela conversa, mas corrigir a camada de armazenagem em geral fará a correção tanto das camadas de backend quanto de frontend.

Entretanto, suponha que os backends se comuniquem entre si. Por exemplo, os backends podem fazer proxy de requisições uns aos outros para mudar o dono de um usuário quando a camada de armazenagem não puder servir a uma requisição. Essa comunicação intracamada pode ser problemática por diversos motivos:

- A comunicação é suscetível a um deadlock distribuído. Os backends

podem usar o mesmo pool de threads para esperar RPCs enviadas a backends remotos que estejam simultaneamente recebendo requisições de backends remotos. Suponha que o pool de threads do backend *A* esteja cheio. O backend *B* envia uma requisição ao backend *A* e utiliza uma thread no backend *B* até que o pool de threads do backend *A* tenha espaço. Esse comportamento pode fazer a saturação do pool de threads se espalhar.

- Se a comunicação intracamada aumentar em resposta a algum tipo de falha ou de condição de carga intensa (por exemplo, redistribuição de carga que seja mais ativa quando houver carga intensa), a comunicação intracamada poderá mudar rapidamente de um modo de requisição intracamada baixa para alta quando a carga aumentar o suficiente.

Por exemplo, suponha que um usuário tenha um backend principal e um backend hot standby secundário predeterminado em um cluster diferente que possa assumir o usuário. O backend principal faz proxies de requisições ao backend secundário como resultado de erros da camada mais baixa ou em resposta a uma carga intensa no mestre. Se todo o sistema estiver sobrecarregado, um proxying do principal para o secundário provavelmente aumentará e acrescentará mais carga ainda ao sistema por causa do custo adicional de parsing e espera pela requisição enviada ao secundário, no backend principal.

- Conforme a criticidade da comunicação entre camadas, reiniciar o sistema pode se tornar mais complexo.

Em geral, é melhor evitar comunicações intracamada – isto é, possíveis ciclos no caminho de comunicação – no caminho da requisição do usuário. Em vez disso, faça o cliente fazer a comunicação. Por exemplo, se um frontend conversa com um backend, mas supõe o backend errado, o backend não deve fazer proxy para o backend correto. Em vez disso, o backend deve dizer ao frontend para fazer uma nova tentativa de sua requisição no backend correto.

Condições para disparo de falhas em cascata

Quando um serviço é suscetível a falhas em cascata, há vários distúrbios possíveis que podem iniciar um efeito dominó. Esta seção identifica alguns dos fatores que disparam falhas em cascata.

Morte de processo

Algumas tarefas do servidor podem morrer, reduzindo o volume de capacidade disponível. As tarefas podem morrer por causa de uma Query of Death (uma RPC cujo conteúdo dispare uma falha no processo), problemas de cluster, falhas de asserção ou uma série de outros motivos. Um evento muito pequeno (por exemplo, algumas falhas ou tarefas reescalonadas em outras máquinas) pode derrubar um serviço que esteja prestes a cair.

Atualizações de processos

Implantar uma nova versão de binário ou atualizar sua configuração podem iniciar uma falha em cascata se um grande número de tarefas for afetado simultaneamente. Para evitar esse cenário, leve em consideração um overhead necessário de capacidade quando configurar a infraestrutura de atualização de um serviço ou faça uma implantação fora do horário de pico. Ajustar dinamicamente o número de atualizações de tarefas em andamento com base no volume de requisições e na capacidade disponível pode ser uma abordagem funcional.

Novos rollouts

Um novo binário, mudanças de configuração ou uma mudança na pilha da infraestrutura subjacente podem resultar em mudanças nos perfis das requisições, no uso e limite dos recursos, nos backends ou em uma série de outros componentes do sistema capazes de disparar uma falha em cascata.

Durante uma falha em cascata, geralmente verificar mudanças recentes e considerar revertê-las é uma atitude sábia, principalmente se essas alterações afetaram a capacidade ou modificaram o perfil das requisições.

Seu serviço deve implementar algum tipo de logging de mudanças, que pode ajudar a identificar rapidamente as alterações recentes.

Crescimento orgânico

Em muitos casos, uma falha em cascata não é disparada por uma mudança específica no serviço, mas porque um crescimento no uso não foi acompanhado por um ajuste na capacidade.

Mudanças, drenagens e desativações planejadas

Se seu serviço estiver hospedado em vários lugares, parte de sua capacidade poderá estar indisponível devido a manutenções ou interrupções de serviço em um cluster. De modo semelhante, uma das dependências críticas do serviço pode sofrer drenagem, resultando em uma redução de capacidade para o serviço de upstream por causa de dependências da drenagem, ou um aumento de latência por causa da necessidade de enviar as requisições a um cluster mais distante.

Mudanças no perfil das requisições

Um serviço de backend pode receber requisições de clusters diferentes porque um serviço de frontend desviou seu tráfego devido a alterações na configuração de distribuição de carga, mudanças no tipo de tráfego ou por causa de um cluster cheio. Além disso, o custo médio para tratar um payload individual pode ter mudado por causa de alterações no código do frontend ou de configuração. De modo semelhante, os dados tratados pelo serviço podem ter mudado organicamente devido a um aumento de uso ou um uso diferente por parte dos usuários existentes: por exemplo, tanto o número quanto o tamanho das imagens *por usuário* em um serviço de armazenagem de fotos tendem a aumentar com o tempo.

Limites de recursos

Alguns sistemas operacionais de cluster permitem ter folga nos recursos. A CPU é um recurso intercambiável; com frequência, algumas máquinas têm um pouco de folga de CPU disponível, o que oferece uma espécie de rede de proteção contra picos de uso de CPU. A disponibilidade dessa folga de CPU difere entre células, e também entre máquinas na célula.

Depender dessa folga de CPU como sua rede de proteção é perigoso. Sua

disponibilidade é totalmente dependente do comportamento de outros jobs no cluster, portanto ela pode desaparecer a qualquer instante. Por exemplo, se uma equipe inicia um MapReduce que consuma muita CPU e que seja escalonado em muitas máquinas, o volume agregado de folga de CPU pode repentinamente diminuir e disparar condições de starvation de CPU para jobs não relacionados a esse MapReduce. Ao realizar testes de carga, garanta que você permanecerá dentro dos limites de recursos com os quais se comprometeu.

Testes para falhas em cascata

As maneiras específicas pelas quais um serviço falhará podem ser muito difíceis de prever desde o início. Esta seção discute as estratégias de testes que podem detectar se os serviços são suscetíveis a falhas em cascata.

Teste seu serviço para determinar como ele se comportará com carga intensa para estar confiante de que ele não entrará em uma falha em cascata em diversas circunstâncias.

Teste até falhar, e um pouco além

Entender o comportamento do serviço em condições de carga intensa 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 impulsionar o processo de depuração pelos engenheiros de plantão quando uma emergência ocorrer.

Faça testes de carga nos componentes até eles falharem. À medida que a carga aumentar, em geral, um componente tratará as requisições com sucesso até atingir um ponto em que não poderá mais tratar requisições. Nesse momento, o ideal é que o componente comece a servir erros ou resultados degradados em resposta à carga adicional, mas não reduza significativamente a taxa com que trata requisições com sucesso. Um componente que seja altamente suscetível a uma falha em cascata começará a falhar ou a servir uma alta taxa de erros quando ficar sobrecarregado; por outro lado, um componente com um bom design será capaz de rejeitar algumas requisições e

sobreviverá.

Testes de carga também revelam onde está o ponto de ruptura – um conhecimento que é fundamental para o processo de planejamento de capacidade. Ele permite testar regressões, fazer o provisionamento de limites no pior caso e conduzir negociações entre utilização e margens de segurança.

Por causa dos efeitos de caching, aumentar gradualmente a carga pode produzir resultados diferentes em relação a aumentá-la de imediato para os níveis esperados. Desse modo, considere testar padrões de cargas tanto graduais quanto repentinhas.

Você também deve testar e compreender como o componente se comporta à medida que retorna à carga nominal após ter sido forçado para muito além dessa carga. Um teste desse tipo pode responder a perguntas como:

- Se um componente entrar em modo de degradação quando sujeito a carga intensa, ele será capaz de sair desse modo sem intervenção humana?
- Se alguns servidores falharem com carga intensa, de quanto a carga deve ser reduzida para que o sistema se estabilize?

Se você estiver fazendo testes de carga em um serviço com estados (stateful) ou em um serviço que empregue caching, seu teste deve monitorar os estados entre as várias interações e conferir se estão corretos em condição de carga intensa que, com frequência, é quando bugs de concorrência sutis ocorrem.

Tenha em mente que componentes individuais podem ter pontos de ruptura diferentes, portanto faça testes de carga em cada componente de modo separado. Você não saberá com antecedência qual componente poderá atingir seu limite antes, mas vai querer saber como o seu sistema se comportará quando isso ocorrer.

Se você acredita que seu sistema tem proteções adequadas contra a sobrecarga, considere fazer testes de falha em uma pequena parte da produção a fim de descobrir o ponto em que os componentes de seu sistema falharão em uma condição de tráfego real. Esses limites podem não se refletir de modo apropriado em um tráfego de teste de carga sintético, portanto testes com tráfego real podem oferecer resultados mais realistas que os testes de carga, porém com o risco de causar problemas visíveis ao usuário. Tome

cuidado quando fizer testes com tráfego real: certifique-se de que você tenha capacidade extra disponível caso suas proteções automáticas não funcionem e você precise fazer um failover manual. Você pode considerar alguns dos seguintes testes em produção:

- Reduzir o número de tarefas de forma rápida ou lenta com o tempo, para além dos padrões de tráfego esperados.
- Perder a capacidade de um cluster rapidamente.
- Fazer vários backends atuarem como black holes.

Teste clientes populares

Entenda como os maiores clientes usam o seu serviço. Por exemplo, você vai querer saber se os clientes:

- São capazes de enfileirar o trabalho enquanto o serviço está inativo.
- Utilizam backoff exponencial aleatório quando houver erros.
- São vulneráveis a fatores externos que disparem um grande volume de carga (por exemplo, uma atualização de software disparada externamente pode limpar o cache de um cliente offline).

Conforme o seu serviço, você poderá ou não estar no controle de todo o código de cliente que conversa com o seu serviço. No entanto, compreender como os maiores clientes que interagem com o seu serviço se comportarão continua sendo uma boa ideia.

Os mesmos princípios se aplicam a grandes clientes internos. Encene falhas de sistema com os maiores clientes para ver como eles reagem. Pergunte aos clientes internos como eles acessam o seu serviço e quais são os mecanismos que eles utilizam para tratar falhas de backend.

Teste backends não críticos

Teste seus backends não críticos e certifique-se de que sua indisponibilidade não interfira nos componentes críticos de seu serviço.

Por exemplo, suponha que seu frontend tenha backends críticos e não críticos. Com frequência, uma dada requisição inclui tanto componentes

críticos (por exemplo, resultados de consultas) quanto componentes não críticos (por exemplo, sugestões para ortografia). Suas requisições poderão ficar significativamente mais lentas e consumir recursos à espera dos backends não críticos terminarem suas tarefas.

Além de testar o comportamento quando o backend não crítico estiver indisponível, teste como o frontend se comportará se o backend não crítico não responder (por exemplo, se estiver se comportando como um black hole para as requisições). Os backends considerados não críticos ainda podem causar problemas nos frontends quando as requisições tiverem tempos de espera longos. O frontend não deve começar a rejeitar várias requisições, ficar sem recursos nem servir com alta latência quando um backend não crítico se comportar como um black hole.

Passos imediatos para tratar falhas em cascata

Depois de ter identificado que seu serviço está passando por uma falha em cascata, você poderá usar algumas estratégias diferentes para remediar a situação – e, é claro, uma falha em cascata é uma boa oportunidade para utilizar o seu protocolo de gerenciamento de incidentes (Capítulo 14).

Aumente os recursos

Se seu sistema estiver executando com capacidade degradada e você tem recursos ociosos, adicionar tarefas pode ser a maneira mais prática de se recuperar de uma interrupção no serviço. No entanto, se o serviço entrou em algum tipo de espiral mortal, adicionar mais recursos talvez não seja suficiente para se recuperar.

Interrompa as falhas de verificação de sanidade/mortes

Alguns sistemas de escalonamento de cluster, como o Borg, fazem uma verificação da sanidade das tarefas em um job e reiniciam as tarefas que não estejam saudáveis. Essa prática pode criar um modo de falha em que a própria verificação de sanidade deixa o serviço não saudável. Por exemplo, se metade das tarefas não for capaz de realizar nenhum trabalho porque está iniciando e a outra metade logo será encerrada porque está sobrecarregada e

está falhando nas verificações de sanidade, desabilitar temporariamente essa verificação poderá permitir que o sistema se estabilize até que todas as tarefas estejam executando.

A verificação de sanidade dos processos (“este binário está respondendo a *algo*?”) e a verificação de sanidade do serviço (“este binário é capaz de responder a *essa classe de requisições* neste momento?”) são duas operações conceitualmente distintas. A verificação de sanidade de processos é relevante ao escalonador de cluster, enquanto a verificação de sanidade do serviço é relevante ao distribuidor de carga. Fazer uma distinção clara entre os dois tipos de verificação de sanidade pode ajudar a evitar esse cenário.

Reinic peace os servidores

Se os servidores, de algum modo, estiverem travados ou não estiverem fazendo progresso, reiniciá-los pode ajudar. Tente reiniciar os servidores quando:

- os servidores Java estiverem em uma espiral da morte por causa de GC;
- algumas requisições em andamento não tiverem tempos de espera, mas estão consumindo recursos, levando-as a bloquear threads, por exemplo;
- os servidores estiverem em deadlock.

Certifique-se de que você tenha identificado a origem da falha em cascata antes de reiniciar seus servidores. Garanta que tomar essa atitude não irá simplesmente mudar a carga de lugar. Faça um teste canário dessa alteração, e faça-o de forma lenta. Suas ações poderão amplificar uma falha em cascata existente se a interrupção de serviço for, na verdade, por causa de um problema como cache frio.

Desacerte tráfego

Descartar tráfego é uma grande ferramenta, geralmente reservada para situações em que você tem uma verdadeira falha em cascata em mãos e não consegue corrigi-la por outros meios. Por exemplo, se uma carga intensa fizer com que a maioria dos servidores falhe assim que se tornam saudáveis, você poderá fazer o serviço funcionar novamente se:

1. Tratar a condição inicial de disparo (adicionando capacidade, por exemplo).
2. Reduzir a carga o suficiente para que as falhas parem. Considere ser agressivo nesse caso – se todo o serviço estiver em um ciclo de falhas, permita apenas que, digamos, 1% do tráfego passe.
3. Permitir que a maioria dos servidores se torne saudável.
4. Aumentar gradualmente a carga.

Essa estratégia permite que o cache aqueça, as conexões sejam estabelecidas etc., antes que a carga retorne aos níveis normais.

Obviamente, essa tática causará muitos danos visíveis aos usuários. O fato de você ser capaz ou não de (ou até mesmo se *deveria*) descartar tráfego indiscriminadamente depende de como o serviço está configurado. Se você tiver algum mecanismo para descartar o tráfego menos importante (por exemplo, prefetching), utilize-o primeiro.

É importante ter em mente que essa estratégia permite recuperar-se de uma interrupção de serviço causada por uma falha em cascata depois que o problema subjacente estiver resolvido. Se o problema que iniciou a falha em cascata não for corrigido (por exemplo, capacidade global insuficiente), então a falha em cascata poderá ser disparada logo depois que todo o tráfego retornar. Assim, antes de usar essa estratégia, considere corrigir a causa-raiz ou a condição que dispara a falha (ou pelo menos ache uma solução temporária). Por exemplo, se o serviço ficou sem memória e está agora em uma espiral da morte, acrescentar mais memória ou tarefas deverá ser o seu primeiro passo.

Entre no modo de degradação

Sirva resultados degradados fazendo menos trabalho ou descartando o tráfego que não seja importante. Essa estratégia deve ser planejada em seu serviço e pode ser implementada somente se você souber qual tráfego pode sofrer degradação e tiver a capacidade de diferenciar entre os vários payloads.

Elimine a carga em batch

Alguns serviços têm carga importante, mas não crítica. Considere desativar esses originadores de carga. Por exemplo, se atualizações de índice, cópias de dados ou coleta de estatísticas consumirem recursos do caminho usado para servir às requisições, considere desativar esses originadores de carga durante uma interrupção de serviço.

Elimine o tráfego ruim

Se algumas consultas estiverem gerando uma carga intensa ou falhas (por exemplo, Queries of Death), considere bloqueá-las ou eliminá-las por outros meios.

Falhas em cascata e o Shakespeare

Um documentário sobre as obras de Shakespeare é transmitido no Japão e indica explicitamente o nosso serviço Shakespeare como um excelente lugar para conduzir pesquisas adicionais. Após a transmissão, o tráfego para o nosso datacenter na Ásia tem um surto que vai além da capacidade do serviço. Esse problema de capacidade é exacerbado por uma grande atualização no serviço Shakespeare que ocorre simultaneamente nesse datacenter.

Felizmente, uma série de proteções está implantada, as quais ajudam a atenuar o potencial para falhas. O processo Production Readiness Review (Análise de Prontidão para a Produção) identificou alguns problemas que a equipe já tratou. Por exemplo, os desenvolvedores incluíram uma degradação elegante no serviço. À medida que a capacidade se torna escassa, o serviço não devolverá mais imagens juntamente com o texto, nem pequenos mapas que ilustram o lugar em que uma história acontece. Além disso, dependendo de seu propósito, não é feita uma retentativa de uma RPC que sofra timeout (por exemplo, no caso das imagens mencionadas antes), ou uma retentativa é feita com um backoff exponencial aleatório. Apesar dessas proteções, as tarefas falham uma a uma e são então reiniciadas pelo Borg, o que reduz ainda mais o número de tarefas em operação.

Como resultado, alguns gráficos no painel de controle do serviço exibem

um tom de vermelho alarmante e a SRE é acionada com um page. Em resposta, os SREs adicionam capacidade temporariamente ao datacenter asiático, aumentando o número de tarefas disponíveis ao job Shakespeare. Ao fazer isso, eles conseguem restaurar o serviço Shakespeare no cluster asiático.

Depois disso, a equipe de SRE escreve um postmortem detalhando a cadeia de eventos, o que deu certo, o que poderia ter sido feito de um modo melhor e uma série de ações para evitar que esse cenário ocorra novamente. Por exemplo, no caso de uma sobrecarga do serviço, o distribuidor de carga GSLB redirecionará parte do tráfego aos datacenters vizinhos. Além disso, a equipe de SRE ativará uma escala sem intervenção humana para que o número de tarefas aumente automaticamente com o tráfego; assim, eles não precisarão se preocupar com esse tipo de problema novamente.

Observações finais

Quando os sistemas estão sobrecarregados, é preciso abrir mão de algo para remediar a situação. Depois que um serviço ultrapassa seu ponto de ruptura, é melhor permitir que alguns erros visíveis aos usuários ou resultados de mais baixa qualidade sejam permitidos do que tentar servir a todas as requisições de forma completa. Entender em que lugar estão esses pontos de ruptura e como o sistema se comportará além desses pontos é crucial para os proprietários de serviços que queiram evitar falhas em cascata.

Sem o cuidado apropriado, algumas mudanças de sistema cujos propósitos sejam reduzir erros em segundo plano ou melhorar o estado estável podem expor o serviço a um risco maior ou a uma interrupção completa do serviço. Fazer novas tentativas em caso de falhas, distribuir a carga que estava em servidores não saudáveis, encerrar servidores não saudáveis, acrescentar caches para melhorar o desempenho ou reduzir a latência: tudo isso pode ser implementado para melhorar o funcionamento normal, porém pode aumentar a chance de causar uma falha em larga escala. Tome cuidado ao avaliar as mudanças para garantir que uma interrupção de serviço não seja trocada por

outra.

1 Veja a Wikipédia, “Positive feedback” (Feedback positivo), https://en.wikipedia.org/wiki/Positive_feedback.

2 Um watchdog frequentemente é implementado como uma thread que acorda de forma periódica para ver se houve algum trabalho feito desde a sua última verificação. Se não houve, ele supõe que o servidor está travado e o encerra. Por exemplo, requisições de um tipo conhecido podem ser enviadas ao servidor a intervalos regulares; se uma delas não for recebida ou processada conforme esperado, isso pode indicar uma falha – do servidor, do sistema enviando as requisições ou da rede intermediária.

3 N.T.: Um acerto de cache (cache hit) ocorre quando o cache é acessado e a informação desejada está presente.

4 N.T.: Um “erro” de cache (cache miss) ocorre quando o cache é acessado, mas a informação desejada não está presente.

5 Geralmente, essa não é uma boa suposição por questões geográficas; veja também a seção “Organização de jobs e dados”.

6 Um exercício instrutivo para o leitor: escreva um simulador simples e veja como a quantidade de trabalho útil que o backend pode fazer varia conforme o seu volume de sobrecarga e o número de retentativas permitidas.

7 Às vezes, você verá que uma proporção significativa da sua capacidade propriamente dita de servir é uma função de servir a partir de um cache e, se você perder o acesso a esse cache, não será capaz de servir a essa quantidade de consultas. Uma observação semelhante vale para a latência: um cache pode ajudar você a atingir as metas de latência (reduzindo o tempo médio de resposta quando a consulta puder ser servida a partir do cache) que, possivelmente, não poderiam ser atendidas sem esse cache.

CAPÍTULO 23

Administrando estados críticos: consenso distribuído para confiabilidade

Escrito por Laura Nolan

Editado por Tim Harvey

Processos falham ou podem precisar de reinicialização. Discos rígidos falham. Desastres naturais podem tirar muitos datacenters do ar em uma região. Os Site Reliability Engineers (Engenheiros de Confiabilidade de Sites) devem prever esses tipos de falha e desenvolver estratégias para manter os sistemas executando, apesar deles. Essas estratégias geralmente implicam executar esses 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, o que é uma empreitada mais difícil, com mais nuances.

Grupos de processos podem querer entrar em acordo, de modo confiável, em questões como:

- Qual processo é o líder de um grupo de processos?
- Quais são os processos que estão em um grupo?
- Uma mensagem foi enviada com sucesso a uma fila distribuída?
- O processo tem um lease ou não?
- Qual é o valor de uma dada chave em um banco de dados?

Percebemos que o consenso distribuído (distributed consensus) é eficaz para criar sistemas confiáveis e altamente disponíveis, que exijam uma visão

consistente de algum estado do sistema. O problema do consenso distribuído diz respeito a chegar a um acordo em um grupo de processos conectados por meio de uma rede de comunicações não confiável. Por exemplo, vários processos em um sistema distribuído talvez precisem ter a capacidade de compor uma visão consistente de uma parte crítica da configuração, saber se um lock distribuído está em uso ou se uma mensagem em uma fila foi processada. É um dos conceitos mais básicos do processamento distribuído e com o qual contamos para, virtualmente, qualquer serviço que oferecemos. A Figura 23.1 apresenta um modelo simples de como um grupo de processos pode ter uma visão consistente do estado do sistema por meio do consenso distribuído.

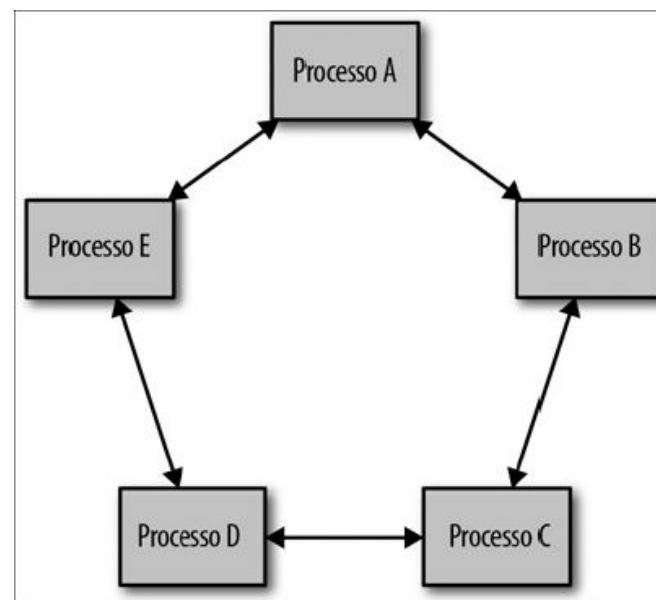


Figura 23.1 – Consenso distribuído: concordância entre um grupo de processos.

Sempre que você vir eleição de líder, estado crítico compartilhado ou locking distribuído, recomendamos usar *sistemas de consenso distribuído que tenham sido formalmente comprovados e testados de modo completo*. Abordagens informais para resolver esse problema podem levar a interrupções de serviço e, de modo mais insidioso, a problemas de consistência de dados sutis e difíceis de corrigir, que podem prolongar desnecessariamente as interrupções de serviço de seu sistema.

Teorema CAP

O teorema CAP ([Fox99], [Bre12]) afirma que um sistema distribuído não pode ter todas as três propriedades a seguir simultaneamente:

- Visões consistentes dos dados em todos os nós
- Disponibilidade de dados em todos os nós
- Tolerância a particionamentos de rede [Gil02]

A lógica é intuitiva: se dois nós não puderem se comunicar (porque a rede está particionada), então o sistema como um todo pode parar de servir parte das requisições, ou todas elas, em alguns ou em todos os nós (reduzindo, assim, a disponibilidade), ou ele pode servir às requisições como faria normalmente, o que resultaria em visões inconsistentes dos dados em cada nó.

Como os particionamentos de rede são inevitáveis (cabos se rompem, pacotes são perdidos ou sofrem atrasos por causa de congestionamento, hardwares falham, componentes de rede são erroneamente configurados etc.), entender o consenso distribuído realmente ajuda a compreender como a consistência e a disponibilidade funcionam para a sua aplicação em particular. Pressões comerciais muitas vezes exigem altos níveis de disponibilidade, e muitas aplicações precisam ter visões consistentes de seus dados.

Engenheiros de sistema e de software geralmente têm familiaridade com a semântica tradicional ACID (Atomicity, Consistency, Isolation, and Durability, ou Atomicidade, Consistência, Isolamento e Durabilidade) de bancos de dados, mas um número crescente de tecnologias de bancos de dados distribuídos oferece um conjunto diferente de semânticas conhecido como BASE (Basically Available, Soft state, and Eventual consistency, ou Basicamente Disponível, Estado soft e Consistência Eventual). Bancos de dados que suportam a semântica BASE têm aplicações úteis para determinados tipos de dados e são capazes de tratar grandes volumes de dados e transações que seriam muito mais custosos e talvez totalmente impraticáveis com bancos de dados que aceitem a semântica ACID.

A maioria desses sistemas que aceitam a semântica BASE conta com replicação multimaster, em que as escritas podem ser confiadas a diferentes processos de modo concorrente, e há algum mecanismo para resolução de

conflitos (muitas vezes tão simples quanto “o timestamp mais recente vence”). Essa abordagem geralmente é conhecida como *consistência eventual* (eventual consistency). No entanto, a consistência eventual pode levar a resultados surpreendentes [Lu15], em particular no caso de *perda de sincronização entre relógios* (clock drift) – que é inevitável em sistemas distribuídos – ou particionamento de rede [Kin15].¹

Também é difícil aos desenvolvedores projetar sistemas que funcionem bem com bancos de dados que aceitam apenas a semântica BASE. Jeff Shute [Shu13], por exemplo, afirmou que “percebemos que os desenvolvedores gastam uma fração significativa de seu tempo criando mecanismos extremamente complexos e suscetíveis a erros para lidar com a consistência eventual e tratar dados que podem estar desatualizados. Achamos que essa é uma carga inaceitável a ser colocada sobre os desenvolvedores, e que os problemas de consistência devem ser resolvidos no nível do banco de dados”.

Designers de sistemas não podem sacrificar a correção para conseguir confiabilidade ou desempenho, particularmente no que diz respeito a um estado crítico. Por exemplo, considere um sistema que trate transações financeiras: os requisitos de confiabilidade ou de desempenho não agregarão muito valor se os dados financeiros estiverem incorretos. Os sistemas precisam ser capazes de sincronizar estados críticos de modo confiável entre vários processos. Os algoritmos de consenso distribuído oferecem essa funcionalidade.

Motivação para o uso do consenso: falha na coordenação de sistemas distribuídos

Sistemas distribuídos são complexos e sutis para entender, monitorar e ter os problemas resolvidos. Os engenheiros que operam sistemas como esses muitas vezes se surpreendem com o comportamento na presença de falhas. As falhas são eventos relativamente raros, e testar sistemas nessas condições não é uma prática comum. É muito difícil pensar no comportamento do sistema durante as falhas. Os particionamentos de rede são particularmente desafiadores – um problema que pareça ter sido causado por um particionamento completo pode, na verdade, ser o resultado de:

- Uma rede muito lenta
- Algumas mensagens descartadas, mas não todas
- Throttle (estrangulamento) ocorrendo em uma direção, mas não em outra

As próximas seções apresentam exemplos de problemas que ocorreram em sistemas distribuídos do mundo real e discutem como a eleição de líder e os algoritmos de consenso distribuído poderiam ser usados para evitar esses problemas.

Estudo de caso 1: o problema do split-brain

Um determinado serviço é um repositório de conteúdos que permite colaboração entre vários usuários. Ele utiliza conjuntos de dois servidores de arquivo replicados em racks diferentes por questões de confiabilidade. O serviço deve evitar a escrita de dados simultânea nos dois servidores de arquivo de um conjunto, pois fazer isso poderia resultar em dados corrompidos (e, possivelmente, em dados irrecuperáveis).

Cada par de servidores de arquivo tem um líder e um seguidor. Os servidores monitoram um ao outro por meio de heartbeats. Se um servidor de arquivo não puder entrar em contato com seu par, ele executa um comando STONITH (Shoot The Other Node in the Head, ou Atire na Cabeça do Outro Nó) para seu nó parceiro a fim de desativá-lo e, então, assume seus arquivos. Essa prática é um método-padrão de mercado para reduzir ocorrências de split-brain, embora, como podemos observar, do ponto de vista conceitual, não seja robusto.

O que aconteceria se a rede ficasse lenta ou começasse a descartar pacotes? Nesse cenário, os servidores de arquivo teriam seus timeouts excedidos e, conforme projetado, enviariam comandos STONITH aos seus nós parceiros e passariam a ser os mestres. No entanto, alguns comandos poderão não ser entregues por causa da rede comprometida. Pares de servidores de arquivos poderão estar agora em um estado em que se espera que os dois nós estejam ativos para o mesmo recurso, ou em que ambos estejam inativos porque os dois deram e receberam comandos STONITH. Isso resulta em dados corrompidos ou indisponíveis.

O problema, nesse caso, é que o sistema está tentando resolver um problema de eleição de líder usando timeouts simples. A eleição de líder é uma reformulação do problema de consenso distribuído assíncrono, que não pode ser resolvido corretamente com o uso de heartbeats.

Estudo de caso 2: failover exige intervenção humana

Um sistema de banco de dados altamente fragmentado tem um primário para cada shard (fragmento), que é replicado sincronamente para um secundário em outro datacenter. Um sistema externo verifica a saúde dos primários e, se eles não estiverem saudáveis, promove o secundário para primário. Se o primário não puder determinar a saúde de seu secundário, ele se deixa ficar indisponível e escala para um ser humano a fim de evitar o cenário de split-brain que vimos no Estudo de Caso 1.

Essa solução não apresenta riscos de perda de dados, mas tem impactos negativos na disponibilidade. Ela também aumenta desnecessariamente a carga operacional dos engenheiros que operam o sistema, e a intervenção humana escala de modo precário. 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 maior de infraestrutura, quando os engenheiros respondendo ao problema já podem estar sobrecarregados com outras tarefas. Se a rede já está tão negativamente afetada a ponto de um sistema de consenso distribuído não conseguir eleger um mestre, um ser humano provavelmente não estará em uma posição melhor para fazer isso.

Estudo de caso 3: algoritmo de pertencimento a grupo com falha

Um sistema tem um componente que faz serviços de indexação e pesquisa. Na inicialização, os nós utilizam um protocolo de “fofoca” (gossip protocol) para descobrir uns aos outros e juntar-se ao cluster. O cluster elege um líder, que faz a coordenação. No caso de um particionamento de rede que divide o cluster, cada lado (incorretamente) elege um mestre e aceita escritas e remoções de dados, levando a um cenário de split-brain e a dados corrompidos.

O problema de determinar uma visão consistente de pertencimento ao grupo

em um grupo de processos é outra ocorrência do problema de consenso distribuído.

De fato, muitos problemas de sistemas distribuídos acabam sendo versões diferentes do consenso distribuído, incluindo eleição de mestre, pertencimento a grupo, todos os tipos de locking distribuído e leasing, enfileiramento distribuído e troca de mensagens confiáveis e manutenção de qualquer tipo de estados críticos compartilhados que devam ser visualizados de modo consistente entre um grupo de processos. Todos esses problemas devem ser resolvidos usando apenas algoritmos de consenso distribuído que tenham provado ser formalmente corretos e cujas implementações tenham sido intensamente testadas. Meios *ad hoc* de resolver esses tipos de problemas (como heartbeats e protocolos de “fofoca”) sempre apresentarão problemas de confiabilidade na prática.

Como o consenso distribuído funciona

O problema do consenso tem diversas variantes. Ao lidar com sistemas de software distribuído, estamos interessados no *consenso distribuído assíncrono*, que se aplica a ambientes com possíveis atrasos ilimitados na troca de mensagens. (O *consenso síncrono* se aplica a sistemas de tempo real, em que um hardware dedicado implica que as mensagens serão sempre passadas com garantias de tempos específicos.)

Os algoritmos de consenso distribuído podem ser *crash-fail* (que supõem que nós com falha jamais retornam ao sistema) ou *crash-recover*. Os algoritmos *crash-recover* são muito mais úteis, pois a maioria dos problemas em sistemas reais é transitória por natureza devido a rede lenta, reinicializações etc.

Os algoritmos podem lidar com falhas bizantinas (Byzantine failures) ou não bizantinas (non-Byzantine failures). Uma *falha bizantina* ocorre quando um processo passa mensagens incorretas por causa de um bug ou de uma atividade maliciosa e, em comparação com outras falhas, é custosa para tratar e encontrada com menos frequência.

Tecnicamente, resolver o problema do consenso distribuído assíncrono em

um tempo limitado é impossível. Como comprovado pelo *resultado da impossibilidade de FLP* [Fis85], vencedor do Prêmio Dijkstra, 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 terá réplicas saudáveis e conectividade de rede suficientes para fazer progressos de forma confiável na maior parte do tempo. Além disso, o sistema deve ter backoffs com esperas aleatórias. Essa configuração impede que retentativas provoquem efeitos em cascata e evita o problema de proponentes em disputa (dueling proposers), descrito mais adiante neste capítulo. Os protocolos garantem a segurança, e uma redundância apropriada do sistema incentiva a vitalidade.

A solução original ao problema do consenso distribuído foi o protocolo Paxos de Lamport [Lam98], mas há outros protocolos que resolvem o problema, incluindo Raft [Ong14], Zab [Jun11] e Mencius [Mao08]. O próprio Paxos tem muitas variações cujo propósito é melhorar o desempenho [Zoo14]. Em geral, elas variam apenas quanto a um único detalhe, por exemplo, atribuir um papel especial de líder a um processo para simplificar o protocolo.

Visão geral do Paxos: um protocolo de exemplo

O Paxos funciona como uma sequência de propostas que podem ou não ser aceitas por uma maioria de processos no sistema. Se uma proposta não for aceita, ela falha. Cada proposta tem um número de sequência, que impõe uma sequência rigorosa em todas as operações do sistema.

Na primeira fase do protocolo, o proponente (proposer) envia um número sequencial aos aceitadores (acceptors). Cada aceitador concordará em aceitar a proposta somente se ainda não viu uma proposta com um número sequencial maior. Os proponentes podem tentar novamente com um número sequencial maior, se for necessário. Os proponentes devem utilizar números sequenciais únicos (extraídos de conjuntos disjuntos, ou devem incorporar seus nomes de host no número sequencial, por exemplo).

Se um proponente tiver a concordância de uma maioria de aceitadores, ele pode confirmar a proposta enviando uma mensagem de commit com um

valor.

O sequenciamento rigoroso das propostas resolve qualquer problema relacionado à ordenação das mensagens no sistema. O requisito de que uma maioria deva concordar significa que dois valores diferentes não podem ser confirmados para a mesma proposta, pois haverá sobreposição de quaisquer dois conjuntos de maioria em pelo menos um nó. Os aceitadores devem escrever um registro (jurnal) em um repositório persistente sempre que concordarem em aceitar uma proposta, pois devem honrar essas garantias após uma reinicialização.

O Paxos, por si só, não é tão útil assim: tudo que ele permite fazer é chegar a um acordo sobre um valor e um número de proposta uma vez. Como apenas um quórum de nós precisa concordar com um valor, qualquer dado nó pode não ter uma visão completa do conjunto de valores para o qual houve um acordo. Essa limitação é válida para a maioria dos algoritmos de consenso distribuído.

Padrões de arquitetura de sistema para o consenso distribuído

Os algoritmos de consenso distribuído são primitivos e de baixo nível: eles simplesmente permitem que um conjunto de nós concorde com um valor, uma vez. Eles não são bem mapeados para tarefas reais de design. O que torna o consenso distribuído útil é o acréscimo de componentes de mais alto nível no sistema, como bancos de dados, repositórios de configuração, filas, locking e serviços de eleição de líder para oferecer as funcionalidades práticas de sistema que os algoritmos de consenso distribuído não tratam. O uso de componentes de alto nível reduz a complexidade para os designers do sistema. Isso também permite que algoritmos subjacentes de consenso distribuído sejam modificados, se for necessário, em resposta a alterações no ambiente em que o sistema executa ou a mudanças em requisitos não funcionais.

Muitos sistemas que utilizam algoritmos de consenso com sucesso, na verdade, fazem isso 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 impulso no mercado, pois era fácil de usar, mesmo com aplicações que não haviam sido projetadas para utilizar o consenso distribuído. O serviço Chubby ocupa um nicho semelhante no Google. Seus autores [Bur06] destacam que oferecer primitivas de consenso como um serviço, em vez de oferecê-las na forma de bibliotecas que os engenheiros possam incluir em suas aplicações, deixa os mantenedores da aplicação livres de terem que implantar seus sistemas de uma maneira que seja compatível com um serviço de consenso altamente disponível (executando o número correto de réplicas, lidando com pertencimento a grupos, tratando desempenho etc.).

Máquinas de estado replicadas confiáveis

Uma *RSM* (Replicated State Machine, ou Máquina de Estados Replicada) é um sistema que executa o mesmo conjunto de operações, na mesma ordem, em vários processos. As RSMs são o bloco de construção fundamental de componentes e serviços úteis de sistemas distribuídos, como armazenagem de dados ou de configuração, locking e eleição de líder (descrito em detalhes mais adiante).

As operações em uma RSM são ordenadas globalmente por meio de um algoritmo de consenso. Esse é um conceito eficaz: vários artigos ([Agu10], [Kir08], [Sch90]) mostram que qualquer programa determinístico pode ser implementado como um serviço replicado altamente disponível se for implementado como uma RSM.

Conforme mostra a Figura 23.2, máquinas de estado replicadas são um sistema implementado em uma camada lógica acima do algoritmo de consenso. O algoritmo de consenso lida com a concordância na sequência de operações, e a RSM executa as operações nessa ordem. Como nem todos os membros do grupo de consenso são necessariamente um membro do quórum de cada consenso, as RSMs talvez precisem sincronizar o estado com seus pares. Conforme descrito por Kirsch e Amir [Kir08], você pode usar um *protocolo de janela móvel* (sliding-window protocol) para reconciliar o estado entre processos pares em uma RSM.



Figura 23.2 – O relacionamento entre algoritmos de consenso e máquinas de estado replicadas.

Bancos de dados e repositórios de configuração replicados e confiáveis

Repositórios de dados replicados confiáveis são uma aplicação das máquinas de estado replicadas. Repositórios de dados replicados utilizam algoritmos de consenso no caminho crítico de seu trabalho. Desse modo, desempenho, throughput e escalabilidade são muito importantes nesse tipo de design. Como ocorre com os repositórios de dados implementados com outras tecnologias subjacentes, os repositórios baseados em consenso podem oferecer uma variedade de semânticas de consistência para operações de leitura, o que faz uma enorme diferença no modo como o repositório de dados escala. Essas análises de custo-benefício serão discutidas na seção “Desempenho do consenso distribuído”.

Outros sistemas (não distribuídos, baseados em consenso) muitas vezes simplesmente contam com timestamps para prover limites para o prazo de validade dos dados devolvidos. Os timestamps são altamente problemáticos em sistemas distribuídos porque é impossível garantir que os relógios estejam sincronizados entre várias máquinas. O Spanner [Cor12] trata esse problema modelando a incerteza envolvida no pior caso e deixando o processamento mais lento quando for necessário para resolver essa incerteza.

Processamento altamente disponível usando eleição de líder

A eleição de líder em sistemas distribuídos é um problema equivalente para o consenso distribuído. Serviços replicados que utilizem um único líder para

executar algum tipo específico de tarefa no sistema são muito comuns; o sistema de líder único é uma maneira de garantir exclusão mútua em um nível geral.

Esse tipo de design é apropriado quando o trabalho do líder do serviço pode ser realizado por um processo ou quando ele está fragmentado. Os designers do sistema podem implementar um serviço altamente disponível escrevendo-o como se fosse um programa simples, replicando esse processo e usando a eleição de líder para garantir que apenas um líder esteja trabalhando em qualquer instante no tempo (como mostra a Figura 23.3). Com frequência, o trabalho do líder é coordenar algum pool de workers no sistema. Esse padrão foi usado no GFS [Ghe03] (que foi substituído pelo Colossus) e no repositório de chave-valor Bigtable [Cha06].

Nesse tipo de componente, de modo diferente do repositório de dados replicado, o algoritmo de consenso não está no caminho crítico da tarefa principal que o sistema faz, portanto o throughput geralmente não é uma grande preocupação.

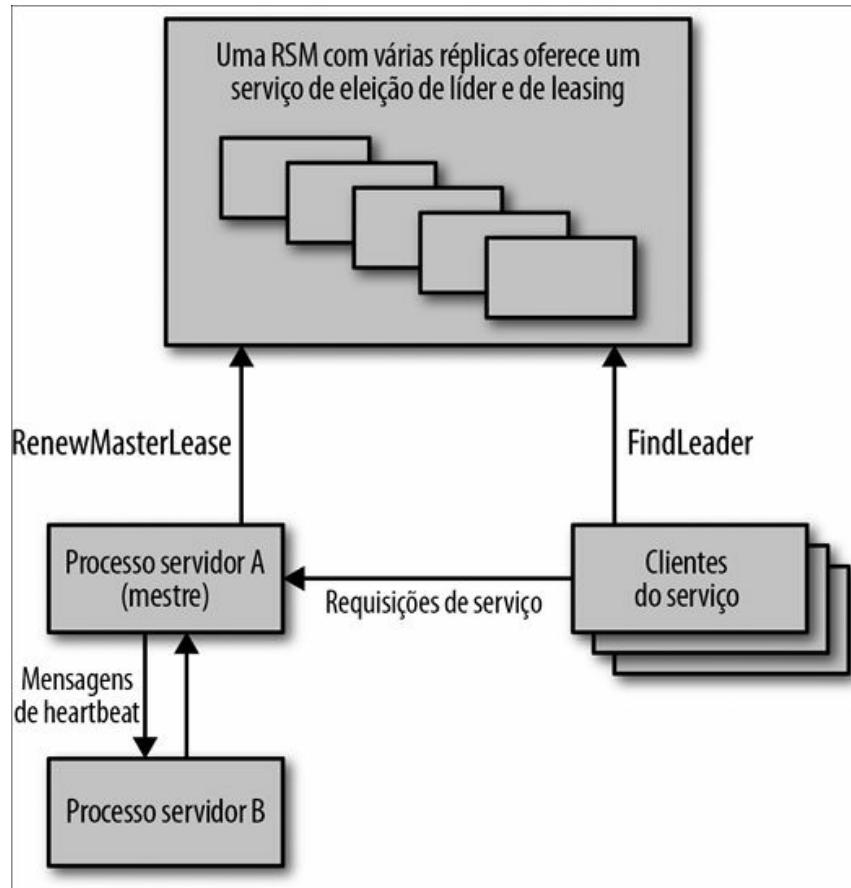


Figura 23.3 – Sistema altamente disponível usando um serviço replicado para eleição de mestre.

Coordenação distribuída e serviços de locking

Uma *barreira* em um processamento distribuído é uma primitiva que bloqueia um grupo de processos, impedindo-o de continuar até que alguma condição seja atendida (por exemplo, até que todas as partes de uma fase de um processamento estejam concluídas). O uso de uma barreira separa um processamento distribuído de modo eficiente em fases lógicas. Por exemplo, como mostra a Figura 23.4, uma barreira poderia ser usada na implementação do modelo MapReduce [Dea04] para garantir que toda a fase Map seja concluída antes que a parte Reduce do processamento prossiga.

A barreira poderia ser implementada por um único processo coordenador, porém essa implementação adiciona um único ponto de falha que, geralmente, é inaceitável. A barreira também pode ser implementada como uma RSM. O serviço de consenso Zookeeper é capaz de implementar o

padrão de barreira: veja [Hun10] e [Zoo14].

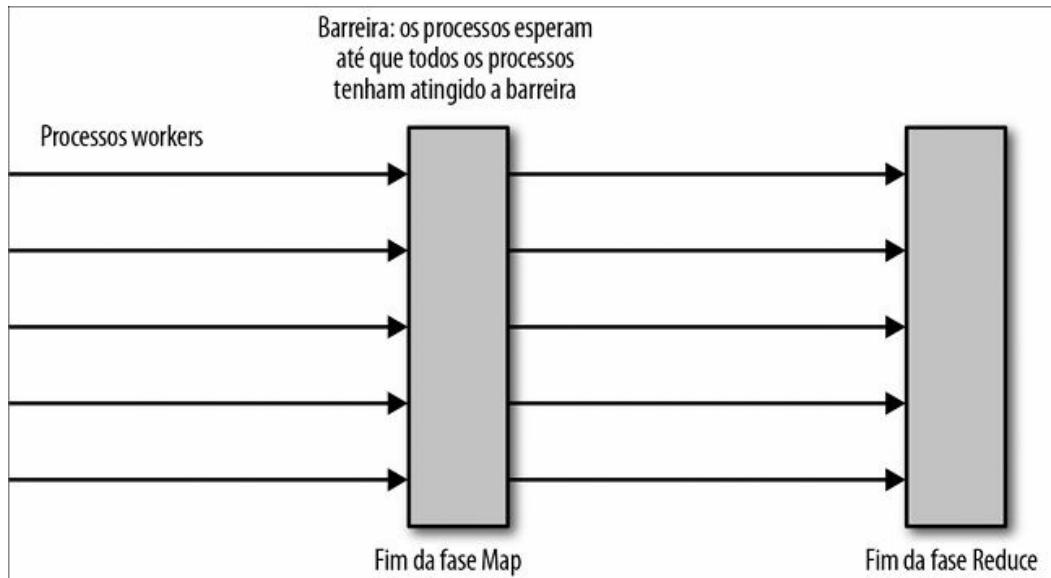


Figura 23.4 – Barreiras para coordenação de processos no tratamento do MapReduce.

Locks são outra primitiva de coordenação útil que pode ser implementada como uma RSM. Considere um sistema distribuído em que processos workers consumam atomicamente alguns arquivos de entrada e gravem resultados. Locks distribuídos podem ser usados para evitar que vários workers processem o mesmo arquivo de entrada. Na prática, é essencial usar leases renováveis com timeouts, em vez de utilizar locks indeterminados, pois fazer isso evita que os locks sejam mantidos indefinidamente por processos que falhem. O locking distribuído está além do escopo deste capítulo, mas tenha em mente que eles são uma primitiva de sistemas de baixo nível, e devem ser usados com cuidado. A maioria das aplicações deve usar um sistema de mais alto nível que ofereça transações distribuídas.

Enfileiramento distribuído e troca de mensagem confiáveis

As filas são uma estrutura de dados comum, com frequência usadas como uma maneira de distribuir tarefas entre vários processos workers.

Sistemas baseados em fila são capazes de tolerar falhas e perda de nós workers de forma relativamente simples. No entanto, o sistema deve garantir que as tarefas requisitadas sejam processadas com sucesso. Para isso, um

sistema de lease (discutido anteriormente no contexto de locks) é recomendado, em vez de fazer uma remoção imediata da fila. A desvantagem dos sistemas baseados em fila é que a perda da fila impede que o sistema como um todo funcione. Implementar a fila como uma RSM pode minimizar o risco e deixa o sistema todo muito mais robusto.

O *broadcast atômico* é uma primitiva de sistemas distribuídos em que mensagens são recebidas de forma confiável e na mesma sequência por todos os participantes. Esse é um conceito extremamente eficaz em sistemas distribuídos e muito útil no design de sistemas práticos. Há um grande número de infraestruturas de troca de mensagens do tipo publicar-inscrever (publish-subscribe) que os designers de sistema podem usar, embora nem todas ofereçam garantias de atomicidade. Chandra e Toueg [Cha96] demonstram a equivalência entre broadcast atômico e consenso.

O padrão *enfileiramento como distribuição de trabalho* (queuing-as-work-distribution), que utiliza a fila como um dispositivo de distribuição de carga, como mostra a Figura 23.5, pode ser considerado um sistema de mensagens ponto a ponto. Os sistemas de mensagens em geral também implementam uma fila do tipo publicar-inscrever, em que as mensagens podem ser consumidas por muitos clientes que se inscrevem junto a um canal ou um assunto. Nesse caso de um para muitos (one-to-many), as mensagens na fila são armazenadas como uma lista ordenada persistente. Sistemas do tipo publicar-inscrever podem ser usados por vários tipos de aplicações que exijam que os clientes se inscrevam para receber notificações de algum tipo de evento. Sistemas desse tipo também podem ser usados para implementar caches distribuídos coerentes.

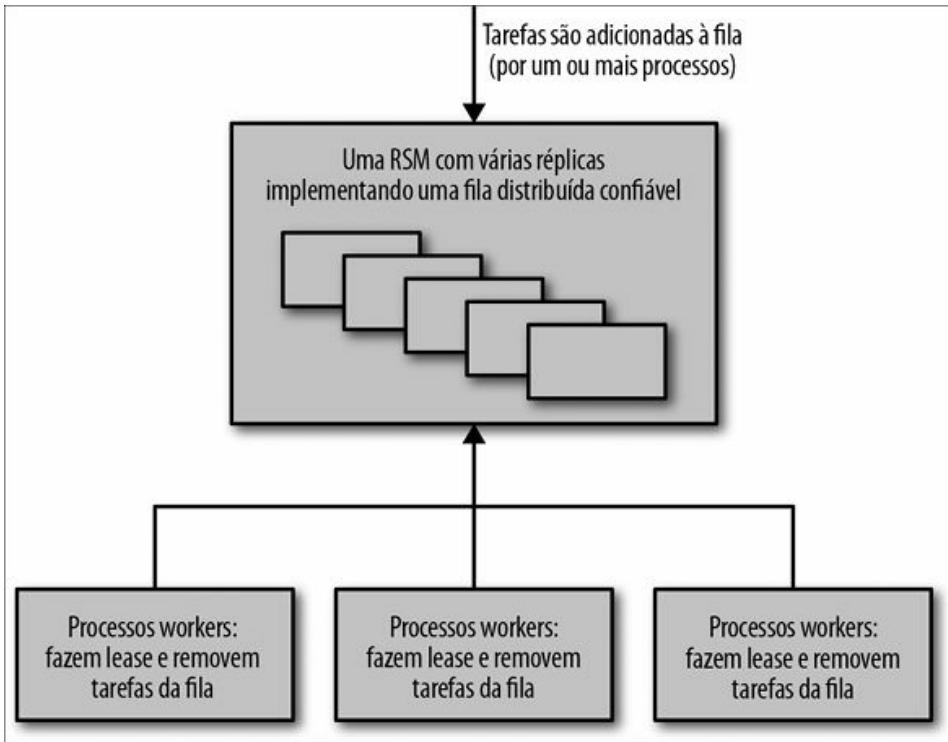


Figura 23.5 – Um sistema de distribuição de tarefas baseado em fila, usando um componente confiável de enfileiramento baseado em consenso.

Sistemas de enfileiramento e mensagens com frequência exigem um throughput excelente, mas não precisam de uma latência extremamente baixa (pois raramente estão diretamente voltados aos usuários). No entanto, latências muito altas em um sistema como o que acabamos de descrever, que tem vários workers requisitando tarefas de uma fila, podem se tornar um problema se o percentual do tempo de processamento para cada tarefa crescer de forma significativa.

Desempenho do consenso distribuído

A sabedoria convencional em geral considera os algoritmos de consenso muito lentos e custosos para muitos sistemas que exijam throughput alto e baixa latência [Bol11]. Essa concepção simplesmente não é verdadeira – embora as implementações possam ser lentas, há uma série de truques que podem melhorar o desempenho. Os algoritmos de consenso distribuído estão no núcleo de muitos sistemas críticos do Google, descritos em [Ana13], [Bur06], [Cor12] e [Shu13], e foi comprovado que são extremamente

eficazes na prática. A escala do Google não é uma vantagem nesse caso: na verdade, nossa escala é mais uma desvantagem, pois introduz dois desafios importantes: nossos conjuntos de dados tendem a ser grandes e nossos sistemas estão distribuídos em extensas regiões geográficas. Conjuntos maiores de dados multiplicados por várias réplicas representam custos significativos de processamento, e distâncias geográficas maiores aumentam a latência entre as réplicas, o que, por sua vez, reduz o desempenho.

Não há um consenso distribuído e um algoritmo de replicação de máquinas de estado “melhores” para o desempenho, pois ele é dependente de uma série de fatores relacionados a carga de trabalho, objetivos de desempenho do sistema e o modo como o sistema deve ser implantado.² Embora algumas das próximas seções apresentem a pesquisa, com o objetivo de melhorar a compreensão do que é possível conseguir com o consenso distribuído, muitos dos sistemas descritos estão disponíveis e em uso atualmente.

As *cargas de trabalho* podem variar de diversas 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 quanto a:

- Throughput: o número de propostas feitas por unidade de tempo com a carga no pico.
- Tipo de requisições: a proporção das operações que mudam de estado.
- A semântica de consistência necessária para operações de leitura.
- Tamanhos das requisições, se o tamanho do payload de dados puder variar.

As estratégias de implantação também variam. Por exemplo:

- A implantação é em uma área local ou ampla?
- Quais tipos de quórum são usados e onde está a maioria dos processos?
- O sistema usa sharding (fragmentação), pipelining e batching?

Muitos sistemas de consenso utilizam um processo líder distinto e exigem que todas as requisições sejam encaminhadas a esse nó especial. Como mostra a Figura 23.6, como resultado o desempenho do sistema, conforme percebido pelos clientes em localizações geográficas distintas, pode variar de

forma considerável, simplesmente porque nós mais distantes têm tempos de ida e volta mais demorados até o processo líder.

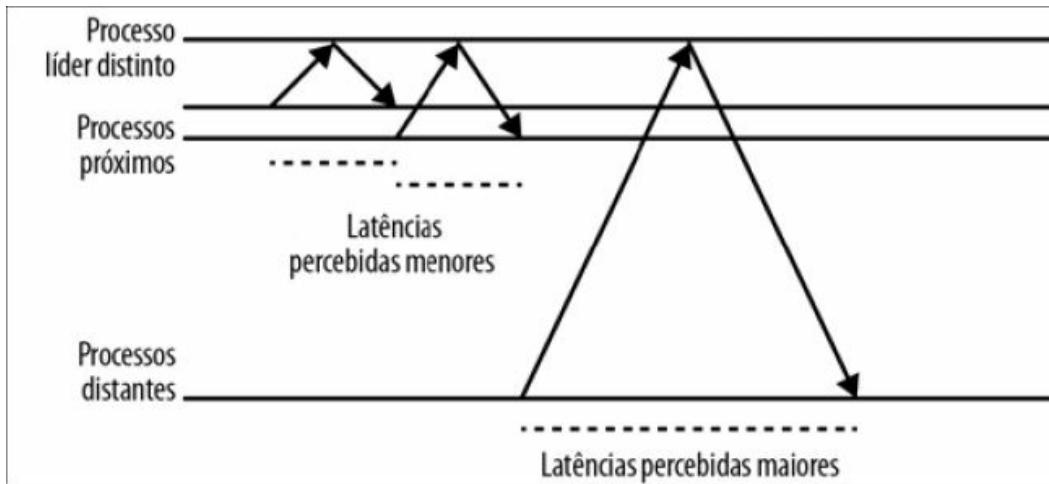


Figura 23.6 – O efeito da distância de um processo servidor na latência percebida pelo cliente.

Multi-Paxos: fluxo de mensagens detalhado

O protocolo Multi-Paxos utiliza um *processo de líder forte*: a menos que um líder ainda não tenha sido eleito ou uma falha ocorra, apenas uma viagem de ida e volta do proponente para um quórum de aceitadores é necessária para chegar a um consenso. Utilizar um processo de líder forte é ideal em termos do número de mensagens a ser passado e é característico de muitos protocolos de consenso.

A Figura 23.7 mostra um estado inicial, com um novo proponente executando a primeira fase Prepare/Promise do protocolo. Executar essa fase estabelece uma nova visão (view) numerada, ou termo de líder. Em execuções subsequentes do protocolo, enquanto a visão permanecer a mesma, a primeira fase será desnecessária, pois o proponente que determinou a visão pode simplesmente enviar mensagens Accept, e o consenso será alcançado quando um quórum de respostas for recebido (incluindo o próprio proponente).

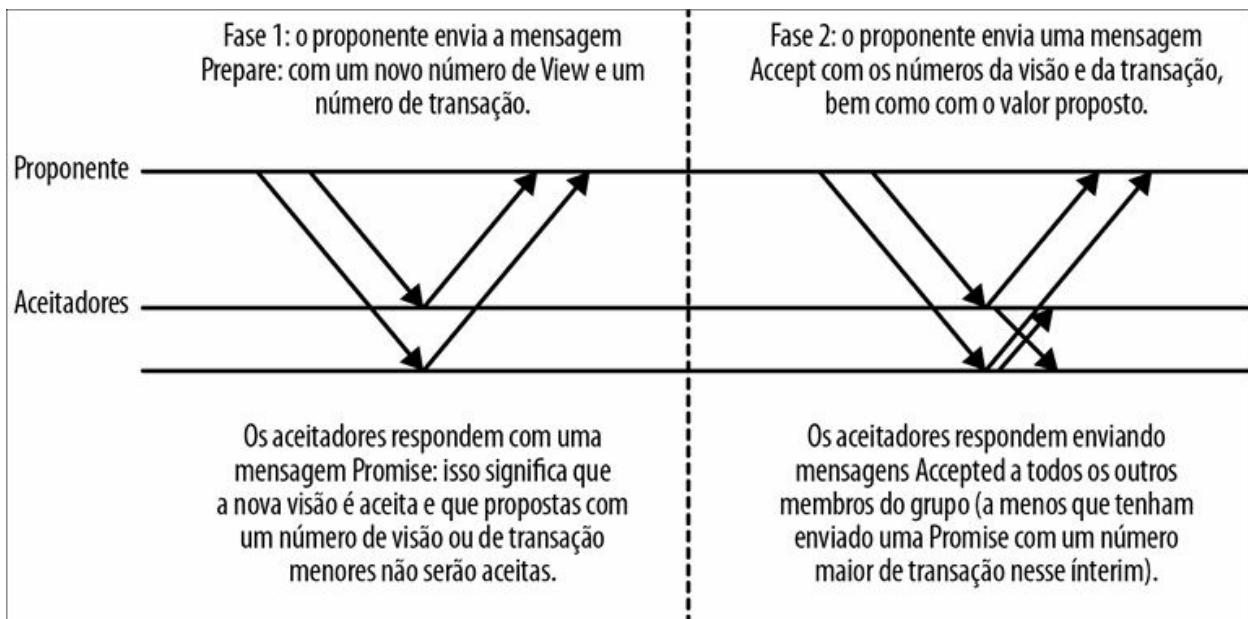


Figura 23.7 – Fluxo de mensagens básico no Multi-Paxos.

Outro processo do grupo pode assumir o papel de proponente para propor mensagens a qualquer momento, mas mudar o proponente tem um custo para o desempenho. É necessária uma viagem de ida e volta extra para executar a Fase 1 do protocolo, mas, acima de tudo, pode haver uma situação de *proponentes em disputa*, em que as propostas interrompem umas às outras repetidamente e nenhuma delas pode ser aceita, como mostra a Figura 23.8. Como esse cenário é uma forma de livelock, ele pode continuar indefinidamente.

Todos os sistemas práticos de consenso tratam esse problema de colisões, geralmente elegendo um processo proponente, que faz todas as propostas no sistema, ou usando um proponente rotativo que aloca posições reservadas particulares a cada processo para suas propostas.

Para sistemas que usam um processo líder, o processo de eleição de líder deve ser ajustado com cuidado para achar um equilíbrio entre a indisponibilidade do sistema resultante de não haver um líder presente e o risco de proponentes em disputa. É importante implementar os timeouts corretos e as estratégias de backoff. Se vários processos determinarem que não há um líder e todos tentarem se tornar líderes ao mesmo tempo, então é provável que nenhum dos processos tenha sucesso (novamente, proponentes em disputa). Introduzir aleatoriedade é a melhor abordagem. Raft [Ong14],

por exemplo, apresenta um método bem planejado para abordar o processo de eleição de líder.

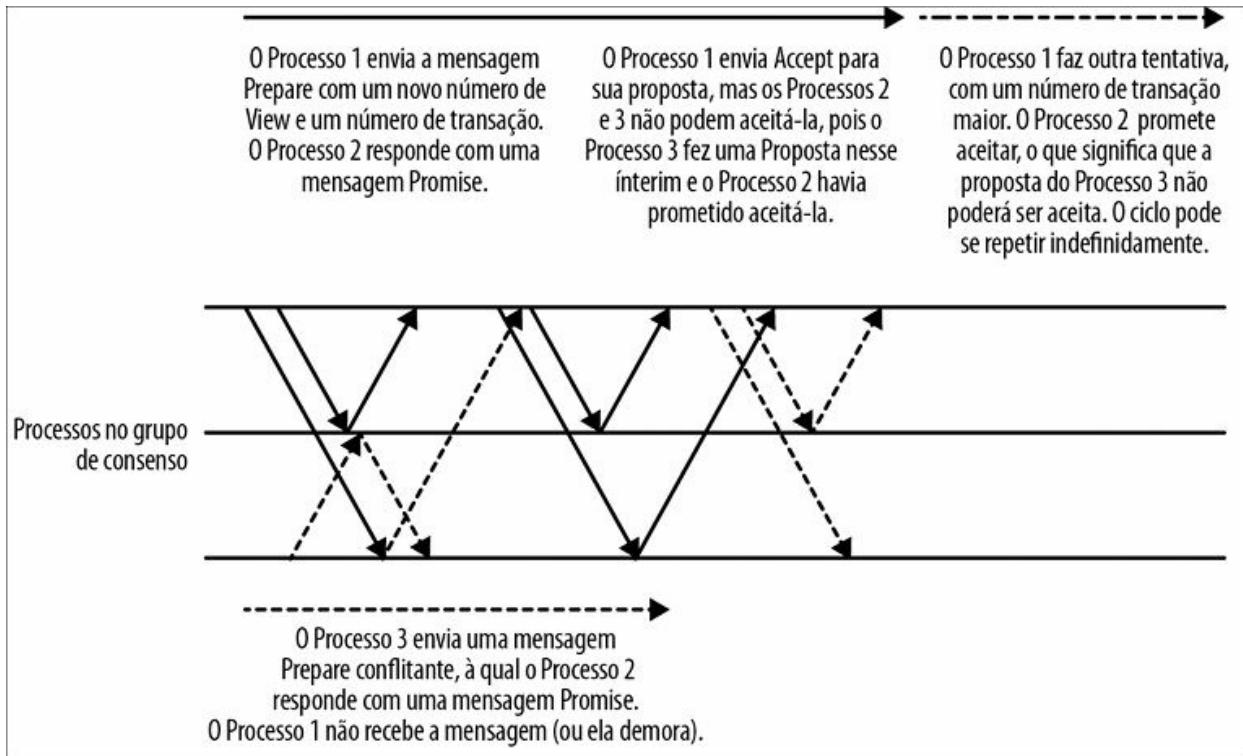


Figura 23.8 – Proponentes em disputa no Multi-Paxos.

Escalando cargas de trabalho intensas em leitura

Escalar cargas de trabalho de leitura muitas vezes é crucial, pois muitas cargas de trabalho têm bastante leitura. Repositórios de dados replicados têm a vantagem de os dados estarem disponíveis em vários lugares, o que significa que se uma consistência forte não for necessária para todas as leituras, os dados poderão ser lidos de *qualquer réplica*. Essa técnica de leitura de réplicas funciona bem para determinadas aplicações, como o sistema Photon do Google [Ana13], que utiliza consenso distribuído para coordenar o trabalho de vários pipelines. O Photon utiliza uma operação atômica de comparação-e-definição para modificação de estado (inspirada nos registros atômicos), que deve ser absolutamente consistente; porém, as operações de leitura podem ser servidas a partir de qualquer réplica, pois dados antigos resultam em trabalho extra, mas não em resultados incorretos [Gup15]. O custo-benefício vale a pena.

Para garantir que os dados lidos estejam atualizados e sejam consistentes com qualquer mudança feita antes de a leitura ser realizada, é necessário executar uma das ações a seguir:

- Fazer uma operação de consenso somente de leitura.
- Ler dados de uma réplica que se garanta ser a mais atualizada. Em um sistema que utilize um processo líder estável (como muitas implementações de consenso distribuído fazem), o líder pode oferecer essa garantia.
- Usar leases de quórum, em que algumas réplicas recebem um lease sobre todos ou parte dos dados do sistema, permitindo leituras locais fortemente consistentes à custa da execução de algumas escritas. Essa técnica será discutida em detalhes na próxima seção.

Leases de quórum

Leases de quórum [Mor14] são uma otimização de desempenho de consenso distribuído recentemente desenvolvida cujo propósito é reduzir a latência e aumentar o throughput de operações de leitura. Conforme mencionamos antes, no caso do Paxos clássico e da maior parte dos demais protocolos de consenso distribuído, fazer uma leitura fortemente consistente (isto é, uma em que se garanta que temos a visão mais atualizada do estado) exige uma operação de consenso distribuído que leia de um quórum de réplicas, ou uma réplica líder estável que se garanta que tenha visto todas as operações recentes de mudança de estado. Em muitos sistemas, as operações de leitura superam enormemente as escritas; portanto, contar com uma operação distribuída ou com uma única réplica prejudica a latência e o throughput do sistema.

A técnica de leasing de quórum simplesmente concede um lease de leitura para algum subconjunto do estado de um repositório de dados replicado a um quórum de réplicas. O lease serve para um período de tempo específico (geralmente breve). Qualquer operação que altere o estado desses dados deve ser reconhecida por todas as réplicas no quórum de leitura. Se alguma dessas réplicas se tornar indisponível, os dados não poderão ser modificados até que o lease expire.

Os leases de quórum são particularmente úteis para cargas de trabalho com bastante leitura, em que as leituras de subconjuntos particulares de dados estão concentradas em uma única região geográfica.

Desempenho do consenso distribuído e latência de rede

Sistemas de consenso enfrentam duas limitações físicas importantes relacionadas ao desempenho quando efetuam mudanças de estado. Uma é o tempo de ida e volta na rede e a outra é o tempo que demora para escrever os dados em um repositório persistente, que será analisado mais adiante.

Os tempos de ida e volta na rede variam enormemente de acordo com a localização da origem e do destino, e sofrem impactos tanto da distância física entre a origem e o destino quanto do volume de congestionamento da rede. Em um único datacenter, os tempos de ida e volta entre máquinas devem ser da ordem de um milissegundo. Um RTT (Round-Trip-Time, ou Tempo de Ida e Volta) típico 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 assíncrono líder-seguidor [Bol11], como usado em vários bancos de dados tradicionais para replicação. No entanto, muitas das vantagens de disponibilidade dos sistemas de consenso distribuído exigem que as réplicas estejam “distantes” umas das outras para estarem em diferentes domínios de falha.

Muitos sistemas de consenso usam TCP/IP como seu protocolo de comunicação. O TCP/IP é orientado a conexão e oferece algumas garantias fortes de confiabilidade quanto à ordenação de mensagens em FIFO. No entanto, estabelecer uma nova conexão TCP/IP exige uma viagem de ida e volta na rede para executar o handshake de três vias (three-way handshake) que estabelece uma conexão antes de qualquer dado poder ser enviado ou recebido. O início lento do TCP/IP limita a largura de banda da conexão até que seu limite tenha sido definido. Os tamanhos de janela iniciais do TCP/IP variam de 4 a 15 KB.

O início lento do TCP/IP provavelmente não é um problema para os processos que compõem um grupo de consenso: eles estabelecerão conexões

uns com os outros e manterão essas conexões abertas para reutilização, pois estarão em comunicação frequente. No entanto, para sistemas com um número bem elevado de clientes, talvez não seja prático que todos os clientes mantenham uma conexão persistente aberta para os clusters de consenso, pois conexões TCP/IP abertas consomem alguns recursos (por exemplo, descritores de arquivo), além de gerar tráfego keepalive. Esse overhead pode ser um problema importante para as aplicações que usem repositórios de dados altamente fragmentados baseados em consenso, contendo milhares de réplicas e com um número maior ainda de clientes. Uma solução é usar um pool de proxies regionais, como mostra a Figura 23.9, que mantém conexões TCP/IP persistentes com o grupo de consenso a fim de evitar o overhead de estabelecer conexões em longas distâncias. Os proxies também podem ser uma boa maneira de encapsular a fragmentação e as estratégias de distribuição de carga, bem como a descoberta de membros e líderes de clusters.

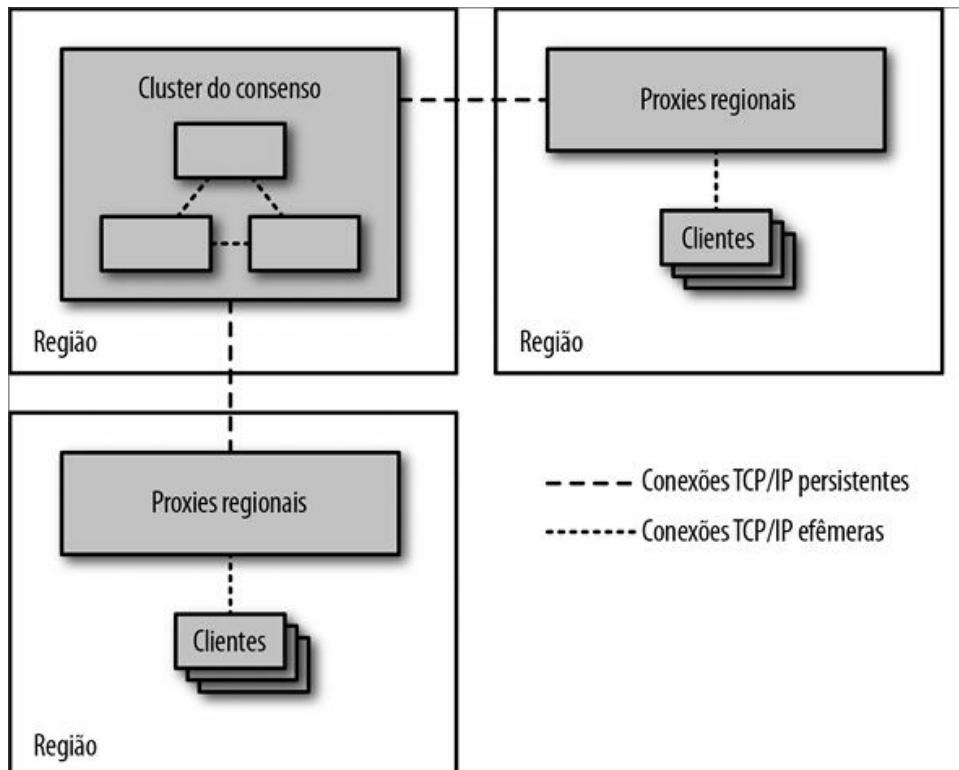


Figura 23.9 – Usando proxies para reduzir a necessidade de os clientes estabelecerem conexões TCP/IP entre regiões.

Pensando no desempenho: Fast Paxos

O Fast Paxos [Lam06] é uma versão do algoritmo Paxos projetado para melhorar seu desempenho em redes de longa distância. Ao usar o Fast Paxos, cada cliente pode enviar mensagens `Propose` diretamente a cada membro de um grupo de aceitadores, em vez de fazê-lo por meio de um líder, como no Paxos Clássico ou no Multi-Paxos. A ideia é substituir um envio de mensagem paralelo do cliente a todos os aceitadores no Fast Paxos por duas operações de envio de mensagem no Paxos Clássico:

- Uma mensagem do cliente para um único proponente
- Uma operação paralela de envio de mensagem do proponente para as outras réplicas

Intuitivamente, pode parecer que o Fast Paxos deveria ser sempre mais rápido que o Paxos Clássico. Contudo, isso não é verdade: se o cliente no sistema Fast Paxos tiver um RTT (Round-Trip Time, ou Tempo de Ida e Volta) alto até os aceitadores, e esses tiverem conexões rápidas uns com os outros, substituímos N mensagens paralelas pelos links de rede mais lentos (no Fast Paxos) por uma mensagem pelo link mais lento mais N mensagens paralelas pelos links mais rápidos (Paxos Clássico). Por causa do efeito de cauda da latência, na maior parte do tempo, uma única viagem de ida e volta por um link lento com uma distribuição de latências é mais rápida que um quórum (como mostrado em [Jun07]) e, desse modo, o Fast Paxos é mais lento que o Paxos Clássico nesse caso.

Muitos sistemas reúnem várias operações em batch em uma única transação no aceitador para melhorar o throughput. Fazer os clientes atuarem como proponentes também dificulta muito mais reunir as propostas em batch. O motivo é que as propostas chegam de modo independente nos aceitadores, portanto não será possível reuni-las em batch de uma maneira consistente.

Líderes estáveis

Vimos como o 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 permite otimizações de leitura, pois o líder tem o estado mais

atualizado, mas também apresenta vários problemas:

- Todas as operações que alteram estados devem ser enviadas por meio do líder – um requisito que acrescenta latência de rede aos clientes que não estejam localizados próximos do líder.
- A banda de rede de saída do processo líder é um gargalo do sistema [Mao08], pois a mensagem Accept do líder contém todos os dados relacionados a qualquer proposta, enquanto outras mensagens contêm apenas reconhecimentos de uma transação numerada, sem payload de dados.
- Se, por acaso, o líder estiver em uma máquina com problemas de desempenho, o throughput de todo o sistema será reduzido.

Quase todos os sistemas de consenso distribuído projetados com desempenho em mente utilizam o padrão de líder estável único ou um sistema de rotação de liderança, em que cada algoritmo de consenso distribuído numerado é atribuído previamente a uma réplica (geralmente, calculando o módulo simples com o ID da transação). Algoritmos que utilizam essa abordagem incluem o Mencius [Mao08] e o Egalitarian Paxos (Paxos Igualitário) [Mor12a].

Em uma rede de longa distância, com clientes distribuídos geograficamente e réplicas do grupo de consenso localizadas razoavelmente próximas aos clientes, uma eleição de líder como essa resulta em uma latência percebida menor para os clientes, pois seu RTT de rede para a réplica mais próxima, em média, será menor que a latência para um líder arbitrário.

Batching

O batching, conforme descrito na seção “Pensando no desempenho: Fast Paxos”, aumenta o throughput do sistema, porém ainda deixa réplicas ociosas enquanto elas esperam respostas às mensagens que enviaram. As ineficiências apresentadas por réplicas ociosas podem ser resolvidas por meio de *pipelining*, que permite que várias propostas estejam em andamento ao mesmo tempo. Essa otimização é muito semelhante ao caso do TCP/IP, em que o protocolo tenta “manter o pipe cheio” usando uma abordagem de janela deslizante. O pipelining geralmente é utilizado em conjunto com o batching.

Os batches de requisições no pipeline continuam ordenados globalmente, com um número de visão e um número de transação; portanto, esse método não viola as propriedades de ordenação global exigidas para executar uma máquina de estados replicada. Esse método de otimização é discutido em [Bol11] e em [San11].

Acesso a disco

Fazer logging em um repositório persistente é necessário para que um nó, após ter falhado e retornado ao cluster, honre quaisquer compromissos anteriores relacionados às transações de consenso em andamento. No protocolo Paxos, por exemplo, os aceitadores não podem concordar com uma proposta quando já tiverem concordado com outra que tenha um número de sequência maior. Se detalhes das propostas aceitas e confirmadas não forem registrados em log em um repositório persistente, um aceitador poderá violar o protocolo se ele falhar e for reiniciado, levando o sistema a um estado inconsistente.

O tempo necessário para escrever uma entrada em um log em disco varia enormemente, dependendo do hardware ou do ambiente virtualizado em uso, mas é provável que demore entre um e vários milissegundos.

O fluxo de mensagem do Multi-Paxos foi discutido na seção “Multi-Paxos: fluxo de mensagens detalhado”, mas essa seção não mostrou em que lugar o protocolo deve fazer log das mudanças de estado em disco. Uma escrita em disco deve ocorrer sempre que um processo assuma um compromisso que ele deva honrar. Na segunda fase do Multi-Paxos, que é crítica quanto ao desempenho, esses pontos ocorrem antes que um aceitador envie uma mensagem Accepted em resposta a uma proposta, e antes de o proponente enviar a mensagem Accept, pois essa mensagem Accept também é uma mensagem Accepted implícita [Lam98].

Isso significa que a latência para uma única operação de consenso envolve o seguinte:

- Uma escrita em disco no proponente
- Mensagens paralelas aos aceitadores

- Escritas em disco paralelas nos aceitadores
- Mensagens de retorno

Há uma versão do protocolo Multi-Paxos que é conveniente para os casos em que o tempo de escrita em disco seja predominante: essa variante não considera a mensagem Accept do proponente como uma mensagem Accepted implícita. Em vez disso, o proponente escreve em disco em paralelo com os outros processos e envia uma mensagem Accept explícita. A latência, então, se torna proporcional ao tempo necessário para enviar duas mensagens e um quórum de processos executar uma escrita síncrona no disco em paralelo.

Se a latência para fazer uma pequena escrita aleatória em disco for da ordem de 10 milissegundos, a taxa das operações de consenso será limitada a aproximadamente 100 por segundo. Esses tempos pressupõem que os tempos de ida e volta na rede sejam desprezíveis e que o proponente faça seu logging em paralelo com os aceitadores.

Como já vimos antes, os algoritmos de consenso distribuído com frequência são usados como base para construir uma máquina de estados replicada. As RSMs também precisam manter logs de transações com vistas à recuperação (pelos mesmos motivos que qualquer repositório de dados). O log do algoritmo de consenso e o log de transações da RSM podem ser combinados em um único log. Combinar esses logs evita a necessidade de alternar constantemente entre escrever em dois lugares físicos diferentes no disco [Bol11], reduzindo o tempo gasto em operações de seek (busca). Os discos poderão suportar mais operações por segundo e, desse modo, o sistema como um todo poderá realizar mais transações.

Em um repositório de dados, os discos têm outros propósitos além de armazenar logs de manutenção: o estado do sistema geralmente é mantido em disco. Escritas em log devem ser descarregadas diretamente no disco, mas as escritas para mudanças de estado podem ser gravadas em um cache de memória e descarregadas depois em disco, reordenadas de modo a usar a sequência mais eficiente [Bol11].

Outra otimização possível é o batching de várias operações de cliente em uma só operação no proponente ([Ana13], [Bol11], [Cha07], [Jun11], [Mao08], [Mor12a]). Isso amortiza os custos fixos de logging em disco e a latência de

rede sobre o número maior de operações, melhorando o throughput.

Implantando sistemas baseados em consenso distribuído

As decisões mais críticas que os designers de sistema devem tomar quando implantam um sistema baseado em consenso dizem respeito ao número de réplicas a ser implantado e à localização dessas réplicas.

Número de réplicas

Em geral, sistemas baseados em consenso usando *quóruns majoritários*, isto é, um grupo de $2f + 1$ réplicas, pode tolerar f falhas (se tolerância a falhas bizantinas, em que o sistema é resistente a réplicas, devolvendo resultados incorretos, for necessária, então $3f + 1$ réplicas poderão tolerar f falhas [Cas99]). Para falhas não bizantinas, o número mínimo de réplicas que pode ser implantado é três – se duas forem implantadas, então não haverá tolerância a falhas em nenhum processo. Três réplicas podem tolerar uma falha. A maior parte do downtime de um sistema é resultado de uma manutenção planejada [Ken12]: três réplicas permitem que um sistema opere normalmente quando uma réplica estiver inativa para manutenção (supondo que as duas réplicas restantes sejam capazes de tratar a carga do sistema com um desempenho aceitável).

Se uma falha não planejada ocorrer durante uma janela de manutenção, o sistema de consenso ficará indisponível. A indisponibilidade do sistema de consenso geralmente é inaceitável, portanto cinco réplicas devem ser executadas, permitindo que o sistema funcione com até duas falhas. Nenhuma intervenção é necessariamente exigida se quatro de cinco réplicas restarem em um sistema de consenso, porém, se sobrarem três, uma ou duas réplicas adicionais devem ser acrescentadas.

Se um sistema de consenso perder muitas de suas réplicas a ponto de não conseguir formar um quórum, esse sistema, teoricamente, estará em um estado irrecuperável, pois os logs duráveis de pelo menos uma das réplicas ausentes não poderão ser acessados. Se não restar quórum, é possível que

uma decisão que tenha sido vista apenas pelas réplicas ausentes tenha sido tomada. Os administradores podem ser capazes de forçar uma alteração nos membros do grupo e adicionar novas réplicas que se sincronizarão com a réplica existente para continuar, mas sempre haverá a possibilidade de perda de dados – uma situação que deve ser evitada, se for possível.

Em um desastre, os administradores precisam decidir se devem realizar uma reconfiguração forçada como essa ou esperar um período de tempo para que as máquinas com o estado do sistema se tornem disponíveis. Quando uma decisão como essa for feita, o tratamento do log do sistema (além da monitoração) se torna crucial. Artigos teóricos com frequência destacam que o consenso pode ser utilizado para criar um log replicado, mas falham em discutir o modo de lidar com réplicas que possam falhar e se recuperar (e, desse modo, perder parte da sequência das decisões de consenso) ou se atrasar por causa de lentidão. Para manter a robustez do sistema, é importante que essas réplicas se sincronizem.

O *log replicado* nem sempre recebe a devida atenção na teoria de consenso distribuído, mas é um aspecto muito importante dos sistemas em produção. O Raft descreve um método para administrar a consistência de logs replicados [Ong14], definindo explicitamente de que modo qualquer lacuna no log de uma réplica deve ser preenchida. Se um sistema Raft com cinco instâncias perder todos os seus membros, exceto seu líder, ainda se garante que ele terá total conhecimento de todas as decisões efetuadas. Por outro lado, se a maioria dos membros ausentes incluir o líder, nenhuma garantia robusta pode ser feita em relação a quão atualizadas estão as réplicas remanescentes.

Há um relacionamento entre desempenho e o número de réplicas em um sistema que não precisam fazer parte de um quórum: uma minoria de réplicas mais lentas pode ficar para trás, permitindo que o quórum de réplicas com melhor desempenho execute mais rápido (desde que o líder tenha um bom desempenho). Se o desempenho das réplicas variar de forma significativa, então toda falha poderá reduzir o desempenho do sistema como um todo, pois outras réplicas mais lentas serão necessárias para ter um quórum. Quanto mais falhas ou réplicas atrasadas um sistema puder tolerar, maior será a probabilidade de o desempenho geral do sistema ser melhor.

A questão do custo também deve ser considerada ao administrar réplicas: cada réplica utiliza recursos de processamento custosos. Se o sistema em questão for um único cluster de processos, o custo de executar réplicas provavelmente não será uma consideração importante. No entanto, o custo das réplicas pode ser uma consideração séria para sistemas como o Photon [Ana13], que utiliza uma configuração fragmentada, em que cada fragmento (shard) é um grupo completo de processos executando um algoritmo de consenso. À medida que o número de fragmentos aumenta, o mesmo ocorre com o custo de cada réplica adicional, pois um número de processos igual ao número de fragmentos deve ser adicionado ao sistema.

Desse modo, a decisão sobre o número de réplicas em qualquer sistema é uma negociação entre os seguintes fatores:

- A necessidade de confiabilidade
- A frequência de manutenções planejadas que afetam o sistema
- Risco
- Desempenho
- Custo

Esse cálculo será diferente para cada sistema: os sistemas têm objetivos diferentes de nível de serviço para disponibilidade, algumas empresas fazem manutenção com mais regularidade do que outras e as empresas usam hardware de custo, qualidade e confiabilidade variados.

Localização das réplicas

Decisões sobre onde implantar os processos que compõem um cluster de consenso são tomadas com base em dois fatores: uma negociação entre os domínios de falha que o sistema deve tratar e os requisitos de latência do sistema. Várias questões complexas estão em cena para decidir a localização das réplicas.

Um *domínio de falha* é o conjunto de componentes de um sistema que pode se tornar indisponível como resultado de uma única falha. Exemplos de domínios de falha incluem:

- Uma máquina física.

- Um rack em um datacenter servido por uma única fonte de alimentação.
- Vários racks em um datacenter servidos por um único equipamento de rede.
- Um datacenter que pode ficar indisponível por causa do rompimento de um cabo de fibra óptica.
- Um conjunto de datacenters em uma única área geográfica que possa ser afetado por um único desastre natural, por exemplo, um furacão.

Em geral, à medida que a distância entre as réplicas aumenta, o mesmo ocorre com o tempo de ida e volta entre as réplicas, assim como o tamanho da falha que o sistema será capaz de tolerar. Para a maioria dos sistemas de consenso, aumentar o tempo de ida e volta entre as réplicas também elevará a latência das operações.

Até que ponto a latência importa, assim como a capacidade de sobreviver a uma falha em um domínio em particular, é muito dependente do sistema. Algumas arquiteturas de sistemas de consenso não exigem particularmente um throughput alto ou uma latência baixa: por exemplo, um sistema de consenso que exista para oferecer serviços de pertencimento a grupo e eleição de líder para um serviço altamente disponível provavelmente não estará intensamente carregado, e se o tempo de transação do consenso for apenas uma fração do tempo de lease do líder, então seu desempenho não será crítico. Sistemas orientados a batch também são menos afetados pela latência: os tamanhos dos lotes de operações podem ser aumentados para melhorar o throughput.

Nem sempre faz sentido aumentar continuamente o tamanho do domínio da falha cuja perda o sistema é capaz de suportar. Por exemplo, se todos os clientes que usam um sistema de consenso estiverem executando em um domínio de falha em particular (digamos, na área de Nova York) e implantar um sistema baseado em consenso distribuído em uma área geográfica mais ampla permitisse que ele continuasse servindo a requisições durante interrupções de serviço nesse domínio de falha (por exemplo, o Furacão Sandy), isso valeria a pena? Provavelmente não, pois os clientes do sistema também estarão inativos, portanto o sistema não verá nenhum tráfego. O custo extra em termos de latência, throughput e recursos de processamento

não proporcionaria nenhuma vantagem.

Você deve levar a recuperação de desastre em consideração ao decidir a localização de suas réplicas: em um sistema que armazene 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 snapshots frequentes em outro lugar, mesmo no caso de sistemas sólidos baseados em consenso implantados em vários domínios de falha diversificados. Há dois domínios de falha dos quais você não consegue jamais escapar: o próprio software e o erro humano da parte dos administradores do sistema. Bugs no software podem surgir em circunstâncias incomuns e provocar perda de dados, enquanto erros de configuração do sistema podem causar efeitos semelhantes. Operadores humanos também podem errar ou sabotar o sistema, provocando perda de dados.

Ao tomar decisões sobre a localização das réplicas, lembre-se de que a medida de desempenho mais importante é a percepção do cliente: o ideal é que o tempo de ida e volta de rede, dos clientes até as réplicas do sistema de consenso, seja minimizado. Em uma rede de longa distância, protocolos sem líderes, como Mencius ou Egalitarian Paxos, podem ter uma vantagem quanto ao desempenho, particularmente se as restrições de consistência da aplicação implicarem que é possível executar operações somente de leitura em qualquer réplica do sistema, sem realizar uma operação de consenso.

Capacidade e distribuição de carga

Ao fazer o design de uma implantação, você deve garantir que haja capacidade suficiente para tratar a carga. No caso de *implantações fragmentadas*, você pode ajustar a capacidade ajustando o número de fragmentos (shards). No entanto, para sistemas que podem ler de membros do grupo de consenso que não sejam o líder, você pode aumentar a capacidade de leitura adicionando mais réplicas. Acrescentar mais réplicas tem um custo: em um algoritmo que utilize um líder forte, adicionar réplicas impõe mais carga no processo líder, enquanto, em um protocolo ponto a ponto (peer-to-peer), acrescentar réplicas impõe mais carga em todos os processos. Porém,

se houver bastante capacidade para operações de escrita, mas uma carga intensa de leitura está estressando o sistema, acrescentar mais réplicas pode ser a melhor abordagem.

Devemos observar que a adição de uma réplica em um sistema com quórum majoritário pode possivelmente reduzir a disponibilidade do sistema de alguma forma (como mostra a Figura 23.10). Uma implantação típica para Zookeeper ou Chubby utiliza cinco réplicas, portanto um quórum majoritário exige três réplicas. O sistema continuará fazendo progressos se duas réplicas, ou 40%, estiverem indisponíveis. Com seis réplicas, um quórum exige quatro réplicas: somente 33% das réplicas pode estar indisponível para o sistema permanecer ativo.

As considerações relacionadas aos domínios de falha, desse modo, se aplicam mais intensamente quando uma sexta réplica é adicionada: se uma empresa tiver cinco datacenters e geralmente executa grupos de consenso com cinco processos, um em cada datacenter, então a perda de um datacenter ainda deixará uma réplica extra em cada grupo. Se uma sexta réplica for implantada em um dos cinco datacenters, uma interrupção de serviço nesse datacenter removerá as duas réplicas extras do grupo, reduzindo assim a capacidade em 33%.

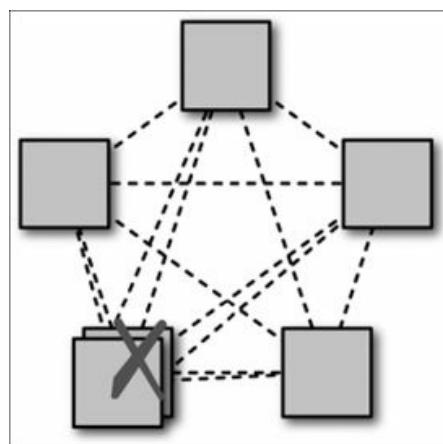


Figura 23.10 – Adicionar uma réplica extra em uma região pode reduzir a disponibilidade do sistema. Colocar várias réplicas em um único datacenter pode reduzir a disponibilidade do sistema: nesse caso, há um quórum sem redundância restante.

Se houver uma alta densidade de clientes em uma determinada região

geográfica, é melhor localizar as réplicas próximas aos clientes. No entanto, decidir onde, exatamente, as réplicas devem estar localizadas pode exigir um planejamento cuidadoso em termos de distribuição de carga e no modo como um sistema lida com a sobrecarga. Como mostra a Figura 23.11, se um sistema simplesmente encaminha requisições de leitura de clientes para a réplica mais próxima, um pico grande de carga concentrado em uma região poderá sobrecarregar a réplica mais próxima e, depois, a réplica mais próxima seguinte, e assim sucessivamente – é uma *falha em cascata* (veja o Capítulo 22). Esse tipo de sobrecarga muitas vezes pode ocorrer como resultado de jobs em batch iniciando, especialmente se vários deles começarem ao mesmo tempo.

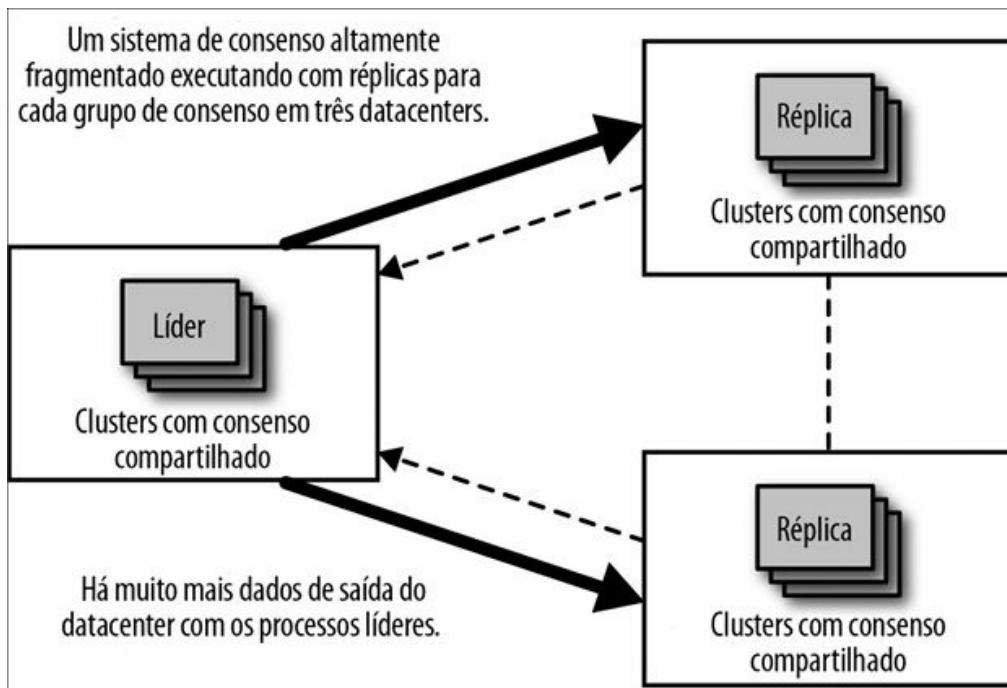


Figura 23.11 – Colocar os processos líderes no mesmo lugar resulta em utilização irregular de banda.

Já vimos o motivo pelo qual muitos sistemas de consenso distribuído utilizam um processo líder para melhorar o desempenho. Contudo, é importante entender que as réplicas líderes utilizarão mais recursos computacionais, particularmente capacidade de saída de rede. Isso ocorre porque o líder envia mensagens de proposta que incluem os dados propostos, mas as réplicas enviam mensagens menores, geralmente contendo apenas uma concordância

com um determinado ID de transação de consenso. As empresas que operam sistemas de consenso altamente fragmentados, com um número muito alto de processos, talvez achem necessário garantir que os processos líderes para os diferentes fragmentos estejam relativamente平衡ados de modo uniforme entre datacenters distintos. Fazer isso evita que o sistema como um todo tenha um gargalo na capacidade de saída de rede para apenas um datacenter, e proporciona uma capacidade muito maior para o sistema como um todo.

Outra desvantagem de implantar grupos de consenso em vários datacenters (mostrado na Figura 23.11) é a mudança muito radical no sistema que pode ocorrer se o datacenter que hospeda os líderes sofrer uma falha de proporções amplas (alimentação, falha em equipamento de rede ou rompimento de fibra, por exemplo). Como mostra a Figura 23.12, nesse cenário de falha, todos os líderes devem fazer failover para outro datacenter, separando-se uniformemente ou desviando em massa para um datacenter. Qualquer que seja o caso, o link entre os outros dois datacenters repentinamente receberá muito mais tráfego de rede desse sistema. Esse seria um momento inoportuno para descobrir que a capacidade nesse link é insuficiente.

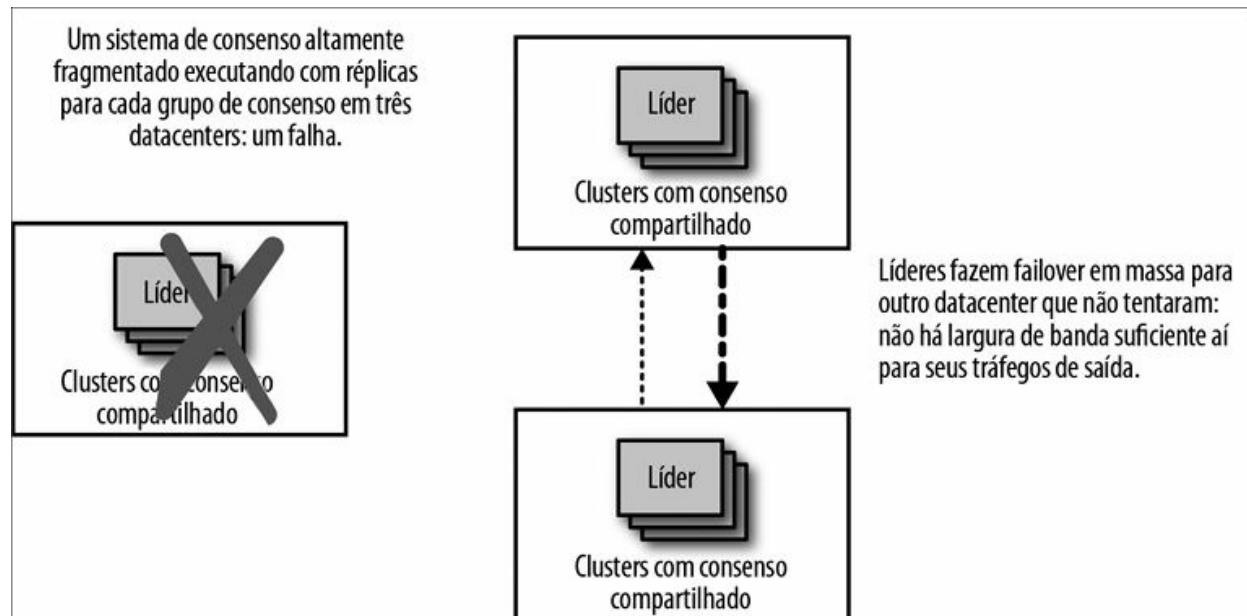


Figura 23.12 – Quando líderes com localização compartilhada fazem failover em massa, os padrões de utilização de rede mudam drasticamente.

Entretanto, esse tipo de implantação poderia ser facilmente um resultado não intencional de processos automáticos no sistema que influenciam o modo

como os líderes são escolhidos. Por exemplo:

- Os clientes terão uma latência melhor para qualquer operação tratada por meio do líder se esse estiver mais próximo a eles. Um algoritmo que tente colocar os líderes próximos à grande maioria dos clientes poderia tirar proveito desse insight.
- Um algoritmo poderia tentar colocar os líderes em máquinas com os melhores desempenhos. Uma armadilha dessa abordagem é que, se um dos três datacenters tiver máquinas mais rápidas, uma quantidade desproporcional de tráfego será enviada a esse datacenter, resultando em mudanças drásticas de tráfego se esse datacenter ficar offline. Para evitar esse problema, o algoritmo também deve levar em consideração um equilíbrio na distribuição com base nos recursos da máquina ao selecioná-la.
- Um algoritmo de eleição de líder pode favorecer processos que estejam executando há mais tempo. Processos executando há mais tempo provavelmente estarão correlacionados com a localização se o lançamento de novas versões de software for feito por datacenter.

Composição do quórum

Ao determinar a localização das réplicas em um grupo de consenso, é importante considerar o efeito da distribuição geográfica (ou, mais exatamente, as latências de rede entre as réplicas) no desempenho do grupo.

Uma abordagem é distribuir as réplicas o mais uniformemente possível, com RTTs semelhantes entre todas as réplicas. Considerando que todos os demais fatores sejam iguais (por exemplo, carga de trabalho, hardware e desempenho de rede), essa organização deve resultar em um desempenho razoavelmente consistente entre todas as regiões, não importando a localização do líder do grupo (ou de cada membro do grupo de consenso, se um protocolo sem líder estiver em uso).

A geografia pode complicar bastante essa abordagem. Isso é particularmente verdadeiro para tráfego intracontinental *versus* transpacífico e transatlântico. Considere um sistema que se estenda pela América do Norte e pela Europa: é impossível deixar as réplicas equidistantes umas das outras, pois sempre

haverá um atraso maior para o tráfego transatlântico em relação ao tráfego intracontinental. Independentemente de tudo o mais, as transações de uma região precisarão fazer uma viagem transatlântica de ida e volta para atingir o consenso.

No entanto, como mostra a Figura 23.13, na tentativa de distribuir o tráfego o mais uniformemente possível, os designers do sistema podem optar por usar cinco réplicas, com duas réplicas aproximadamente no centro dos Estados Unidos, uma na costa leste e duas na Europa. Uma distribuição como essa significaria que, no caso médio, o consenso poderia ser conseguido na América do Norte, sem esperar pelas respostas da Europa, ou que na Europa o consenso seria alcançado por meio de troca de mensagens somente com a réplica da costa leste. A réplica da costa leste atua como uma espécie de peça central, em que dois quóruns possíveis se sobrepõem.



Figura 23.13 – Quóruns que se sobrepõem, com uma réplica atuando como uma ligação.

Como mostra a Figura 23.14, a perda dessa réplica significa que a latência do sistema provavelmente mudará de forma drástica: em vez de ser extremamente influenciada pelo RTT do centro dos Estados Unidos para a costa leste ou pelo RTT da Europa para a costa leste, a latência será baseada no RTT da Europa até o centro dos Estados Unidos, que é aproximadamente 50% maior que o RTT da Europa até a costa leste norte-americana. A

distância geográfica e o RTT de rede entre o quórum mais próximo possível aumenta bastante.



Figura 23.14 – A perda da réplica de ligação resulta imediatamente em um RTT maior para qualquer quórum.

Esse cenário é um ponto fraco fundamental para o quórum de maioria simples quando aplicado a grupos compostos de réplicas com RTTs muito diferentes entre os membros. Em casos como esse, uma abordagem de quórum hierárquico pode ser conveniente. Como mostra o diagrama 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 uma maioria dos membros do grupo estiver disponível. Isso significa que uma réplica pode ser perdida no grupo central sem incorrer em um grande impacto no desempenho do sistema como um todo, pois o grupo central ainda poderá votar nas transações com duas de suas três réplicas.

Há, porém, um custo de recurso associado à execução de um número maior de réplicas. Em um sistema altamente fragmentado, com uma carga intensa de leituras que, em sua maior parte, seja atendida pelas réplicas, podemos atenuar esse custo utilizando menos grupos de consenso. Uma estratégia como essa significa que o número geral de processos no sistema pode não mudar.



Figura 23.15 – Quóruns hierárquicos podem ser usados para reduzir a dependência da réplica central.

Monitorando sistemas de consenso distribuído

Como já vimos, algoritmos de consenso distribuído estão no núcleo de muitos sistemas críticos do Google ([Ana13], [Bur06], [Cor12], [Shu13]). Todos os sistemas importantes de produção precisam de monitoração para detectar interrupções de serviço ou problemas, e resolvê-los. A experiência tem mostrado que há certos aspectos específicos dos sistemas de consenso distribuído que exigem uma atenção especial. São eles:

Número de membros executando em cada grupo de consenso e o status de cada processo (saudável ou não saudável)

Um processo pode estar executando, mas ser incapaz de fazer progresso por algum motivo (por exemplo, um problema relacionado ao hardware).

Réplicas que se atrasam de forma persistente

Membros saudáveis de um grupo de consenso ainda podem estar possivelmente em vários estados diferentes. O membro de um grupo pode estar recuperando estados de seus pares após uma inicialização, ou pode estar atrasado em relação ao quórum do grupo, ou pode estar atualizado, com participação total, e pode ser o líder.

Existência ou não de um líder

Um sistema baseado em um algoritmo como o Multi-Paxos, que utiliza um papel de líder, deve ser monitorado para garantir que um líder exista, pois se o sistema não tiver nenhum 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 mudança, portanto esse número oferece uma métrica útil a ser monitorada. Um aumento muito rápido nas mudanças de líder indica que o líder está alternando, talvez por causa de problemas de conectividade de rede. Uma diminuição no número da visão poderia indicar um bug sério.

Número de transação do consenso

Os operadores devem saber se o sistema de consenso está ou não fazendo progressos. A maioria dos algoritmos de consenso utiliza um número de transação de consenso crescente para sinalizar progresso. Devemos ver esse número aumentando com o tempo se o sistema estiver saudável.

Número de propostas vistas; número de propostas para as quais houve acordo

Esses números indicam se o sistema está ou não funcionando corretamente.

Throughput e latência

Embora não sejam específicas de sistema de consenso distribuído, essas características de seu sistema de consenso devem ser monitoradas e compreendidas pelos administradores.

Para compreender o desempenho do sistema e ajudar a resolver problemas de desempenho, você também pode monitorar os seguintes dados:

- Distribuições das latências para aceitação de propostas
- Distribuições de latências de rede observadas entre partes do sistema em locais diferentes

- Quantidade de tempo que os aceitadores gastam em logging durável
- Bytes, em geral, aceitos por segundo no sistema

Conclusão

Exploramos a definição do problema de consenso distribuído e apresentamos alguns padrões de arquitetura para sistemas baseados em consenso distribuído. Também analisamos as características de desempenho, além de algumas das preocupações operacionais em torno de sistemas baseados em consenso distribuído.

Propositalmente, evitamos uma discussão profunda sobre algoritmos, protocolos ou implementações específicos neste capítulo. Os sistemas de coordenação distribuída e as tecnologias subjacentes estão evoluindo rapidamente, e essas informações estariam rapidamente desatualizadas, de modo diferente dos fundamentos discutidos aqui. No entanto, esses fundamentos, juntamente com os artigos referenciados ao longo deste capítulo, permitirão que você use as ferramentas de coordenação distribuída disponíveis atualmente, assim como em softwares futuros.

Se você não se lembrar de mais nada deste capítulo, tenha em mente os tipos de problemas que o consenso distribuído pode resolver e os tipos de problemas que podem surgir quando métodos *ad hoc*, como heartbeats, forem usados no lugar do consenso distribuído. Sempre que você vir eleição de líder, estado crítico compartilhado ou locking distribuído, pense no consenso distribuído: qualquer abordagem inferior será uma bomba-relógio esperando explodir em seus sistemas.

¹ Kyle Kingsbury escreveu uma longa série de artigos sobre a correção de sistemas distribuídos, que contém muitos exemplos de comportamentos inesperados e incorretos nesses tipos de bancos de dados. Veja <https://aphyr.com/tags/jepsen>.

² Em particular, o desempenho do algoritmo Paxos original não é ideal, mas foi bastante melhorado ao longo dos anos.

CAPÍTULO 24

Escalonamento periódico e distribuído com o cron

Escrito por Štěpán Davidovič¹

Editado por Kavita Guliani

Este capítulo descreve a implementação do Google de um serviço cron distribuído que serve a uma grande maioria das equipes internas que precisa de escalonamento periódico de jobs de processamento. Ao longo da existência do cron, aprendemos muitas lições sobre como fazer o design e implementar o que poderia parecer um serviço básico. Neste capítulo, discutiremos os problemas que os crons distribuídos enfrentam e apresentaremos algumas soluções possíveis.

O cron é um utilitário Unix comum, projetado para iniciar jobs arbitrários periodicamente nos horários ou a intervalos definidos pelo usuário. Inicialmente analisaremos os princípios básicos do cron e suas implementações mais comuns e, em seguida, veremos como uma aplicação como o cron pode funcionar em um ambiente grande e distribuído de modo a aumentar a confiabilidade do sistema contra falhas em uma única máquina. Descreveremos um sistema de cron distribuído implantado em um pequeno número de máquinas, mas que pode iniciar cron jobs em todo um datacenter em conjunto com um sistema de escalonamento de datacenter como o Borg [Ver15].

Cron

Vamos discutir como o cron geralmente é usado no caso de uma máquina única antes de explorarmos sua execução como um serviço em vários

datacenters.

Introdução

O cron foi projetado de modo que os administradores de sistemas e os usuários comuns possam especificar comandos a serem executados, e quando esses comandos devem executar. O cron executa vários tipos de jobs, incluindo coleta de lixo (garbage collection) e análise periódica de dados. O formato mais comum de especificação de horário se chama “crontab”. Esse formato aceita intervalos simples (por exemplo, “uma vez ao dia ao meio-dia” ou “a cada hora em hora cheia”). Intervalos complexos, como “todo sábado, que também seja o trigésimo dia do mês”, também podem ser configurados.

O cron geralmente é implementado com um único componente, comumente conhecido como crond. O crond é um daemon que carrega a lista de cron jobs agendados. Os jobs são iniciados de acordo com seus horários de execução especificados.

Ponto de vista da confiabilidade

Vários aspectos do serviço cron merecem destaque do ponto de vista da confiabilidade:

- O domínio de falhas do cron é essencialmente uma única máquina. Se a máquina não estiver executando, nem o escalonador cron nem os jobs que ele inicia poderão executar.² Considere um caso de distribuição bem simples, com duas máquinas, em que seu escalonador cron inicia jobs em uma máquina de trabalho diferente (por exemplo, usando SSH). Esse cenário apresenta dois domínios de falha distintos que podem impactar a nossa capacidade de iniciar jobs: a máquina escalonadora ou a máquina de destino poderiam falhar.
- O único estado que deve ser persistente entre as reinicializações do crond (incluindo reinicializações de máquinas) é a própria configuração crontab. As iniciações de jobs do cron são do tipo disparar-e-esquecer, e o crond não faz nenhuma tentativa de monitorar essas iniciações.
- O anacron é uma exceção digna de destaque. O anacron tenta iniciar jobs

que teriam sido iniciados quando o sistema estava inativo. Tentativas de reiniciar jobs estão limitadas aos jobs que executam diariamente ou com menos frequência. Essa funcionalidade é muito útil para executar jobs de manutenção em estações de trabalho e em notebooks, e é facilitada por um arquivo que mantém o timestamp do último início de todos os cron jobs registrados.

Cron jobs e idempotência

Os cron jobs foram projetados para realizar tarefas periódicas, mas, além disso, é difícil saber com antecedência que função eles têm. A variedade de requisitos que o conjunto diversificado de cron jobs implica, obviamente tem impacto nos requisitos de confiabilidade.

Alguns cron jobs, como os processos de coleta de lixo, são idempotentes. No caso de mau funcionamento do sistema, é seguro iniciar esses jobs várias vezes. Outros cron jobs, como um processo que envia uma newsletter por email a uma lista de distribuição extensa, não devem ser iniciados mais de uma vez.

Para complicar mais ainda a situação, uma falha em iniciar um job é aceitável para alguns cron jobs, mas não para outros. Por exemplo, um cron job de coleta de lixo agendado para executar a cada cinco minutos pode pular um início, mas um cron job de folha de pagamento agendado para executar uma vez por mês não deve ser ignorado.

Essa grande variedade de cron jobs dificulta pensar nos modos de falha: em um sistema como o serviço cron, não há uma única resposta adequada a todas as situações. Em geral, favorecemos pular inícios de jobs em vez de arriscar a fazer inícios duplos, por mais que a infraestrutura permita. Isso ocorre porque recuperar-se de um início de job que não ocorreu é mais factível do que recuperar-se de um início duplo. Os proprietários dos cron jobs podem (e devem!) monitorar seus cron jobs: por exemplo, um proprietário de cron jobs pode fazer o serviço cron expor os estados dos cron jobs que ele administra, ou esse administrador pode configurar um sistema independente de monitoração dos efeitos dos cron jobs. No caso de um início de job que não ocorreu, os proprietários dos cron jobs podem executar uma ação apropriada

que esteja de acordo com a natureza do cron job. No entanto, desfazer um início duplo, por exemplo, o caso da newsletter mencionada antes, pode ser difícil ou até mesmo totalmente impossível. Assim, preferimos “pecar por falta” a fim de evitar a criação de um estado ruim para todo o sistema.

Cron em larga escala

Passar de máquinas isoladas para implantações em larga escala exige repensar alguns fundamentos sobre como fazer o cron funcionar bem em um ambiente desse tipo. Antes de apresentar os detalhes da solução do Google para o cron, discutiremos as diferenças entre implantação em pequena escala e em larga escala e descreveremos as mudanças de design exigidas pelas implantações em larga escala.

Infraestrutura estendida

Em suas implementações “normais”, o cron está limitado a uma única máquina. Implantações de sistemas em larga escala estendem nossa solução de cron a várias máquinas.

Hospedar seu serviço cron em uma única máquina poderia ser catastrófico no que concerne à confiabilidade. Suponha que essa máquina esteja localizada em um datacenter com exatamente 1.000 máquinas. Uma falha de apenas um milésimo de suas máquinas disponíveis poderia derrubar todo o serviço cron. Por motivos óbvios, essa implementação não é aceitável.

Para aumentar a confiabilidade do cron, desacoplamos os processos das máquinas. Se você quiser executar um serviço, basta especificar os requisitos do serviço e em qual datacenter ele deve executar. O sistema de escalonamento do datacenter (ele próprio deve ser confiável) determina a máquina ou as máquinas em que seu serviço deve ser implantado, além de tratar os casos em que as máquinas morrem. Desse modo, iniciar um job em um datacenter passa a ser efetivamente o envio de uma ou mais RPCs ao escalonador do datacenter.

Esse processo, porém, não é instantâneo. Descobrir que há uma máquina inativa implica timeouts de verificação de sanidade, enquanto reescalonar seu

serviço em uma máquina diferente exige tempo para instalar o software e inicializar o novo processo.

Como passar um processo para uma máquina diferente pode significar perda de qualquer estado local armazenado na máquina antiga (a menos que uma migração ao vivo seja empregada), e o tempo de reescalonar pode exceder o menor intervalo de escalonamento de um minuto, precisamos de procedimentos definidos para atenuar tanto a perda de dados quanto os requisitos rigorosos de tempo. Para manter o estado local da máquina antiga, você pode simplesmente fazer a persistência do estado em um sistema de arquivos distribuído, como o GFS, e utilizar esse sistema de arquivos durante a inicialização para identificar os jobs que deixaram de iniciar devido ao reescalonamento. No entanto, essa solução deixa a desejar em termos de expectativas de hora exata: se você executa um cron job a cada cinco minutos, um atraso de um a dois minutos causado pelo overhead total do sistema cron sendo reescalonado poderá ser significativamente inaceitável. Nesse caso, substitutos funcionando em hot standby, capazes de entrar em cena rapidamente e assumir a operação, podem reduzir essa janela de tempo de forma significativa.

Requisitos estendidos

Sistemas com uma única máquina geralmente colocam todos os processos em execução no mesmo lugar, apenas com um isolamento limitado. Embora contêineres não sejam comuns, não é necessário nem usual utilizar contêineres para isolar todos os componentes de um serviço implantado em uma única máquina. Desse modo, se o cron fosse implantado em uma única máquina, o crond e todos os cron jobs executados por ele provavelmente não estariam isolados.

A implantação em escala de datacenter comumente significa uma implantação em contêineres que garantem isolamento. O isolamento é necessário porque a expectativa básica é que processos independentes executando no mesmo datacenter não devem causar impactos negativos uns nos outros. Para garantir que essa expectativa seja atendida, é necessário conhecer a quantidade de recursos que você deve adquirir com antecedência

para qualquer dado processo que você queira executar – tanto para o sistema cron quanto para os jobs que ele inicia. Um cron job pode ser postergado se um datacenter não tiver recursos disponíveis para atender às demandas do cron job. Os requisitos de recursos, além da exigência do usuário para monitorar os inícios dos cron jobs, implicam que precisamos monitorar o estado completo dos inícios de nossos cron jobs, desde o início agendado até o término.

Desacoplar os inícios dos processos das máquinas específicas expõe o sistema cron a falhas parciais de início. A versatilidade das configurações dos cron jobs também implica que iniciar um novo cron job em um datacenter pode exigir várias RPCs, de modo que, às vezes, veremos um cenário em que algumas RPCs foram bem-sucedidas, enquanto outras não (por exemplo, porque o processo enviando as RPCs morreu no meio da execução dessas tarefas). O procedimento de recuperação do cron também deve levar esse cenário em consideração.

Quanto ao modo de falha, um datacenter é um ecossistema substancialmente mais complexo do que uma única máquina. O serviço cron que começa como um binário relativamente simples em uma única máquina agora tem várias dependências óbvias ou não 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, uma queda parcial de energia elétrica ou problemas com serviços de armazenagem), o serviço seja capaz de continuar funcionando. Ao exigir que o escalonador do datacenter coloque réplicas do cron em várias localidades no datacenter, evitamos o cenário em que a falha em uma única unidade de distribuição de energia deixa todos os processos do serviço cron inativos.

Talvez seja possível implantar um único serviço cron pelo mundo, mas implantar o cron em um único datacenter tem suas vantagens: o serviço desfruta de baixa latência e tem o mesmo destino do escalonador do datacenter, que é a principal dependência do cron.

Desenvolvendo o cron no Google

Esta seção aborda os problemas que devem ser resolvidos para oferecer uma

implantação distribuída e de larga escala do cron, de forma confiável. Também destacamos algumas decisões importantes feitas em relação ao cron distribuído no Google.

Monitorando o estado dos cron jobs

Conforme discutimos nas seções anteriores, precisamos armazenar alguns estados dos cron jobs e devemos ser capazes de restaurar essas informações rapidamente em caso de falha. Além do mais, a consistência desse estado é fundamental. Lembre-se de que muitos cron jobs, por exemplo, uma execução de folha de pagamento ou o envio de uma newsletter via email, não são idempotentes.

Temos duas opções para monitorar o estado dos cron jobs:

- Armazenar os dados externamente em repositórios distribuídos disponíveis de modo geral.
- Utilizar um sistema que armazene um pequeno volume de estados como parte do próprio serviço cron.

Ao fazer o design do cron distribuído, escolhemos a segunda opção. Fizemos essa escolha por vários motivos:

- Sistemas de arquivo distribuídos como o GFS ou o HDFS com frequência visam ao caso de uso de arquivos bem grandes (por exemplo, a saída de programas de web crawling), enquanto a quantidade de informações que precisamos armazenar sobre os cron jobs é bem pequena. Escritas menores em um sistema de arquivos distribuído são muito custosas, e apresentam alta latência, pois o sistema de arquivos não está otimizado para esses tipos de escrita.
- Serviços básicos para os quais as interrupções de serviço têm um impacto amplo (por exemplo, o cron) devem ter poucas dependências. Mesmo que partes do datacenter fiquem inativas, o serviço cron deve ser capaz de funcionar no mínimo por breves períodos de tempo. Contudo, esse requisito não quer dizer que a armazenagem deva fazer parte do processo cron diretamente (o modo como a armazenagem é tratada é essencialmente um detalhe de implementação). No entanto, o cron deve

ser capaz de funcionar de modo independente dos sistemas de downstream projetados para tratar um número elevado de usuários internos.

O uso do Paxos

Implantamos várias réplicas do serviço cron e usamos o algoritmo de consenso distribuído Paxos (veja o Capítulo 23) para garantir que elas tenham estados consistentes. Desde que a maioria dos membros do grupo esteja disponível, o sistema distribuído como um todo pode processar novas mudanças de estado com sucesso, apesar de falhas de subconjuntos limitados da infraestrutura.

Como mostra a Figura 24.1, o cron distribuído utiliza um único job líder, que é a única réplica que pode modificar o estado compartilhado, assim como é a única réplica que pode iniciar cron jobs. Tiramos proveito do fato de que a variante do Paxos que usamos, o Fast Paxos [Lam06], utiliza uma réplica líder internamente como otimização – a réplica líder do Fast Paxos também atua como o líder do serviço cron.

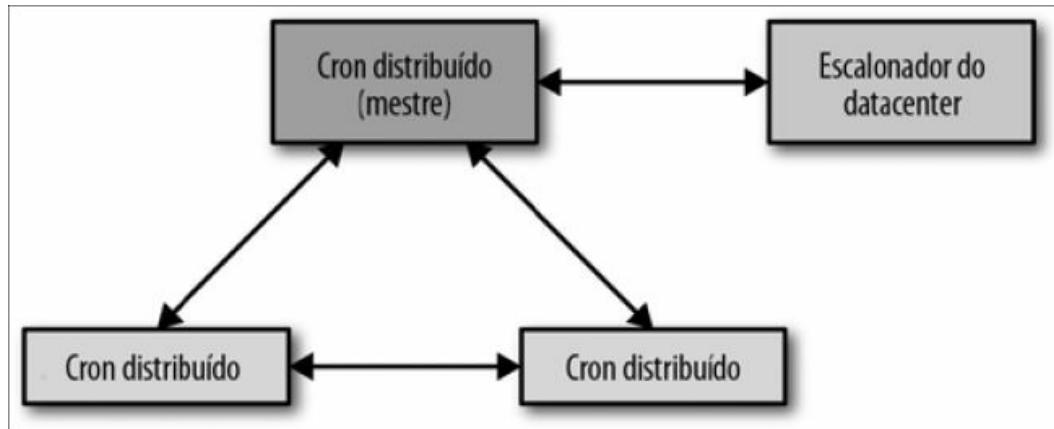


Figura 24.1 – As interações entre as réplicas do cron distribuído.

Se a réplica líder morrer, o mecanismo de verificação de sanidade do grupo Paxos identifica esse evento rapidamente (em segundos). Como outro processo cron já iniciou e está disponível, podemos eleger um novo líder. Assim que o novo líder é eleito, seguimos um protocolo de eleição de líder específico do serviço cron, que é responsável por assumir todo o trabalho não concluído pelo líder anterior. O líder específico do serviço cron é o mesmo

líder do Paxos, porém o serviço cron deve executar ações adicionais por ocasião da promoção. O tempo de reação rápido para areeleição do líder nos permite permanecer dentro de um tempo de failover geralmente tolerável de um minuto.

O estado mais importante mantido no Paxos é a informação relacionada a quais cron jobs são iniciados. Informamos sincronamente a um quórum de réplicas sobre o começo e o fim do início agendado de cada cron job.

Os papéis de líder e de seguidor

Como acabamos de descrever, nosso uso do Paxos e sua implantação no serviço cron fazem com que haja dois papéis atribuídos: o de líder e o de seguidor. As próximas seções descrevem cada papel.

O líder

A réplica líder é a única réplica que inicia ativamente os cron jobs. O líder tem um escalonador interno que, de modo muito semelhante ao crond simples descrito no início deste capítulo, mantém a lista dos cron jobs ordenada pelo horário de início agendado. A réplica líder espera até o horário de início agendado para o primeiro job.

Ao atingir esse horário de início agendado, a réplica líder anuncia que está prestes a iniciar esse cron job em particular, e calcula o novo horário de início agendado, assim como uma implementação normal de crond faria. É claro que, com um crond normal, a especificação de início de um cron job pode ter mudado desde a última execução, e essa especificação de início deve ser mantida em sincronia com os seguidores também. Simplesmente identificar o cron job não é suficiente: devemos também identificar unicamente o início em particular usando o horário; caso contrário, pode haver ambiguidade na monitoração de início dos cron jobs. (Uma ambiguidade desse tipo é especialmente provável no caso de cron jobs muito frequentes, como aqueles que executam a cada minuto.) Como vemos na Figura 24.2, essa comunicação é feita por intermédio do Paxos.

É importante que a comunicação do Paxos permaneça síncrona e que o início propriamente dito do cron job não ocorra até que ele receba a confirmação de

que o quórum do Paxos recebeu a notificação de início. O serviço cron precisa entender se cada cron job foi iniciado a fim de decidir o próximo curso de ação no caso de failover do líder. Não realizar essa tarefa de modo síncrono poderia implicar que o início completo do cron job ocorreria no líder, sem informar as réplicas seguidoras. No caso de failover, as réplicas seguidoras poderiam tentar efetuar o mesmo início de job novamente, pois não estariam cientes de que o início já havia ocorrido.

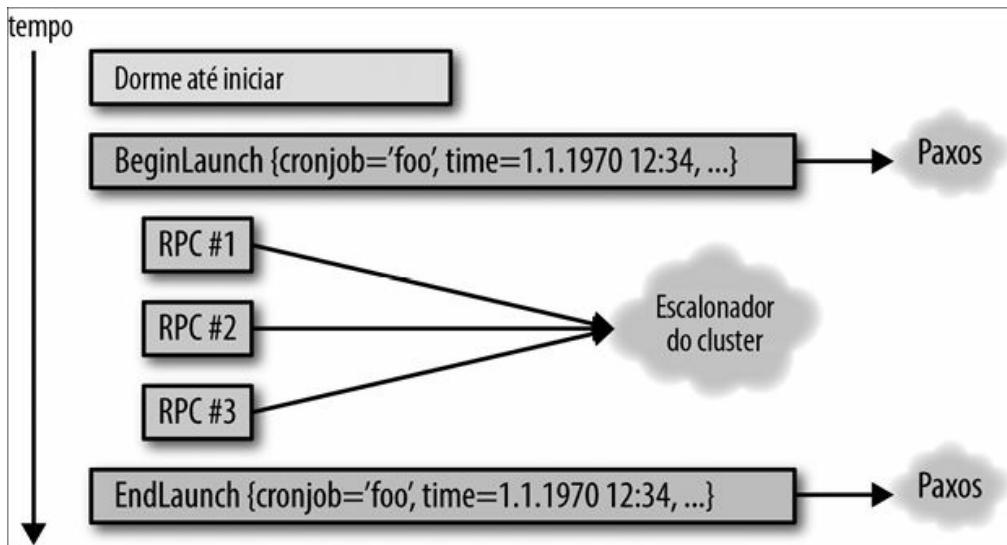


Figura 24.2 – Ilustração do progresso do início de um cron job, do ponto de vista do líder.

A conclusão do início de um cron job é anunciada por meio do Paxos às outras réplicas de forma síncrona. Observe que não importa se o início do job teve sucesso ou falhou por motivos externos (por exemplo, se o escalonador do datacenter estava indisponível). Nesse caso, estamos apenas mantendo o controle do fato de que o serviço cron tentou iniciar o job no horário específico agendado. Também devemos ser capazes de resolver falhas do sistema cron no meio dessa operação, conforme discutiremos na próxima seção.

Outra funcionalidade extremamente importante do líder é que, assim que perder a liderança por algum motivo, ele deverá parar imediatamente de interagir com o escalonador do datacenter. Manter a liderança deve garantir uma exclusão mútua no acesso ao escalonador do datacenter. Na ausência dessa condição de exclusão mútua, o antigo líder e o novo poderiam executar

ações conflitantes no escalonador do datacenter.

O seguidor

As réplicas seguidoras mantêm o controle do estado do mundo, conforme fornecido pelo líder, a fim de assumir o controle de imediato, se for necessário. Todas as mudanças de estado monitoradas pelas réplicas seguidoras são informadas pela réplica líder por meio do Paxos. De modo muito semelhante ao líder, os seguidores também mantêm uma lista de todos os cron jobs do sistema, e essa lista deve estar consistente entre as réplicas (por meio do Paxos).

Na recepção de uma notificação sobre o início de um job, a réplica seguidora atualiza seu próximo horário local de início agendado para o dado cron job. Essa mudança de estado muito importante (realizada de forma síncrona) garante que todos os agendamentos de cron jobs no sistema estejam consistentes. Monitoramos todos os inícios dos jobs em aberto (que já começaram, mas não foram concluídos).

Se uma réplica líder morrer ou tiver um mau funcionamento (por exemplo, for separada de outras réplicas da rede), um seguidor deverá ser eleito como o novo líder. A eleição deve convergir em menos de um minuto para evitar o risco de perder ou atrasar o início de um cron job de um modo que não seja razoável. Depois que um líder é eleito, todos os inícios de job abertos (isto é, com falhas parciais) devem ser concluídos. Esse processo pode ser bem complicado, impondo requisitos adicionais tanto ao sistema cron quanto à infraestrutura do datacenter. A próxima seção discute como resolver falhas parciais desse tipo.

Resolvendo falhas parciais

Conforme mencionamos, a interação entre a réplica líder e o escalonador do datacenter pode falhar entre o envio de várias RPCs que descrevem o início lógico de um único cron job. Nossos sistemas devem ser capazes de tratar essa condição.

Lembre-se de que todo início de cron job tem dois pontos de sincronização:

- Quando estamos prestes a iniciar o job

- Quando acabamos de iniciá-lo

Esse dois pontos nos permitem delimitar o início do job. Mesmo que o início do job seja constituído de uma única RPC, como podemos saber se a RPC foi realmente enviada? Considere o caso em que sabemos que o início agendado começou, mas não fomos notificados de sua conclusão antes de a réplica líder morrer.

Para determinar se a RPC foi realmente enviada, uma das condições a seguir deve ser atendida:

- Todas as operações em sistemas externos, que talvez precisemos continuar após uma reeleição, devem ser idempotentes (isto é, podemos executar as operações novamente de forma segura).
- Devemos ser capazes de buscar o estado de todas as operações em sistemas externos a fim de determinar, de uma maneira que não seja ambígua, se elas foram concluídas ou não.

Cada uma dessas condições impõe restrições significativas, e podem ser difíceis de implementar, mas ser capaz de atender a pelo menos uma delas é fundamental para a operação precisa de um serviço cron em um ambiente distribuído que possa sofrer uma única falha ou várias falhas parciais. Não tratar essa situação de modo apropriado pode resultar em inícios perdidos ou duplicados do mesmo cron job.

A maior parte da infraestrutura que inicia jobs lógicos em datacenters (o Mesos, por exemplo) permite atribuir nomes a esses jobs do datacenter, possibilitando consultar seus estados, interrompê-los ou realizar outros serviços de manutenção. Uma solução razoável ao problema de idempotência é compor os nomes dos jobs com antecedência (evitando, assim, provocar qualquer operação de mutação no escalonador do datacenter) e então distribuir os nomes a todas as réplicas de seu serviço cron. Se o líder do serviço cron morrer durante o início de um job, o novo líder simplesmente buscará o estado de todos os nomes previamente calculados e iniciará os nomes ausentes.

Observe que, semelhante ao nosso método de identificar inícios individuais de cron jobs pelo nome e pelo horário de início, é importante que os nomes

dos jobs que foram compostos no escalonador do datacenter incluem o horário particular de início agendado (ou que essa informação seja recuperável de outra maneira). Em operação normal, o serviço cron deve fazer um failover rápido em caso de falha do líder, mas isso nem sempre acontece.

Lembre-se de que monitoramos o horário de início agendado ao manter o estado interno entre as réplicas. De modo semelhante, precisamos tirar a ambiguidade de nossa interação com o escalonador do datacenter, usando também o horário de início agendado. Por exemplo, considere um cron job de curta duração, porém executado com frequência. O cron job inicia; porém, antes de o início ser comunicado a todas as réplicas, o líder falha e um failover excepcionalmente longo – demorado o suficiente para o cron job terminar com sucesso – ocorre. O novo líder consulta o estado do cron job, nota que ele foi concluído e tenta iniciar o job novamente. Se o horário de início tivesse sido incluído, o novo líder saberia que o job no escalonador do datacenter é o resultado desse início de cron job em particular, e esse início duplo não teria ocorrido.

A implementação propriamente dita tem um sistema mais complicado para consulta de estados, determinada pelos detalhes de implementação da infraestrutura subjacente. No entanto, a descrição anterior inclui os requisitos independentes de implementação de qualquer sistema desse tipo. Conforme a infraestrutura disponível, talvez você precise considerar também o custo-benefício de correr o risco de um início duplo e o risco de pular o início de um job.

Armazenando o estado

Usar o Paxos para chegar a um consenso é apenas uma parte do problema de como tratar o estado. O Paxos é essencialmente um log contínuo de mudanças de estado, ao qual novas informações são concatenadas sincronamente à medida que mudanças de estado ocorrem. Essa característica do Paxos tem duas implicações:

- O log deve ser compactado para evitar que cresça infinitamente.
- O log propriamente dito deve ser armazenado em algum lugar.

Para evitar o crescimento infinito do log do Paxos, podemos simplesmente obter um snapshot (imagem instantânea) do estado atual, o que significa que podemos reconstruir o estado sem precisar reproduzir todas as entradas de mudanças de estado do log que levaram ao estado atual. Para apresentar um exemplo: se nossas mudanças de estado armazenadas em logs forem “Incremente um contador em 1”, então, após mil iterações, teremos mil entradas de log que poderão ser facilmente alteradas para um snapshot “Defina o contador com 1.000”.

No caso de logs perdidos, só perderemos o estado desde o último snapshot. Os snapshots, na verdade, são o nosso estado mais crítico – se perdermos nossos snapshots, teremos essencialmente que começar do zero novamente, pois teremos perdido nosso estado interno. Perder logs, por outro lado, simplesmente causa uma perda de estado limitada e envia o sistema cron de volta no tempo, até o momento em que o último snapshot foi obtido.

Temos duas opções principais para armazenar nossos dados:

- Externamente, em um repositório distribuído disponível de modo geral
- Em um sistema que armazene o pequeno volume de estados como parte do próprio serviço cron

Quando fizemos o design do sistema, combinamos elementos das duas opções.

Armazenamos os logs do Paxos no disco local da máquina em que as réplicas do serviço cron estão escalonadas. Ter três réplicas na operação default implica que temos três cópias dos logs. Armazenamos os snapshots no disco local também. No entanto, como eles são críticos, também fazemos backup deles em um sistema de arquivos distribuído, nos protegendo assim contra falhas que afetem todas as três máquinas.

Não armazenamos os logs em nosso sistema de arquivos distribuído. De modo consciente, decidimos que perder os 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 implicar em uma penalidade substancial no desempenho, causada por pequenas escritas frequentes. A perda simultânea de todas as três máquinas é

improvável, e se uma perda desse tipo realmente ocorrer, fazemos uma restauração automática a partir do snapshot. Assim, perdemos apenas uma pequena quantidade dos logs: aqueles obtidos desde o último snapshot, criado a intervalos configuráveis. É claro que essas negociações podem ser diferentes de acordo com os detalhes da infraestrutura, bem como com os requisitos impostos ao sistema cron.

Além dos logs e dos snapshots armazenados no disco local e dos backups de snapshots no sistema de arquivos distribuído, uma réplica recém-iniciada pode buscar o snapshot dos estados e todos os logs de uma réplica já em execução pela rede. Essa capacidade torna a inicialização da réplica independente de qualquer estado na máquina local. Desse modo, reescalonar uma réplica em uma máquina diferente na reinicialização (ou se a máquina morrer) não é essencialmente um problema para a confiabilidade do serviço.

Executando o cron em larga escala

Há outras implicações menores, porém igualmente interessantes, na execução de uma implantação de larga escala do cron. Um cron tradicional é pequeno: no máximo, contém provavelmente da ordem de dezenas de cron jobs. No entanto, se você executar um serviço cron para milhares de máquinas em um datacenter, seu uso crescerá, e você poderá se deparar com problemas.

Esteja ciente de um problema importante e bem conhecido dos sistemas distribuídos: o thundering herd. Conforme a configuração do usuário, o serviço cron pode causar picos significativos no uso do datacenter. Quando as pessoas pensam em um “cron job diário”, elas normalmente configuram esse job para executar à meia-noite. Essa configuração funciona bem se o cron job inicia na mesma máquina, mas e se seu cron job puder iniciar um MapReduce com milhares de workers? E se 30 equipes diferentes decidirem executar um cron job diário como esse no mesmo datacenter? Para resolver esse problema, introduzimos uma extensão ao formato crontab.

No crontab comum, os usuários especificam o minuto, a hora, o dia do mês (ou semana) e o mês em que o cron job deve iniciar, ou asterisco para especificar qualquer valor. Executar à meia-noite, diariamente, teria então uma especificação no crontab igual a "0 0 * * *" (isto é, no minuto zero, na

hora zero, todos os dias do mês, todos os meses e todos os dias da semana). Também introduzimos o uso do ponto de interrogação, que significa que qualquer valor é aceitável, e o sistema cron tem a liberdade de escolher o valor. Os usuários escolhem esse valor calculando o hash da configuração do cron job no intervalo de tempo dado (por exemplo, 0..23 para a hora), distribuindo assim esses inícios de job de modo mais uniforme.

Apesar dessa mudança, a carga provocada pelos cron jobs ainda apresenta muitos picos. O gráfico na Figura 24.3 mostra o número global agregado de inícios de cron jobs no Google. Esse gráfico ilustra os picos frequentes nos inícios dos cron jobs, que muitas vezes são provocados pelos cron jobs que devem ser iniciados em um horário específico – por exemplo, por causa de uma dependência temporal de eventos externos.

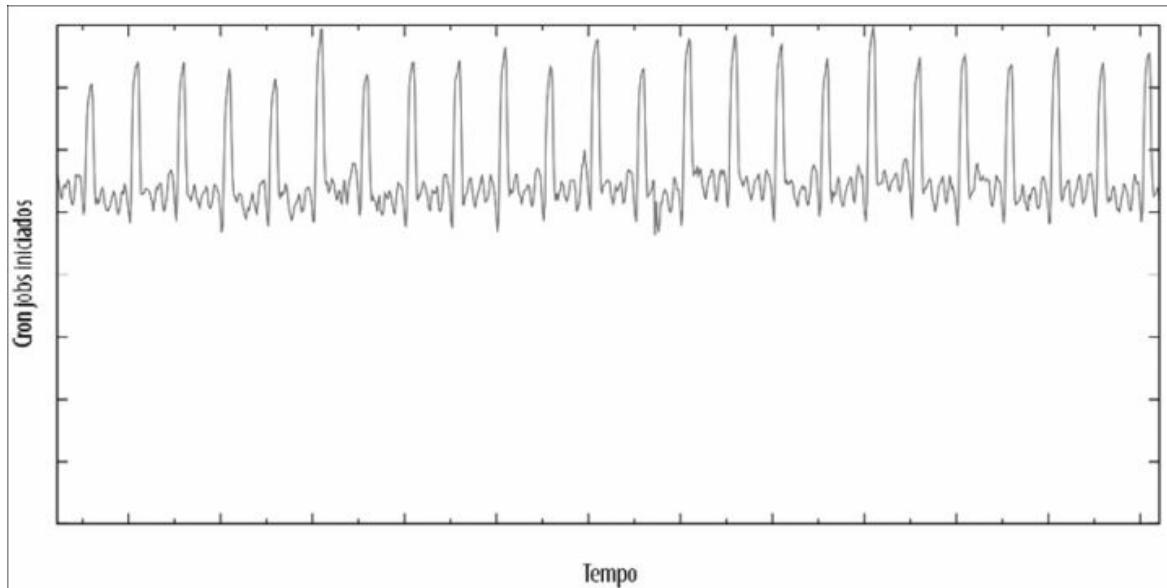


Figura 24.3 – O número de cron jobs iniciados globalmente.

Resumo

Um serviço cron tem sido um recurso fundamental de sistemas Unix há várias décadas. O movimento do mercado em direção a sistemas distribuídos de grande porte, em que um datacenter pode ser a menor unidade efetiva de hardware, exige mudanças em porções grandes da pilha. O cron não é exceção nessa tendência. Uma observação cuidadosa das propriedades exigidas de um serviço cron e dos requisitos dos cron jobs orienta o novo

design do Google.

Discutimos as novas restrições impostas por um ambiente de sistema distribuído e um possível design do serviço cron baseado na solução do Google. Essa solução exige garantias fortes de consistência no ambiente distribuído. Assim, o núcleo da implementação do cron distribuído é o Paxos – um algoritmo comum para alcançar um consenso em um ambiente não confiável. O uso do Paxos e a análise correta de novos modos de falha dos cron jobs em um ambiente distribuído de larga escala nos permitiram criar um serviço cron robusto, intensamente utilizado no Google.

¹ Este capítulo foi anteriormente publicado em parte na *ACM Queue* (março de 2015, vol. 13, nº 3).

² Falhas em jobs individuais estão além do escopo desta análise.

CAPÍTULO 25

Pipelines de processamento de dados

Escrito por Dan Dennison

Editado por Tim Harvey

Este capítulo enfoca os desafios da vida real relacionados à administração de pipelines de processamento de dados profundos e complexos. Serão considerados o contínuo de frequência entre pipelines periódicos que executam bem raramente até pipelines contínuos que nunca param de executar, e as descontinuidades que podem gerar problemas operacionais significativos. Uma nova abordagem do modelo líder-seguidor será apresentada como uma alternativa mais confiável e escalável ao pipeline periódico para processamento de Big Data.

Origem do padrão de projeto pipeline

A abordagem clássica para o processamento de dados consiste em escrever um programa que leia os dados, transforme-os de algum modo desejado e gere novos dados. Geralmente, o programa é agendado para executar sob o controle de um programa de escalonamento periódico, como o cron. Esse padrão de projeto é chamado de *pipeline de dados*. Pipelines de dados são tão antigos quanto as corrotinas [Con63], os arquivos de comunicação DTSS [Bul80], o pipe UNIX [McI86] e, mais tarde, os pipelines ETL¹, mas esses pipelines ganharam uma atenção cada vez maior com o surgimento do “Big Data”, ou “conjuntos de dados tão grandes e tão complexos que as aplicações tradicionais de processamento de dados são inadequadas”.²

Efeito inicial do Big Data no padrão pipeline simples

Os programas que realizam transformações periódicas ou contínuas em Big

Data geralmente são chamados de “pipelines simples monofásicos”.

Dadas a escala e a complexidade do processamento inerente ao Big Data, os programas em geral são organizados como uma série encadeada, com a saída de um programa se tornando a entrada do próximo. Pode haver raciocínios variados por trás dessa organização, mas geralmente ela é projetada para facilitar a compreensão do sistema, em vez de estar voltada à eficiência operacional. Programas organizados dessa maneira são chamados de *pipelines multifase*, pois cada programa na cadeia atua como uma fase de processamento de dados discreta.

O número de programas encadeados em série é uma medida conhecida como a *profundidade* de um pipeline. Assim, um pipeline raso pode ter apenas um programa com uma profundidade de pipeline correspondente igual a um, enquanto um pipeline profundo pode ter uma profundidade de dezenas ou de centenas de programas.

Desafios com o padrão de pipeline periódico

Pipelines periódicos em geral são estáveis quando há workers suficientes para o volume de dados e a demanda para execução está dentro da capacidade de processamento. Além disso, instabilidades como gargalos no processamento são evitadas quando o número de jobs encadeados e o throughput relativo entre jobs permanece uniforme.

Pipelines periódicos são úteis e práticos, e nós os executamos regularmente no Google. Eles são escritos com frameworks como MapReduce [Dea04] e Flume [Cha10], entre outros.

No entanto, segundo a experiência coletiva da SRE, o modelo de pipeline periódico é frágil. Percebemos que quando um pipeline periódico é inicialmente instalado com tamanho de worker, periodicidade, técnica de chunking e outros parâmetros cuidadosamente ajustados, o desempenho é confiável no começo. Contudo, o crescimento orgânico e as mudanças inevitavelmente começam a estressar o sistema, e os problemas surgem. Exemplos de problemas desse tipo incluem jobs que excedem seus prazos de execução, esgotamento de recursos e chunks (porções) de processamento

pendentes que implicam uma carga operacional correspondente.

Problemas causados por distribuição de carga irregular

A principal inovação do Big Data é a aplicação disseminada dos algoritmos “embarrasosamente paralelos” (embarrassingly parallel) [Mol86] para dividir uma carga de trabalho grande em chunks pequenos o suficiente para caber em máquinas individuais. Às vezes, os chunks exigem uma quantidade irregular de recursos em relação a outros chunks, e o motivo pelo qual determinados chunks exigem quantidades diferentes de recursos raramente é óbvio no início. Por exemplo, em uma carga de trabalho particionada pelo cliente, chunks de dados para alguns clientes podem ser muito maiores que outros.

Como o cliente é o ponto de indivisibilidade, o tempo de execução fim a fim é então limitado pelo tempo de execução do maior cliente.

O problema do hanging chunk (porção pendente) pode surgir quando recursos são atribuídos quando há diferenças entre máquinas em um cluster ou superalocação a um job. Esse problema surge devido à dificuldade de algumas operações de tempo real em streams, como ordenar dados de “steaming”. O padrão de código típico de usuário é esperar que o processamento todo seja concluído antes de prosseguir para o próximo estágio do pipeline, normalmente porque uma ordenação pode estar envolvida, o que exige todos os dados para prosseguir. Isso pode atrasar significativamente o tempo de conclusão do pipeline porque ele será bloqueado pelo desempenho de pior caso, conforme determinado pela metodologia de chunking em uso.

Se esse problema for identificado pelos engenheiros ou pela infraestrutura de monitoração de cluster, a resposta poderá piorar a situação. Por exemplo, a resposta “sensata” ou “padrão” para um chunk pendente é matar imediatamente o job e então permitir que ele reinicie, pois o bloqueio pode muito bem ser o resultado de fatores não determinísticos. No entanto, como as implementações de pipeline, por design, geralmente não incluem pontos de verificação (checkpoint), o trabalho em todos os chunks será reiniciado desde

o princípio, fazendo com que haja desperdício de tempo, de ciclos de CPU e de esforços humanos investidos no ciclo anterior.

Desvantagens de pipelines periódicos em ambientes distribuídos

Pipelines periódicos em Big Data são amplamente utilizados no Google; desse modo, a solução de gerenciamento de clusters do Google inclui um mecanismo alternativo de escalonamento desses pipelines. Esse mecanismo é necessário porque, de modo diferente de pipelines que executam continuamente, os pipelines periódicos geralmente executam como jobs batch de baixa prioridade. Uma designação de baixa prioridade funciona bem nesse caso porque os trabalhos em batch não são sensíveis à latência do mesmo modo que o são os serviços web voltados à internet. Além disso, para controlar os custos maximizando a carga de trabalho das máquinas, o Borg (o sistema de gerenciamento de clusters do Google, [Ver15]) atribui tarefas em batch às máquinas disponíveis. Essa prioridade pode resultar em uma degradação na latência de inicialização, de modo que os jobs de pipeline podem possivelmente sofrer atrasos de inicialização sem prazos para terminar.

Os jobs invocados por meio desse mecanismo têm uma série de limitações naturais, resultando em diversos comportamentos distintos. Por exemplo, jobs escalonados nas lacunas deixadas por jobs de serviços web voltados aos usuários podem sofrer impactos em termos de disponibilidade de recursos de baixa latência, custos e estabilidade de acesso aos recursos. Os custos de execução são inversamente proporcionais ao atraso de inicialização solicitado e diretamente proporcionais aos recursos consumidos. Embora o escalonamento de batch possa funcionar suavemente na prática, um uso excessivo do escalonador de batch (Capítulo 24) coloca os jobs em risco de preempções (veja a seção 2.5 de [Ver15]) quando a carga do cluster é alta porque outros usuários sofrem starvation de recursos de batch. Diante das negociações de risco, executar um pipeline periódico bem ajustado com sucesso é conseguir um equilíbrio delicado entre o custo alto de recursos e o risco de preempções.

Atrasos de até algumas horas podem ser aceitáveis para pipelines que executem diariamente. Contudo, à medida que a frequência de execução agendada aumenta, o tempo mínimo entre as execuções pode alcançar rapidamente o ponto de atraso médio mínimo, impondo um limite inferior na latência que um pipeline periódico espera atingir. Reduzir o intervalo de execução do job abaixo desse limite inferior efetivo simplesmente resulta em um comportamento indesejável, em vez de resultar em um aumento de progresso. O modo de falha específico depende da política de escalonamento de batch em uso. Por exemplo, cada nova execução pode se acumular no escalonador do cluster porque a execução anterior não foi concluída. Pior ainda, a execução atual, quase concluída, poderia ser encerrada quando a próxima execução for escalonada para iniciar, interrompendo totalmente todo o progresso em nome do aumento de execuções.

Observe o ponto de intersecção entre a linha de intervalo ocioso descendente e o atraso no escalonamento na Figura 25.1. Nesse cenário, reduzir o intervalo de execução para muito abaixo de 40 minutos para esse job de aproximadamente 20 minutos resulta em possíveis sobreposições de execuções, com consequências indesejadas.

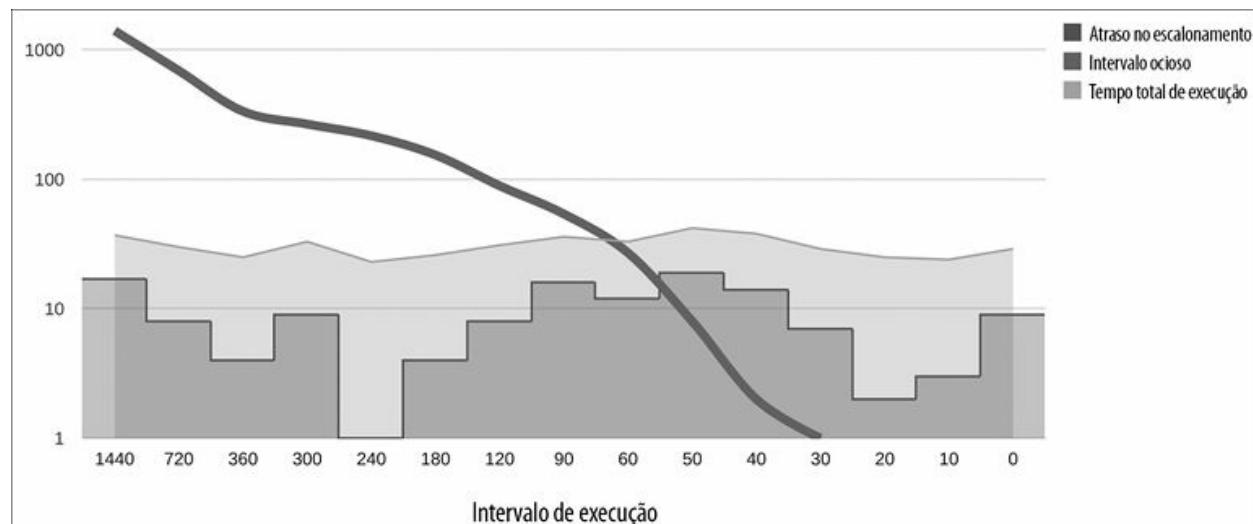


Figura 25.1 – Intervalos de execução de um pipeline periódico versus o tempo de ociosidade (escala logarítmica).

A solução para esse problema é garantir que haja capacidade suficiente no servidor para uma operação apropriada. No entanto, a aquisição de recursos

em um ambiente compartilhado e distribuído está sujeita à oferta e à demanda. Conforme esperado, as equipes de desenvolvimento tendem a ficar relutantes em passar pelos processos de aquisição de recursos quando esses devem contribuir com um pool comum e compartilhado. Para resolver isso, uma distinção entre recursos para escalonamento em batch *versus* recursos com prioridade em produção deve ser feita para racionalizar os custos de aquisição de recursos.

Monitorando problemas em pipelines periódicos

Para pipelines com duração de execução suficiente, ter informações em tempo real sobre as métricas de desempenho em tempo de execução pode ser tão importante, se não for mais, quanto conhecer as métricas em geral. Isso ocorre porque dados em tempo real são importantes para oferecer suporte operacional, incluindo respostas a emergências. Na prática, o modelo-padrão de monitoração envolve coletar métricas durante a execução de jobs e informá-las somente após a conclusão. Se o job falhar durante a execução, nenhuma estatística será fornecida.

Pipelines contínuos não compartilham desses problemas, pois suas tarefas estão constantemente executando e sua telemetria é projetada de modo que as métricas de tempo real estejam disponíveis rotineiramente. Pipelines periódicos não devem ter problemas inerentes de monitoração, mas temos observado uma forte associação.

Problemas de “thundering herd”

Somando-se aos desafios de execução e de monitoração, temos o problema endêmico do “thundering herd” em sistemas distribuídos, também discutido no Capítulo 24. Dado um pipeline periódico bem grande, para cada ciclo, possivelmente milhares de workers começam a trabalhar imediatamente. Se houver workers demais ou se os workers estiverem erroneamente configurados ou forem chamados por uma lógica de retentativas com falha, os servidores em que eles executam ficarão sobrecarregados, assim como os serviços compartilhados do cluster subjacente e qualquer infraestrutura de rede que estava sendo usada.

Para piorar mais ainda essa situação, se a lógica de retentativas não estiver implementada, poderá haver problemas de correção quando trabalhos forem descartados em situação de falha, e não haverá nova tentativa de execução do job. Se a lógica de retentativas estiver presente, mas for ingênuas ou precariamente implementadas, tentar novamente em caso de falha poderá piorar o problema.

Uma intervenção humana também pode colaborar com esse cenário. Engenheiros com experiência limitada em administrar pipelines tendem a amplificar esse problema adicionando mais workers em seu pipeline quando o job não é concluído no período de tempo desejado.

Independentemente da origem do problema de “thundering herd”, nada é mais difícil na infraestrutura de clusters, e aos SREs responsáveis pelos diversos serviços em um cluster, do que um job de pipeline problemático com 10.000 workers.

Padrão de carga Moiré

Às vezes, o problema de “thundering herd” pode não ser tão óbvio para ser identificado isoladamente. Um problema relacionado a ele, que chamamos de “padrão de carga Moiré”, ocorre quando dois ou mais pipelines executam simultaneamente e suas sequências de execução ocasionalmente se sobreponham, fazendo com que eles consumam um recurso compartilhado ao mesmo tempo. Esse problema pode ocorrer mesmo em pipelines contínuos, embora seja menos comum quando a carga chega de maneira mais uniforme.

Os padrões de carga Moiré são mais aparentes em gráficos de uso de recursos compartilhados por pipelines. Por exemplo, a Figura 25.2 identifica o uso de recursos de três pipelines periódicos. Na Figura 25.3, que é uma versão do gráfico anterior com os dados combinados, o impacto de pico que provoca problemas para o plantão ocorre quando a carga agregada se aproxima de 1,2M.

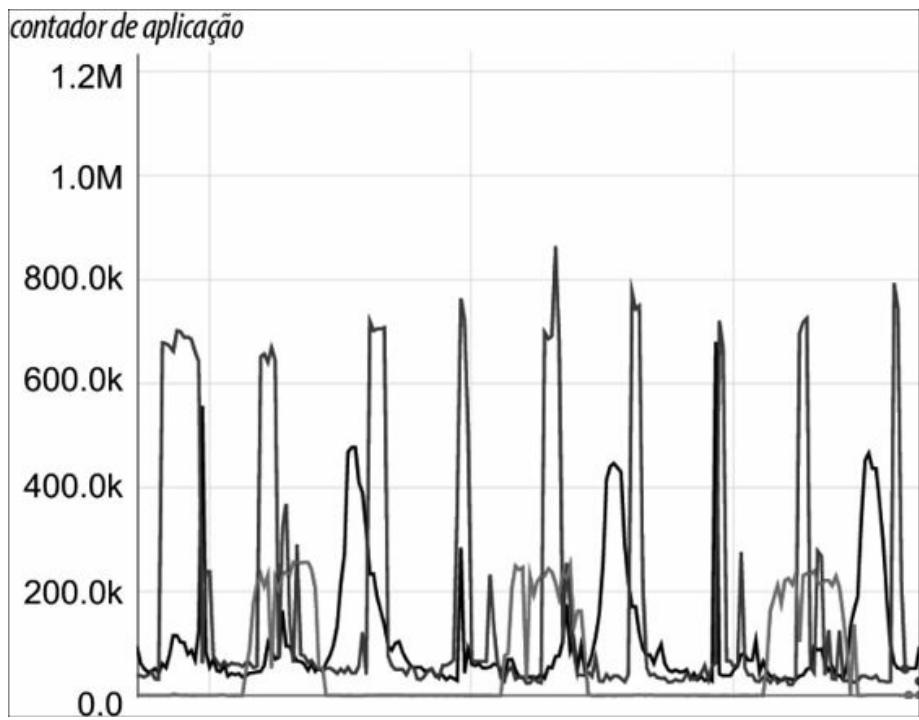


Figura 25.2 – Padrão de carga Moiré em uma infraestrutura separada.

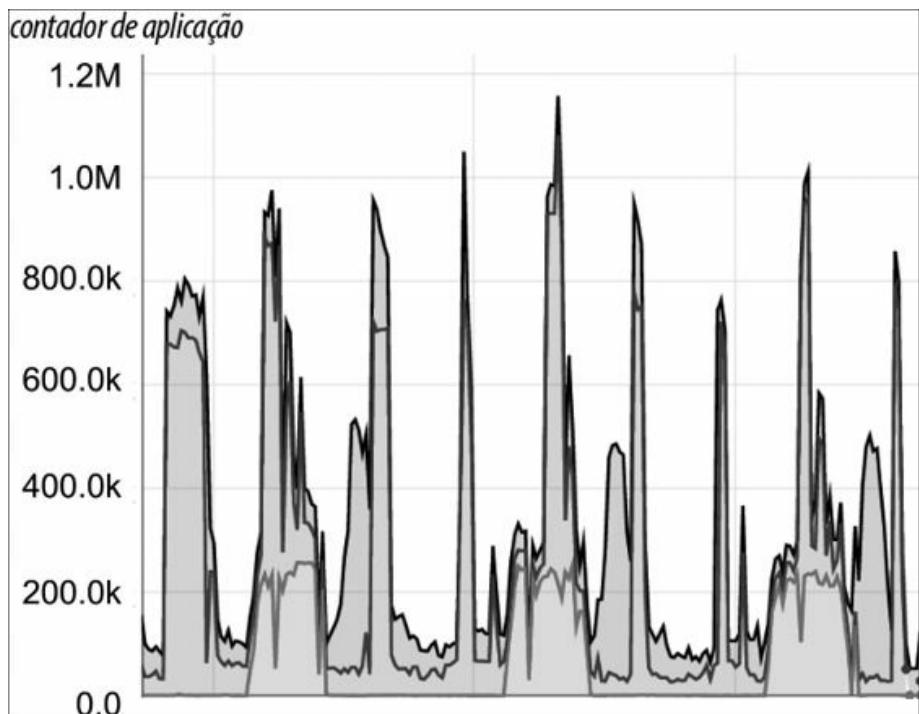


Figura 25.3 – Padrão de carga Moiré em uma infraestrutura compartilhada.

Introdução ao Google Workflow

Quando um pipeline em batch inherentemente executado de uma só vez fica sobrecarregado por demandas de negócio que exigem dados continuamente atualizados, a equipe de desenvolvimento do pipeline geralmente considera refatorar o design original para satisfazer às demandas atuais ou passar para um modelo de pipeline contínuo. Infelizmente, as demandas do negócio em geral ocorrem no momento menos conveniente para refatorar o sistema de pipeline para transformá-lo em um sistema online de processamento contínuo. Clientes mais novos e maiores, que precisam encarar problemas de aumento de escala, geralmente também querem incluir novas funcionalidades, e esperam que esses requisitos sejam atendidos com prazos imutáveis. Para antecipar-se a esse desafio, é importante averiguar diversos detalhes no início do design de um sistema que envolva um pipeline de dados proposto. Certifique-se de determinar o escopo da trajetória de crescimento esperada³, da demanda por modificações de design, dos recursos adicionais e dos requisitos de latência esperados para o negócio.

Diante dessas necessidades, o Google desenvolveu um sistema em 2003, chamado “Workflow”, que faz com que um processamento contínuo esteja disponível em escala. O Workflow utiliza o padrão de projeto de sistemas distribuídos líder-seguidor (workers) [Sha00] e o padrão de projeto prevalência de sistema (system prevalence).⁴ Essa combinação permite ter pipelines de dados transacionais de larga escala, garantindo a correção com uma semântica exactly-once (exatamente uma vez).

Workflow como um padrão Modelo-Visão-Controlador

Por causa do modo como o padrão de projeto prevalência de sistema funciona, talvez seja conveniente pensar no Workflow como o equivalente para sistemas distribuídos do padrão modelo-visão-controlador (model-view-controller), conhecido do desenvolvimento de interface de usuários.⁵ Como mostra a Figura 25.4, esse padrão de projeto divide uma dada aplicação de software em três partes interconectadas para separar as representações internas das informações das maneiras pelas quais essas informações são apresentadas ou aceitas dos usuários.⁶

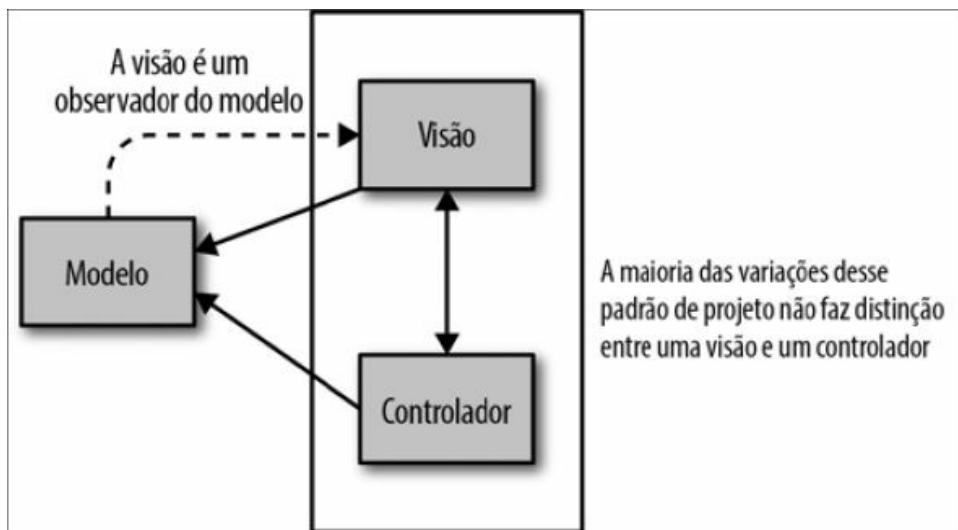


Figura 25.4 – O padrão modelo-visão-controlador usado em design de interface de usuários.

Ao adaptar esse padrão para o Workflow, o *modelo* é mantido em um servidor chamado “Task Master”. O Task Master utiliza o padrão prevalência de sistema para armazenar os estados de todos os jobs em memória a fim de ter disponibilidade rápida, ao mesmo tempo que registra sincronamente as mudanças em um diário (journal) em um disco persistente. A *visão* são os workers que continuamente atualizam o estado do sistema de modo transacional junto ao mestre, de acordo com seus pontos de vista como um subcomponente do pipeline. Embora todos os dados do pipeline possam ser armazenados no Task Master, o melhor desempenho geralmente é conseguido quando apenas ponteiros para o trabalho são armazenados no Task Master, e os dados de entrada e saída propriamente ditos são armazenados em um sistema de arquivos comum ou em outro repositório. Dando suporte a essa analogia, os workers não têm nenhum estado (são stateless) e podem ser descartados a qualquer momento. Um *controlador* pode, opcionalmente, ser adicionado como um terceiro componente do sistema para tratar de modo eficiente uma série de atividades auxiliares do sistema que afetam o pipeline, por exemplo, escalar o pipeline em tempo de execução, criar snapshots, controlar o estado do ciclo de trabalho, fazer rollback do estado do pipeline ou até mesmo fazer uma interdição global para continuidade dos negócios. A Figura 25.5 ilustra o padrão de projeto.

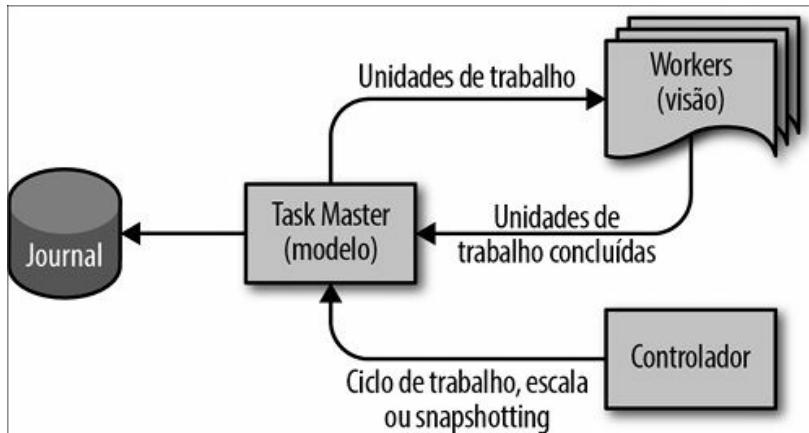


Figura 25.5 – O padrão de projeto modelo-visão-controlador, conforme adaptado para o Google Workflow.

Estágios de execução no Workflow

Podemos aumentar a profundidade do pipeline para qualquer nível no Workflow ao subdividir o processamento em grupos de tarefas mantidos no Task Master. Cada grupo de tarefas mantém o trabalho correspondente a um estágio do pipeline, que pode realizar operações arbitrárias em algum dado. É relativamente simples mapear, embaralhar, ordenar, separar, combinar ou fazer qualquer outra operação em qualquer estágio.

Um estágio geralmente tem algum tipo de worker associado. Pode haver várias instâncias concorrentes de um dado tipo de worker, e os workers podem se autoescalonar, no sentido de que podem procurar diferentes tipos de trabalho e escolher qual deles realizar.

O worker consome unidades de trabalho de um estágio anterior e gera unidades de saída. A saída pode ser um ponto final ou uma entrada para outro estágio do processamento. No sistema, é fácil garantir que todo o trabalho seja executado ou, pelo menos, refletido em um estado permanente, exatamente uma vez.

Garantia de que o Workflow está correto

Não é prático armazenar *todos* os detalhes do estado do pipeline no Task Master, pois ele é limitado pelo tamanho da RAM. No entanto, uma garantia dupla de correção persiste, pois o mestre armazena uma coleção de ponteiros

para dados unicamente nomeados, e cada unidade de trabalho tem um lease mantido de forma exclusiva. Os workers adquirem trabalho com um lease e podem realizar trabalhos de tarefas para as quais eles possuam um lease válido no momento.

Para evitar a situação em que um worker órfão possa continuar trabalhando em uma unidade de trabalho, destruindo assim o trabalho do worker atual, cada arquivo de saída aberto por um worker tem um nome único. Dessa maneira, mesmo workers órfãos são capazes de continuar escrevendo de forma independente do mestre, até tentarem fazer commit. Ao tentar um commit, eles serão incapazes de fazê-lo, pois outro worker tem o lease para essa unidade de trabalho. Além do mais, workers órfãos não podem destruir o trabalho gerado por um worker válido, pois o esquema de nome de arquivo único garante que todo worker escreva em um arquivo distinto. Dessa maneira, a garantia dupla de correção se mantém: os arquivos de saída são sempre únicos e o estado do pipeline está sempre correto por causa das tarefas com leases.

Como se uma garantia dupla de correção não fosse suficiente, o Workflow também atribui versões a todas as tarefas. Se a tarefa for atualizada ou se seu lease mudar, cada operação gerará uma nova tarefa única, substituindo a anterior, com um novo ID atribuído à tarefa. Como toda a configuração do pipeline no Workflow é armazenada no Task Master no mesmo formato que as próprias unidades de trabalho, para fazer commit de um trabalho, um worker deve possuir um lease ativo e referenciar o número de ID da tarefa da configuração usada para gerar seu resultado. Se a configuração mudou enquanto a unidade de trabalho estava em andamento, todos os workers desse tipo serão incapazes de fazer commit, apesar de estarem de posse dos leases atuais. Assim, todo trabalho realizado após uma mudança de configuração será consistente com a nova configuração, à custa de jogar fora o trabalho feito por workers que, infelizmente, estavam com leases antigos.

Essas medidas oferecem uma garantia tripla de correção: configuração, posse de lease e unicidade nos nomes de arquivos. No entanto, mesmo isso não é suficiente para todos os casos.

Por exemplo, o que aconteceria se o endereço de rede do Task Master

mudasse e um Task Master diferente o substituísse no mesmo endereço? E se uma memória corrompida alterasse o endereço IP ou o número da porta, resultando em outro Task Master na outra extremidade? Mais comum ainda, e se alguém configurasse (erroneamente) o seu Task Master inserindo um distribuidor de carga na frente de um conjunto de Task Masters independentes?

O Workflow inclui um token de servidor, que é um identificador único para esse Task Master em particular, nos metadados de cada tarefa para evitar que um Task Master falso ou incorretamente configurado corrompa o pipeline. Tanto o cliente quanto o servidor verificam o token a cada operação, evitando um erro muito sutil de configuração, em que todas as operações executem suavemente até que uma colisão de identificador de tarefa ocorra.

Em suma, as quatro garantias de correção do Workflow são:

- A saída do worker por meio de tarefas de configuração cria barreiras para garantir o trabalho.
- Todos os trabalhos para os quais foi feito um commit exigem que o worker esteja de posse de um lease válido no momento.
- Os arquivos de saída são unicamente nomeados pelos workers.
- O cliente e o servidor validam o próprio Task Master verificando um token de servidor em todas as operações.

A essa altura, talvez ocorra a você que seria mais simples deixar de lado o Task Master especializado e utilizar o Spanner [Cor12] ou outro banco de dados. No entanto, o Workflow é especial porque cada tarefa é única e imutável. Essas duas propriedades evitam que muitos problemas potencialmente sutis com distribuição de trabalho em larga escala ocorram.

Por exemplo, o lease obtido pelo worker faz parte da própria tarefa, exigindo uma nova tarefa, mesmo para mudanças de lease. Se um banco de dados for usado diretamente e seus logs de transação atuarem como um “diário” (journal), toda e qualquer leitura deverá fazer parte de uma transação de execução demorada. É quase certo que essa configuração seja possível, porém é terrivelmente ineficiente.

Garantindo a continuidade do negócio

Pipelines de Big Data devem continuar a processar, apesar de falhas de todos os tipos, incluindo rompimento de fibra, fenômenos da natureza e falhas em cascata da rede elétrica. Esses tipos de falha podem deixar datacenters inteiros inativos. Além do mais, os pipelines que não empregam a prevalência de sistema para obter garantias robustas sobre a conclusão de jobs muitas vezes são desabilitados e entram em um estado indefinido. Essa lacuna na arquitetura resulta em uma estratégia de continuidade de negócios frágil e implica uma duplicação em massa custosa de esforços para restaurar os pipelines e os dados.

O Workflow resolve esse problema de modo conclusivo para pipelines de processamento contínuo. Para ter consistência global, o Task Master armazena diários no Spanner, usando-o como um sistema de arquivos globalmente disponível e consistente, mas com baixo throughput. Para determinar qual Task Master pode escrever, cada Task Master utiliza o serviço de lock distribuído chamado Chubby [Bur06] para eleger quem vai escrever, e o resultado é armazenado no Spanner de forma persistente. Por fim, os clientes consultam o Task Master atual usando serviços internos de nomes.

Como o Spanner não é um sistema de arquivos com throughput alto, os Workflows distribuídos globalmente empregam dois ou mais Workflows locais executando em clusters distintos, além de uma noção de tarefas de referência armazenadas no Workflow global. À medida que unidades de trabalho (tarefas) são consumidas por meio de um pipeline, tarefas de referência equivalentes são inseridas no Workflow global pelo binário identificado como “estágio 1” na Figura 25.6. Conforme as tarefas terminam, as tarefas de referência são removidas de modo transacional do Workflow global, como representado pelo “estágio n” da Figura 25.6. Se as tarefas não puderem ser removidas do Workflow global, o Workflow local ficará bloqueado até que o Workflow global se torne disponível novamente, garantindo a correção transacional.

Para automatizar o failover, um binário auxiliar identificado como “estágio 1” na Figura 25.6 é executado em cada Workflow local. O Workflow local,

por outro lado, não é alterado, conforme descrito na caixa “executa tarefas” no diagrama. Esse binário auxiliar atua como um “controlador” no sentido MVC, e é responsável por criar tarefas de referência, assim como atualizar uma tarefa especial de heartbeat no Workflow global. Se a tarefa de heartbeat não for atualizada durante o período de tempo até o timeout, o binário auxiliar do Workflow remoto assume o trabalho em progresso, conforme documentado pelas tarefas de referência, e o pipeline continua, sem ser importunado com o que quer que o ambiente possa fazer com o trabalho.

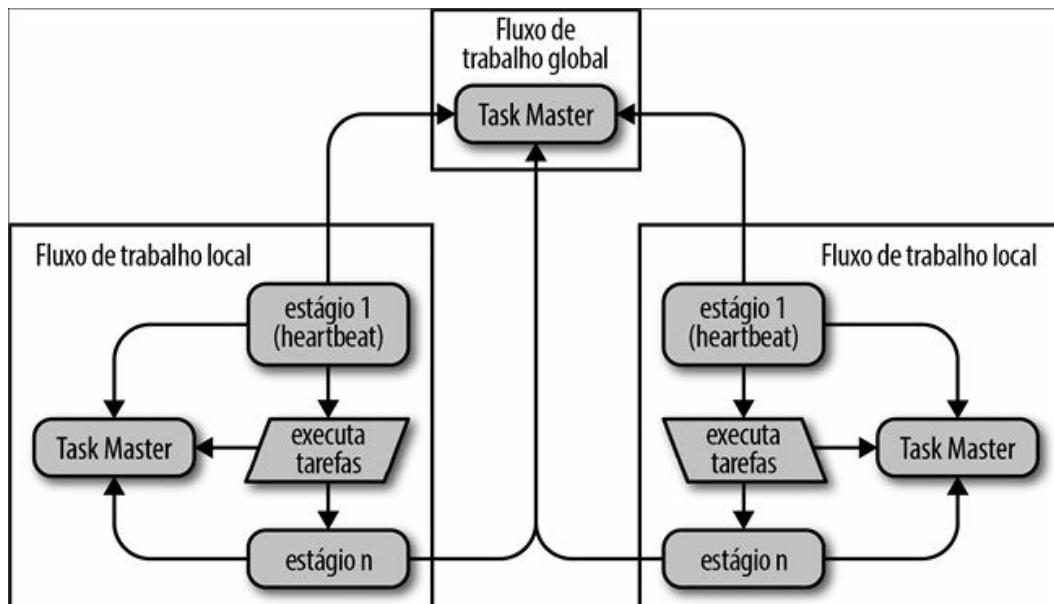


Figura 25.6 – Um exemplo de dados distribuídos e fluxo de processos usando pipelines Workflow.

Resumo e considerações finais

Os pipelines periódicos são importantes. No entanto, se um problema de processamento de dados é contínuo ou crescerá organicamente até se tornar contínuo, não utilize um pipeline periódico. Em vez disso, utilize uma tecnologia com características semelhantes à do Workflow.

Percebemos que um processamento de dados contínuo com garantias robustas, conforme oferecido pelo Workflow, apresenta um bom desempenho e capacidade de escalar em uma infraestrutura de clusters distribuídos, geralmente produz resultados com os quais os usuários podem contar e é um

sistema estável e confiável para a equipe de Site Reliability Engineering (Engenharia de Confiabilidade de Sites) administrar e manter.

- 1 Wikipedia: Extract, Transform, Load (Extrair, Transformar, Carregar),
http://en.wikipedia.org/wiki/Extract,_transform,_load
- 2 Wikipedia: Big data, http://en.wikipedia.org/wiki/Big_data
- 3 A palestra de Jeff Dean em “Software Engineering Advice from Building Large-Scale Distributed Systems” (Conselhos para engenharia de software baseados na criação de sistemas distribuídos de larga escala) é um excelente recurso [Dea07].
- 4 Wikipedia: System Prevalence (Prevalência de sistema),
http://en.wikipedia.org/wiki/System_Prevalence
- 5 O padrão “modelo-visão-controlador” é uma analogia para sistemas distribuídos, emprestada de Smalltalk de forma muito livre, e originalmente usado para descrever a estrutura de design das interfaces gráficas de usuário [Fow08].
- 6 Wikipedia: Model-view-controller, Model-view-controller,
<http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>

CAPÍTULO 26

Integridade de 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 estão em primeiro lugar, a integridade é aquilo que eles acham que ela é.

Podemos dizer que *a integridade de dados é uma medida da acessibilidade e da precisão dos repositórios de dados, necessárias para oferecer um nível de serviço adequado aos usuários*. Contudo, essa definição é insuficiente.

Por exemplo, se um bug na interface de usuário do Gmail exibir uma caixa de correio vazia por tempo demais, os usuários poderão acreditar que dados foram perdidos. Desse modo, mesmo que nenhum dado tenha sido *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 de manutenção por tempo demais enquanto “apenas alguns metadados” estão sendo corrigidos, a confiança dos usuários do Google, de modo semelhante, sofreria prejuízos.

Quanto é “tempo demais” para os dados ficarem indisponíveis? Conforme demonstrado por um incidente real com o Gmail em 2011 [Hic11], quatro dias é muito tempo – talvez “tempo demais”. Como consequência, acreditamos que 24 horas é um bom ponto de partida para determinar o limite para “tempo demais” nos Google Apps.

Um raciocínio semelhante pode ser usado em aplicações como Google Photos, Drive, Cloud Storage e Cloud Datastore, pois os usuários não

necessariamente fazem distinção entre esses produtos discretos (pensando: “este produto ainda é do Google” ou “Google, Amazon, não importa; este produto ainda faz parte da nuvem”). Perda de dados, dados corrompidos e indisponibilidade estendida geralmente são indistinguíveis aos usuários. Desse modo, a integridade de dados se aplica a todos os tipos de dados em todos os serviços. Ao considerar a integridade de dados, o que importa é que *os serviços na nuvem permaneçam acessíveis aos usuários. O acesso dos usuários aos dados é especialmente importante.*

Requisitos rigorosos da integridade de dados

Ao considerar as necessidades de confiabilidade de um dado sistema, pode parecer que as necessidades de uptime (disponibilidade do serviço) sejam mais rigorosas que as de integridade de dados. Por exemplo, os usuários podem achar que uma hora de downtime de email seja inaceitável, mas poderiam ficar de mau-humor durante um período de quatro dias para recuperar uma caixa de correio. No entanto, há uma maneira mais apropriada de considerar as exigências de uptime *versus* integridade de dados.

Um SLO de 99,99% de uptime deixa espaço para apenas uma hora de downtime em um ano todo. Esse SLO define uma meta bem alta, que provavelmente excede as expectativas da maioria dos usuários de internet e dos usuários corporativos.

Em comparação, um SLO de 99,99% de bytes bons em um artefato de 2 GB resultaria em documentos, executáveis e bancos de dados corrompidos (até 200 KB adulterados). Esse nível de dados corrompidos é *catastrófico* na maioria dos casos – resultando em executáveis com opcodes aleatórios e bancos de dados totalmente impossíveis de carregar.

Do ponto de vista dos usuários, então, todo serviço tem requisitos independentes de uptime e de integridade de dados, mesmo que esses requisitos sejam implícitos. A pior hora para discordar dos usuários a respeito desses requisitos é depois da perda de seus dados!



Para revisar nossa definição anterior de integridade de dados, podemos dizer que *integridade de dados significa que os serviços na nuvem devem permanecer acessíveis aos usuários*. O acesso dos usuários aos dados é especialmente importante, portanto esse acesso deve permanecer em ótimo estado.

Agora suponha que um artefato seja corrompido ou perdido exatamente uma vez por ano. Se a perda for irrecuperável, o uptime do artefato afetado será *perdido* naquele ano. O meio mais provável de evitar uma perda como essa é por meio de uma detecção proativa, em conjunto com uma correção rápida.

Em um universo alternativo, suponha que os dados corrompidos tenham sido imediatamente detectados antes de os usuários terem sido afetados e que o artefato tenha sido removido, corrigido e devolvido ao serviço em meia hora. Ignorando qualquer outro downtime durante esses 30 minutos, um objeto como esse estaria disponível 99,99% naquele ano.

De modo surpreendente, pelo menos do ponto de vista do usuário, nesse cenário, a integridade de dados continua sendo de 100% (ou próxima a 100%) durante o tempo de vida acessível do objeto. Conforme mostrado nesse exemplo, o *segredo para uma integridade de dados superior é uma detecção proativa e uma correção e recuperação rápidas*.

Escolhendo uma estratégia para uma integridade de dados superior

Há várias estratégias possíveis para detecção rápida, correção e recuperação de dados perdidos. Todas essas estratégias implicam uma negociação entre uptime e integridade de dados no que diz respeito aos usuários afetados. Algumas estratégias funcionam melhor do que outras, e algumas exigem um investimento mais complexo de engenharia. Com tantas opções disponíveis, quais estratégias você deve usar? A resposta depende de seu paradigma

computacional.

A maioria das aplicações de computação em nuvem procura se otimizar em relação a uma combinação de uptime, latência, escala, velocidade e privacidade. A seguir, apresentamos uma definição funcional para cada um desses termos:

Uptime

Também chamado de *disponibilidade*, é a proporção de tempo em que um serviço é utilizável por seus usuários.

Latência

O quanto responsivo um serviço se apresenta aos seus usuários.

Escala

O volume de usuários de um serviço e a combinação de cargas de trabalho que o serviço é capaz de tratar antes que a latência sofra ou o serviço se desintegre.

Velocidade

A rapidez com que um serviço é capaz de se inovar para oferecer mais valor aos usuários a um custo razoável.

Privacidade

Esse conceito impõe requisitos complexos. Como uma simplificação, este capítulo limita seu escopo à discussão de privacidade na remoção de dados: os dados devem ser destruídos em um período de tempo razoável após os usuários os terem apagado.

Muitas aplicações em nuvem evoluem continuamente com base em uma combinação de APIs ACID¹ e BASE² para atender às demandas desses cinco componentes.³ BASE permite mais disponibilidade que ACID em troca de uma garantia menos rigorosa de consistência distribuída. Especificamente, BASE apenas garante que, depois que um dado não estiver mais atualizado, seu valor se tornará consistente *em algum momento* (eventually) nos repositórios (potencialmente distribuídos).

O cenário a seguir apresenta um exemplo de como as negociações entre

uptime, latência, escala, velocidade e privacidade podem ocorrer.

Quando a velocidade triunfa sobre os demais requisitos, as aplicações resultantes contam com uma coleção arbitrária de APIs que são mais familiares aos desenvolvedores que trabalham na aplicação.

Por exemplo, uma aplicação pode tirar proveito de uma API de repositório BLOB⁴ eficiente, como o Blobstore, que negligencia a consistência distribuída em favor da escala para cargas de trabalho intensas com uptime alto, baixa latência e baixo custo. Para compensar:

- A mesma aplicação pode confiar pequenas quantidades de metadados autoritativos pertencentes aos seus blobs a um serviço baseado em Paxos, com mais latência, menos disponibilidade e custo mais alto, como o Megastore [Bak11], [Lam98].
- Determinados clientes da aplicação podem fazer cache de parte desses metadados localmente e acessar os blobs de forma direta, reduzindo mais ainda a latência do ponto de vista dos usuários.
- Outra aplicação pode manter metadados no Bigtable, sacrificando uma consistência distribuída forte porque seus desenvolvedores, por acaso, tinham familiaridade com o Bigtable.

Aplicações em nuvem como essas enfrentam uma variedade de desafios de integridade de dados em tempo de execução, como integridade referencial entre repositórios de dados (no exemplo anterior, entre Blobstore, Megastore e caches do lado cliente). Os caprichos resultantes da alta velocidade determinam que as mudanças em esquema, as migrações de dados, o empilhamento de novas funcionalidades sobre as antigas, as reescritas e a evolução de pontos de integração com outras aplicações conspirem para produzir um ambiente repleto de relacionamentos complexos entre vários dados que nenhum engenheiro dominará completamente.

Para evitar que os dados de uma aplicação desse tipo se degradem diante dos olhos dos usuários, um sistema out-of-band (fora da banda) de verificações e balanceamentos é necessário nos repositórios de dados e entre eles. A seção “Terceira camada: detecção precoce”, discute um sistema desse tipo.

Além do mais, se uma aplicação como essa depender de backups

independentes e não coordenados de vários repositórios de dados (no exemplo anterior, Blobstore e Megastore), então sua capacidade de fazer um uso eficiente dos dados restaurados durante um esforço de recuperação de dados será complicada devido à variedade de relacionamentos entre os dados restaurados e os dados ativos. Nossa aplicação de exemplo teria que classificar e fazer a distinção entre blobs restaurados *versus* Megastore ativo, Megastore restaurado *versus* blobs ativos, blobs restaurados *versus* Megastore restaurado e interações com caches do lado cliente.

Considerando essas dependências e complicações, qual é o volume de recursos que deverá ser investido nos esforços de integridade de dados e onde devemos investi-lo?

Backups versus arquivamentos

Tradicionalmente, as empresas “protegem” dados contra a perda investindo em estratégias de backup. No entanto, o foco real de esforços de backup como esse deve estar na recuperação de dados, que distingue backups *reais* de arquivamentos (archives). Como às vezes se observa, ninguém realmente *quer* fazer backups; o que as pessoas *realmente* querem são as *recuperações*.

Será que seu “backup” é, na verdade, um arquivamento, em vez de estar apropriado para uso na recuperação de desastres?



A diferença mais importante entre backups e arquivamentos é que os backups *podem* ser carregados de volta em uma aplicação, enquanto os archives *não podem*. Desse modo, backups e arquivamentos têm casos de uso bem distintos.

Os *archives* guardam os dados de forma segura por longos períodos de tempo para atender às necessidades de auditoria, descoberta e conformidade. A recuperação de dados para esses propósitos geralmente não precisa ser

concluída dentro dos requisitos de uptime de um serviço. Por exemplo, talvez você precise reter dados de transações financeiras durante sete anos. Para alcançar essa meta, você poderia transferir logs de auditoria acumulados para um repositório de arquivamento de longo prazo, em um local externo, uma vez por mês. Obter e recuperar os logs durante uma auditoria financeira de um mês pode demorar uma semana, e essa janela de uma semana para recuperação pode ser aceitável para um arquivamento.

Por outro lado, quando um desastre ocorre, os dados devem ser recuperados rapidamente de *backups reais*, de preferência, bem dentro das necessidades de uptime de um serviço. Caso contrário, os usuários afetados ficarão sem acesso útil à aplicação, desde o início do problema com a integridade dos dados até o término dos esforços de recuperação.

Também é importante considerar que, como os dados mais recentes estão correndo riscos até que um backup seja feito de modo seguro, talvez seja ideal agendar backups reais (em oposição a arquivamentos) diariamente, toda hora ou com mais frequência, usando abordagens completas e incrementais ou contínuas (*streaming*).

Assim, ao formular uma estratégia de backup, considere a rapidez com que você deve ser capaz de se recuperar de um problema e quantos dados recentes você pode se dar ao luxo de perder.

Requisitos do ambiente de nuvem em perspectiva

Os ambientes na nuvem introduzem uma combinação única de desafios técnicos:

- Se o ambiente utiliza uma mistura de soluções de backup e recuperação transacionais e não transacionais, os dados recuperados não estarão necessariamente corretos.
- Se os serviços devem evoluir sem serem tirados do ar para manutenção, diferentes versões da lógica de negócios poderão atuar nos dados em paralelo.
- Se os serviços em interação receberem versões de modo independente, versões incompatíveis de diferentes serviços poderão interagir

momentaneamente, aumentando mais as chances de os dados serem corrompidos ou perdidos por acidente.

Além do mais, para manter a economia de escala, os provedores de serviços devem oferecer apenas um número limitado de APIs. Essas APIs devem ser simples e fáceis de usar para a grande maioria das aplicações; caso contrário, poucos clientes as usarão. Ao mesmo tempo, as APIs devem ser suficientemente robustas para entender o seguinte:

- Localidade e caching dos dados
- Distribuição de dados local e global
- Consistência forte e/ou eventual
- Durabilidade dos dados, backup e recuperação

Do contrário, clientes sofisticados não poderão migrar as aplicações para a nuvem e aplicações simples que crescerem em tamanho e complexidade precisarão de reescritas completas para utilizar APIs diferentes e mais complexas.

Quando as funcionalidades de API mencionadas antes forem usadas em determinadas combinações, poderão surgir problemas. Se o provedor do serviço não resolver esses problemas, as aplicações que se depararem com esses desafios deverão identificá-los e resolvê-los de forma independente.

Objetivos da SRE do Google na manutenção da integridade de dados e da disponibilidade

Embora o objetivo da SRE de “manter a integridade de dados persistentes” seja uma boa visão, nós nos destacamos em objetivos concretos com indicadores mensuráveis. A SRE define métricas essenciais que usamos para definir expectativas para as capacidades de nossos sistemas e processos por meio de testes e para monitorar seu desempenho durante um evento real.

A integridade dos dados é o meio, a disponibilidade é a meta

A integridade de dados refere-se à precisão e à consistência dos dados durante seu tempo de vida. Os usuários devem saber que as informações

estarão corretas e que não mudarão de alguma forma inesperada entre a primeira vez em que foram registradas e a última em que forem observadas. No entanto, essa garantia é suficiente?

Considere o caso de um provedor de emails que sofreu uma interrupção de uma semana no serviço [Kinc09]. Em um intervalo de dez dias, os usuários tiveram que encontrar outros métodos temporários para conduzir seus negócios, com a expectativa de que logo retornariam às suas contas de email definidas, suas identidades e os históricos acumulados.

Então, a pior notícia possível chegou: o provedor anunciou que, apesar das expectativas anteriores, o baú de emails e contatos passados, na verdade, estava perdido – evaporou-se e jamais seria visto novamente. Parece que uma série de contratemplos na administração da integridade de dados havia conspirado para deixar o provedor do serviço sem backups utilizáveis. Usuários furiosos ficaram presos às suas identidades criadas nesse ínterim ou criaram novas identidades, abandonando seu antigo provedor de emails problemático.

Mas, espere! Vários dias depois da declaração de perda absoluta dos dados, o provedor anunciou que as informações pessoais dos usuários *poderiam* ser recuperadas. Não houve perda de dados; foi apenas uma interrupção no serviço. Tudo estava bem!

Exceto que *tudo não estava bem*. Os dados dos usuários haviam sido preservados, mas ficaram inacessíveis às pessoas que precisavam deles por tempo demais.

Moral desse exemplo: do ponto de vista do usuário, a integridade de dados sem a disponibilidade esperada e regular é efetivamente o mesmo que não ter nenhum dado.

Entregando um sistema de recuperação, e não um sistema de backup

Fazer backups é uma tarefa classicamente negligenciada, delegada e postergada na administração de sistemas. Os backups não têm alta prioridade para ninguém – consomem tempo e recursos continuamente e não proporcionam nenhum benefício imediato visível. Por esse motivo, uma falta

de zelo em implementar uma estratégia de backup geralmente é vista com um vóler de olhos complacente. Alguém pode argumentar que, assim como para a maioria das medidas de proteção contra perigos de baixo risco, uma atitude como essa é pragmática. O problema fundamental com essa estratégia que deixa a desejar é que os perigos que ela implica podem ter risco baixo, mas também têm alto impacto. Quando os dados de seu serviço estiverem indisponíveis, sua resposta poderá ser determinante para o serviço, o produto ou até mesmo para a sua empresa.

Em vez de ter como foco o trabalho ingrato de fazer um backup, é muito mais útil, é até mais fácil, motivar a participação em fazer backups, concentrando-se em uma tarefa que tenha uma recompensa visível: a *restauração!*! Os *backups* são como um imposto, que é pago continuamente para o serviço municipal pela garantia da disponibilidade dos dados. Em vez de enfatizar o imposto, desvie sua atenção para o serviço custeado por ele: a disponibilidade dos dados. Não fazemos as equipes “treinarem” seus backups; em vez disso:

- As equipes definem SLOs (Service Level Objectives, ou Objetivos de Nível de Serviço) para a disponibilidade de dados em diversos modos de falha.
- Uma equipe treina e demonstra sua capacidade em atender a esses SLOs.

Tipos de falha que levam à perda de dados

Conforme mostra a Figura 26.1, em nível bem geral, há 24 tipos distintos de falhas quando os três fatores podem ocorrer em qualquer combinação. Você deve considerar cada uma dessas falhas em potencial quando fizer o design de um programa de integridade de dados. Os fatores que determinam os modos de falha de integridade de dados são:

Causa-raiz

Uma perda irrecuperável de dados pode ser causada por uma série de fatores: ação do usuário, erro do operador, bugs na aplicação, defeitos na infraestrutura, hardware com problema ou catástrofes no site.

Escopo

Algumas perdas se espalham, afetando muitas entidades. Outras são

restritas e direcionadas, apagando ou corrompendo dados específicos a um pequeno subconjunto de usuários.

Classificação

Algumas perdas de dados são um evento do tipo Big Bang (por exemplo, um milhão de linhas são substituídas por apenas dez linhas em um único minuto), enquanto outras são insidiosas (por exemplo, dez linhas de dados são apagadas por minuto no curso de semanas).

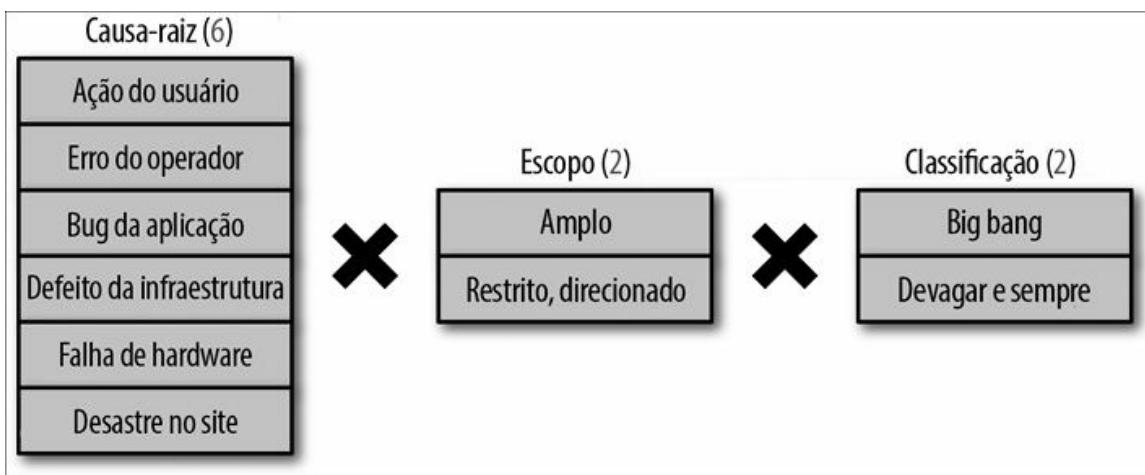


Figura 26.1 – Fatores que determinam os modos de falha de integridade de dados.

Um plano de restauração eficiente deve levar em conta qualquer um desses modos de falha ocorrendo em qualquer combinação concebível. O que pode ser uma estratégia perfeitamente eficiente para se proteger contra uma perda de dados causada por um bug de aplicação insidioso pode não ser de nenhuma ajuda se seu datacenter com colocation (compartilhamento de localização) pegar fogo.

Um estudo de 19 esforços de recuperação de dados no Google revelou que os cenários mais comuns de perda de dados visíveis aos usuários envolveram remoção de dados ou perda de integridade referencial provocadas por bugs de software. As variantes mais desafiadoras envolveram dados mais discretos apagados ou corrompidos, descobertos semanas ou meses após os bugs terem sido introduzidos no ambiente de produção. Desse modo, as proteções empregadas pelo Google devem ser bem adequadas para evitar esses tipos de perda ou para recuperar-se deles.

Para se recuperar desses cenários, uma aplicação grande e bem-sucedida deve recuperar dados, quem sabe de milhões de usuários, distribuídos em dias, semanas ou meses. A aplicação talvez precise recuperar também cada artefato afetado em relação a um único ponto no tempo. Esse cenário de recuperação de dados se chama “recuperação de um ponto no tempo” (point-in-time recovery) fora do Google e “viagem no tempo” (time-travel) no Google.

Uma solução de backup e recuperação que ofereça uma recuperação em um ponto no tempo para uma aplicação em seus repositórios de dados ACID e BASE, ao mesmo tempo que atenda a metas rigorosas de uptime, latência, escalabilidade, velocidade e custo, é uma quimera nos dias de hoje!

Resolver esse problema com seus próprios engenheiros implica sacrificar a velocidade. Muitos projetos fazem um compromisso adotando uma estratégia de backup em camadas, sem recuperação em um ponto no tempo. Por exemplo, as APIs abaixo de sua aplicação podem ter suporte para uma variedade de mecanismos de recuperação de dados. “Snapshots” locais custosos podem oferecer uma proteção limitada contra bugs de aplicação e uma funcionalidade de restauração rápida, portanto você pode manter esses tipos de “snapshots” locais para alguns dias, obtidos a intervalos de algumas horas. Cópias completas e incrementais de custo eficiente feitas a cada dois dias podem ser mantidas por mais tempo. A recuperação em um ponto no tempo é uma funcionalidade muito interessante para ter se uma ou mais dessas estratégias tiverem suporte a ela.

Considere as opções de recuperação de dados oferecidas pelas APIs de nuvem que você está prestes a usar. Faça a análise de custo-benefício entre uma recuperação em um ponto no tempo e uma estratégia em camadas se for necessário, mas não deixe de usar uma delas! Se você puder ter as duas funcionalidades, faça isso. Cada uma dessas funcionalidades (ou ambas) será valiosa em algum momento.

Desafios de manter a integridade de dados ampla e profunda

Ao fazer o design de um programa de integridade de dados, é importante reconhecer que *a replicação e a redundância não equivalem à capacidade de recuperação*.

Problemas de escalabilidade: backups e restaurações completas e incrementais e as forças em disputa

Uma resposta clássica, porém incorreta, à pergunta “Você tem um backup?” é “Temos algo melhor ainda que um backup – temos replicação!”. A replicação proporciona muitas vantagens, incluindo localidade dos dados e proteção contra um desastre específico a um local, mas não poderá proteger você de várias causas de perda de dados. Os repositórios de dados que fazem a sincronização automática de várias réplicas garantem que uma linha de banco de dados corrompida ou uma remoção errática sejam enviadas a todas as suas cópias, provavelmente antes de você poder isolar o problema.

Para cuidar dessa preocupação, você pode fazer cópias não servidoras de seus dados em algum outro formato, como exportações frequentes do banco de dados para um arquivo nativo. Essa medida adicional agrega proteção para os tipos de erros contra os quais a replicação não protege – erros de usuários e bugs na camada de aplicação –, mas não serve para se proteger contra perdas introduzidas em uma camada inferior. Essa medida também introduz um risco de bugs durante a conversão de dados (nas duas direções) e durante a armazenagem do arquivo nativo, além de possíveis incompatibilidades de semântica entre os dois formatos. Pense em um ataque dia zero (zero-day)⁵ em algum nível baixo de sua pilha, por exemplo, no sistema de arquivos ou no device driver. Qualquer cópia que conte com o componente de software comprometido, incluindo as exportações de banco de dados gravadas no mesmo sistema de arquivos que tem o backup de seu banco de dados, estará vulnerável.

Desse modo, vemos que a diversidade é fundamental: proteger-se contra uma falha na camada X exige armazenar dados em componentes diversificados nessa camada. O isolamento de mídia protege contra falhas de mídia: um bug ou um ataque em um device driver de disco provavelmente não afetará drives de fita. Se pudéssemos, faríamos cópias de backup de nossos dados valiosos em tabuletas de argila.⁶

As forças entre dados mais recentes e restauração completa competem contra uma proteção abrangente. Quanto mais para baixo na pilha você fizer um snapshot de seus dados, mais demorará para uma cópia ser feita, o que

significa que a frequência das cópias diminuirá. No nível do banco de dados, uma transação pode demorar da ordem de segundos para ser replicada. Exportar o snapshot de um banco de dados para o sistema de arquivos abaixo pode demorar 40 minutos. Um backup completo do sistema de arquivos subjacente pode demorar horas.

Nesse cenário, você poderá perder até 40 minutos dos dados mais recentes ao fazer a restauração do último snapshot. Uma restauração do backup do sistema de arquivos pode implicar horas de transações faltando. Além disso, a restauração provavelmente demorará tanto quanto fazer o backup, portanto carregar os dados pode demorar horas. Obviamente, você vai querer ter os dados mais recentes de volta o mais rápido possível, mas, dependendo do tipo de falha, essa cópia imediatamente disponível, contendo os dados mais recentes, pode não ser uma opção.

Retenção

A retenção – quanto tempo você guarda cópias de seus dados por aí – é outro fator a ser considerado em seus planos de recuperação de dados.

Embora seja provável que você ou seus clientes rapidamente percebam o esvaziamento súbito de todo um banco de dados, pode demorar dias para que uma perda de dados mais gradual atraia a atenção da pessoa correta. Restaurar os dados perdidos no último cenário exige snapshots feitos muito tempo atrás. Ao retroceder todo esse tempo, é provável que você queira combinar os dados restaurados com o estado atual. Fazer isso complica significativamente o processo de restauração.

Como a SRE do Google enfrenta o desafio da integridade de dados

De modo semelhante à nossa suposição de que os sistemas subjacentes do Google são suscetíveis a falhas, supomos que qualquer um de nossos mecanismos de proteção também está sujeito às mesmas forças e pode falhar das mesmas maneiras, nos momentos mais inconvenientes possíveis. Manter uma garantia de integridade de dados em larga escala – um desafio que se torna mais complicado pela alta taxa de mudanças dos sistemas de software

envolvidos – exige uma série de práticas complementares, porém desacopladas, em que cada uma é escolhida para oferecer um alto grau de proteção por si só.

As 24 combinações de modos de falha de integridade de dados

Dadas as várias maneiras pelas quais os dados podem ser perdidos (conforme descrito anteriormente), não há uma solução mágica para se proteger contra as muitas combinações dos modos de falha. Em vez disso, você precisa de uma defesa em profundidade. A defesa em profundidade é constituída de várias camadas, com cada camada sucessiva de defesa conferindo proteção contra cenários cada vez menos comuns de perda de dados. A Figura 26.2 mostra a jornada de um objeto, da remoção soft até a destruição, e as estratégias de recuperação de dados que devem ser empregadas ao longo dessa jornada para garantir uma defesa em profundidade.

A primeira camada é a *remoção soft* (ou “remoção preguiçosa” [lazy deletion], no caso de APIs oferecidas a desenvolvedores), que tem se mostrado ser uma defesa eficaz em cenários em que os dados são apagados de forma inadvertida. A segunda linha de defesa são os *backups e seus métodos relacionados de recuperação*. A terceira e última camada é a *validação de dados regular*, discutida na seção “Terceira camada: detecção precoce”. Em todas essas camadas, a presença de *replicação* é ocasionalmente útil para a recuperação de dados em cenários específicos (embora os planos de recuperação de dados não devam contar com a replicação).

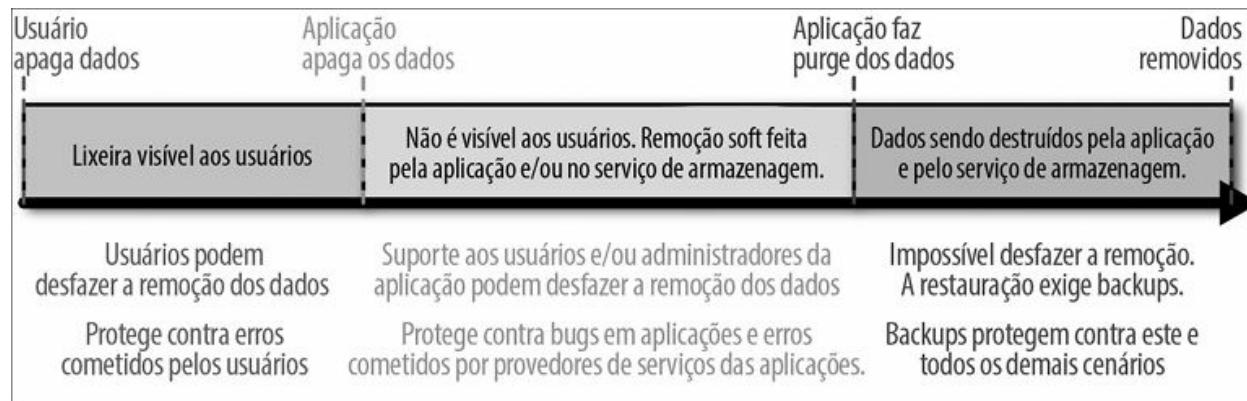


Figura 26.2 – A jornada de um objeto, da remoção soft até sua destruição.

Primeira camada: remoção soft

Quando a velocidade é alta e a privacidade é importante, bugs em aplicações são responsáveis pela grande maioria dos eventos de perda de dados e de dados corrompidos. De fato, bugs de remoção de dados podem se tornar tão comuns que a capacidade de desfazer a remoção por um tempo limitado passa a ser a principal linha de defesa contra a maioria das perdas de dados ocorridas inadvertidamente que, de outro modo, seriam permanentes.

Qualquer produto que garanta a privacidade de seus usuários deve permitir que eles apaguem subconjuntos selecionados e/ou todos os seus dados. Produtos assim podem representar um fardo para o suporte por causa de uma remoção de dados acidental. Dar aos usuários a capacidade de desfazer a remoção de seus dados (por exemplo, por meio de uma pasta de lixeira) reduz, mas não elimina completamente, esse fardo do suporte, em particular se seu serviço também aceita add-ons de terceiros que também possam apagar dados.

A remoção soft pode reduzir drasticamente ou até mesmo eliminar por completo esse fardo ao suporte. Essa remoção implica que dados apagados são imediatamente marcados como tal, ficando inutilizáveis por todos, exceto pelos caminhos de código administrativos da aplicação. Caminhos de código administrativos podem incluir descoberta legal, recuperação de contas sequestradas, administração corporativa, suporte a usuários, resolução de problemas e suas funcionalidades relacionadas. Faça uma remoção soft quando um usuário esvaziar sua lixeira e ofereça uma ferramenta de suporte ao usuário que permita aos administradores autorizados desfazer a remoção de qualquer item apagado acidentalmente pelos usuários. O Google implementa essa estratégia nas aplicações mais populares em produção; caso contrário, a carga de engenharia para o suporte a usuários seria impraticável.

Você pode estender mais ainda a estratégia de remoção soft oferecendo aos usuários a opção de recuperar dados removidos. Por exemplo, a pasta de lixeira do Gmail permite que os usuários acessem mensagens apagadas há menos de 30 dias.

Outra causa comum de remoção de dados indesejada ocorre como resultado de sequestro de conta. Em cenários de sequestro de contas, um sequestrador

geralmente apaga os dados do usuário original antes de usar a conta para enviar spams e para outros propósitos ilegais. Ao combinar o caso comum de remoção acidental de dados pelo usuário com o risco de remoção de dados por sequestradores, o argumento para ter uma remoção soft por meio de programa e uma interface para desfazer a remoção de dados em sua aplicação ou abaixo dela torna-se claro.

A remoção soft implica que, uma vez que os dados tenham sido marcados dessa forma, eles serão destruídos após um tempo razoável. Esse período de tempo depende das políticas de uma empresa e das leis aplicáveis, dos recursos de armazenagem disponíveis e seu custo e do preço do produto e seu posicionamento no mercado, em especial nos casos que envolvam dados de duração muito curta. Opções comuns de períodos de tempo para manter os dados após uma remoção soft são 15, 30, 45 ou 60 dias. De acordo com a experiência do Google, a maior parte dos problemas com sequestro de contas e integridade de dados é informada ou detectada em 60 dias. Desse modo, o argumento para remoção soft de dados para mais de 60 dias pode não ser convincente.

O Google também percebeu que os casos mais graves e devastadores de remoção de dados são causados pelos desenvolvedores da aplicação que não têm familiaridade com o código existente, mas trabalham com um código relacionado à remoção de dados, especialmente com pipelines de processamento em batch (por exemplo, um MapReduce offline ou um pipeline de Hadoop). É vantajoso fazer o design de suas interfaces de modo a criar obstáculos aos desenvolvedores sem familiaridade com o seu código, para dificultar sua aproximação das funcionalidades de remoção soft com códigos novos. Uma maneira eficiente de conseguir isso é oferecer soluções para computação em nuvem que tenham remoção soft embutida e APIs para desfazer a remoção de dados, garantindo que *esse recurso esteja habilitado*.⁷ Mesmo a melhor das armaduras será inútil se você não usá-la.

As estratégias de remoção soft incluem funcionalidades de remoção de dados em produtos para clientes como Gmail e Google Drive, mas e se, em vez disso, você dá suporte à computação em nuvem? Supondo que sua solução de computação em nuvem já suporte uma remoção soft via programação e a

funcionalidade para desfazer uma remoção de dados com defaults razoáveis, os cenários restantes de remoção acidental de dados terão origem em erros cometidos pelos seus próprios desenvolvedores internos ou pelos seus clientes desenvolvedores.

Em casos assim, pode ser conveniente introduzir uma camada adicional de remoção soft, que chamaremos de “remoção preguiçosa” (lazy deletion). Você pode pensar na remoção preguiçosa como um purge interno, controlado pelo sistema de armazenagem (enquanto a remoção soft é controlada pela aplicação cliente ou pelo serviço e é explícita a eles). Em um cenário de remoção preguiçosa, os dados apagados por uma aplicação em nuvem se tornam imediatamente inacessíveis à aplicação, mas são preservados pelo provedor de serviço de nuvem por até algumas semanas até serem destruídos. A remoção preguiçosa não é aconselhável em todas as estratégias de defesa em profundidade: um período longo de remoção preguiçosa é custoso em sistemas com muitos dados de curta duração e é impraticável em sistemas que devam garantir a destruição de dados apagados em um período de tempo razoável (isto é, aqueles que oferecem garantias de privacidade).

Sintetizando a primeira camada de defesa em profundidade:

- Uma pasta de lixeira que permita aos usuários desfazer a remoção de dados é a principal defesa contra erros de usuário.
- A remoção soft é a principal defesa contra erros de desenvolvedores e a segunda defesa contra erros de usuários.
- Nas soluções oferecidas aos desenvolvedores, a remoção preguiçosa é a principal defesa contra erros de desenvolvedores internos e a segunda defesa contra erros de desenvolvedores externos.

E o que dizer do *histórico de revisões*? Alguns produtos oferecem a capacidade de reverter itens a estados anteriores. Quando uma funcionalidade como essa estiver disponível aos usuários, ela será uma forma de lixeira. Quando disponível aos desenvolvedores, ela poderá ou não ser um substituto para a remoção soft, dependendo de sua implementação.

No Google, o histórico de revisões tem se mostrado útil na recuperação de certos cenários de dados corrompidos, mas não na recuperação da maioria

dos cenários de perda de dados que envolvam remoção acidental pelo programa ou de outros tipos. Isso ocorre porque algumas implementações de histórico de revisões tratam a remoção de dados como um caso especial, em que estados anteriores devem ser removidos, em oposição a modificar um item cujo histórico possa ser mantido por um determinado período de tempo. Para oferecer uma proteção adequada contra uma remoção de dados indesejada, aplique os princípios da remoção preguiçosa e/ou soft também ao histórico de revisões.

Segunda camada: backups e seus métodos relacionados de recuperação

Backups e recuperação de dados são a segunda linha de defesa após a remoção soft. O princípio mais importante nessa camada é que os backups não importam; o que importa é a recuperação. Os fatores que dão suporte a uma recuperação bem-sucedida devem orientar suas decisões de backup, e não o inverso.

Em outras palavras, os cenários em que você quer que seus backups ajudem a recuperar dados devem determinar o seguinte:

- Quais métodos de backup e de recuperação devem ser usados.
- Com que frequência você define pontos de restauração fazendo backups completos ou incrementais de seus dados.
- O local em que você armazena seus backups.
- Por quanto tempo você retém os backups.

Qual é o volume de dados recentes que você pode se dar ao luxo de perder durante um esforço de recuperação? Quanto menos dados você puder perder, mais seriamente deverá considerar 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 frequentes e completos são caros de outras maneiras. Notadamente, eles impõem uma carga computacional nos repositórios de dados ativos de seu serviço enquanto esses servem os usuários, levando seu serviço a se aproximar de seus limites

de escalabilidade e de desempenho. Para aliviar essa carga, você pode fazer backups completos fora dos horários de pico e, então, fazer 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 socorridos, mais locais devem ser seus backups. Com frequência, o Google retém snapshots⁸ custosos, porém rápidos de restaurar, por períodos de tempo muito curtos dentro da instância de armazenagem, e armazena backups menos recentes em repositórios distribuídos de acesso aleatório no mesmo datacenter (ou em um datacenter próximo) por um período de tempo um pouco maior. Uma estratégia como essa, por si só, não protegeria contra falhas no nível local, portanto esses backups, com frequência, são transferidos para localidades próximas ou offline por um período de tempo mais longo antes de expirarem em favor de novos backups.

Quanto tempo atrás seus backups devem alcançar? Sua estratégia de backup será mais custosa quanto mais para trás no tempo você quiser alcançar, ao mesmo tempo que os cenários dos quais você pode esperar se recuperar aumentam (embora esse aumento esteja sujeito a retornos cada vez menores).

Segundo a experiência do Google, bugs que provoquem modificação ou remoção de dados mais discretos no código de uma aplicação exigem um período de alcance mais distante no tempo, pois alguns desses bugs foram percebidos meses após a primeira perda de dados ter iniciado. Casos como esse sugerem que você vai querer ter a capacidade de alcançar um período de tempo o mais distante possível.

Por outro lado, em um ambiente de desenvolvimento de alta velocidade, mudanças no código e em esquemas podem resultar em backups antigos custosos ou impossíveis de usar. Além do mais, é desafiador recuperar diferentes subconjuntos de dados para diferentes pontos de restauração, pois fazer isso envolveria vários backups. Contudo, esse é exatamente o tipo de esforço de recuperação exigido por cenários de dados mais discretos sendo corrompidos ou removidos.

As estratégias descritas na seção “Terceira camada: detecção precoce”, têm o propósito de agilizar a detecção de modificação ou remoção de dados mais discretos no código da aplicação, evitando, pelo menos parcialmente, a

necessidade desse tipo de esforço complexo de recuperação. Ainda assim, como você confere uma proteção razoável antes de saber quais tipos de problemas devem ser detectados? O Google optou por traçar a linha entre 30 e 90 dias de backups para muitos serviços. O ponto em que um serviço se enquadra nesse intervalo depende de sua tolerância à perda de dados e de seus investimentos relativos na detecção precoce.

Para resumir nosso conselho para se proteger contra as 24 combinações de modos de falha de integridade de dados: abordar uma grande variedade de cenários com um custo razoável exige uma estratégia de backup em camadas. A primeira camada é constituída de muitos backups frequentes e rapidamente restaurados, armazenados próximo aos repositórios de dados ativos, talvez usando as mesmas tecnologias ou tecnologias semelhantes de armazenagem utilizadas pelos dados originais. Fazer isso confere proteção contra a maioria dos cenários que envolvam bugs de software e erros de desenvolvedor. Devido ao custo alto relativo, os backups são retidos nessa camada por qualquer período, que varia de horas a poucos dias, e podem levar minutos para ser restaurados.

A segunda camada é constituída de menos backups, mantidos de alguns dias a pouco menos de um mês, em sistemas de arquivos distribuídos de acesso aleatório, locais ao site. Esses backups podem exigir horas para restauração e conferem uma proteção adicional para contratemplos que afetem tecnologias particulares de armazenagem em sua pilha de serviços, mas não as tecnologias usadas para conter os backups. Essa camada também oferece proteção contra bugs em sua aplicação que sejam detectados tarde demais para contar com a primeira camada de sua estratégia de backup. Se você introduz novas versões de seu código em produção duas vezes por semana, talvez faça sentido reter esses backups por, no mínimo, uma ou duas semanas antes de apagá-los.

Camadas subsequentes tiram proveito de repositórios próximos, como bibliotecas de fitas dedicadas e armazenamento externo da mídia de backup (por exemplo, fitas ou drives de disco). Os backups nessas camadas conferem proteção contra problemas no nível local, como uma interrupção no fornecimento de energia elétrica no datacenter ou um sistema de arquivos

distribuído corrompido por causa de um bug.

É custoso passar grandes quantidades de dados entre as camadas. Por outro lado, a capacidade de armazenagem nas camadas subsequentes não compete com o crescimento das instâncias de armazenagem ativas em produção para o seu serviço. Como resultado, os backups nessas camadas tendem a ser feitos com menos frequência, mas são retidos por mais tempo.

Camada abrangente: replicação

Em um mundo ideal, todas as instâncias de armazenagem, incluindo as instâncias que contêm seus backups, seriam replicadas. Durante um esforço de recuperação de dados, a última coisa que você quer é descobrir que seus próprios backups perderam os dados necessários ou que o datacenter contendo o backup mais útil está em manutenção.

À medida que o volume de dados aumenta, a replicação de toda instância de armazenagem nem sempre é viável. Em casos assim, faz sentido distribuir backups sucessivos em locais diferentes, em que cada um possa falhar de modo independente, e gravar seus backups usando um método de redundância como RAID, código de correção Reed-Solomon ou replicação em estilo GFS.⁹

Ao escolher um sistema de redundância, não conte com um esquema usado raramente, cujos únicos “testes” de eficácia sejam suas próprias tentativas não frequentes de recuperação de dados. Em vez disso, escolha um esquema popular que seja de uso comum e contínuo por muitos de seus usuários.

1T versus 1E: não é “apenas” um backup maior

Processos e práticas aplicados a volumes de dados medidos em T (terabytes) não escalam bem para 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 de seu modelo de transações, esse exercício não apresenta nenhum desafio especial. Geralmente, você só precisará adquirir os recursos de máquina para iterar pelos seus dados, realizar alguma lógica de validação e prover repositórios suficientes para guardar algumas cópias de

seus dados.

Vamos agora elevar a aposta: em vez de alguns gigabytes, vamos tentar proteger e validar 700 petabytes de dados estruturados. Supondo um desempenho ideal do SATA 2.0 de 300 MB/s, uma única tarefa que itere por todos os seus dados e realize mesmo a mais básica das verificações de validação demorará oito décadas. Fazer alguns backups completos, supondo que você tenha a mídia, demorará no mínimo o mesmo tempo. O período de restauração, com algum pós-processamento, será mais demorado ainda. Estamos agora diante de quase um século inteiro para restaurar um backup que teria quase 80 anos quando você iniciasse a restauração. Obviamente, uma estratégia como essa precisa ser reavaliada.

A técnica mais comum e amplamente eficiente, utilizada para fazer backup de quantidades gigantescas de dados, é determinar “pontos de confiança” (trust points) em seus dados – porções de seus dados armazenados que são verificadas após terem sido consideradas imutáveis, geralmente pela passagem do tempo. Depois que soubermos que um dado perfil de usuário ou transação é fixo e não estará sujeito a mais mudanças, podemos conferir seu estado interno e fazer cópias adequadas com vistas à recuperação. Você pode, então, fazer backups incrementais que incluem apenas dados modificados ou acrescentados desde o seu último backup. Essa técnica deixa o seu tempo de backup alinhado ao seu tempo de processamento “convencional”, o que significa que backups incrementais frequentes poderão evitar o trabalho monolítico de 80 anos de verificação e cópia.

No entanto, lembre-se de que nos importamos com as *restaurações*, e não com os backups. Vamos supor que fizemos um backup completo há três anos e viemos fazendo backups incrementais diárias desde então. Uma restauração completa de nossos dados processaria serialmente uma cadeia de mais de mil backups altamente interdependentes. Cada backup independente representa um risco adicional de falha, sem mencionar a carga logística de escalonamento e de custo de tempo de execução desses jobs.

Outra maneira pela qual podemos reduzir o tempo de nossos jobs de cópia e verificação é distribuir a carga. Se fragmentarmos nossos dados de modo conveniente, será possível executar N tarefas em paralelo, em que cada tarefa

será responsável por copiar e verificar $1/N$ de nossos dados. Fazer isso exige um pouco de organização prévia e planejamento do design do esquema e da implantação física de nossos dados para:

- Fazer o balanceamento dos dados corretamente
- Garantir a independência de cada fragmento (shard)
- Evitar contenção entre as tarefas-irmãs concorrentes

Entre distribuir a carga horizontalmente e restringir a tarefa a fatias verticais de dados demarcadas por tempo, podemos reduzir essas oito décadas de tempo em várias ordens de magnitude, deixando nossas restaurações relevantes.

Terceira camada: detecção precoce

Dados “ruins” não ficam ociosamente parados: eles se propagam. Referências a dados ausentes ou corrompidos são copiadas, links se multiplicam e, a cada atualização, a qualidade geral de seu repositório de dados diminui. Transações dependentes feitas subsequentemente e mudanças no formato de dados em potencial tornam a restauração de um dado backup mais difícil à medida que o tempo passa. Quanto antes você ficar sabendo de uma perda de dados, mais fácil e mais completa poderá ser a sua recuperação.

Desafios enfrentados pelos desenvolvedores de nuvem

Em ambientes de alta velocidade, a aplicação em nuvem e os serviços de infraestrutura enfrentam muitos desafios de integridade de dados em tempo de execução, como:

- Integridade referencial entre repositórios de dados
- Mudanças de esquema
- Códigos que envelhecem
- Migrações de dados com downtime zero
- Pontos de integração que evoluem com outros serviços

Sem um esforço consciente de engenharia para monitorar os relacionamentos que surgem em seus dados, a qualidade dos dados de um serviço bem-

sucedido e crescente se degradará com o tempo.

Com frequência, um desenvolvedor de nuvem iniciante que escolha uma API de armazenagem consistente e distribuída (como o Megastore) delega a integridade dos dados da aplicação ao algoritmo distribuído e consistente implementado abaixo da API (como o Paxos; veja o Capítulo 23). O desenvolvedor pensa que a API selecionada, por si só, manterá os dados da aplicação em bom estado. Como resultado, esses desenvolvedores reúnem todos os dados da aplicação em uma única solução de armazenagem que garanta uma consistência distribuída, evitando problemas de integridade referencial em troca de desempenho e/ou escala reduzidos.

Embora algoritmos como esses sejam infalíveis teoricamente, suas implementações muitas vezes estão repletas de hacks, otimizações, bugs e palpites. Por exemplo: em teoria, o Paxos ignora nós computacionais com falha e é capaz de fazer progressos, desde que um quórum de nós em funcionamento seja preservado. Na prática, porém, ignorar um nó em falha pode corresponder a timeouts, retentativas e outras abordagens para tratamento de falhas abaixo da implementação do Paxos em particular [Cha07]. Por quanto tempo o Paxos deve tentar entrar em contato com um nó que não esteja respondendo antes de considerar que houve timeout? Quando uma máquina em particular falha (talvez, de modo intermitente) de determinada maneira, em certos instantes no tempo e em um datacenter em particular, comportamentos imprevisíveis poderão resultar. Quanto maior a escala de uma aplicação, mais frequentemente ela será afetada por essas inconsistências, sem tomar conhecimento disso. Se essa lógica for verdadeira mesmo quando aplicada às implementações de Paxos (como tem sido para o Google), então ela deverá ser mais verdadeira para implementações com consistência eventual como o Bigtable (para o qual a lógica também se mostrou ser verdadeira). As aplicações afetadas não têm nenhuma maneira de saber que 100% de seus dados estão bons até verificarem: confie nos sistemas de armazenagem, mas verifique!

Para complicar esse problema, para se recuperar de cenários de dados mais discretos corrompidos ou removidos, devemos recuperar subconjuntos diferentes de dados em pontos de restauração diferentes usando backups

distintos, enquanto mudanças de código e de esquema podem deixar backups antigos ineficientes em ambientes de alta velocidade.

Validação de dados out-of-band

Para evitar que a qualidade dos dados se degrade diante dos olhos dos usuários e detectar cenários de dados mais discretos corrompidos ou perdidos antes que se tornem irrecuperáveis, um sistema de verificações out-of-band (fora da banda) e balanceamentos é necessário dentro dos repositórios de dados de uma aplicação e entre eles.

Com muita frequência, esses pipelines de validação de dados são implementados como coleções de jobs MapReduce ou Hadoop. Geralmente, pipelines como esses são acrescentados como uma ideia *a posteriori* para serviços que já são populares e bem-sucedidos. Às vezes, tentamos usar esses pipelines pela primeira vez quando os serviços alcançam os limites de escalabilidade e são reconstruídos do zero. O Google desenvolveu validadores em resposta a cada uma dessas situações.

Desviar alguns desenvolvedores para trabalhar em um pipeline de validação de dados pode reduzir a velocidade de engenharia no curto prazo. Porém, dedicar recursos de engenharia para validação de dados dará coragem a outros desenvolvedores para serem mais rápidos no longo prazo, pois os engenheiros sabem que bugs que corrompam dados serão menos prováveis de se insinuar em produção sem que sejam percebidos. De modo semelhante aos efeitos desfrutados quando testes de unidade são introduzidos precocemente no ciclo de vida dos projetos, um pipeline de validação de dados resulta em uma aceleração geral dos projetos de desenvolvimento de software.

Para citar um exemplo específico: o Gmail tem uma série de validadores de dados, em que cada um já detectou verdadeiros problemas de integridade de dados em produção. Os desenvolvedores do Gmail se sentem confortados em saber que bugs que introduzem inconsistências nos dados de produção são detectados em 24 horas, e estremecem só de pensar que seus validadores sejam executados com menos frequência que diariamente. Esses validadores, juntamente com uma cultura de testes de unidade e de regressão e outras boas práticas, têm dado coragem aos desenvolvedores do Gmail para introduzir

modificações de código na implementação da armazenagem em produção do Gmail com uma frequência maior que uma vez por semana.

A validação de dados out-of-band é intrincada para ser implementada corretamente. Se for rigorosa demais, mesmo alterações simples e apropriadas farão a validação falhar. Como resultado, os engenheiros abandonarão totalmente a validação de dados. Se a validação de dados não for rigorosa o suficiente, dados corrompidos que afetem a experiência do usuário poderão passar despercebidos. Para encontrar o equilíbrio correto, valide apenas as invariantes que causem devastação aos usuários.

Por exemplo, o Google Drive periodicamente valida se os conteúdos dos arquivos estão alinhados às listagens das pastas do Drive. Se esses dois elementos não estiverem alinhados, alguns arquivos terão dados faltando – um resultado desastroso. Os desenvolvedores da infraestrutura do Drive estavam tão comprometidos com a integridade dos dados que também aperfeiçoaram seus validadores para que corrigissem essas inconsistências automaticamente. Essa proteção transformou uma potencial situação de emergência de perda de dados em um negócio do tipo “apresentem-se todos; ai, meu Deus, todos os arquivos estão desaparecendo!” em 2013, em uma situação comum do tipo “vamos para casa e corrigir a causa-raiz na segunda-feira”. Ao transformar emergências nos negócios em algo comum, os validadores elevam o moral da engenharia, melhoram sua qualidade de vida e a previsibilidade.

Validadores out-of-band podem ser custosos em escala. Uma parte significativa dos recursos de processamento do Gmail dá suporte a uma coleção de validadores diários. Aumentando mais ainda esse custo, esses validadores também reduzem as taxas de acertos de cache (cache hits) do lado do servidor, reduzindo a capacidade de resposta percebida pelos usuários. Para atenuar esse impacto na capacidade de ser responsável, o Gmail oferece uma variedade de controles para limitar a taxa de seus validadores e periodicamente refatorá-los a fim de reduzir a contenção de disco. Em um desses esforços de refatoração, reduzimos a contenção para eixos de disco em 60% sem reduzir significativamente o escopo das invariantes incluídas. Embora a maioria dos validadores do Gmail execute diariamente, a carga de

trabalho do maior validador é dividida entre 10 a 14 fragmentos (shards), com um fragmento validado por dia por questões de escala.

O Google Compute Storage é outro exemplo dos desafios impostos pela escala na validação de dados. Quando seus validadores out-of-band não podiam mais terminar em um dia, os engenheiros do Compute Storage tiveram que conceber uma maneira mais eficiente de verificar seus metadados, em vez de usar apenas a força bruta. De modo semelhante à sua aplicação na recuperação de dados, uma estratégia em camadas também pode ser útil na validação de dados out-of-band. À medida que um serviço escalar, sacrifique o rigor em validadores diários. Garanta que os validadores diários continuem capturando os cenários mais desastrosos em 24 horas, mas continue com uma validação mais rigorosa com frequência reduzida para conter custos e latência.

Resolver problemas de validações que falharam pode exigir esforços significativos. As causas de uma validação com falha intermitente podem desaparecer em minutos, horas ou dias. Desse modo, a capacidade de explorar os logs de auditoria de validação rapidamente é essencial. Os serviços mais maduros do Google oferecem aos engenheiros de plantão uma documentação abrangente e ferramentas para resolução de problemas. Por exemplo, os engenheiros de plantão para o Gmail têm à disposição:

- Um conjunto de entradas no manual que descrevem como responder a um alerta de falha de validação
- Uma ferramenta de investigação do tipo BigQuery
- Um painel de controle para validação de dados

Uma validação de dados out-of-band eficaz exige todos os itens a seguir:

- Gerenciamento dos jobs de validação
- Monitoração, alertas e painéis de controle
- Funcionalidades para limitação de taxa
- Ferramentas para resolução de problemas
- Manuais de produção
- APIs de validação de dados que facilitem adicionar e refatorar validadores

A maioria das equipes pequenas de engenharia que opera em alta velocidade não é capaz de fazer o design, construir e manter todos esses sistemas. Se forem pressionadas a fazer isso, o resultado, com frequência, será soluções específicas, frágeis, limitadas, que representam um desperdício e se desintegram rapidamente. Portanto, estruture suas equipes de engenharia de modo que uma equipe de infraestrutura central ofereça um framework de validação de dados para várias equipes de engenharia de produto. A equipe da infraestrutura central mantém o framework de validação de dados out-of-band, enquanto as equipes de engenharia de produto mantêm a lógica de negócios personalizada no coração do validador para manter o compasso com seus produtos em evolução.

Saber se a recuperação de dados funcionará

Quando uma lâmpada deixa de funcionar? Quando o acionamento do interruptor deixa de acender a luz? Nem sempre – com frequência, a lâmpada já deixou de funcionar e você só percebeu a falha quando não houve uma resposta ao acionamento do interruptor. A essa altura, a sala estará escura e você já terá tropeçado.

De modo semelhante, as dependências de sua recuperação (que significam, em sua maior parte, mas não apenas, o seu backup) podem estar em um estado latente de falha, do qual você não estará ciente até tentar recuperar os dados.

Se descobrir que seu processo de restauração tem falhas antes de precisar contar com ele, você poderá tratar a vulnerabilidade antes de se tornar uma vítima dela: você poderá fazer outro backup, fornecer recursos adicionais e mudar seu SLO. Contudo, para realizar essas ações de forma proativa, você deve saber previamente que elas são necessárias. Para detectar essas vulnerabilidades:

- Teste continuamente o processo de recuperação como parte de suas operações normais.
- Configure alertas que disparem quando um processo de recuperação falhar em fornecer uma indicação imediata de seu sucesso.

O que pode dar errado com seu processo de recuperação? Qualquer coisa e

tudo – motivo pelo qual o único teste que poderá deixar você dormir à noite é um teste completo fim a fim. Deixe que o próprio processo comprove isso. Mesmo que você tenha executado recentemente uma recuperação com sucesso, partes de seu processo de recuperação ainda podem conter falhas. Se você aprender apenas uma lição com este capítulo, lembre-se de que *você só saberá que pode recuperar seu estado recente se realmente fizer isso*.

Se os testes de recuperação forem um evento manual e em fases, os testes se tornarão uma espécie de tarefa enfadonha, que não será realizada de forma suficientemente profunda nem frequente para merecer sua confiança. Portanto, automatize esses testes sempre que for possível e então execute-os continuamente.

São vários os aspectos de seu plano de recuperação que você deve confirmar:

- Seus backups são válidos e completos ou eles estão vazios?
- Você tem recursos de máquina suficientes para executar todas as tarefas de configuração, restauração e pós-processamento que compõem a sua recuperação?
- O processo de recuperação termina em um tempo razoável?
- Você é capaz de monitorar o estado de seu processo de recuperação à medida que ele progride?
- Você está livre de dependências críticas de recursos que estejam fora de seu controle, como acesso a um conjunto de repositórios de mídia externo, que não esteja disponível 24/7?

Nossos testes identificaram as falhas mencionadas anteriormente, assim como falhas em muitos outros componentes de uma recuperação de dados bem-sucedida. Se não tivéssemos descoberto essas falhas em testes regulares – isto é, se tivéssemos nos deparado com elas somente quando precisássemos recuperar dados de usuários em emergências reais –, é bem possível que alguns dos produtos mais bem-sucedidos do Google atualmente não tivessem passado pelo teste do tempo.

Falhas são inevitáveis. Se você espera descobri-las quando estiver com a arma apontada em sua direção, encarando uma perda de dados real, estará brincando com fogo. Se os testes forçarem a ocorrência de falhas antes que

verdadeiras catástrofes ocorram, você poderá corrigir os problemas antes que qualquer dano se concretize.

Estudos de caso

A vida imita a arte (ou, nesse caso, a ciência) e, conforme previmos, a vida real nos tem apresentado oportunidades infelizes e inevitáveis para colocar nossos sistemas e processos de recuperação de dados em teste, sob pressão do mundo real. Duas dessas oportunidades mais dignas de nota e interessantes serão discutidas aqui.

Gmail – fevereiro de 2011: restauração a partir do GTape

O primeiro estudo de caso de recuperação que analisaremos foi único sob dois aspectos: o número de falhas que coincidiram para levar à perda de dados e o fato de ter sido 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 se juntar a uma chamada de conferência. O evento que há muito temíamos – na verdade, o motivo para a existência do sistema de backup – havia começado: o Gmail havia perdido uma quantidade significativa de dados de usuário. Apesar das muitas proteções do sistema e de verificações internas e redundâncias, os dados desapareceram do Gmail.

Esse foi o primeiro uso em larga escala do GTape: um sistema de backup global do Gmail para restaurar dados ativos dos clientes. Felizmente, não era a primeira restauração desse tipo, pois situações semelhantes haviam sido anteriormente simuladas várias vezes. Desse modo, fomos capazes de:

- Fornecer uma estimativa de quanto tempo demoraria para restaurar a maior parte das contas de usuário afetadas.
- Restaurar todas as contas em algumas horas após a nossa estimativa inicial.
- Recuperar mais de 99% dos dados antes do período de tempo estimado ter

se esgotado.

A habilidade para formular uma estimativa como essa foi sorte? Não – nosso sucesso foi fruto de planejamento, aderência às melhores práticas, trabalho árduo e cooperação, e ficamos felizes em ver que nosso investimento em cada um desses elementos teve resultados tão bons quanto os que tivemos. O Google foi capaz de restaurar os dados perdidos prontamente, executando um plano projetado de acordo com as melhores práticas de *Defesa em Profundidade e Preparo 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 um misto de surpresa e divertimento. Fita? O Google não tem muitos discos e uma rede rápida para replicar dados importantes assim? É claro que o Google tem esses recursos, mas o princípio da Defesa em Profundidade determina que devemos oferecer várias camadas de proteção contra falhas ou comprometimento de qualquer sistema de proteção. Fazer backup de sistemas online como o Gmail oferece defesa em profundidade em duas camadas:

- Uma falha de redundância interna do Gmail e de subsistemas de backup.
- Uma falha ampla ou uma vulnerabilidade dia zero (zero-day) em um device driver ou em um sistema de arquivos que afetem a mídia subjacente de armazenagem (disco).

Essa falha em particular resultou do primeiro cenário – embora o Gmail tivesse meios internos para recuperar dados perdidos, essa perda foi além do ponto de recuperação por esses meios internos.

Um dos aspectos mais celebrados internamente da recuperação de dados do Gmail foi o grau de cooperação e a coordenação tranquila que fizeram parte da recuperação. Muitas equipes, algumas totalmente não relacionadas ao Gmail ou à recuperação de dados, se ofereceram para ajudar. A recuperação não teria sido bem-sucedida de forma tão tranquila sem um plano central para coordenar um esforço hercúleo tão amplamente distribuído; esse plano era o produto de ensaios e dry runs regulares. A devoção do Google em estar preparado para emergências nos levou a ver essas falhas como inevitáveis. Ao aceitar essa inevitabilidade, não esperamos nem apostamos em evitar

esses desastres, mas prevemos que eles ocorrerão. Desse modo, precisamos de um plano para lidar não só com as falhas previsíveis, mas para uma quantidade de falhas aleatórias indiferenciadas também.

Em suma, sempre *soubemos* que a aderência às melhores práticas é importante, e foi bom ver que a máxima se comprovou ser verdadeira.

Google Music – março de 2012: detecção de remoção furtiva

A segunda falha que analisaremos implica desafios de logística que são únicos para a escala do repositório de dados recuperado: onde você armazenaria mais de 5.000 fitas e como você leria de modo eficiente (ou, até mesmo, viável) essa quantidade de dados de mídias offline em um período de tempo razoável?

Terça-feira, 6 de março de 2012, meio da tarde

Descobrindo o problema

Um usuário do Google Music informa que faixas de áudio anteriormente sem problemas estavam sendo puladas. A equipe responsável pela interface com os usuários do Google Music notifica os engenheiros desse serviço. O caso é investigado como um possível problema de streaming de mídia.

No dia 7 de março, o engenheiro investigando a situação descobre que há uma referência faltando nos metadados da faixa de áudio que não podia ser reproduzida e que deveria apontar para os dados de áudio propriamente ditos. Ele fica surpreso. A correção 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 resolução dos problemas na raiz, portanto o engenheiro faz uma exploração mais profunda.

Quando descobre a causa do lapso na integridade de dados, ele quase tem um ataque cardíaco: a referência de áudio havia sido removida por um pipeline de remoção de dados para proteção de privacidade. Essa parte do Google Music havia sido projetada para apagar um número muito grande de faixas de áudio em tempo recorde.

Avaliando o dano

A política de privacidade do Google protege os dados pessoais de um usuário. Conforme aplicada especificamente no Google Music, nossa política de privacidade implica que arquivos de música e metadados relevantes sejam removidos em um período de tempo razoável após os usuários os terem apagado. À medida que a popularidade do Google Music aumentava, a quantidade de dados cresceu rapidamente, portanto a implementação original de remoção de dados precisou ser reprojetada em 2012 para que fosse mais eficiente. Em 6 de fevereiro, o pipeline atualizado de remoção de dados fez sua execução inaugural para remover metadados relevantes. Nada parecia estar errado na ocasião, portanto um segundo estágio do pipeline foi permitido para remover também os dados de áudio associados.

Os piores pesadelos do engenheiro teriam se tornado realidade? Ele imediatamente fez soar o alarme, aumentando a prioridade do caso para a classificação mais urgente do Google e informando o problema à gerência de engenharia e à Site Reliability Engineering (Engenharia de Confiabilidade de Sites). Uma equipe pequena de desenvolvedores do Google Music e de SREs se reuniu para atacar o problema, e o pipeline problemático foi temporariamente desabilitado para conter a onda de casualidades entre os usuários externos.

Uma verificação manual dos metadados de milhões a bilhões de arquivos organizados em vários datacenters seria impensável. Assim, a equipe implementou um job MapReduce rápido para avaliar os danos e esperou desesperadamente que o job terminasse. Eles ficaram paralisados quando o resultado foi apresentado no dia 8 de março: o pipeline refatorado de remoção de dados havia apagado aproximadamente 600.000 referências de áudio que não deviam ter sido removidas, afetando arquivos de áudio de 21.000 usuários. Como o pipeline de diagnóstico criado às pressas havia feito algumas simplificações, a verdadeira extensão dos danos poderia ser pior.

Mais de um mês havia se passado desde que o pipeline problemático de remoção de dados havia executado pela primeira vez, e a própria execução inaugural havia removido centenas de milhares de faixas de áudio que não deviam ter sido removidas. Havia alguma esperança de ter os dados de volta? Se as faixas não fossem recuperadas, ou não fossem recuperadas de forma

rápida o suficiente, o Google teria que enfrentar a ira de seus usuários. Como não havíamos percebido esse glitch?

Resolvendo o problema

Identificação do bug e esforços de recuperação em paralelo. O primeiro passo para resolver o problema foi identificar o bug propriamente dito e determinar como e por que ele havia ocorrido. Enquanto a causa-raiz não fosse identificada e corrigida, qualquer esforço de recuperação seria em vão. Estaríamos sob pressão para reativar o pipeline a fim de respeitar as solicitações de usuários que haviam apagado faixas de áudio, mas fazer isso causaria prejuízos a usuários inocentes, que continuariam a perder músicas compradas em lojas ou, pior ainda, seus próprios arquivos de áudio gravados com tanto esforço. A única maneira de escapar do Catch-22¹⁰ era corrigir o problema em sua raiz, e fazê-lo rapidamente.

Apesar disso, não havia tempo a perder para organizar o esforço de recuperação. As faixas de áudio propriamente ditas tinham backup em fita, mas de modo diferente de nosso estudo de caso com o Gmail, as fitas de backup criptografadas do Google Music eram transportadas para locais de armazenagem externos, pois essa opção oferecia mais espaço para os backups volumosos dos dados de áudio dos usuários. Para restaurar rapidamente a experiência dos usuários afetados, a equipe decidiu resolver o problema da causa-raiz, ao mesmo tempo que buscava as fitas de backup que estavam em outro lugar (uma opção de restauração que exigia bastante tempo) em paralelo.

Os engenheiros se dividiram em dois grupos. Os SREs mais experientes trabalharam no esforço de recuperação, enquanto os desenvolvedores analisavam o código de remoção de dados e tentavam corrigir o bug de perda de dados em sua raiz. Por causa do conhecimento incompleto da raiz do problema, a recuperação teria que ser feita em vários passos divididos em etapas. O primeiro conjunto de quase meio milhão de faixas de áudio foi identificado, e a equipe responsável pela manutenção do sistema de backup em fita foi notificada do esforço de recuperação emergencial às 16h34min. Fuso do Pacífico, dia 8 de março.

A equipe de recuperação tinha um fator trabalhando em seu favor: esse esforço de recuperação estava ocorrendo apenas algumas semanas depois do exercício anual de teste de recuperação de desastres da empresa [Kri12]. A equipe de backup em fita já conhecia as capacidades e as limitações de seus subsistemas, que haviam sido o tema dos testes de DiRT, e havia começado a exercitar uma nova ferramenta testada durante um exercício de DiRT. Usando a nova ferramenta, a equipe de recuperação em conjunto deu início à árdua tarefa de mapear centenas de milhares de arquivos de áudio aos backups registrados no sistema de backup em fita, e então mapear os arquivos dos backups às fitas propriamente ditas.

Dessa maneira, a equipe determinou que o esforço inicial de recuperação envolveria trazer mais de 5.000 fitas de backup por caminhão. Depois disso, os técnicos do datacenter precisariam criar espaço para as fitas nas bibliotecas. Um processo longo e complexo de registrar as fitas e extrair seus dados viria a seguir, envolvendo soluções alternativas e medidas de atenuação em caso de fitas ou drives ruins e interações inesperadas com o sistema.

Infelizmente, apenas 436.223 das aproximadamente 600.000 faixas de áudio perdidas foram encontradas nas fitas de backup, o que significava que cerca de 161.000 outras faixas de áudio haviam sido removidas antes que um backup pudesse ser feito. A equipe de recuperação decidiu que descobriria como recuperar as 161.000 faixas em falta depois que tivesse iniciado o processo de recuperação para as faixas com backups em fita.

Enquanto isso, a equipe de causa-raiz havia perseguido e abandonado uma pista falsa: inicialmente, os desenvolvedores acharam que um serviço de armazenagem do qual o Google Music dependia havia fornecido dados com problemas, confundindo os pipelines de remoção de dados e fazendo com que eles, erroneamente, removessem os dados de áudio incorretos. Ao investigar melhor, essa teoria se mostrou ser falsa. A equipe de causa-raiz coçava suas cabeças e continuava a procurar o bug esquivo.

Primeira onda de recuperação. Depois que a equipe de recuperação havia identificado as fitas de backup, a primeira onda de recuperação teve início em 8 de março. Requisitar 1,5 petabyte de dados distribuídos em milhares de

fitas de um repositório externo era um problema, mas extrair os dados das fitas era outra questão completamente diferente. A pilha personalizada de software de backup em fita não havia sido projetada para tratar uma única operação de restauração tão grande, portanto a recuperação inicial foi dividida em 5.475 jobs de restauração. Isso exigiria um operador humano digitando um comando de restauração por minuto durante mais de três dias para solicitar essa quantidade enorme de restaurações e, sem dúvida, qualquer operador humano cometeria muitos erros. Solicitar a restauração do sistema de backup em fitas, por si só, exigiu que a SRE desenvolvesse uma solução por meio de programação.¹¹

Por volta da meia-noite de 9 de março, o SRE de Music havia terminado de requisitar todas as 5.475 restaurações. O sistema de backup em fita começou a fazer sua mágica. Quatro horas depois, ele apresentou uma lista de 5.337 fitas de backup a serem trazidas de outros locais. Em mais oito horas, as fitas chegaram em um datacenter em uma série de entregas feitas por caminhão.

Enquanto os caminhões estavam a caminho, técnicos do datacenter desativaram várias bibliotecas de fita para manutenção e removeram milhares de fitas a fim de criar espaço para a operação gigantesca de recuperação de dados. Em seguida, os técnicos começaram a carregar arduamente as fitas de forma manual, à medida que milhares delas chegavam nas primeiras horas da manhã. Nos exercícios anteriores de DiRT, esse processo manual se provou ser centenas de vezes mais rápido para restaurações gigantescas do que os métodos baseados em robôs oferecidos pelos fornecedores de bibliotecas de fitas. Em três horas, as bibliotecas estavam de volta, fazendo scanning das fitas e executando milhares de jobs de restauração em repositórios de processamento distribuído.

Apesar da experiência da equipe com DiRT, a recuperação gigantesca de 1,5 petabyte demorou mais 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 das 3.475 fitas de backup recuperadas, para repositórios de sistemas de arquivos distribuídos em um cluster de processamento próximo. As outras 1.862 fitas de backup haviam sido omitidas do processo de recuperação de fitas por um fornecedor. Além disso, o processo de

recuperação sofreu atrasos por causa de 17 fitas ruins. Prevendo uma falha devido a fitas ruins, uma codificação redundante havia sido usada para gravar os arquivos de backup. Entregas adicionais por caminhão foram acionadas para recuperar as fitas redundantes, juntamente com as outras 1.862 fitas que haviam sido omitidas na primeira recuperação externa.

Na manhã de 11 de março, mais de 99,95% da operação de restauração havia sido concluída, e a recuperação de fitas adicionais redundantes para os arquivos remanescentes estava em progresso. Embora os dados estivessem seguros em sistemas de arquivos distribuídos, passos adicionais na recuperação de dados foram necessários para deixá-los acessíveis aos usuários. A Equipe do Google Music começou a exercitar esses passos finais do processo de recuperação de dados em paralelo em uma pequena amostra de arquivos de áudio recuperados para garantir que o processo continuava funcionando conforme esperado.

Naquele momento, os pagers de produção do Google Music foram acionados devido a uma falha de produção não relacionada ao problema, mas que afetava os usuários de modo crítico – uma falha que envolveu totalmente a equipe do Google Music por dois dias. O esforço de recuperação de dados foi retomado no dia 13 de março, quando todas as 436.223 faixas de áudio se tornaram novamente acessíveis aos seus usuários. Em pouco menos de sete dias, 1,5 petabyte de dados de áudio foram restabelecidos aos usuários com a ajuda de backups em fita externos; os esforços propriamente ditos de recuperação de dados consumiram cinco dos sete dias.

Segunda onda de recuperação. Com a primeira onda do processo de recuperação para trás, a equipe mudou seu foco para os outros 161.000 arquivos de áudio em falta, que haviam sido apagados pelo bug antes que seu backup fosse feito. A maioria desses arquivos era constituída de faixas de áudio compradas em lojas e promocionais, e as cópias originais das lojas não haviam sido afetadas pelo bug. Essas faixas foram rapidamente restabelecidas, de modo que os usuários afetados pudessem desfrutar de suas músicas novamente.

No entanto, o upload de uma pequena parcela dos 161.000 arquivos de áudio havia sido feito pelos próprios usuários. A Equipe do Google Music pediu

aos seus servidores que solicitassem que os clientes do Google Music dos usuários afetados refizessem o upload de arquivos cujas datas fossem de 14 de março em diante.

Esse processo demorou mais de uma semana. Assim, concluímos todo o esforço de recuperação do incidente.

Tratando a causa-raiz

Mais tarde, a Equipe do Google Music identificou a falha em seu pipeline refatorado de remoção de dados. Para entender essa falha, você precisa antes conhecer o contexto de como os sistemas de processamento de dados offline evoluem em um ambiente de larga escala.

Para um serviço grande e complexo, constituído de vários subsistemas e serviços de armazenagem, mesmo uma tarefa tão simples quanto remover dados apagados deve ser realizada em etapas, cada uma envolvendo diferentes repositórios de dados.

Para que o processamento de dados termine rapidamente, ele se dá em paralelo, executando em dezenas de milhares de máquinas, exigindo uma carga alta em vários subsistemas. Essa distribuição pode deixar mais lento o serviço aos usuários ou fazer o serviço falhar diante da carga intensa.

Para evitar esses cenários indesejáveis, os engenheiros da computação em nuvem muitas vezes criam uma cópia de curta duração dos dados em um repositório secundário, no qual o processamento dos dados então é feito. A menos que a idade relativa das cópias secundárias dos dados seja cuidadosamente coordenada, essa prática introduz condições de concorrência (race conditions).

Por exemplo, dois estágios de um pipeline podem ser projetados para executar em estrita sucessão, com três horas de intervalo, de modo que o segundo estágio possa fazer uma suposição simplificada sobre a correção de seus dados de entrada. Sem essa suposição simplificadora, a lógica do segundo estágio pode ser difícil de ser executada em paralelo. Porém, os estágios podem demorar mais para serem concluídos à medida que o volume de dados aumenta. Em algum momento, as suposições do design original poderão não ser mais válidas para determinadas porções de dados necessárias

ao segundo estágio.

À primeira vista, essa condição de concorrência pode ocorrer para uma minúscula fração dos dados. Porém, conforme o volume de dados aumenta, uma fração cada vez maior dos dados corre o risco de disparar uma condição de concorrência. Um cenário como esse é probabilístico – o pipeline funciona corretamente para a grande maioria dos dados e na maior parte do tempo. Quando condições de concorrência como essa ocorrem em um pipeline de remoção de dados, dados indevidos poderão ser apagados de forma não determinística.

O pipeline de remoção de dados do Google Music foi projetado com coordenação e amplas margens de erro implantadas. Porém, quando os estágios de upstream do pipeline começaram a exigir cada vez mais tempo à medida que o serviço crescia, otimizações de desempenho foram implementadas para que o Google Music pudesse continuar a atender aos requisitos de privacidade. Como resultado, a probabilidade de uma condição de concorrência que apagava inadvertidamente os dados nesse pipeline começou a aumentar. Quando o pipeline foi refatorado, essa probabilidade aumentou novamente de modo significativo, até um ponto em que as condições de concorrência passaram a ocorrer com mais regularidade.

Na esteira do esforço de recuperação, o Google Music refez o projeto de seu pipeline de remoção de dados a fim de eliminar esse tipo de condição de concorrência. Além disso, aperfeiçoamos os sistemas de monitoração de produção e de alertas para detectar bugs semelhantes de remoção furtiva em larga escala, com o propósito de detectar e de corrigir problemas como esse antes que os usuários percebam.¹²

Princípios gerais de SRE conforme aplicados à integridade de dados

Princípios gerais de SRE podem ser aplicados às especificidades da integridade de dados e da computação em nuvem, conforme descrito nesta seção.

Mente de principiante

Serviços complexos, de larga escala, têm bugs inerentes que não podem ser totalmente dominados. Jamais pense que você entende suficientemente um sistema complexo a ponto de dizer que ele não falhará de determinada maneira. Confie, mas verifique e aplique uma defesa em profundidade. (Observação: a “mente de principiante” *não* sugere colocar um novo contratado responsável por aquele pipeline de remoção de dados!)

Confie, mas verifique

Qualquer API da qual você dependa não funcionará perfeitamente o tempo todo. É certo que, independentemente da qualidade de sua engenharia ou do rigor de seus testes, a API terá defeitos. Verifique se os elementos mais críticos de seus dados estão corretos utilizando validadores de dados out-of-band, mesmo que a semântica da API sugira que você não precise fazer isso. Algoritmos perfeitos podem não ter implementações perfeitas.

Esperança não é uma estratégia

Componentes de sistema que não sejam continuamente exercitados falham quando você mais precisa deles. Prove que sua recuperação de dados funciona com exercícios regulares; do contrário, sua recuperação de dados não funcionará. Seres humanos não têm a disciplina para exercitar continuamente os componentes do sistema, portanto a automação é sua amiga. Entretanto, se usar engenheiros que tenham prioridades concorrentes para esses esforços de automação, você poderá acabar com soluções temporárias para tapar buracos.

Defesa em profundidade

Mesmo um sistema que seja ao máximo à prova de balas é suscetível a bugs e a erros de operador. Para que os problemas de integridade de dados possam ser corrigidos, os serviços devem detectar esses problemas rapidamente. Toda estratégia, em algum momento, falha em ambientes em que há mudanças. As melhores estratégias de integridade de dados têm várias camadas – diversas estratégias que vão usando as próximas como alternativa e tratam uma ampla

variedade de cenários em conjunto, a um custo razoável.

Revise e reanalise

O fato de que seus dados “estavam seguros ontem” não ajudará você amanhã, ou nem mesmo hoje. Os sistemas e a infraestrutura mudam, e você deve provar que suas suposições e processos permanecem relevantes diante do progresso. Considere o cenário a seguir.

O serviço Shakespeare recebeu muita atenção positiva da mídia, e sua base de usuários está aumentando continuamente. Nenhuma atenção real foi dada à integridade de dados à medida que o serviço foi desenvolvido. É claro que não queremos servir bits *ruins*, mas se o índice do Bigtable for perdido, podemos facilmente recriá-lo a partir dos textos originais de Shakespeare e de um MapReduce. Fazer isso exigiria pouco tempo, portanto nunca fizemos backups dos índices.

Agora, uma nova funcionalidade permite que os usuários façam anotações textuais. Repentinamente, nosso conjunto de dados não pode mais ser facilmente recriado, ao mesmo tempo que os dados dos usuários são cada vez mais importantes para eles. Desse modo, precisamos rever nossas opções de replicação – não estamos replicando apenas por causa de latência ou largura de banda, mas também pela integridade dos dados. Portanto, precisamos criar e testar um procedimento de backup e de restauração. Esse procedimento também é testado periodicamente por um exercício de DiRT para garantir que podemos restaurar as anotações dos usuários a partir dos backups no tempo definido pelo SLO.

Conclusão

A disponibilidade dos dados deve ser uma preocupação primordial de qualquer sistema centrado em dados. Em vez de colocar o foco nos meios para um fim, a SRE do Google acha útil fazer um empréstimo do desenvolvimento orientado a testes, provando que nossos sistemas são capazes de manter a disponibilidade dos dados com um downtime máximo previsto. Os meios e os mecanismos que usamos para atingir esse objetivo

final são um mal necessário. Ao manter nossos olhos fixos na meta, evitamos cair na armadilha em que “a operação foi um sucesso, mas o sistema morreu”.

Reconhecer que não só *algo* pode dar errado, mas que *tudo* dará errado, é um passo significativo para estar preparado para qualquer emergência real. Uma matriz de todas as combinações possíveis de desastres, com planos para tratar cada um deles, permite que você durma profundamente por pelo menos uma noite; manter seus planos de recuperação atualizados e exercitados permite que você durma nas outras 364 noites do ano.

À medida que você se tornar mais eficaz para se recuperar de qualquer falha em um tempo N razoável, encontre maneiras de reduzir esse tempo por meio de detecções mais rápidas e específicas de perda, com a meta de se aproximar de $N = 0$. Você pode, então, passar do planejamento de recuperação para o planejamento de prevenção, com o objetivo de chegar ao Santo Graal de *todos os dados, o tempo todo*. Atinja essa meta, e você poderá dormir na praia naquelas suas bem merecidas férias.

¹ Atomicity, Consistency, Isolation, Durability (Atomicidade, Consistência, Isolamento, Durabilidade); veja <https://en.wikipedia.org/wiki/ACID>. Bancos de dados SQL, como MySQL e PostgreSQL, se esforçam para ter essas propriedades.

² Basically Available, Soft state, Eventual consistency (Basicamente disponível, Estado soft, Consistência eventual); veja https://en.wikipedia.org/wiki/Eventual_consistency. Sistemas BASE, como Bigtable e Megastore, com frequência também são descritos como “NoSQL”.

³ Para outras leituras sobre APIs ACID e BASE, veja [Gol14] e [Bai13].

⁴ Binary Large Object (Objeto Binário Grande); veja https://en.wikipedia.org/wiki/Binary_large_object.

⁵ Veja [https://en.wikipedia.org/wiki/Zero-day_\(computing\)](https://en.wikipedia.org/wiki/Zero-day_(computing)).

⁶ As tabuletas de argila são o exemplo mais antigo conhecido de escrita. Para uma discussão mais ampla sobre preservação de dados no longo prazo, veja [Con96].

⁷ Ao ler esse conselho, alguém poderia perguntar: como você precisa oferecer uma API sobre o repositório de dados para implementar uma remoção soft, por que parar na remoção soft quando você poderia oferecer muitas outras funcionalidades para proteção contra usuários apagando dados acidentalmente? Para tomar um exemplo específico proveniente da experiência do Google, considere o Blobstore: em vez de permitir que os clientes apaguem dados e metadados do Blob diretamente, as APIs do Blob implementam muitos recursos de segurança, incluindo políticas default de backup (réplicas offline), checksums fim a fim e tempos de vida default para objetos removidos (remoção soft). O fato é que, em várias ocasiões, a remoção soft salvou clientes do Blobstore de perda de dados que poderiam ter sido muito, muito piores. Certamente, há muitas funcionalidades para proteção contra remoção de dados que valem a pena, mas para empresas com exigência de prazos para remoção de dados, a remoção soft foi a proteção mais pertinente contra bugs e remoção acidental de

dados, no caso dos clientes do Blobstore.

- 8 “Snapshot”, nesse caso, refere-se a uma visão estática, somente para leitura, de uma instância de armazenagem, como snapshots de bancos de dados SQL. Os snapshots muitas vezes são implementados usando uma semântica de “cópia na escrita” (copy-on-write) para eficiência de armazenagem. Podem ser custosos por dois motivos: em primeiro lugar, eles competem com a mesma capacidade de armazenagem que os repositórios de dados ativos; em segundo lugar, quanto mais rápido seus dados mudarem, menos eficiente será a estratégia de “cópia na escrita”.
- 9 Para mais informações sobre a replicação em estilo GFS, consulte [Ghe03]. Para mais informações sobre códigos de correção Reed-Solomon, veja https://en.wikipedia.org/wiki/Reed–Solomon_error_correction.
- 10 Veja [http://en.wikipedia.org/wiki/Catch-22_\(logic\)](http://en.wikipedia.org/wiki/Catch-22_(logic)).
- 11 Na prática, implementar uma solução por meio de programação não foi um obstáculo, pois a maioria dos SREs são engenheiros de software experientes, como era o caso aqui. A expectativa de uma experiência como essa faz com que encontrar e contratar SREs seja uma tarefa notoriamente difícil e, a partir desse estudo de caso e de outros dados, você poderá começar a apreciar por que a SRE contrata engenheiros de software experientes; veja [Jon15].
- 12 Segundo a nossa experiência, os engenheiros de computação em nuvem com frequência ficam relutantes em configurar alertas de produção para taxas de remoção de dados por causa da variação natural das taxas de remoção de dados por usuário no tempo. Entretanto, como a intenção de um alerta desse tipo é detectar anomalias em taxas de remoção globais, e não locais, seria mais útil alertar quando a taxa de remoção de dados global, resultante da agregação de todos os usuários, cruzasse um limite extremo (por exemplo, dez vezes maior que aquela observada no 95º percentil), em oposição a alertas de taxa de remoção por usuário, menos úteis.

CAPÍTULO 27

Lançamento de produtos confiáveis em escala

Escrito por Rhandeev Singh e Sebastian Kirsch com Vivek Rau

Editado por Betsy Beyer

As empresas de internet como o Google são capazes de lançar novos produtos e funcionalidades em iterações muito mais rápidas que as empresas tradicionais. O papel da Site Reliability (Confiabilidade de Sites) nesse processo é permitir um ritmo rápido de mudanças, sem comprometer a estabilidade do site. Criamos uma equipe dedicada de “Launch Coordination Engineers” (Engenheiros de Coordenação de Lançamentos) para consultar as equipes de engenharia sobre os aspectos técnicos de um lançamento bem-sucedido.

A equipe também aperfeiçoou uma “checklist de lançamento” com perguntas rotineiras a serem feitas sobre um lançamento e receitas para resolver problemas comuns. A checklist se mostrou uma ferramenta útil para garantir que lançamentos confiáveis sejam reproduzíveis.

Considere um serviço comum do Google – por exemplo, o Keyhole, que serve imagens de satélite para o Google Maps e o Google Earth. Em um dia normal, o Keyhole serve até vários milhares de imagens de satélite por segundo. Porém, na véspera do Natal de 2011, ele recebeu 25 vezes mais que seu tráfego de pico normal – acima de um milhão de requisições por segundo. O que provocou esse surto enorme no tráfego?

Papai Noel estava chegando.

Alguns anos atrás, o Google colaborou com o NORAD (North American

Aerospace Defense Command, ou Comando de Defesa Aeroespacial Norte-American) para hospedar um site com tema de Natal que monitorasse o progresso de Papai Noel pelo mundo, permitindo que os usuários o assistissem entregando presentes em tempo real. Parte da experiência era um “voo virtual”, que usava imagens de satélite para acompanhar o progresso de Papai Noel sobre um mundo simulado.

Embora um projeto como o NORAD Acompanha Papai Noel possa parecer excêntrico, ele tinha todas as características que definem um lançamento difícil e arriscado: um prazo fixo (o Google não poderia pedir ao Papai Noel que viesse uma semana depois se o site não estivesse pronto), muita publicidade, um público-alvo de milhões de pessoas e uma elevação rápida de tráfego (todos iriam observar o site na véspera de Natal). Jamais subestime o poder de milhões de crianças ansiosas por presentes – esse projeto tinha uma real possibilidade de deixar os servidores do Google de joelhos.

A equipe de Site Reliability Engineering (Engenharia de Confiabilidade de Sites) do Google trabalhou arduamente para preparar a nossa infraestrutura para esse lançamento, garantindo que Papai Noel pudesse entregar todos os seus presentes a tempo, sob os olhares atentos de um público-alvo cheio de expectativas. A última coisa que queríamos era fazer as crianças chorarem porque não puderam ver Papai Noel entregando os presentes. De fato, apelidamos as várias kill switches incluídas na experiência para proteger nossos serviços de “Switches para fazer as crianças chorarem” (Make-children-cry switches). Prever as várias maneiras pelas quais esse lançamento poderia dar errado e fazer a coordenação entre os diferentes grupos de engenharia envolvidos no lançamento coube a uma equipe especial na Site Reliability Engineering: a equipe de LCE.

Lançar um novo produto ou funcionalidade é a hora da verdade para toda empresa – o momento em que meses ou anos de esforço são apresentados ao mundo. Empresas tradicionais lançam novos produtos a uma taxa razoavelmente baixa. O ciclo de lançamentos em empresas de internet é marcadamente diferente. Os lançamentos e as iterações rápidas são muito mais simples porque novas funcionalidades podem ser implantadas do lado do servidor, não exigindo rollout de software em estações de trabalho

individuais dos clientes.

O Google define como lançamento qualquer código novo que introduza uma mudança externamente visível a uma aplicação. Conforme as características de um lançamento – a combinação de atributos, o prazo, o número de passos envolvidos 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 taxa rápida de mudança oferece tanto o raciocínio quanto a oportunidade para criar um processo de lançamento organizado. Uma empresa que lance apenas um produto a cada três anos não precisa de um processo detalhado de lançamento. Na época em que um novo lançamento ocorrer, a maioria dos componentes do processo de lançamento desenvolvido antes estará desatualizada. As empresas tradicionais tampouco têm a oportunidade de definir um processo detalhado de lançamento, pois não acumulam experiência suficiente fazendo lançamentos de modo a gerar um processo robusto e maduro.

Launch Coordination Engineering

Bons engenheiros de software têm uma boa dose de expertise em programação e design, e compreendem muito bem a tecnologia de seus próprios produtos. No entanto, os mesmos engenheiros podem não ter familiaridade com os desafios e as armadilhas de lançar um produto para milhões de usuários, ao mesmo tempo que minimizam as interrupções de serviço e maximizam o desempenho.

O Google abordou os desafios inerentes aos lançamentos criando uma equipe de consultoria dedicada dentro da SRE, responsável pelo lado técnico do lançamento de um novo produto ou funcionalidade. Composta de engenheiros de software e engenheiros de sistema – alguns com experiência em outras equipes de SRE –, essa equipe é especializada em orientar os desenvolvedores em direção ao desenvolvimento de produtos confiáveis e rápidos, que atendam aos padrões de robustez, escalabilidade e confiabilidade do Google. Essa equipe de consultoria, LCE (Launch Coordination Engineering, ou Engenharia de Coordenação de Lançamentos), facilita

executar um processo de lançamento tranquilo de algumas maneiras:

- Fazendo a auditoria de produtos e serviços para ver se estão de acordo com os padrões de confiabilidade e as melhores práticas do Google e apresentando ações específicas para melhorar a confiabilidade.
- Atuando como uma ligação entre as várias equipes envolvidas em um lançamento.
- Orientando acerca dos aspectos técnicos de um lançamento, garantindo que as tarefas mantenham a dinâmica.
- Atuando como guardiões e assinando embaixo de lançamentos considerados “seguros”.
- Educando desenvolvedores nas melhores práticas e no modo de se integrar aos serviços do Google, disponibilizando documentação interna e recursos de treinamento para agilizar o aprendizado.

Membros da equipe de LCE fazem auditoria em vários momentos durante o ciclo de vida dos serviços. A maior parte das auditorias é conduzida antes de um novo produto ou serviço ser lançado. Se uma equipe de desenvolvimento de produto realizar um lançamento sem o suporte da SRE, a LCE oferece o conhecimento apropriado do domínio para garantir um lançamento tranquilo. Porém, mesmo produtos que já tenham um suporte sólido da SRE muitas vezes se envolvem com a equipe de LCE durante lançamentos críticos. Os desafios enfrentados pelas equipes no lançamento de um novo produto são substancialmente diferentes da operação cotidiana de um serviço confiável (uma tarefa em que as equipes de SRE já se destacam), e a equipe de LCE pode tirar proveito da experiência de centenas de lançamentos. Essa equipe também facilita as auditorias de serviço quando novos serviços se envolvem pela primeira vez com a SRE.

O papel do Launch Coordination Engineer

Nossa equipe de Launch Coordination Engineering é composta de LCEs contratados diretamente para essa função, ou SREs com experiência prática na operação de serviços do Google. Os LCEs devem ter os mesmos requisitos técnicos que qualquer outro SRE, além de se esperar que tenham habilidades

sólidas de comunicação e liderança – um LCE faz com que partes muito díspares se reúnem a fim de trabalhar em direção a um objetivo comum, faz a mediação de conflitos ocasionais, além de direcionar, orientar e educar os colegas engenheiros.

Uma equipe dedicada a coordenar lançamentos oferece as seguintes vantagens:

Experiência ampla

Como uma verdadeira equipe para vários produtos, os membros são ativos em quase todas as áreas de produto do Google. Um conhecimento de vários produtos e relacionamentos com muitas equipes na empresa fazem dos LCEs excelentes veículos de transferência de conhecimento.

Perspectiva multifuncional

Os LCEs têm uma visão holística do lançamento, o que lhes permite fazer a coordenação entre equipes díspares em SRE, desenvolvimento e gerenciamento de produto. Essa abordagem holística é particularmente importante para lançamentos complicados que envolvam mais de meia dúzia de equipes em vários fusos horários.

Objetividade

Como um conselheiro apartidário, um LCE tem a função de criar um equilíbrio e fazer a mediação entre stakeholders, incluindo SRE, desenvolvedores de produto, gerentes de produto e marketing.

Como o LCE é um cargo da SRE, os LCEs são incentivados a dar prioridade à confiabilidade em relação a outras preocupações. Uma empresa que não compartilhe as metas de confiabilidade do Google, mas compartilhe sua taxa rápida de mudanças, pode optar por uma estrutura diferente de incentivos.

Definindo um processo de lançamento

O Google aperfeiçoou seu processo de lançamento ao longo de um período de mais de dez anos. Com o tempo, identificamos vários critérios que caracterizam um bom processo de lançamento:

Leve

Fácil para os desenvolvedores.

Robusto

Captura erros óbvios.

Completo

Trata detalhes importantes de forma consistente e reproduzível.

Escalável

Acomoda tanto um número alto de lançamentos simples quanto menos lançamentos complexos.

Adaptável

Funciona bem tanto para tipos comuns de lançamentos (por exemplo, adicionar uma nova linguagem de UI em um produto) quanto para novos tipos (por exemplo, o lançamento inicial do navegador Chrome ou do Google Fiber).

Como podemos ver, alguns desses requisitos estão em evidente conflito. Por exemplo, é difícil conceber um processo que seja simultaneamente leve e completo. Achar um equilíbrio entre esses critérios exige um trabalho contínuo. O Google tem empregado algumas táticas com sucesso, que nos ajudam a atender aos critérios a seguir:

Simplicidade

Faça o básico corretamente. Não planeje para todas as eventualidades.

Uma abordagem de alto nível

Engenheiros experientes personalizam o processo para adequá-lo a cada lançamento.

Caminhos comuns rápidos

Identifique classes de lançamentos que sempre sigam um padrão comum (por exemplo, lançar um produto em um novo país) e ofereça um processo de lançamento simplificado para essa classe.

A experiência tem mostrado que é provável que os engenheiros deixem de

lado os processos que considerem pesados demais ou que não agreguem valor suficiente – em especial, quando uma equipe já está em plena operação e o processo de lançamento é visto como apenas mais um item bloqueando o lançamento. Por esse motivo, a LCE deve otimizar a experiência de lançamento continuamente a fim de atingir o equilíbrio correto entre custo e benefício.

A checklist de lançamento

As checklists são usadas para reduzir falhas e garantir consistência e completude entre uma variedade de disciplinas. Exemplos comuns incluem as checklists antes de voos e as checklists para cirurgias [Gaw09]. De modo semelhante, a LCE emprega uma checklist de lançamento para qualificá-lo. A checklist (Apêndice E) ajuda um LCE a avaliar o lançamento e oferece à equipe de lançamento uma lista de ações e referências para mais informações. Eis alguns exemplos de itens que uma checklist pode incluir:

- Pergunta: Você precisa de um novo nome de domínio?
 - Ação: Coordene com o marketing sobre o nome de domínio desejado e solicite o registro do domínio. Eis um link para o formulário de marketing.
- Pergunta: Você está armazenando dados persistentes?
 - Ação: Certifique-se de ter implementado backups. Eis as instruções para implementar backups.
- Pergunta: Um usuário poderia potencialmente abusar de seu serviço?
 - Ação: Implemente limitação de taxa e cotas. Utilize o serviço compartilhado a seguir.

Na prática, há quase um número infinito de perguntas a serem feitas sobre qualquer sistema, e a checklist pode aumentar facilmente até atingir um tamanho impossível de administrar. Manter uma carga administrável sobre os desenvolvedores exige um tratamento cuidadoso da checklist. Em um esforço para conter o seu crescimento, em certo momento, adicionar novas perguntas à checklist de lançamento do Google passou a exigir a aprovação de um vice-presidente. A LCE atualmente utiliza as seguintes diretrizes:

- A importância de cada pergunta deve ser bem fundamentada, idealmente por um desastre em um lançamento anterior.
- Toda instrução deve ser concreta, prática e razoável para os desenvolvedores executarem.

A checklist precisa receber atenção contínua para permanecer relevante e atualizada: recomendações mudam com o tempo, sistemas internos são substituídos por sistemas diferentes e áreas de preocupação de lançamentos anteriores se tornam obsoletas por causa de novas políticas e processos. Os LCEs cuidam da checklist continuamente e fazem pequenas atualizações quando os membros da equipe percebem que há itens que devem ser modificados. Uma ou duas vezes por ano, um membro da equipe revisa toda a checklist para identificar itens obsoletos e, então, trabalha com os proprietários dos serviços e os experts no assunto para modernizar as seções da checklist.

Levando à convergência e à simplificação

Em uma empresa de grande porte, os engenheiros podem não estar cientes da infraestrutura disponível para tarefas comuns (como limitação de taxa). Sem uma orientação apropriada, é provável que eles reimplementem soluções existentes. Convergir para um conjunto comum de bibliotecas de infraestrutura evita esse cenário e proporciona vantagens óbvias à empresa: reduz o esforço duplicado, deixa o conhecimento mais facilmente transferível entre os serviços e resulta em uma engenharia de mais alto nível e mais qualidade de serviço por causa da atenção concentrada, dedicada à infraestrutura.

Quase todos os grupos do Google participam de um processo de lançamento comum, o que faz da checklist de lançamento um veículo para levar a convergência a uma infraestrutura comum. Em vez de implementar uma solução personalizada, a LCE pode recomendar a infraestrutura existente como blocos de construção – uma infraestrutura que já se tornou robusta ao longo de anos de experiência e que pode ajudar a atenuar os riscos à capacidade, ao desempenho e à escalabilidade. Exemplos incluem a infraestrutura comum para limitação de taxa ou uso de cotas de usuários,

envio de novos dados aos servidores ou atualização de novas versões de um binário. Esse tipo de padronização ajudou a simplificar radicalmente a checklist de lançamento: por exemplo, seções longas da checklist que lidavam com requisitos para limitação de taxas puderam ser substituídas por uma única linha que dizia: “Implemente a limitação de taxa usando o sistema X”.

Por causa da amplitude da experiência englobando todos os produtos do Google, os LCEs também estão em uma posição única para identificar oportunidades de simplificação. Enquanto trabalham em um lançamento, eles observam os obstáculos em primeira mão: quais partes de um lançamento apresentam mais dificuldade, quais passos exigem uma quantidade de tempo desproporcional, quais problemas são resolvidos várias vezes de forma independente, mas semelhante, em que pontos falta uma infraestrutura comum ou há duplicação na infraestrutura comum.

Os LCEs têm diversas maneiras de organizar a experiência de lançamento e atuar como advogados para as equipes de lançamento. Por exemplo, os 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 pontos complicados aos proprietários da infraestrutura comum e abrir um diálogo com os clientes. Ao tirar proveito da experiência obtida no curso de vários lançamentos anteriores, os LCEs podem dedicar mais atenção às preocupações e sugestões individuais.

Lançando o inesperado

Quando um projeto entra no espaço ou na vertical de um novo produto, um LCE talvez precise criar uma checklist apropriada do zero. Fazer isso muitas vezes envolve sintetizar a experiência de experts nos domínios relevantes. Ao criar uma versão preliminar de uma nova checklist, talvez seja conveniente estruturá-la em torno de temas amplos, como confiabilidade, modos de falha e processos.

Por exemplo, antes do lançamento do Android, o Google raramente havia lidado com dispositivos de consumo em massa com uma lógica do lado

cliente que não controlássemos diretamente. Embora possamos, mais ou menos, corrigir facilmente um bug no Gmail em horas ou dias enviando novas versões de JavaScript aos navegadores, essas correções não são uma opção com dispositivos móveis. Assim, os LCEs que trabalhavam nos lançamentos para dispositivos móveis envolveram experts em domínios móveis para determinar quais seções das checklists existentes se aplicavam ou não, e em que pontos novas perguntas na checklist eram necessárias. Nessas conversas, é importante ter o *propósito* de cada pergunta em mente a fim de evitar aplicar negligentemente uma pergunta concreta ou uma ação que não sejam relevantes ao design do produto único sendo lançado. Um LCE que esteja diante de um lançamento incomum deve retornar aos princípios iniciais básicos de como executar um lançamento seguro e, então, especializá-los novamente para deixar a checklist concreta e útil aos desenvolvedores.

Desenvolvendo uma checklist de lançamento

Uma checklist é fundamental para lançar novos serviços e produtos com confiabilidade reproduzível. Nossa checklist de lançamento cresceu com o tempo e foi periodicamente aperfeiçoada por membros da equipe de Launch Coordination Engineering. Os detalhes de uma checklist de lançamento serão diferentes para cada empresa, pois as especificidades devem ser personalizadas conforme os serviços internos e a infraestrutura de uma empresa. Nas próximas seções, extrairemos uma série de temas das checklists de LCE do Google e mostraremos exemplos de como esses temas podem ser complementados.

Arquitetura e dependências

Uma revisão de arquitetura permite determinar se o serviço está usando corretamente a infraestrutura compartilhada e identifica os proprietários dessa infraestrutura como stakeholders adicionais no lançamento. O Google tem um grande número de serviços internos utilizados como blocos de construção para novos produtos. Durante as últimas etapas do planejamento de capacidade (veja [Hix15a]), a lista de dependências identificada nessa seção

da checklist pode ser usada para garantir que todas as dependências estejam corretamente provisionadas.

Exemplos de perguntas da checklist

- Qual é o fluxo de sua requisição, do usuário ao frontend e até o backend?
- Há tipos diferentes de requisições com requisitos diferentes de latência?

Exemplos de ações

- Isole as requisições voltadas aos usuários das requisições não voltadas aos usuários.
- Valide as suposições de volume de requisições. Uma visão de página pode se transformar em muitas requisições.

Integração

Os serviços de muitas empresas executam em um ecossistema interno que implica diretrizes sobre como configurar máquinas, novos serviços e a monitoração, como integrar-se com a distribuição de carga, configurar endereços de DNS e assim por diante. Esses ecossistemas internos geralmente crescem com o tempo e, com frequência, têm suas próprias idiossincrasias e armadilhas para navegar. Assim, essa seção da checklist variará muito de empresa para empresa.

Exemplos de ações

- Configure um novo nome de DNS para o seu serviço.
- Configure distribuidores de carga para conversar com o seu serviço.
- Configure a monitoração para o seu novo serviço.

Planejamento de capacidade

Novas funcionalidades podem exibir um aumento temporário de uso no lançamento, que diminuirá após alguns dias. O tipo de carga de trabalho ou combinação de tráfego de um pico de lançamento pode ser substancialmente diferente do estado estável, invalidando os resultados de testes de carga. O interesse do público é notoriamente difícil de prever, e alguns produtos do

Google tiveram que acomodar picos de lançamento até 15 vezes maiores que o valor estimado inicialmente. Lançar inicialmente em uma região ou um país de cada vez ajuda a desenvolver a confiança necessária para tratar lançamentos maiores.

A capacidade interage com a redundância e a disponibilidade. Por exemplo, se você precisa de três implantações replicadas para servir a 100% de seu tráfego no pico, será necessário manter quatro ou cinco implantações, uma ou duas das quais serão redundantes, para proteger os usuários de manutenções e mau funcionamento inesperado. Os recursos de datacenter e de rede com frequência têm um tempo de espera longo e devem ser solicitados com bastante antecedência para que sua empresa possa obtê-los.

Exemplos de perguntas da checklist

- Esse lançamento está ligado a um comunicado de imprensa, propagandas, postagem de blog ou outra forma de promoção?
- Qual é o volume de tráfego e a taxa de crescimento esperados durante e após o lançamento?
- Você adquiriu todos os recursos computacionais necessários para dar suporte ao seu tráfego?

Modos de falha

Uma observação sistemática dos possíveis modos de falha de um novo serviço garante alta confiabilidade desde o início. Nessa parte da checklist, analise cada componente e dependência e identifique o impacto de sua falha. O serviço é capaz de lidar com falhas em máquinas individuais? Com interrupções de datacenter? Falhas de rede? Como lidamos com dados de entrada ruins? Estamos preparados para a possibilidade de um ataque de DoS (denial-of-service, ou negação de serviço)? O serviço continuará a servir em modo degradado se uma de suas dependências falhar? Como lidamos com a indisponibilidade de uma dependência na inicialização do serviço? E durante a execução?

Exemplos de perguntas da checklist

- Você tem algum ponto único de falha em seu design?
- Como você atenua a indisponibilidade de suas dependências?

Exemplos de ações

- Implemente tempos de espera para requisições a fim de evitar ficar sem recursos em requisições de longa duração.
- Implemente o descarte de carga para rejeitar novas requisições mais cedo em situações de sobrecarga.

Comportamento do cliente

Em um site tradicional, raramente há a necessidade de levar em consideração um comportamento abusivo por parte de usuários legítimos. Quando toda requisição é acionada por uma ação do usuário (por exemplo, um clique em um link), as taxas de requisições 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 de usuários – por exemplo, um aplicativo de celular que sincronize periodicamente seus dados na nuvem ou um site que se atualize periodicamente. Em qualquer um desses cenários, um comportamento abusivo do cliente pode ameaçar facilmente a estabilidade de um serviço. (Há também a questão de proteger um serviço de um tráfego abusivo, como scrapers e ataques de negação de serviço – que é diferente de fazer o design de um comportamento seguro para clientes diretos.)

Exemplo de pergunta da checklist

- Você tem funcionalidades como salvar automaticamente/completar automaticamente/heartbeat?

Exemplos de ações

- Certifique-se de que seus clientes façam backoffs exponencialmente em caso de falha.
- Certifique-se de incluir jitter nas requisições automáticas.

Processos e automação

O Google incentiva os engenheiros a usar ferramentas padrões para automatizar processos comuns. No entanto, a automação nunca é perfeita, e todo serviço tem processos que devem ser executados por um ser humano: criar uma nova versão, passar o serviço para um datacenter diferente, restaurar dados de backups e assim por diante. Por questões de confiabilidade, nos esforçamos para minimizar pontos únicos de falha, o que inclui seres humanos.

Esse processo restante deve ser documentado antes do lançamento para garantir que as informações sejam traduzidas da mente de um engenheiro para o papel enquanto ainda estão frescas e estejam disponíveis em caso de emergência. Os processos devem ser documentados de modo que qualquer membro da equipe possa executar um dado processo em uma emergência.

Exemplo de pergunta da checklist

- Há algum processo manual necessário para manter o serviço executando?

Exemplos de ações

- Documente todos os processos manuais.
- Documente o processo para transferir seu serviço para um novo datacenter.
- Automatize o processo para construir e disponibilizar uma nova versão.

Processo de desenvolvimento

O Google faz um uso intenso de controle de versões, e quase todos os processos de desenvolvimento estão profundamente integrados ao sistema de controle de versões. Muitas de nossas melhores práticas giram em torno do uso eficiente do sistema de controle de versões. Por exemplo, realizamos a maior parte do desenvolvimento no branch da linha principal, mas as versões são construídas em branches separados. Essa configuração facilita corrigir bugs em uma versão sem incluir modificações da linha principal, não relacionadas a esses bugs.

O Google também utiliza controle de versões para outros propósitos, como

armazenar arquivos de configuração. Muitas das vantagens do controle de versões – acompanhamento de histórico, atribuição das mudanças aos indivíduos e revisões de código – se aplicam também aos arquivos de configuração. Em alguns casos, também propagamos as mudanças do sistema de controle de versões para os servidores ativos automaticamente, de modo que um engenheiro só precisa submeter uma mudança para que ela seja ativada.

Exemplos de ações

- Inclua todo o código e os arquivos de configuração no sistema de controle de versões.
- Gere cada versão em um novo branch de release.

Dependências externas

Às vezes, um lançamento depende de fatores que estão além do controle da empresa. Identificar esses fatores permite atenuar a imprevisibilidade que eles implicam. Por exemplo, a dependência pode ser uma biblioteca de código mantida por terceiros, ou um serviço ou dados fornecidos por outra empresa. Se uma interrupção de serviço, bug, erro sistemático, problema de segurança ou um limite inesperado de escalabilidade em um terceiro ocorrer, um planejamento com antecedência permitirá evitar ou atenuar os danos aos seus usuários. Na história de lançamentos do Google, utilizamos proxies de filtro e/ou reescrita, pipelines de transcodificação de dados e caches para atenuar alguns desses riscos.

Exemplos de perguntas da checklist

- De quais códigos de terceiros, dados, serviços ou eventos o serviço ou o lançamento dependem?
- Algum parceiro depende de seu serviço? Em caso afirmativo, eles precisam ser notificados de seu lançamento?
- O que acontecerá se você ou o fornecedor não puderem atender a um prazo fixo de lançamento?

Planejamento do rollout

Em sistemas distribuídos de grande porte, poucos eventos acontecem instantaneamente. Por questões de confiabilidade, esse imediatismo, de qualquer modo, muitas vezes não é ideal. Um lançamento complicado pode exigir a habilitação de funcionalidades individuais em vários subsistemas diferentes, e cada uma dessas mudanças de configuração pode demorar horas para ser concluída. Ter uma configuração funcional em uma instância de teste não garante que a mesma configuração poderá ser implantada na instância ativa. Às vezes, uma dança complicada ou uma funcionalidade especial é necessária para que todos os componentes sejam lançados de forma limpa, na ordem correta.

Requisitos externos de equipes como marketing e relações públicas podem acrescentar outras complicações. Por exemplo, uma equipe talvez precise que uma funcionalidade esteja disponível a tempo para uma apresentação em uma conferência, mas precisa deixar a funcionalidade invisível até esse momento.

Medidas de contingência são outra parte do planejamento de rollout. E se você não conseguir habilitar a funcionalidade a tempo para a apresentação? Às vezes, essas medidas de contingência são tão simples quanto preparar um conjunto de slides backup que diga “Lançaremos essa funcionalidade nos próximos dias” em vez de “Lançamos essa funcionalidade”.

Exemplos de ações

- Defina um plano de lançamento que identifique as ações necessárias para lançar o serviço. Identifique o responsável por cada ação.
- Identifique o risco nos passos individuais do lançamento e implemente medidas de contingência.

Técnicas selecionadas para lançamentos confiáveis

Conforme descrito em outras partes deste livro, o Google desenvolveu uma série de técnicas para operar sistemas confiáveis ao longo dos anos. Algumas dessas técnicas são particularmente bem adequadas para lançar produtos de forma segura. Elas também oferecem vantagens durante a operação normal

do serviço, mas é particularmente importante usá-las de forma correta durante a fase de lançamento.

Rollouts graduais e em fases

Um adágio da administração de sistemas é “nunca mude um sistema em execução”. Qualquer mudança representa um risco, e os riscos devem ser minimizados para garantir a confiabilidade de um sistema. O que é verdadeiro para qualquer sistema pequeno é duplamente verdadeiro para sistemas globalmente distribuídos e altamente replicados, como aqueles executados pelo Google.

Poucos lançamentos no Google são do tipo “apertar um botão”, em que lançamos um novo produto em um horário específico para o mundo todo usar. Com o tempo, o Google desenvolveu uma série de padrões que permitem lançar produtos e funcionalidades gradualmente e, desse modo, minimizar o risco; veja o Apêndice B.

Quase todas as atualizações nos serviços do Google ocorrem gradualmente, de acordo com um processo definido, com passos apropriados de verificação intercalados. Um novo servidor pode ser instalado em algumas máquinas em um datacenter e observado por um período de tempo definido. Se tudo parecer bem, o servidor é instalado em todas as máquinas em um datacenter, observado novamente e, então, instalado em todas as máquinas globalmente. As primeiras fases de um rollout geralmente são chamadas de “canários” (canaries) – uma alusão aos canários levados por mineiros às minas de carvão para detectar gases perigosos. Nossos servidores canários detectam efeitos perigosos resultantes do comportamento do novo software sujeito a tráfego real de usuário.

Teste canário (canary testing) é um conceito incluído em muitas das ferramentas internas do Google usadas para fazer mudanças automatizadas, assim como em sistemas que alterem arquivos de configuração. As ferramentas que administram a instalação de um software novo geralmente observam o servidor recém-iniciado por um tempo, garantindo que ele não vá falhar ou apresentar um mau comportamento. Se a mudança não passar pelo período de validação, um rollback será feito automaticamente.

O conceito de rollouts graduais se aplica até mesmo a softwares que não executam nos servidores do Google. Novas versões de um aplicativo Android podem sofrer rollout gradualmente, em que a versão atualizada é oferecida como um subconjunto das instalações para upgrade. O percentual de instâncias atualizadas aumenta gradualmente com o tempo até atingir 100%. Esse tipo de rollout é particularmente conveniente se a nova versão resultar em tráfego adicional aos servidores de backend nos datacenters do Google. Dessa maneira, podemos observar o efeito em nossos servidores à medida que fazemos rollout da nova versão gradualmente e detectamos os problemas com antecedência.

O sistema de convite é outro tipo de rollout gradual. Frequentemente, em vez de permitir inscrições gratuitas para um novo serviço, apenas um número limitado de usuários tem permissão para se inscrever por dia. Inscrições com taxa limitada geralmente são combinadas com um sistema de convite, em que um usuário pode enviar um número limitado de convites aos amigos.

Frameworks de flag para funcionalidades

Com frequência, o Google expande os testes de pré-lançamento com estratégias que atenuem o risco de uma interrupção no serviço. Um mecanismo para fazer rollout das mudanças lentamente, permitindo observar o comportamento do sistema como um todo com cargas de trabalho reais, pode compensar o investimento em engenharia em termos de confiabilidade, velocidade de engenharia e tempo para chegar ao mercado (time to market). Esses mecanismos se provaram particularmente úteis em casos em que ambientes de teste realistas são impraticáveis ou em lançamentos particularmente complexos, para os quais os efeitos podem ser difíceis de prever.

Além do mais, nem todas as mudanças são iguais. Às vezes, você simplesmente quer verificar se um pequeno ajuste na interface melhora a experiência de seus usuários. Pequenas mudanças como essa não devem envolver milhares de linhas de código nem um processo pesado de lançamento. Você pode testar centenas dessas mudanças em paralelo.

Por fim, às vezes, você vai querer saber se uma pequena amostra dos usuários

gosta de usar um protótipo de uma nova funcionalidade difícil de implementar. Você não vai querer investir meses de esforço em engenharia para deixar uma nova funcionalidade robusta e servir milhões de usuários, somente para descobrir que ela é um fiasco.

Para acomodar os cenários anteriores, vários produtos do Google conceberam frameworks de flag para funcionalidades. Alguns desses frameworks foram projetados para fazer o rollout gradual de novas funcionalidades, de 0% a 100% dos usuários. Sempre que um produto introduzia um framework desse tipo, deixávamos o próprio framework o mais robusto possível para que a maioria de suas aplicações não precisasse de nenhum envolvimento da LCE. Esses frameworks geralmente atendem aos seguintes requisitos:

- Fazem rollout de muitas mudanças em paralelo, cada um para alguns servidores, usuários, entidades ou datacenters.
- Aumentam gradualmente até atingir um grupo grande, porém limitado, de usuários, em geral entre 1% a 10%.
- Direcionam o tráfego por meio de servidores diferentes, de acordo com os usuários, as sessões, os objetos e/ou as localidades.
- Tratam automaticamente falhas dos novos caminhos de código, por design, sem afetar os usuários.
- Revertam independentemente cada uma dessas mudanças de forma imediata, caso haja bugs sérios ou efeitos colaterais.
- Avaliam a extensão com que cada mudança melhora a experiência do usuário.

Os frameworks de flag de funcionalidades do Google se enquadram em duas classes gerais:

- Aqueles que primordialmente facilitam melhorias na interface de usuário
- Aqueles que dão suporte a mudanças arbitrárias do lado do servidor e na lógica de negócios

O framework mais simples de flag de funcionalidades para mudanças na interface de usuário em um serviço sem estado (stateless) é um framework de reescrita de payload HTTP nos servidores de aplicações de frontend, limitado

a um subconjunto de cookies ou outro atributo semelhante de requisição/resposta HTTP. Um sistema de configuração pode especificar um identificador associado aos novos caminhos de código e o escopo da mudança (por exemplo, hash do cookie mod intervalo), listas brancas e listas negras.

Serviços com estado (stateful) tendem a limitar as flags de funcionalidades a um subconjunto de identificadores únicos de usuários que fizeram login ou às entidades do produto acessadas, como o ID de documentos, planilhas ou objetos de armazenagem. Em vez de reescrever payloads HTTP, é mais provável que os serviços com estado façam proxy ou reencaminhem requisições para servidores diferentes, conforme a mudança, conferindo a capacidade de testar uma lógica de negócios melhorada e novas funcionalidades mais complexas.

Lidando com comportamentos abusivos de clientes

O exemplo mais simples de comportamento abusivo de cliente é um julgamento errôneo das taxas de atualização. Um novo cliente que se sincronize a cada 60 segundos, em oposição a fazê-lo a cada 600 segundos, provoca uma carga dez vezes maior no sistema. O comportamento de retentativas tem uma série de armadilhas que afetam requisições iniciadas por usuários, assim como requisições iniciadas por clientes. Tome o exemplo de um serviço que esteja sobrecarregado e, desse modo, deixe algumas requisições falharem: se os clientes fizerem retentativas para as requisições com falha, eles adicionarão carga a um serviço já sobrecarregado, resultando em mais retentativas e em mais requisições. Em vez disso, os clientes devem reduzir a frequência das retentativas, geralmente adicionando atrasos exponencialmente crescentes entre elas, além de considerar com cuidado os tipos de erro que mereçam uma retentativa. Por exemplo, um erro de rede normalmente autoriza uma retentativa, porém um erro HTTP 4xx (que indica um erro do lado do cliente), em geral, não autoriza.

Uma sincronização intencional ou inadvertida de requisições automatizadas em um thundering herd (semelhante àquelas descritas nos Capítulos 24 e 25) é outro exemplo comum de comportamento abusivo do cliente. Um

desenvolvedor de aplicativo para celulares pode decidir que duas horas da manhã é uma boa hora para fazer download de atualizações, pois é bem provável que o usuário esteja dormindo e o download não lhe será inconveniente. No entanto, um design como esse resulta em um bombardeio de requisições ao servidor de download às duas da manhã todas as noites, e em quase nenhuma requisição em qualquer outro horário. Em vez disso, cada cliente deveria escolher o horário para esse tipo de requisição de forma aleatória.

A aleatoriedade também deve ser injetada em outros processos periódicos. Para retornar às retentativas mencionadas antes: vamos tomar o exemplo de um cliente que envie uma requisição e, quando encontra uma falha, faça retentativas após um segundo, depois dois segundos, quatro segundos, e assim sucessivamente. Sem a aleatoriedade, um rápido pico de requisições que leve a uma taxa de erro maior poderia se repetir por causa das retentativas após um segundo, então dois segundos e depois quatro. Para distribuir esses eventos sincronizados, cada espera deve ter um jitter (isto é, deve ser ajustada por um valor aleatório).

A capacidade de controlar o comportamento de um cliente do lado do servidor se mostrou ser uma ferramenta importante no passado. Para um aplicativo em um dispositivo, um controle desse tipo pode significar instruir o cliente para verificar periodicamente o servidor e fazer download de um arquivo de configuração. O arquivo deve habilitar ou desabilitar determinadas funcionalidades ou definir parâmetros, por exemplo, a frequência com que o cliente se sincroniza ou faz retentativas.

A configuração do cliente pode até mesmo habilitar completamente uma nova funcionalidade voltada ao usuário. Ao hospedar o código que dá suporte à nova funcionalidade na aplicação cliente antes de ativá-la, reduzimos enormemente o risco associado a um lançamento. Lançar uma nova versão será muito mais fácil se não precisarmos manter caminhos paralelos para a versão com a nova funcionalidade *versus* a versão sem a funcionalidade. Isso é particularmente verdadeiro se não estivermos lidando com uma única nova funcionalidade, mas com um conjunto independente delas, que possa ser lançado com cronogramas diferentes, o que exigiria manter uma explosão

combinatória de diferentes versões.

Ter esse tipo de funcionalidade dormente também faz com que abortar lançamentos seja mais fácil quando efeitos adversos são descobertos durante um rollout. Em casos assim, podemos simplesmente desativar a funcionalidade, iterar e lançar uma versão atualizada do aplicativo. Sem esse tipo de configuração de cliente, teríamos que oferecer uma nova versão do aplicativo sem a funcionalidade e atualizá-la nos telefones de todos os usuários.

Comportamento de sobrecarga e testes de carga

Situações de sobrecarga são um modo de falha particularmente complexo e, desse modo, merecem uma atenção especial. Um sucesso rápido geralmente é a causa mais bem-vinda para uma sobrecarga quando um novo serviço é lançado, mas há uma variedade de outras causas, incluindo falhas na distribuição de carga, interrupções de serviço em máquinas, comportamento sincronizado de clientes e ataques externos.

Um modelo ingênuo supõe que o uso de CPU em uma máquina que ofereça um serviço em particular escala linearmente com a carga (por exemplo, o número de requisições ou a quantidade de dados processada) e, depois que a CPU disponível se esgotar, o processamento simplesmente ficará mais lento. Infelizmente, os serviços raramente se comportam desse modo ideal no mundo real. Muitos serviços são muito mais lentos quando não estão carregados, em geral por causa do efeito de vários tipos de caches, como caches de CPU, caches JIT e caches de dados específicos do serviço. À medida que a carga aumenta, geralmente há uma janela em que o uso de CPU e a carga no serviço correspondem linearmente e os tempos de resposta permanecem constantes, em sua maior parte.

Em algum momento, muitos serviços alcançam um ponto de não linearidade quando se aproximam do estado de sobrecarga. Na maioria dos casos benignos, os tempos de resposta simplesmente começam a aumentar, resultando em uma experiência de usuário degradada, mas não necessariamente provocando uma interrupção no serviço (embora uma dependência lenta possa causar erros visíveis ao usuário mais para cima na

pilha, por causa de tempos de espera de RPC esgotados). Nos casos mais drásticos, um serviço trava totalmente em resposta à sobrecarga.

Para citar um exemplo específico de comportamento em sobrecarga: um serviço fazia log de informações de depuração em resposta a erros do backend. O fato é que fazer log das informações de depuração era mais custoso que tratar a resposta do backend em um caso normal. Assim, à medida que o serviço ficava sobrecarregado e as respostas do backend sofriam timeout em sua própria pilha de RPC, o serviço gastava mais tempo de CPU ainda para fazer log dessas respostas, fazendo mais requisições sofrerem timeout nesse ínterim, até que o serviço parou completamente. Em serviços que executam na JVM (Java Virtual Machine), um efeito semelhante de travar completamente às vezes é chamado de “thrashing de GC” (garbage collection, ou coleta de lixo). Nesse cenário, o gerenciamento de memória interna da máquina virtual executa em ciclos cada vez mais próximos, tentando disponibilizar memória, até que a maior parte do tempo de CPU é consumida pelo gerenciamento de memória.

Infelizmente, é muito difícil prever desde o início como um serviço reagirá à sobrecarga. Assim, testes de carga são uma ferramenta de valor inestimável, tanto por questões de confiabilidade quanto para planejamento de capacidade, e são necessários para a maioria dos lançamentos.

Desenvolvimento da LCE

Nos anos de formação do Google, o tamanho da equipe de engenharia dobrou anualmente por vários anos seguidos, fragmentando o departamento de engenharia em várias equipes pequenas que trabalhavam em muitos produtos e funcionalidades novos e experimentais. Em um ambiente como esse, engenheiros iniciantes corriam o risco de repetir os erros de seus antecessores, especialmente quando se tratava do lançamento de novas funcionalidades e produtos com sucesso.

Para atenuar a repetição desses erros, absorvendo as lições aprendidas com lançamentos passados, um pequeno grupo de engenheiros experientes, chamados de “Launch Engineers” (Engenheiros de Lançamento), se ofereceu como voluntário para atuar como uma equipe de consultoria. Os Engenheiros

de Lançamento desenvolveram checklists para os lançamentos de novos produtos, incluindo tópicos como:

- Quando fazer consultas ao departamento jurídico
- Como selecionar nomes de domínio
- Como registrar novos domínios sem configurar erroneamente o DNS
- Designs comuns de engenharia e armadilhas da implantação em produção

As “Launch Reviews” (Revisões de Lançamento), como as sessões de consultoria com os Engenheiros de Lançamento passaram a ser chamadas, se tornaram uma prática comum dias ou semanas antes do lançamento de muitos produtos novos.

Em dois anos, os requisitos de implantação de produto na checklist de lançamento aumentaram e se tornaram complexos. Em conjunto com a complexidade crescente do ambiente de implantação do Google, tornou-se cada vez mais desafiador para os engenheiros de produto se manterem atualizados acerca de como fazer mudanças de forma segura. Ao mesmo tempo, a organização SRE estava crescendo rapidamente, e SREs sem experiência às vezes eram demasiadamente cuidadosos e avessos a mudanças. O Google corria o risco de que as negociações resultantes entre essas duas partes reduzissem a velocidade de lançamentos de produtos/funcionalidades.

Para atenuar esse cenário do ponto de vista da engenharia, a SRE montou uma equipe pequena de LCEs em tempo integral em 2004. Eles eram responsáveis por acelerar os lançamentos de novos produtos e funcionalidades, ao mesmo tempo que aplicavam a expertise de SRE para garantir que o Google lançasse produtos confiáveis com alta disponibilidade e baixa latência.

Os LCEs eram responsáveis por garantir que os lançamentos fossem executados rapidamente, sem failover dos serviços, e que se um lançamento falhasse não derrubaria outros produtos. Eles também eram responsáveis por manter os stakeholders informados sobre a natureza e a probabilidade de falhas como essa ocorrerem sempre que um atalho fosse tomado a fim de acelerar o tempo para chegar ao mercado. Suas sessões de consultoria foram

formalizadas como Production Reviews (Revisões para Produção).

Evolução da checklist de LCE

À medida que o ambiente do Google se tornou mais complexo, o mesmo ocorreu com a checklist da LCE (veja o Apêndice E) e o volume de lançamentos. Em três anos e meio, um LCE fez 350 lançamentos por meio da checklist de LCE. Como a equipe tinha uma média de cinco engenheiros durante esse período de tempo, isso se traduz em um fluxo de lançamentos no Google de mais de 1.500 em três anos e meio!

Embora cada pergunta na checklist de LCE seja simples, boa parte da complexidade está embutida no que motivou a pergunta e nas implicações de sua resposta. Para entender totalmente esse grau de complexidade, um novo contratado em LCE exige aproximadamente seis meses de treinamento.

À medida que o volume de lançamentos crescia, mantendo o compasso com a equipe de engenharia do Google dobrando, os LCEs procuraram maneiras de simplificar suas revisões. Eles identificaram categorias de lançamentos de baixo risco, pouco prováveis de enfrentar ou provocar contratempos. Por exemplo, um lançamento de funcionalidade que não envolvesse novos executáveis no servidor e um aumento de tráfego abaixo de 10% seria considerado de baixo risco. Lançamentos como esse tinham uma checklist quase trivial, enquanto lançamentos de alto risco passavam pelo conjunto completo de verificações e balanceamentos. Por volta de 2008, 30% das revisões eram consideradas de baixo risco.

Simultaneamente, o ambiente do Google estava escalando, removendo restrições em muitos lançamentos. Por exemplo, a aquisição do YouTube forçou o Google a ampliar sua rede e utilizar a largura de banda com mais eficiência. Isso significava que muitos produtos menores se encaixariam “entre as rachaduras”, evitando um planejamento de capacidade de rede e processos de provisionamento complexos, acelerando assim seus lançamentos. O Google também começou a criar datacenters bem grandes, capazes de hospedar vários serviços dependentes embaixo do mesmo teto. Esse desenvolvimento simplificou o lançamento de novos produtos que precisavam de um alto volume de capacidade em vários serviços

preexistentes dos quais dependiam.

Problemas que a LCE não resolveu

Embora os LCEs tentassem manter a burocracia das revisões a um nível mínimo, esses esforços não foram suficientes. Por volta de 2009, as dificuldades de lançar um novo serviço pequeno no Google haviam se tornado lendárias. Os serviços que cresciam e passavam a ser de larga escala tinham seu próprio conjunto de problemas que a LCE não era capaz de resolver.

Mudanças de escalabilidade

Quando os produtos têm muito mais sucesso que as estimativas iniciais e seu uso aumenta em mais de duas ordens de magnitude, manter o compasso com a carga exige várias mudanças de design. Essas mudanças de escalabilidade, em conjunto com o acréscimo contínuo de funcionalidades, muitas vezes deixam o produto mais complexo, frágil e difícil de operar. Em algum momento, a arquitetura do produto original se torna impossível de administrar e o produto precisa passar por uma revisão completa de arquitetura. Refazer a arquitetura do produto e então migrar todos os usuários da arquitetura antiga para a nova exige um grande investimento de tempo e de recursos, tanto dos desenvolvedores quanto dos SREs, reduzindo a taxa de desenvolvimento de novas funcionalidades durante esse período.

Carga operacional crescente

Ao executar um serviço após o seu 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 controlá-la. O ruído das notificações automáticas, a complexidade dos procedimentos de implantação e o overhead do trabalho de manutenção manual tendem a aumentar com o tempo e consomem volumes cada vez maiores da capacidade do proprietário do serviço, deixando menos tempo para o desenvolvimento de funcionalidades à equipe. A SRE tem uma meta difundida internamente de manter o trabalho operacional abaixo de um máximo de 50%; veja o Capítulo 5. Ficar abaixo desse máximo exige uma

monitoração constante daquilo que causa a carga operacional, assim como um esforço direcionado para eliminar essas causas.

Mudança na infraestrutura

Se a infraestrutura subjacente (como sistemas para gerenciamento de cluster, armazenagem, monitoração, distribuição de carga e transferência de dados) mudar por causa de um desenvolvimento ativo das equipes de infraestrutura, os proprietários dos serviços que executam sobre essa infraestrutura precisarão investir um grande volume de trabalho simplesmente para acompanhar as mudanças na infraestrutura. À medida que as funcionalidades da infraestrutura das quais os serviços dependem se tornam obsoletas e são substituídas por novas funcionalidades, os proprietários dos serviços devem modificar continuamente as suas configurações e reconstruir seus executáveis, consequentemente “correndo apenas para conseguir ficar no mesmo lugar”. A solução para esse cenário é adotar algum tipo de política de redução de mudanças que proíba os engenheiros de infraestrutura de lançar funcionalidades que não sejam compatíveis com versões anteriores, até que automatizem também a migração de seus clientes para a nova funcionalidade. Criar ferramentas automatizadas para migração a fim de acompanhar as novas funcionalidades minimiza o trabalho imposto aos proprietários de serviços para acompanhar as mudanças de infraestrutura.

Resolver esses problemas exige esforços em toda a empresa, que estão muito além do escopo da LCE: uma combinação de APIs melhores de plataforma e frameworks (veja o Capítulo 32), builds contínuas e automação de testes e uma melhor padronização e automação dos serviços em produção do Google.

Conclusão

As empresas que passam por um rápido crescimento, com uma taxa alta de mudanças em produtos e serviços, podem se beneficiar do equivalente à função de Launch Coordination Engineering. Uma equipe como essa será especialmente valiosa se uma empresa planeja duplicar seus desenvolvedores de produtos a cada um ou dois anos, se tiver que escalar seus serviços para centenas de milhões de usuários e se a confiabilidade, apesar de uma alta taxa

de mudanças, for importante para seus usuários.

A equipe de LCE foi a solução do Google para o problema de ter segurança sem impedir mudanças. Este capítulo apresentou algumas das experiências acumuladas pela nossa função única de LCE em um período de dez anos, exatamente nessas circunstâncias. Esperamos que nossa abordagem ajude a inspirar outras pessoas que estejam diante de desafios semelhantes em suas respectivas empresas.

PARTE IV

Gerenciamento

Nossa última seleção de assuntos inclui trabalhar em equipe e trabalhar como equipe. Nenhum SRE é uma ilha, e há algumas maneiras distintas com que trabalhamos.

Qualquer empresa que aspire seriamente a ter um braço eficiente de SRE deve levar o treinamento em consideração. Ensinar os SREs a pensar em um ambiente complicado e de rápidas mudanças, com um programa de treinamento bem planejado e executado, contém a promessa de instilar as melhores práticas nas primeiras semanas ou meses de um novo contratado que, de outro modo, exigiriam meses ou anos para serem acumuladas. Discutiremos estratégias para fazer exatamente isso no Capítulo 28, *Acelerando os SREs para chegar ao plantão e além*.

Como qualquer pessoa no mundo de operações sabe, a responsabilidade por qualquer serviço significativo vem acompanhada de muitas interrupções: a produção ficando em um estado ruim, pessoas solicitando atualizações de seu binário favorito, uma fila longa de requisições de consultoria... administrar interrupções em condições turbulentas é uma habilidade necessária, conforme discutiremos no Capítulo 29, *Lidando com interrupções*.

Se as condições turbulentas persistirem por tempo suficiente, uma equipe de SRE precisa começar a se recuperar da sobrecarga operacional. Temos o plano de ataque correto para você no Capítulo 30, *Incluindo um SRE para se recuperar da sobrecarga operacional*.

Escrevemos no Capítulo 31, *Comunicação e colaboração em SRE*, sobre as diferentes funções na SRE, comunicação entre equipes, localidades e continentes, como conduzir reuniões de produção e estudos de caso sobre como a SRE teve uma boa colaboração.

Por fim, o Capítulo 32, *O modelo de engajamento da SRE em evolução*, analisa uma pedra angular da operação de SRE: a PRR (Production Readiness Review, ou Revisão de Prontidão para Produção) – um passo crucial para implantar um novo serviço. Discutiremos como conduzir PRRs e como ir além desse modelo bem-sucedido, mas também limitado.

Outras leituras da SRE do Google

Desenvolver sistemas confiáveis exige uma mistura bem calibrada de habilidades que variam de desenvolvimento de software à análise de sistemas indiscutivelmente menos conhecida, além de disciplinas de engenharia. Escrevemos sobre as últimas disciplinas em “The Systems Engineering Side of Site Reliability Engineering” (O lado da engenharia de sistemas da Engenharia de Confiabilidade de Sites) [Hix15b].

Fazer boas contratações de SREs é crucial para ter uma organização de confiabilidade com bom funcionamento, conforme explorado em “Hiring Site Reliability Engineers” (Contratando Engenheiros de Confiabilidade de Sites) [Jon15]. As práticas de contratação do Google estão detalhadas em textos como *Work Rules!* [Boc15]¹, mas contratar SREs tem seu próprio conjunto de particularidades. Mesmo pelos padrões gerais do Google, candidatos a SRE são difíceis de encontrar e mais difíceis ainda de entrevistar de modo eficiente.

¹ Escrito por Laszlo Bock, VP sênior do Google para People Operations (Operações de Pessoas).

CAPÍTULO 28

Acelerando os SREs para chegar ao plantão e além

Como posso colocar um propulsor a jato em meus iniciantes ao mesmo tempo que mantendo os SREs mais experientes trabalhando em velocidade máxima?

Escrito por Andrew Widdowson

Editado por Shylaja Nukala

Você contratou seus próximos SREs; e agora?

Você contratou novos funcionários em sua empresa, e eles estão começando como Site Reliability Engineers (Engenheiros de Confiabilidade de Sites). Agora você deve treiná-los para o cargo. Investir previamente na educação e na orientação técnica de novos SREs os transformará em engenheiros melhores. Um treinamento como esse os acelerará até atingirem um estado de proficiência, ao mesmo tempo que deixará seu conjunto de habilidades mais robusto e equilibrado.

Equipes bem-sucedidas de SRE são criadas com base na confiança – para manter um serviço de forma consistente e global, você precisa ter a confiança de que seus colegas de plantão sabem como seu sistema funciona¹, possam diagnosticar comportamentos atípicos do sistema, sintam-se confortáveis em pedir ajuda e possam reagir sob pressão para salvar o dia. É essencial, então, mas não suficiente, pensar na educação dos SREs através das lentes de “O que um iniciante deve aprender para estar de plantão?”. Dados os requisitos relacionados à confiança, você também deve fazer perguntas como estas:

- Como meus engenheiros de plantão atuais avaliam se o iniciante está pronto para estar de plantão?
- Como podemos controlar o entusiasmo e a curiosidade de nossos novos contratados para garantir que os SREs atuais se beneficiarão?

- Com quais atividades posso comprometer a nossa equipe, que trarão vantagem à educação de todos e que todos gostarão?

Os alunos têm uma grande variedade de preferências de aprendizado. Ao reconhecer que você contratará pessoas com uma combinação dessas preferências, oferecer apenas um estilo em detrimento de outros seria ter uma visão muito restrita. Portanto, não há nenhum estilo de educação que funcione melhor para treinar novos SREs, e certamente não há uma fórmula mágica que funcionará para todas as equipes de SRE. A Tabela 28.1 lista práticas recomendadas de treinamento (e seus antipadrões correspondentes), muito conhecidas da SRE no Google. Essas práticas representam uma grande variedade de opções disponíveis para deixar sua equipe bem treinada nos conceitos de SRE, tanto agora como continuamente.

Tabela 28.1 – Práticas para treinamento de SREs

Padrões recomendados	Antipadrões
Conceber experiências de aprendizado concretas e sequenciais para os alunos seguirem	Encher os alunos com tarefas subalternas (por exemplo, triagem de alertas/tickets) para treiná-los; “ensinar jogando no fogo”
Incentivar a engenharia reversa, o raciocínio estatístico e o trabalho a partir dos princípios fundamentais	Treinar estritamente por meio de procedimentos de operador, checklists e manuais
Incentivar a análise de falhas sugerindo a leitura de postmortems pelos alunos	Tratar interrupções de serviço como segredos a serem enterrados para evitar acusações
Criar falhas contidas, porém realistas, para os alunos corrigirem usando monitoração e ferramentas reais	Ter a primeira chance de corrigir algo somente depois que um aluno já estiver de plantão
Encenar desastres teóricos como um grupo para combinar as abordagens de resolução de problemas de uma equipe	Criar experts na equipe cujas técnicas e conhecimentos sejam compartimentados
Permitir que os alunos acompanhem o rodízio de plantão precocemente,	Pressionar os alunos a estar no plantão principal antes de terem uma

comparando suas anotações com o engenheiro de plantão	compreensão holística de seu serviço
Colocar os alunos em pares com SREs especializados para revisar seções específicas do plano de treinamento para o plantão	Tratar os planos de treinamento para plantão como estáticos e intocáveis, exceto pelos experts no assunto
Definir trabalhos de projeto não triviais para os alunos assumirem, permitindo que tenham responsabilidades parciais na pilha	Conceder todos os novos trabalhos de projeto aos SREs mais experientes, deixando os SREs iniciantes com as sobras

O restante deste capítulo apresenta temas importantes que achamos ser eficientes para acelerar os SREs em direção ao plantão e além. Esses conceitos podem ser visualizados em um diagrama para impulsionar os SREs (Figura 28.1).

Essa ilustração captura as melhores práticas que as equipes de SRE podem escolher para ajudar a impulsionar os novos membros, ao mesmo tempo que mantêm os talentos mais experientes atualizados. Entre as muitas ferramentas apresentadas aqui, você pode escolher as atividades que sejam mais adequadas à sua equipe.

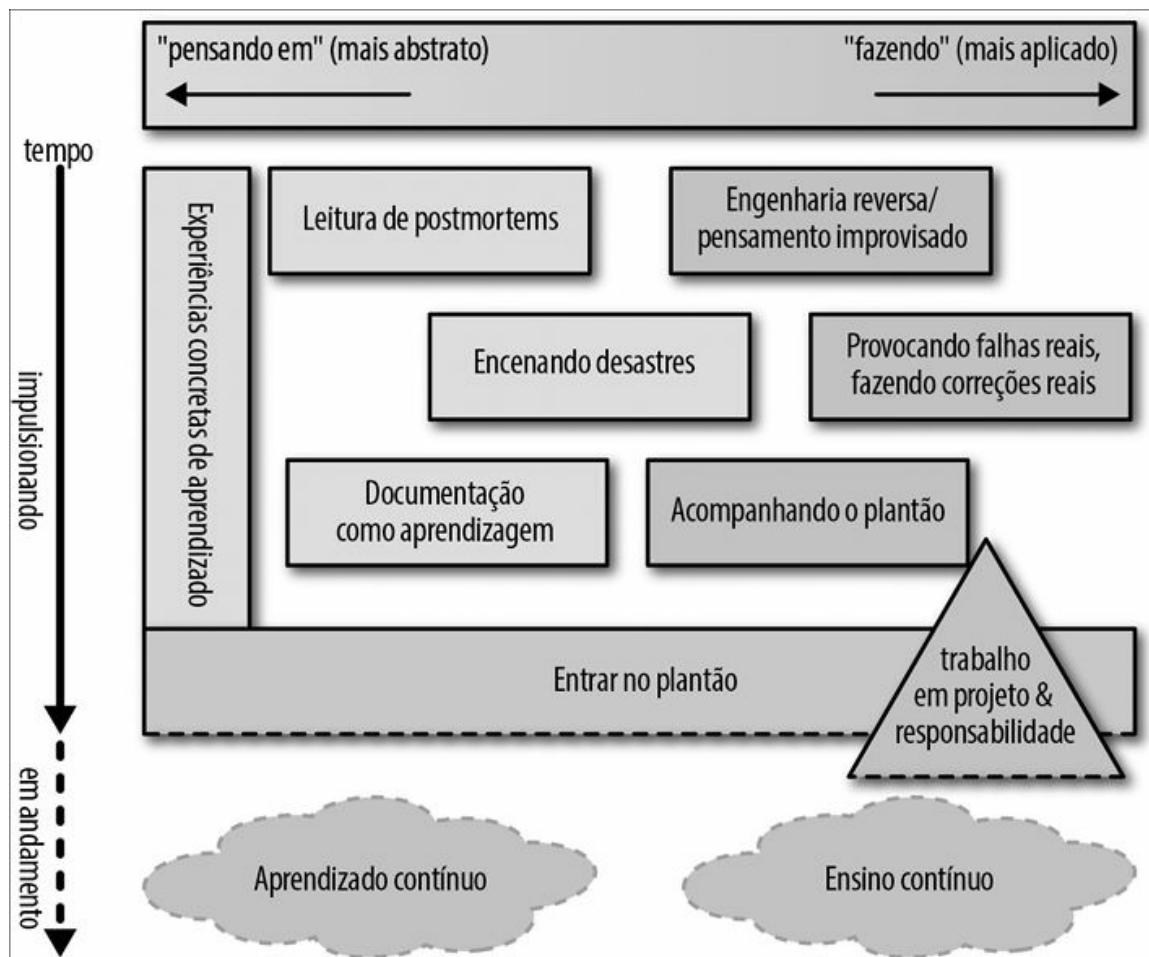


Figura 28.1 – Um diagrama para impulsionar um SRE a fim de chegar ao plantão e ir além.

A ilustração tem dois eixos:

- O eixo x representa o *espectro dos diferentes tipos de trabalho*, que variam de atividades abstratas até as aplicadas.
- O eixo y representa o *tempo*. Lendo de cima para baixo, vemos que os SREs têm muito pouco conhecimento dos sistemas e serviços pelos quais eles serão responsáveis, portanto postmortems que detalhem como esses sistemas falharam no passado são um bom ponto de partida. Novos SREs também podem tentar fazer uma engenharia reversa dos sistemas, partindo dos fundamentos, pois estão começando do zero. Depois que entenderem mais sobre seus sistemas e tiverem feito algum trabalho prático, os SREs estarão prontos para acompanhar o plantão e começar a corrigir documentações incompletas ou desatualizadas.

Dicas para interpretar essa ilustração:

- *Entrar no plantão* é um marco na carreira de um novo SRE; após esse ponto, o aprendizado se torna muito mais nebuloso, indefinido e direcionado por conta própria – daí a linha tracejada em torno das atividades quando um SRE entra no plantão e depois disso.
- A forma triangular de *trabalho em projeto & responsabilidade* indica que o trabalho em projeto começa com um volume pequeno e aumenta com o tempo, tornando-se mais complexo e provavelmente continuando por muito tempo depois de entrar no plantão.
- Algumas dessas atividades e práticas são muito abstratas/passivas, enquanto outras são bastante aplicadas/ativas. Outras atividades misturam os dois tipos. É bom ter uma variedade de modalidades de aprendizagem para que sejam adequadas aos diferentes estilos.
- Para um efeito máximo, as atividades e práticas de treinamento devem ter um ritmo apropriado: algumas são adequadas para serem assumidas de imediato, outras devem ocorrer imediatamente antes de um SRE entrar oficialmente no plantão, e há aquelas que devem ser contínuas e constantes, mesmo para os SREs experientes. *Experiências concretas de aprendizado* devem ocorrer durante o tempo todo até o SRE entrar no plantão.

Experiências iniciais de aprendizado: o caso da estrutura sobre o caos

Conforme discutimos em outro ponto deste livro, as equipes de SRE assumem uma combinação natural de tarefas proativas² e reativas³. Conter e reduzir tarefas reativas sendo amplamente proativo deve ser um objetivo sólido de toda equipe de SRE, e a abordagem que você assumir para engajar seu(s) novo(s) iniciante(s) não deve ser uma exceção. Considere o processo de engajamento a seguir, que é bem comum, porém, infelizmente, não é ideal:

John é o mais novo membro da equipe de SRE do FooServer. Os SREs experientes dessa equipe são responsáveis por muitas tarefas mundanas, como responder a tickets, lidar com alertas e realizar rollouts tediosos de

binários. No primeiro dia de John no cargo, ele recebeu todos os novos tickets de entrada. John foi informado de que pode pedir ajuda a qualquer membro da equipe de SRE a fim de obter o conhecimento prévio necessário para decifrar um ticket. “Certamente, haverá muito que aprender antes”, diz o gerente de John, “mas, em algum momento, cuidar desses tickets será mais rápido. Um dia, bastará *clicar*, e você saberá muito sobre todas as ferramentas que usamos, os procedimentos que seguimos e os sistemas aos quais damos manutenção.” Um membro experiente da equipe comenta: “Estamos jogando você no lado mais fundo da piscina”.

Esse método de “ensinar jogando no fogo” para orientar os iniciantes muitas vezes nasce do ambiente atual de uma equipe; equipes de SRE reativas, orientadas a operações, “treinam” seus membros mais novos fazendo-os... bem, reagir! Repetidamente. Se você estiver com sorte, os engenheiros que já forem bons em navegar por ambiguidades escaparão do buraco em que você os colocar. Porém, as chances são de que essa estratégia aliene vários engenheiros capacitados. Embora uma abordagem como essa possa, em algum momento, gerar ótimos funcionários de operação, seus resultados deixarão a desejar. A abordagem de “ensinar jogando no fogo” também presume que muitos (ou a maioria dos) aspectos de uma equipe podem ser ensinados estritamente fazendo, e não raciocinando. Mesmo que o conjunto de tarefas que uma pessoa encontre na fila de tickets ofereça um treinamento apropriado para o cargo em questão, essa não é a posição da SRE.

Os alunos na SRE terão perguntas como:

- Em que estou trabalhando?
- Quanto progresso já fiz?
- Quando essas atividades me farão acumular experiência suficiente para eu entrar no plantão?

Fazer o salto de uma empresa anterior ou da universidade, ao mesmo tempo que se muda de cargo (do engenheiro de software ou do administrador de sistema tradicionais) para essa função nebulosa de *Site Reliability Engineer* (Engenheiro de Confiabilidade de Sites), com frequência, é suficiente para abater a confiança dos alunos várias vezes. Para personalidades mais

introspectivas (especialmente no que diz respeito às perguntas 2 e 3), as incertezas resultantes de respostas nebulosas ou não muito claras podem levar a um desenvolvimento mais lento ou a problemas de retenção. Em vez disso, considere as abordagens apresentadas nas próximas seções. Essas sugestões são tão concretas quanto qualquer ticket ou alerta, mas também são sequenciais e, desse modo, são muito mais recompensadoras.

Caminhos de aprendizagem cumulativos e organizados

Coloque um pouco de ordem na aprendizagem de seu(s) sistema(s) para que seus novos SREs vejam um caminho diante deles. Qualquer tipo de treinamento é melhor que tickets aleatórios e interrupções, mas faça um esforço consciente para usar a combinação correta entre teoria e aplicação: conceitos abstratos que sejam muito recorrentes na jornada de um iniciante devem vir antes em sua educação, ao mesmo tempo que o aluno também deve receber experiência prática assim que for possível.

Conhecer sua(s) pilha(s) e subsistema(s) exige um ponto de partida. Considere se faz mais sentido agrupar treinamentos por semelhança de propósito ou pela ordem normal de execução. Por exemplo, se sua equipe for responsável por uma pilha de serviços voltada aos usuários em tempo real, considere uma organização curricular como esta:

1) Como uma consulta entra no sistema

Fundamentos de rede e de datacenter, distribuição de carga no frontend, proxies etc.

2) Servir no frontend

Frontend(s) da aplicação, logging de consultas, SLO(s) para a experiência de usuário etc.

3) Serviços intermediários

Caches, distribuição de carga no backend

4) Infraestrutura

Backends, infraestrutura e recursos para processamento

5) Reunindo tudo

Técnicas de depuração, procedimentos para escalar e cenários de emergência

O modo como você opta por apresentar as oportunidades de aprendizagem (conversas informais no quadro-branco, palestras formais ou exercícios de descoberta na prática) cabe a você e aos SREs que estão ajudando a estruturar, projetar e aplicar o treinamento. A equipe de SRE do Google Search estrutura essa aprendizagem por meio de um documento chamado “checklist de aprendizagem para o plantão” (on-call learning checklist). Uma seção simplificada de uma checklist de aprendizagem para o plantão pode ter o aspecto a seguir:

Servidor de Combinação de Resultados (Results Mixing Server, ou “Mixer”)	
Frontend por: Servidor de frontend Backends chamados: Servidor de Obtenção de Resultados (Results Retrieval Server), Servidor de Geolocalização (Geolocation Server), Banco de Dados de Personalização (Personalization Database) Experts na SRE: Sally W, Dave K, Jen P Contatos do desenvolvedor: Jim T, <i>results-team@</i>	Saiba antes de prosseguir: <ul style="list-style-type: none">• Quais clusters têm o Mixer implantado• Como fazer rollback de uma versão do Mixer• Quais backends do Mixer são considerados “caminho crítico” e por quê
Leia e compreenda os seguintes docs: <ul style="list-style-type: none">• Visão Geral do Combinação de Resultados (Results Mixing Overview): Seção “Execução de consultas” (Query execution)• Visão Geral do Combinação de Resultados (Results Mixing Overview): Seção “Produção” (Production)• Manual: Como fazer rollout de uma nova versão do Servidor de Combinação de Resultados (How to Roll Out a New Results Mixing Server)• Uma análise de desempenho do Mixer (A Performance Analysis of Mixer)	Perguntas para verificar a compreensão: <ul style="list-style-type: none">• P: Como o cronograma de novas versões muda se houver um feriado na empresa no dia normal de build da versão?• P: Como você pode corrigir uma atualização de versão ruim do conjunto de dados de geolocalização?

Observe que a seção anterior não apresenta diretamente os procedimentos, os

passos para diagnóstico nem os manuais; em vez disso, é um texto relativamente à prova de futuro, estritamente centrado em enumerar os contatos dos experts no assunto, destacando os recursos de documentação mais úteis, definindo o conhecimento básico que você deve reunir e internalizar e fazendo perguntas de sondagem que só poderão ser respondidas depois que esse conhecimento básico tiver sido absorvido. O documento também oferece resultados concretos para que os alunos saibam quais tipos de conhecimentos e habilidades terão adquirido ao concluir essa seção da checklist de aprendizagem.

É uma boa ideia que todas as partes interessadas tenham uma noção da quantidade de informações que o aluno em treinamento está retendo. Embora esse mecanismo de feedback talvez não precise ser tão formal quanto uma prova, fazer algumas lições de casa que apresentem perguntas sobre como o(s) seu(s) serviço(s) funciona(m) é uma boa prática. Respostas satisfatórias, conferidas por um orientador do aluno, são um sinal de que o aprendizado deve prosseguir para a próxima fase. Perguntas sobre o funcionamento interno de seu serviço podem ser semelhantes a estas:

- Quais backends desse servidor são considerados como “no caminho crítico” e por quê?
- Quais aspectos desse servidor poderiam ser simplificados ou automatizados?
- Em que lugar você acha que está o primeiro gargalo nessa arquitetura? Se esse gargalo se saturasse, quais passos você poderia executar para aliviá-lo?

Conforme o modo como as permissões de acesso estão configuradas em seu serviço, você também pode considerar a implementação de um modelo de acesso em camadas. A primeira camada de acesso permitiria que o aluno tivesse acesso somente de leitura ao funcionamento interno dos componentes, e uma última camada permitiria que eles modificassem o estado em produção. Concluir satisfatoriamente as seções da checklist de aprendizagem para o plantão concederia o direito ao aluno de ter um acesso cada vez mais profundo ao sistema. A equipe de SRE do Search chama esses níveis atingidos de “powerups”⁴ no caminho para o plantão, à medida que os alunos

em treinamento são, em algum momento, adicionados ao nível mais alto de acesso aos sistemas.

Trabalho específico em projeto, e não tarefas braçais

Os SREs são pessoas com habilidade para resolver problemas, portanto dê-lhes um bom problema para resolver! Ao começar, ter um mínimo senso de responsabilidade no serviço da equipe pode fazer maravilhas no aprendizado. No sentido inverso, uma responsabilidade como essa também pode trazer muitos progressos no desenvolvimento da confiança entre os colegas mais experientes, pois eles abordarão seu colega iniciante para conhecer o(s) novo(s) componente(s) ou processo(s). Oportunidades para assumir logo uma responsabilidade são um padrão do Google em geral: todos os engenheiros recebem um projeto para iniciantes cujo propósito é oferecer um tour pela infraestrutura que seja suficiente para que possam fazer uma pequena, mas útil, contribuição mais cedo. Fazer o novo SRE dividir seu tempo entre aprendizagem *e* trabalho em projeto também lhe dará um senso de propósito e de produtividade, que não existiria se ele gastasse seu tempo apenas aprendendo *ou* apenas trabalhando em projetos. Vários padrões de projetos para iniciantes, que parecem funcionar bem, incluem:

- Fazer a mudança de uma funcionalidade trivial visível ao usuário em uma pilha servidora e, subsequentemente, conduzir o lançamento da funcionalidade por todo o caminho, até a produção. Entender tanto a cadeia de ferramentas de desenvolvimento quanto o processo de lançamento de um binário incentiva a empatia entre os desenvolvedores.
- Acrescentar monitoração ao seu serviço nos locais em que há pontos cegos no momento. O iniciante deverá pensar usando a lógica de monitoração, ao mesmo tempo que combina sua compreensão de um sistema com o modo como ele realmente se comporta (mal).
- Automatizar um ponto sofrível, que não seja sofrível o bastante a ponto de já ter sido automatizado, oferecendo ao novo SRE uma apreciação do valor que os SREs atribuem à remoção de tarefas penosas (*toil*) de nossas operações cotidianas.

Criando profissionais espetaculares em engenharia reversa e em raciocínio de improviso

Podemos propor um conjunto de diretrizes para *como* treinar novos SREs, mas *em que* devemos treiná-los? Os materiais de treinamento dependerão das tecnologias usadas no trabalho, mas a pergunta mais importante é: que tipo de engenheiros estamos tentando criar? Na escala e na complexidade com que operam, os SREs não podem se dar ao luxo de serem administradores de sistema tradicionais, simplesmente focados em operações. Além de ter uma mentalidade de engenharia em larga escala, os SREs devem exibir as seguintes características:

- No curso de suas tarefas, eles se depararão com sistemas que nunca viram antes; portanto, precisam ter *habilidades robustas de engenharia reversa*.
- Em escala, haverá anomalias difíceis de detectar; portanto, precisarão ter a habilidade de *pensar estatisticamente*, e não de forma procedural, para desvendar os problemas.
- Quando procedimentos de operação padrões falharem, eles deverão ser capazes de *improvisar totalmente*.

Vamos analisar melhor esses atributos, para que possamos entender como equipar nossos SREs para que tenham essas habilidades e comportamentos.

Engenharia reversa: descobrindo como os sistemas funcionam

Os engenheiros são curiosos sobre o modo como sistemas que eles nunca viram antes funcionam – ou, mais provavelmente, como as versões atuais dos sistemas que eles costumavam conhecer muito bem funcionam. Ao ter uma compreensão básica de como os sistemas funcionam em sua empresa, juntamente com uma disposição para explorar profundamente as ferramentas de depuração, as fronteiras das RPCs e os logs de seus binários para desvendar seus fluxos, os SREs se tornarão mais eficientes para resolver problemas inesperados em arquiteturas de sistema inesperadas. Ensine seus SREs acerca das superfícies de diagnóstico e de depuração de suas aplicações e faça com que eles exercitem fazer inferências a partir das informações reveladas por essas superfícies, de modo que esse comportamento se torne

um reflexo ao lidar com futuras interrupções de serviço.

Pensar de modo estatístico e comparativo: guardiões do método científico sob pressão

Você pode pensar na abordagem de um SRE para uma resposta a um incidente em sistemas de larga escala como navegar por uma gigantesca árvore de decisão que se desdobre diante dele. Na janela de tempo limitada que as exigências da resposta ao incidente permitem, o SRE pode executar algumas ações entre centenas delas, com o objetivo de atenuar a interrupção de serviço, no curto ou no longo prazo. Como o tempo muitas vezes é de extrema importância, o SRE precisa, de maneira eficaz e eficiente, podar sua árvore de decisão. A habilidade de fazer isso é parcialmente adquirida por meio de experiência, que só vem com o tempo e a exposição a uma variedade de sistemas de produção. Essa experiência deve ser combinada com uma construção cuidadosa de hipóteses que, quando comprovadas ou descartadas, restrinjam mais esse espaço de decisão. Falando de outra maneira, rastrear falhas de sistema muitas vezes é como um jogo de “qual dessas coisas não é como as outras?”, em que “coisas” podem implicar versão de kernel, arquitetura de CPU, versão(ões) de binários em sua pilha, combinação de tráfego regional ou uma centena de outros fatores. Do ponto de vista da arquitetura, é responsabilidade da equipe garantir que todos esses fatores possam ser controlados, analisados individualmente e comparados. No entanto, devemos também treinar nossos novos SREs para que se tornem bons analistas e comparadores desde muito cedo em seu cargo.

Artistas do improviso: quando o inesperado acontece

Você testa uma correção para a falha, mas ela não funciona. O(s) desenvolvedor(es) por trás do sistema em falha não é(são) encontrado(s) em lugar nenhum. O que você deve fazer agora? Improvise! Conhecer várias ferramentas que possam resolver partes de seu problema permite que você ponha em prática a defesa em profundidade em seus próprios comportamentos para resolução de problemas. Ser demasiadamente procedural diante de uma interrupção de serviço, esquecendo, assim, suas habilidades analíticas, pode ser a diferença entre ficar sem saber o que fazer e

identificar a causa-raiz. Um caso de resolução de problemas complicado pode piorar se um SRE fizer muitas suposições não testadas sobre a causa de uma interrupção de serviço em sua tomada de decisão. Mostrar que há muitas armadilhas analíticas em que os SREs podem cair, o que exige “diminuir o zoom” e adotar uma abordagem diferente para a resolução, é uma lição valiosa que os SREs devem aprender desde cedo.

Dados esses três atributos desejados para SREs de alto desempenho, quais cursos e experiências podemos oferecer aos novos SREs para enviá-los a um caminho na direção correta? Você deve definir o conteúdo de seu próprio curso de modo a incorporar esses atributos, além dos demais atributos específicos à sua cultura de SRE. Vamos considerar um curso que acreditamos atingir todos os pontos mencionados anteriormente.

Reunindo tudo: engenharia reversa de um serviço em produção

“Quando chegou a hora de conhecer [parte da pilha do Google Maps], [uma nova SRE] perguntou se, em vez de ter alguém explicando passivamente o serviço, ela poderia fazer isso por conta própria – aprender tudo por meio das técnicas das aulas de Engenharia Reversa, e o restante de nós a corrigiria/preencheria as lacunas para o que quer que ela não entendesse ou entendesse de forma incorreta. O resultado? Bem, foi provavelmente mais correto e útil do que teria sido se *eu* tivesse dado a explicação, e eu estou de plantão nesse serviço há mais de cinco anos!”

— Paul Cowan, *SRE do Google*

Um curso popular que oferecemos no Google se chama “Reverse Engineering a Production Service (without help from its owners)” (Engenharia reversa de um serviço em produção [sem a ajuda de seus proprietários]). O cenário do problema apresentado parece simples à primeira vista. A equipe toda do Google News – SREs, engenheiros de software, gerentes de produto e assim por diante – fez uma viagem pela empresa: um cruzeiro pelo Triângulo das Bermudas. Não temos notícia da equipe há 30 dias, portanto nossos alunos são a nova equipe de SRE designada para o Google News. Eles precisam descobrir como a pilha servidora funciona de

ponta a ponta para assumir seu controle e mantê-la executando.

Após ter esse cenário apresentado, os alunos são conduzidos por exercícios interativos, orientados a propósitos, em que traçam o caminho de passagem da consulta, partindo de seu navegador web e fluindo pela infraestrutura do Google. Em cada etapa do processo, enfatizamos que é importante conhecer várias maneiras de descobrir a conectividade entre os servidores de produção para que conexões não sejam perdidas. No meio da aula, desafiamos os alunos a encontrar outro endpoint para o tráfego de entrada, mostrando que nossa suposição inicial tinha um escopo restrito demais. Então, desafiamos nossos alunos a encontrar outros caminhos na pilha. Exploramos a natureza altamente instrumentada de nossos binários de produção, que informam sua conectividade com RPCs, assim como nossas monitorações caixa-branca e caixa-preta disponíveis, a fim de determinar o(s) caminho(s) pelo(s) qual(is) as consultas dos usuários passam.⁵ Nesse processo, montamos um diagrama do sistema, além de discutir os componentes que fazem parte da infraestrutura compartilhada que nossos alunos provavelmente verão de novo no futuro.

No final do curso, os alunos recebem uma tarefa. Cada aluno volta para sua equipe original e pede a um SRE experiente para ajudá-lo a selecionar uma pilha ou uma parte de uma pilha para a qual ele(a) estará de plantão. Usando as habilidades adquiridas nas aulas, o aluno então fará o diagrama dessa pilha sozinho e apresentará suas descobertas ao SRE experiente. Sem dúvida, o aluno perderá alguns detalhes sutis, o que dará elementos para uma boa discussão. Também é provável que o SRE experiente vá aprender algo com o exercício também, expondo lacunas em seu entendimento prévio do sistema em constante mudança. Por causa da rápida mudança nos sistemas de produção, é importante que sua equipe receba de braços abertos qualquer chance de se familiarizar novamente com um sistema, incluindo aprender com os membros mais novos da equipe, em vez de fazê-lo com os mais velhos.

Cinco práticas para os engenheiros que aspiram ao plantão

Estar de plantão não é o único propósito mais importante de qualquer SRE, mas as responsabilidades da engenharia de produção geralmente envolvem algum tipo de tratamento de notificação urgente. Uma pessoa que seja capaz de assumir um plantão com responsabilidade é alguém que entende o sistema em que trabalha com uma profundidade e amplitude razoáveis. Portanto, usaremos “capaz de assumir um plantão” como uma aproximação conveniente para “sabe o bastante e é capaz de descobrir o resto”.

Fome de falhas: lendo e compartilhando postmortems

“Aqueles que não conseguem se lembrar do passado estão condenados a repeti-lo.”

– George Santayana, filósofo e ensaísta

Os postmortems (veja o Capítulo 15) são uma parte importante da melhoria contínua. São uma maneira de chegar às muitas causas-raízes de uma interrupção de serviço significativa ou visível, sem fazer acusações. Ao escrever um postmortem, tenha em mente que o público-alvo que mais o apreciará pode ser um engenheiro que ainda não foi contratado. Sem uma edição radical, mudanças sutis podem ser feitas em nossos melhores postmortems para transformá-los em postmortems “ensináveis”.

Mesmo os melhores postmortems não serão úteis se ficarem abandonados no fundo de um armário de arquivos virtuais. Desse modo, sua equipe deve reunir e cuidar de postmortems valiosos para servirem como recursos educacionais para futuros iniciantes. Alguns postmortems são repetitivos, mas “postmortems ensináveis”, que ofereçam insights para falhas estruturais ou novas em sistemas de larga escala, valem ouro para os novos alunos.

A responsabilidade pelos postmortems não está limitada apenas à autoria. Para muitas equipes, é uma questão de orgulho ter sobrevivido e documentado suas maiores interrupções de serviço. Reúna seus melhores postmortems e deixe-os visivelmente disponíveis para os iniciantes lerem – além de mantê-los disponíveis às partes interessadas em equipes relacionadas e/ou que se integram à sua equipe. Peça para as equipes relacionadas que publiquem seus melhores postmortems em um local em que você possa acessá-los.

Algumas equipes de SRE no Google mantêm “clubes de leitura de postmortems”, em que postmortems fascinantes, que proporcionam muitos insights, circulam, são previamente lidos e então discutidos. O(s) autor(es) original(is) do postmortem pode(m) ser o(s) convidado(s) de honra da reunião. Outras equipes organizam reuniões de “histórias de falhas”, em que o(s) autor(es) do(s) postmortem(s) faz(em) uma apresentação semiformal, narrando a interrupção do serviço e conduzindo, ele(s) mesmo(s), a discussão, de modo eficiente.

Leituras regulares ou apresentações de interrupções de serviço, incluindo as condições de disparo e os passos para atenuação, fazem maravilhas para que um novo SRE crie um mapa mental e entenda a produção e a resposta do plantão. Os postmortems também são um combustível excelente para futuros cenários abstratos de desastres.

Interpretando papéis em situações de desastre

“Uma vez por semana, temos uma reunião em que uma vítima é escolhida para estar no centro das atenções do grupo, e um cenário – geralmente, um cenário real, obtido dos anais da história do Google – é apresentado a ela. A vítima – penso nela como a participante de um show em que há concorrentes – diz ao apresentador do show o que faria ou a quem consultaria para entender ou resolver o problema, e o apresentador diz à vítima o que acontece com cada ação ou observação. É como um *SRE Zork*. Você está em um labirinto de consoles de monitoração emaranhados, todos parecidos. Você deve salvar usuários inocentes de escorregar no Abismo da Latência Excessiva de Consultas, salvar datacenters de uma Desestruturação Quase Certa e nos livrar do constrangimento da Exibição Errônea do Google Doodle.”

– Robert Kennedy, ex-SRE do Google Search e healthcare.gov⁶

Quando você tem um grupo de SREs com níveis de experiência extremamente diferentes, o que pode fazer para reuni-los e permitir que aprendam uns com os outros? Como você imprime a cultura de SRE e a natureza solucionadora de problemas de sua equipe em um iniciante, ao mesmo tempo que também mantém os veteranos grisalhos a par das novas mudanças e funcionalidades em sua pilha? As equipes de SRE do Google

abordam esses desafios por meio de uma antiga tradição de encenar desastres regularmente. Entre outros nomes, esse exercício é comumente chamado de “Wheel of Misfortune” (Roda do Azar) ou “Walk the Plank” (Ande pela Prancha). O senso de perigo com humor nesses títulos deixa os exercícios menos assustadores aos SREs recém-contratados.

Em seu melhor aspecto, esses exercícios se tornam um ritual semanal em que todo membro do grupo aprende algo. A fórmula é simples e apresenta alguma semelhança com um tabuleiro de RPG (Role Playing Game): o GM (“game master”, ou mestre do jogo) escolhe dois membros da equipe para servirem de plantão principal e secundário; esses dois SREs se juntam ao GM na frente da sala. Um page de entrada é anunciado e a equipe de plantão responde com o que ela faria para atenuar e investigar a interrupção de serviço.

O GM prepara cuidadosamente um cenário que está prestes a se desdobrar. Esse cenário pode estar baseado em uma interrupção de serviço anterior em que os membros mais novos da equipe não estavam presentes ou da qual os membros mais antigos já tenham se esquecido. Ou, quem sabe, o cenário é uma incursão em uma falha hipotética de uma funcionalidade nova ou que está prestes a ser lançada na pilha, fazendo com que todos os membros da sala estejam igualmente despreparados para lidar com a situação. Melhor ainda, um colega de trabalho pode encontrar uma nova falha que seja uma novidade em produção, e o cenário de hoje é expandido com base nessa nova ameaça.

Nos próximos 30 a 60 minutos, os engenheiros no plantão principal e secundário tentam determinar a causa-raiz do problema. O GM alegremente fornece um contexto adicional à medida que o problema se desdobra, talvez descrevendo aos engenheiros de plantão (e sua audiência) a aparência dos gráficos em seus painéis de monitoração durante a interrupção do serviço. Se o incidente exigir que o problema seja escalado para fora da equipe original, o GM finge ser um membro dessa outra equipe no caso desse cenário. Nenhum cenário virtual será perfeito, portanto, às vezes, o GM talvez precise direcionar os participantes de volta ao caminho, distanciando-os de pistas falsas, introduzindo urgência e clareza ao acrescentar outros estímulos⁷ ou fazendo perguntas urgentes e específicas.⁸

Quando seu RPG de desastre for bem-sucedido, todos terão aprendido algo: talvez uma nova ferramenta ou truque, um ponto de vista diferente sobre como resolver um problema ou (especialmente gratificante para os novos membros da equipe) uma confirmação de que você poderia ter resolvido o problema dessa semana caso tivesse sido escolhido. Com um pouco de sorte, esse exercício inspirará os colegas de equipe a esperar ansiosamente pela aventura da próxima semana ou pedir que sejam o mestre do jogo em alguma semana no futuro.

Provoque falhas reais, faça correções reais

Um iniciante pode aprender muito sobre SRE lendo documentação, postmortems e fazendo treinamentos. Encenar desastres pode ajudar a fazer a mente de um iniciante entrar no jogo. No entanto, a experiência adquirida por meio de experiência prática provocando falhas e/ou corrigindo sistemas *reais* de produção é melhor ainda. Haverá muito tempo para uma experiência prática, depois que um iniciante entrar no plantão, mas um aprendizado desse tipo deve ocorrer *antes* que um novo SRE alcance esse ponto. Assim, proporcione experiências práticas como essa bem cedo a fim de desenvolver as respostas reflexivas do aluno no uso das ferramentas e monitoração de sua empresa para abordar uma interrupção de serviço em desenvolvimento.

O realismo é fundamental nessas interações. O ideal é que sua equipe tenha uma pilha hospedada em vários lugares e provisionada de modo que você tenha pelo menos uma instância que possa ser desviada do tráfego ativo e emprestada temporariamente para um exercício de aprendizagem. De forma alternativa, você pode ter uma instância de teste ou de QA (Quality Assurance) de sua pilha, pequena, porém totalmente equipada, que possa ser emprestada por um curto período de tempo. Se for possível, aplique uma carga sintética à pilha, que seja próxima do tráfego real de usuários/clientes, além de consumo de recursos.

As oportunidades de aprendizagem com um sistema real em produção com carga sintética são abundantes. SREs experientes terão passado por todos os tipos de problema: erros de configuração, vazamento de memória (memory leaks), regressões de desempenho, consultas com falhas, gargalos em

armazenagem e assim por diante. Nesse ambiente realista, porém relativamente livre de riscos, os instrutores podem manipular o conjunto de jobs de maneiras que alterem o comportamento da pilha, forçando os novos SREs a perceber diferenças, determinar fatores que colaboram com a situação e, em última instância, corrigir sistemas para restaurar o comportamento apropriado.

Como alternativa ao overhead de pedir que um SRE experiente planeje cuidadosamente um tipo específico de falha que o(s) novo(s) SRE(s) deva(m) corrigir, 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 configuração boa conhecida e, lentamente, debilite a pilha em gargalos selecionados, observando os esforços de upstream e de downstream com sua monitoração. A equipe de SRE do Google Search valoriza esse exercício e o chama de “Vamos derrubar totalmente um cluster de pesquisa!”. O exercício é conduzido da seguinte maneira:

1. Como um grupo, discutimos quais características de desempenho observáveis podem mudar à medida que debilitamos a pilha.
2. Antes de infligir os danos planejados, fazemos uma enquete entre os participantes a fim de saber seus palpites e raciocínios para suas previsões sobre como o sistema reagirá.
3. Validamos as suposições e justificamos o raciocínio por trás dos comportamentos que virmos.

Esse exercício, que realizamos trimestralmente, revela novos bugs que corrigimos ansiosamente, pois nossos sistemas nem sempre se degradam de modo elegante, conforme esperado.

Documentação como aprendizado

Muitas equipes de SRE mantêm uma “checklist de aprendizagem para plantão”, que é uma lista organizada de leitura e compreensão das tecnologias e conceitos relevantes ao(s) sistema(s) mantido(s) por elas. Essa lista deve ser internalizada por um aluno antes que ele possa ser considerado apto a acompanhar um engenheiro de plantão. Reserve um momento para rever a checklist de aprendizagem para plantão usada como exemplo na Tabela 28.1.

A checklist de aprendizagem tem propósitos distintos para pessoas diferentes:

- Para o aluno:
 - Esse documento ajuda a definir as fronteiras do sistema ao qual sua equipe dará suporte.
 - Ao estudar essa lista, o aluno terá uma noção de quais sistemas são mais importantes e por quê. Quando compreenderem as informações contidas nesse documento, os alunos poderão passar para outros assuntos que devam conhecer, em vez de continuar aprendendo detalhes excepcionais que poderão ser vistos com o tempo.
- Para os orientadores e gerentes: O progresso dos alunos na checklist de aprendizagem pode ser observado. A checklist responde a perguntas como:
 - Em quais seções você está trabalhando hoje?
 - Quais seções são as mais confusas?
- Para todos os membros da equipe: O documento se transforma em um contrato social pelo qual (ao ser dominado) o aluno se juntará às fileiras de plantão. A checklist de aprendizagem define o padrão a que todos os membros da equipe devem aspirar e se manter fiéis.

Em um ambiente em constantes mudanças, a documentação pode ficar rapidamente desatualizada. Uma documentação desatualizada é menos problemática para SREs mais experientes, que já estão preparados, pois eles guardam o estado do sistema e suas mudanças em suas próprias cabeças. SREs iniciantes precisam muito mais de documentação atualizada, mas podem achar que não têm poder nem conhecimento suficientes para fazer alterações. Quando projetada com a quantidade correta de estrutura, a documentação de plantão pode se transformar em um corpo de trabalho adaptável, que controlará o entusiasmo dos iniciantes e o conhecimento dos mais experientes para que todos permaneçam atualizados.

Na SRE de Search, preparamos a chegada de novos membros da equipe revendo nossa checklist de aprendizagem para o plantão e organizando suas seções conforme estiverem mais atualizadas. Quando os novos membros da equipe chegam, apresentamos a checklist geral de aprendizagem, mas

também lhes atribuímos a tarefa de revisar uma ou mais seções desatualizadas. Como podemos ver na Tabela 28.1, nomeamos o SRE experiente e os contatos do desenvolvedor para cada tecnologia. Incentivamos o aluno a fazer logo uma conexão com os experts nesses assuntos para que eles possam conhecer diretamente o funcionamento interno da tecnologia selecionada. Mais tarde, à medida que tenham mais familiaridade com o escopo e o tom da checklist de aprendizagem, espera-se que os SREs iniciantes contribuam com uma seção revisada da checklist, que deve ser revista por um ou mais SREs experientes listados como experts.

Acompanhando o plantão com antecedência e frequência

Em última instância, nenhuma quantidade de exercícios hipotéticos de desastre ou outros mecanismos de treinamento prepararão completamente um SRE para entrar no plantão. No final das contas, enfrentar interrupções de serviço reais sempre será mais benéfico do ponto de vista de aprendizagem do que engajar-se em situações hipotéticas. Apesar disso, não é justo fazer com que os iniciantes esperem até seu primeiro page real para terem a chance de aprender e reter conhecimento.

Depois que o aluno tiver passado por todos os fundamentos do sistema (concluindo, por exemplo, uma checklist de aprendizagem para plantão), considere configurar seu sistema de alertas para copiar pages de entrada ao iniciante, apenas em horário comercial no início. Conte com sua curiosidade para abrir caminho. Esses turnos de plantão com “sombra” são uma ótima maneira de um orientador adquirir visibilidade quanto ao progresso de um aluno e um aluno ter visibilidade das responsabilidades de estar no plantão. Ao se organizar de modo que o iniciante acompanhe vários membros de sua equipe, essa equipe se sentirá cada vez mais à vontade com a ideia de essa pessoa entrar no rodízio de plantão. Instilar confiança desse modo é um método eficaz de desenvolvê-la, permitindo que os membros mais experientes se afastem quando não estiverem de plantão, ajudando assim a evitar que a equipe se estresse.

Quando um page chegar, o novo SRE não será o engenheiro de plantão designado – uma condição que elimina qualquer pressão de tempo para o

aluno. Os novos SREs terão agora um assento na primeira fila da interrupção de serviço enquanto ela se desdobra, e não depois que o problema foi resolvido. Talvez o aluno e o engenheiro de plantão principal compartilhem uma sessão de terminal ou se sentem um ao lado do outro para comparar anotações. Em um momento que seja conveniente para ambos, após a interrupção do serviço ter acabado, o engenheiro de plantão poderá rever o raciocínio e os processos seguidos para auxiliar o aluno. Esse exercício aumentará a retenção do aluno que acompanha o engenheiro de plantão acerca do que realmente ocorreu.



Se uma interrupção de serviço para a qual escrever um postmortem seja benéfico ocorrer, o engenheiro de plantão deverá incluir o iniciante como coautor. *Não jogue a responsabilidade de escrever o postmortem somente sobre o aluno, pois ele poderá aprender erroneamente que os postmortems são uma espécie de trabalho subalterno, a ser passado para os menos experientes. Seria um erro causar essa impressão.*

Algumas equipes também incluirão um passo final: fazer o engenheiro de plantão experiente ser uma “sombra reversa” do aluno. O iniciante será o engenheiro de plantão principal e será responsável por tudo que for escalado, mas o engenheiro de plantão experiente ficará nas sombras, diagnosticando a situação de forma independente, sem modificar qualquer estado. O SRE experiente estará disponível para oferecer um suporte ativo, ajuda, validação e dicas, conforme forem necessários.

De plantão e além: ritos de passagem e colocando a educação contínua em prática

À medida que a compreensão aumenta, o aluno alcançará um ponto em sua carreira em que será capaz de raciocinar na maior parte da pilha de forma confortável e poderá improvisar nas partes restantes. Nesse ponto, o aluno deverá entrar no plantão de seu serviço. Algumas equipes criam um tipo de exame final que testa seus alunos uma última vez antes de lhes conceder poderes e responsabilidades de plantão. Outros novos SREs submeterão sua conclusão da checklist de aprendizagem para plantão como evidência de que estão prontos. Independentemente de como você controla esse marco, estar de plantão é um rito de passagem e deve ser comemorado como uma equipe.

O aprendizado termina quando um aluno se junta às fileiras do plantão? É claro que não! Para permanecerem vigilantes como SREs, os membros de sua equipe sempre precisarão estar ativos e cientes das mudanças que estão por vir. Enquanto sua atenção estiver em outro lugar, partes de sua pilha podem ganhar uma nova arquitetura e ser estendidas, fazendo com que o conhecimento operacional de sua equipe, no melhor caso, se transforme em história.

Defina uma série regular de aprendizagem para toda a sua equipe, em que apresentações gerais de mudanças novas e que estão por vir em sua pilha sejam feitas pelos SREs que estão conduzindo as mudanças, em conjunto com os desenvolvedores, conforme for necessário. Se puder, grave as apresentações para que você possa montar uma biblioteca de treinamento para futuros alunos.

Com um pouco de prática, você terá muitos envolvimentos oportunos, tanto de SREs em sua equipe quanto de desenvolvedores que trabalhem próximos e ela – tudo isso enquanto mantém as mentes de todos atualizadas em relação ao futuro. Há outros lugares para engajamento educacional também: considere fazer os SREs darem palestras às suas contrapartidas na equipe de desenvolvimento. Quanto melhor seus colegas em desenvolvimento entenderem seu trabalho e os desafios enfrentados pela sua equipe, mais fácil será tomar decisões que sejam totalmente fundamentadas em projetos futuros.

Considerações finais

Um investimento prévio em treinamento de SREs vale absolutamente a pena, tanto para os alunos ansiosos por dominar seu ambiente de produção quanto para as equipes gratas por darem boas-vindas aos alunos às fileiras de plantão. Por meio das práticas aplicáveis apresentadas neste capítulo, você criará SREs bem preparados mais rapidamente, ao mesmo tempo que aperfeiçoará as habilidades da equipe de forma contínua. Cabe a você definir o modo de aplicar essas práticas, mas a mudança é clara: como SRE, você deve escalar seus seres humanos mais rapidamente do que escala suas máquinas. Boa sorte para você e para as suas equipes ao criar uma cultura de aprendizagem e ensino!

-
- 1 E não funciona!
 - 2 Exemplos de tarefas proativas de SRE incluem automação de software, consultoria em design e coordenação de lançamentos.
 - 3 Exemplos de tarefas reativas de SRE incluem depurar, resolver problemas e tratar a necessidade de escalar no plantão.
 - 4 Um aceno de cabeça aos videogames do passado.
 - 5 Essa abordagem de “seguir a RPC” também funciona bem para sistemas batch/pipeline; comece pela operação que dispara o sistema. Para sistemas batch, essa operação pode ser a chegada de dados que devam ser processados, uma transação que deva ser validada ou vários outros eventos.
 - 6 Veja “Life in the Trenches of healthcare.gov” (A vida nas trincheiras do healthcare.gov, <http://www.thedotpost.com/2014/05/robert-kennedy-life-in-the-trenches-of-healthcare-gov>).
 - 7 Por exemplo: “Você tem seu pager acionado por outra equipe que lhe traz mais informações. Eis o que eles dizem...”.
 - 8 Por exemplo: “Estamos perdendo dinheiro rapidamente! Como você poderia estancar o sangramento no curto prazo?”.

CAPÍTULO 29

Lidando com interrupções

Escrito por Dave O'Connor

Editado por Diane Bates

“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 é dono de um carro, você (ou alguém a quem você pague) sempre acabará cuidando dele, colocando combustível ou fazendo outras manutenções regulares para que ele continue realizando sua função.

Qualquer sistema complexo é tão imperfeito quanto seus criadores. Ao administrar a carga operacional gerada por esses sistemas, lembre-se de que seus criadores também são máquinas imperfeitas.

A *carga operacional*, quando aplicada à administração de sistemas complexos, assume muitas formas, algumas mais óbvias do que outras. A terminologia pode mudar, mas a carga operacional se enquadra em três categorias gerais: pages, tickets e atividades operacionais contínuas.

Os *pages* dizem respeito a alertas de produção e seus efeitos, e são acionados em resposta a emergências em produção. Às vezes, eles podem ser monótonos e recorrentes, exigindo pouco raciocínio. Também podem ser engajadores e envolver um raciocínio tático em profundidade. Os pages sempre têm um tempo de resposta esperado (SLO) que, às vezes, é medido em minutos.

Os *tickets* dizem respeito às solicitações do cliente que exigem que você tome alguma atitude. Assim como os pages, os tickets podem ser simples e tediosos ou exigir um verdadeiro raciocínio. Um ticket simples pode exigir uma revisão de código para uma configuração pela qual a equipe seja responsável. Um ticket mais complexo pode implicar uma requisição especial

ou incomum de ajuda com um design ou um plano de capacidade. Os tickets também podem ter um SLO, mas é mais provável que o tempo de resposta seja medido em horas, dias ou semanas.

Responsabilidades operacionais contínuas (também conhecidas como “empurrando com a barriga” e “tarefas penosas” [toil]; veja o Capítulo 5) incluem atividades como códigos sob responsabilidade da equipe, rollouts de flag ou respostas a perguntas *ad hoc* de clientes, com prazos críticos. Embora possam não ter um SLO definido, essas tarefas podem interromper você.

Alguns tipos de carga operacional são facilmente previstos ou planejados, mas boa parte da carga não é, ou pode interromper alguém a qualquer momento, exigindo que essa pessoa determine se o problema pode esperar.

Administrando a carga operacional

O Google tem vários métodos para administrar cada tipo de carga operacional no nível da equipe.

Os *pages* são mais comumente administrados por um engenheiro dedicado no plantão principal. Uma única pessoa responde aos pages e administra os incidentes resultantes ou as interrupções de serviço. O engenheiro de plantão principal também pode administrar comunicações com o suporte a usuários, escalar para desenvolvedores de produto, e assim por diante. Para minimizar a interrupção causada por um page em uma equipe e evitar o efeito de permanecer como espectador, os turnos de plantão do Google são constituídos por um único engenheiro. O engenheiro de plantão pode escalar pages para outro membro da equipe se um problema não for bem compreendido.

Geralmente, um engenheiro de plantão secundário atua como backup do engenheiro principal. As responsabilidades do engenheiro secundário variam. Em alguns rodízios, a única obrigação do secundário é entrar em contato com o primário se pages chegarem até ele como alternativa. Nesse caso, o secundário pode estar em outra equipe. O engenheiro secundário pode ou não se considerar *em interrupções*, conforme suas obrigações.

Os *tickets* são administrados de algumas maneiras diferentes, dependendo da

equipe de SRE: um engenheiro de plantão primário pode trabalhar em tickets enquanto estiver de plantão, um engenheiro secundário pode trabalhar em tickets enquanto estiver de plantão ou uma equipe pode ter uma pessoa dedicada a tickets, mas que *não* esteja de plantão. Os tickets podem ser aleatoriamente distribuídos entre os membros da equipe, ou pode-se esperar que eles os tratem *ad hoc*.

As *responsabilidades operacionais contínuas* também são administradas de formas variadas. Às vezes, o engenheiro de plantão faz o trabalho (atualização de versões, mudança de flags etc.). De modo alternativo, cada responsabilidade pode ser atribuída aos membros da equipe *ad hoc*, ou um engenheiro de plantão pode escolher uma responsabilidade duradoura (isto é, um rollout ou ticket que ocupe várias semanas) que vá além de seu turno semanal.

Fatores que determinam como as interrupções são tratadas

Dando um passo para trás em relação ao modo como a carga operacional é administrada, há uma série de métricas que influenciam como cada uma dessas interrupções é tratada. Algumas equipes de SRE no Google perceberam que as métricas a seguir são úteis para decidir a forma de administrar as interrupções:

- SLO para interrupções ou tempo de resposta esperado
- O número de interrupções geralmente registrado
- A severidade das interrupções
- A frequência das interrupções
- O número de pessoas disponível para tratar um determinado tipo de interrupção (por exemplo, algumas equipes exigem um determinado volume de trabalho em tickets antes que um engenheiro entre no plantão)

Você pode ter percebido que todas essas métricas são adequadas para atender ao menor tempo de resposta possível, sem incorrer em mais custos humanos. Tentar fazer uma avaliação do custo humano e de produtividade é difícil.

Máquinas imperfeitas

Os seres humanos são máquinas imperfeitas. Eles ficam entediados, têm processadores (e, às vezes, UIs) que não são muito bem compreendidos e não são muito eficientes. Reconhecer o elemento humano como “funcionando conforme esperado” e tentar contornar ou aperfeiçoar o modo como os seres humanos trabalham poderia exigir um espaço de discussão muito maior do que está disponível aqui; por enquanto, algumas ideias básicas podem ser úteis para determinar como as interrupções devem funcionar.

Estado de fluxo cognitivo

O conceito de *estado de fluxo*¹ é amplamente aceito e pode ser empiricamente reconhecido por quase todos que trabalham em engenharia de software, administração de sistemas, SRE e na maior parte das demais disciplinas que exijam períodos de concentração. Estar “na zona” (de fluxo) pode aumentar a produtividade, mas também pode melhorar a criatividade artística e científica. Atingir esse estado incentiva as pessoas a realmente dominar e aperfeiçoar a tarefa ou o projeto em que estão trabalhando. Ser interrompido pode tirar você imediatamente desse estado, se a interrupção for suficientemente perturbadora. Você deve maximizar o período de tempo em que estiver nesse estado.

O fluxo cognitivo também se aplica a empreitadas menos criativas, em que o nível de habilidade exigido é menor e os elementos essenciais do fluxo ainda estão presentes (metas claras, feedback imediato, um senso de controle e a distorção do tempo associada); exemplos incluem tarefas domésticas e dirigir.

Você pode entrar nessa zona de fluxo trabalhando em problemas de pouca dificuldade, que exijam pouca habilidade, como jogar um videogame repetitivo. Também pode facilmente chegar lá trabalhando com problemas de alta dificuldade, que exijam muita habilidade, como aqueles enfrentados por um engenheiro. Os métodos de chegar a um estado de fluxo cognitivo diferem, mas o resultado é essencialmente o mesmo.

Estado de fluxo cognitivo: criativo e engajado

A zona é esta: alguém trabalha em um problema por um tempo, está ciente

dos parâmetros do problema e se sente à vontade com eles e acha que pode corrigi-lo ou resolvê-lo. A pessoa trabalha atentamente no problema, perdendo a noção do tempo e ignorando o máximo possível as interrupções. Maximizar a quantidade de tempo que uma pessoa pode passar nesse estado é bastante desejável – ela produzirá resultados criativos e fará um bom trabalho em termos de volume. Além disso, ficará mais satisfeita com o trabalho que faz.

Infelizmente, muitas pessoas em funções do tipo SRE gastam boa parte de seu tempo tentando e falhando em entrar nesse modo, ficando frustradas quando não conseguem ou nem mesmo tentam alcançá-lo, permanecendo no estado de interrupção.

Estado de fluxo cognitivo: irritado

As pessoas gostam de realizar tarefas que sabem como fazer. Com efeito, executar tarefas como essas é um dos caminhos mais claros para o fluxo cognitivo. Alguns SREs estão de plantão quando alcançam um estado de fluxo cognitivo. Pode ser muito satisfatório rastrear as causas dos problemas, trabalhar com outras pessoas e melhorar a saúde geral do sistema de uma forma tangível assim. De modo recíproco, para os engenheiros de plantão mais estressados, o estresse é causado pelo volume de pages ou porque tratam o plantão como uma interrupção. Eles estão tentando programar ou trabalhar em projetos, ao mesmo tempo que estão de plantão ou trabalham em interrupções de tempo integral. Esses engenheiros estão em um estado constante de interrupção, ou *sujeitos à interrupção*. Esse ambiente de trabalho é extremamente estressante.

Por outro lado, quando uma pessoa está todo tempo concentrada em interrupções, *as interrupções deixam de ser interrupções*. Em um nível bem visceral, fazer melhorias incrementais no sistema, atacar tickets e corrigir problemas e interrupções de serviço passam a ser um conjunto claro de objetivos, fronteiras e feedback claro: você encerra X bugs ou para de receber pages. Tudo que resta são distrações. *Ao trabalhar com interrupções, seus projetos serão uma distração*. Mesmo que as interrupções possam representar um uso satisfatório do tempo no curto prazo, em um ambiente que combine projeto/plantão, as pessoas, em última instância, ficarão mais satisfeitas com

um 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 mais os motiva (ou podem achar que sabem, mas você pode discordar).

Fazer bem uma tarefa

Você pode estar se perguntando sobre as implicações práticas do que leu até agora.

as sugestões a seguir, baseadas no que funcionou para várias equipes de SRE que gerenciei no Google, servem principalmente para gerentes de equipe ou aqueles que a influenciam. Esse documento é independente de hábitos pessoais – as pessoas são livres para administrar seu próprio tempo conforme acharem apropriado. Nós nos concentraremos aqui em oferecer diretrizes para estruturar como a própria equipe administra as interrupções, para que as pessoas não estejam fadadas a falhar por causa da função ou da estrutura da equipe.

Possibilidade de sofrer distrações

As maneiras pelas quais um engenheiro pode se distrair e, desse modo, ser impedido de alcançar um estado de fluxo cognitivo, são numerosas. Por exemplo, considere um SRE qualquer chamado Fred. Fred vem trabalhar na segunda-feira de manhã. Ele não está de plantão nem tratando interrupções hoje, portanto ele gostaria claramente de trabalhar em seus projetos. Fred pega um café, coloca seus fones de ouvido “não perturbe” e se senta em sua escrivaninha. Hora de entrar na zona de fluxo, certo?

Exceto que, a qualquer momento, as seguintes situações podem ocorrer:

- A equipe de Fred utiliza um sistema automatizado de tickets para atribuí-los aleatoriamente à equipe. Um ticket é atribuído a ele, com prazo para hoje.
- O colega de Fred está de plantão e recebe um page relacionado a um componente do qual Fred é o expert e o interrompe para fazer perguntas sobre esse componente.
- Um usuário do serviço de Fred eleva a prioridade de um ticket que havia

sido atribuído a ele na semana passada, quando ele estava de plantão.

- Um rollout de flag em implantação há 3 ou 4 semanas, e que foi atribuído a Fred, dá errado, forçando-o a deixar tudo de lado e analisar o rollout, fazer rollback da alteração, e assim por diante.
- Um usuário do serviço de Fred entra em contato com ele para fazer uma pergunta, pois Fred é um rapaz muito solícito.
- E assim por diante.

O resultado final é que, apesar de ter o calendário todo livre para trabalhar em projetos, Fred permanece extremamente sujeito a distrações. Algumas dessas distrações, ele mesmo pode administrar fechando o email, desativando o IM ou tomando outras medidas semelhantes. Outras distrações são provocadas por políticas ou por suposições que giram em torno de interrupções e de responsabilidades contínuas.

Você pode argumentar que algum nível de distração é inevitável por design. Essa suposição está correta: as pessoas se atêm a bugs para os quais elas são o contato principal, além de assumirem outras responsabilidades e obrigações. No entanto, há maneiras de uma equipe poder administrar respostas a interrupções de modo que mais pessoas (em média) possam chegar de manhã ao trabalho e *sentir que não estão sujeitas a distrações*.

Tempo polarizado

A fim de limitar a possibilidade de distrações, você deve tentar minimizar as mudanças de contexto. Algumas interrupções são inevitáveis. Contudo, ver um engenheiro como uma unidade de trabalho que possa ser interrompida, cujas mudanças de contexto são gratuitas, não será ideal se você quiser que as pessoas sejam felizes e produtivas. Atribua um custo às mudanças de contexto. Uma interrupção de 20 minutos enquanto você está trabalhando em um projeto implica duas mudanças de contexto; falando de modo realista, essa interrupção resulta em uma perda de cerca de duas horas de um verdadeiro trabalho produtivo. Para evitar ocorrências constantes de perda de produtividade, procure ter um tempo polarizado entre os estilos de trabalho, em que cada período de trabalho dure o máximo possível. O ideal é que esse período de tempo seja de uma semana, mas um dia ou até mesmo a metade

dele poderão ser mais práticos. Essa estratégia também se enquadra no conceito complementar de *tempo produtivo* (make time) [Gra09].

Um tempo polarizado significa que, quando uma pessoa chega ao trabalho todos os dias, deverá saber se está fazendo *apenas* trabalho de projeto ou *apenas* tratando interrupções. Polarizar seu tempo dessa maneira significa que ela poderá permanecer concentrada por períodos de tempo mais longos na tarefa em questão. Ela não ficará estressada por ser amarrada a tarefas que a arrastarão para longe do trabalho que deveria estar fazendo.

É sério, diga-me o que devo fazer

Se o modelo geral apresentado neste capítulo não funcionar para você, eis algumas sugestões específicas de componentes que você pode implementar aos poucos.

Sugestões gerais

Para qualquer dada classe de interrupções, se o volume delas for muito alto para uma pessoa, *acrescente outra*. Esse conceito se aplica mais obviamente aos tickets, mas pode se aplicar potencialmente aos pages também – o engenheiro de plantão começa a empurrar tarefas ao seu secundário ou reduz pages a tickets.

Plantão

O engenheiro no plantão principal deve focar exclusivamente o trabalho de plantão. Se o pager estiver silencioso para o seu serviço, tickets ou outras tarefas baseadas em interrupções, que possam ser abandonadas de forma razoavelmente rápida, devem fazer parte das obrigações de quem está de plantão. Quando um engenheiro estiver de plantão por uma semana, essa semana deve ser eliminada no que diz respeito a trabalhos em projeto. Se um projeto for muito importante, a ponto de não ser possível ter um atraso de uma semana, essa pessoa não deverá estar de plantão. Escale a fim de designar outra pessoa para o turno de plantão. *Jamais devemos esperar que uma pessoa esteja de plantão, ao mesmo tempo que faz progresso em projetos (ou em qualquer outra tarefa com um custo alto de mudança de contexto).*

Obrigações secundárias dependem do custo dessas obrigações. Se a função do secundário for servir de backup para o principal no caso de haver a necessidade de uma alternativa, talvez você possa supor, de forma segura, que o secundário também possa realizar tarefas de projeto. Se alguém que não o secundário for designado para tratar tickets, considere combinar as funções. Se for esperado que o secundário realmente ajude o principal no caso de um volume alto de pages, eles também deverão trabalhar em interrupções.

(Aparte: *você jamais ficará sem tarefas de limpeza*. O número de tickets poderá ser zero, mas sempre haverá documentação para ser atualizada, configurações que precisam de limpeza etc. Seus futuros engenheiros de plantão agradecerão a você, e isso significa que será menos provável que eles o interrompam em seu precioso tempo produtivo.)

Tickets

Se, atualmente, você atribui tickets de forma aleatória às vítimas em sua equipe, *pare*. Fazer isso é extremamente desrespeitoso com o tempo de sua equipe, e funciona de forma totalmente contrária ao princípio de não ser interrompido o máximo possível.

Os tickets devem estar associados a uma função em tempo integral, por um período de tempo que seja administrável por uma pessoa. Se, por acaso, você estiver na posição não invejável de ter mais tickets que possam ser encerrados pelos engenheiros de plantão principal e secundário em conjunto, estruture seu rodízio de tickets para que haja duas pessoas tratando-os a qualquer momento. Não distribua a carga pela equipe toda. As pessoas não são máquinas, e você só estará gerando mudanças de contexto que terão impacto no tempo de fluxo valioso.

Responsabilidades contínuas

Defina funções que permitam que qualquer pessoa da equipe assuma o comando, da melhor forma possível. Se houver um procedimento bem definido para realizar e conferir atualizações de versão ou mudanças de flag, então não há motivos para uma pessoa precisar conduzir essa mudança durante toda a sua vida, mesmo depois que essa pessoa deixe de estar de

plantão ou pare de tratar interrupções. Defina uma função de *administrador de atualizações*, que possa cuidar delas pelo período em que ele estiver de plantão ou tratando interrupções. Formalize o processo de transferência – é um pequeno preço a pagar para ter um tempo produtivo sem interrupções para as pessoas que não estão de plantão.

Trabalhe nas interrupções, ou não

Às vezes, quando uma pessoa não está tratando interrupções, a equipe recebe uma para a qual essa pessoa é a única qualificada para tratar. Embora o ideal seja que esse cenário jamais ocorra, ocasionalmente isso acontece. Você deve trabalhar para que essas ocorrências sejam raras.

Às vezes, as pessoas trabalham em tickets quando não estão designadas para tratá-las porque é uma maneira fácil de parecer ocupado. Um comportamento desse tipo não é conveniente. Significa que a pessoa é menos eficiente do que deveria ser. Isso distorce os números que dizem respeito a quão administrável é a carga de tickets. Se uma pessoa é designada para tratar tickets, mas há outras duas ou três também trabalhando na fila de tickets, você ainda poderá ter uma fila de tickets não administrável, mesmo sem percebê-lo.

Reduzindo as interrupções

A carga de interrupções de sua equipe pode não ser administrável se exigir que muitos membros da equipe cuidem das interrupções simultaneamente a qualquer momento. Há várias técnicas que você pode usar para reduzir sua carga geral de tickets.

Analise realmente os tickets

Muitos rodízios de tickets ou de plantão funcionam como um “corredor polonês”. Isso é especialmente verdadeiro para rodízios em equipes maiores. Se você estiver tratando interrupções somente a cada dois meses, será fácil passar pelo “corredor polonês”², suspirar com alívio e, então, retornar às obrigações regulares. Seu sucessor fará o mesmo, e as causas-raízes dos tickets jamais serão investigadas. Em vez de fazer progressos, sua equipe terá uma sucessão de pessoas irritadas com os mesmos problemas.

Deve haver um processo de transferência de tickets, assim como do trabalho de plantão. Um processo de transferência mantém um estado compartilhado entre as pessoas que tratam os tickets à medida que a responsabilidade muda de mãos. Mesmo uma introspecção básica nas causas-raízes das interrupções pode oferecer boas soluções para reduzir a taxa total delas. *Muitas* equipes fazem transferência de plantão e de análises de pages. *Poucas* delas fazem o mesmo com tickets.

Sua equipe deve conduzir uma limpeza regular de tickets e pages, em que você deve analisar as classes de interrupções para ver se é possível identificar uma causa-raiz. Se achar que a causa-raiz pode ser corrigida em um período de tempo razoável, *silencie as interrupções até que a causa-raiz seja corrigida conforme esperado*. Fazer isso proporciona alívio para a pessoa que está tratando as interrupções e cria uma garantia conveniente de prazo para a pessoa que estiver corrigindo a causa-raiz.

Respeite a si mesmo e aos clientes também

Essa máxima se aplica mais às interrupções de usuário do que às interrupções automatizadas, embora os princípios sejam válidos para os dois cenários. Se os tickets forem particularmente irritantes ou custosos para resolver, você poderá usar uma política para atenuar a carga, de modo eficiente.

Lembre-se de que:

- Sua equipe define o nível de serviço oferecido pelo seu serviço.
- Não há problema em devolver parte do esforço para seus clientes.

Se sua equipe for responsável por tratar tickets ou interrupções de clientes, muitas vezes você poderá usar uma política para deixar sua carga de trabalho mais administrável. Uma correção de política pode ser temporária ou permanente, dependendo do que fizer sentido. Uma correção como essa deve apresentar um bom equilíbrio entre respeito pelo cliente e respeito a si mesmo. Uma política pode ser uma ferramenta tão eficaz quanto o código.

Por exemplo, se você dá suporte a uma ferramenta particularmente frágil, que não tenha muitos recursos de desenvolvedor, e um pequeno número de clientes precisa dela, considere outras opções. Pense no valor do tempo que você gasta tratando interrupções desse sistema e se está gastando esse tempo

de forma inteligente. Em algum momento, se você não puder obter a atenção necessária para corrigir a causa-raiz dos problemas que estão provocando as interrupções, talvez o componente ao qual você dá suporte não seja tão importante assim. Considere devolver o pager, deixar o sistema obsoleto, substituí-lo ou usar outra estratégia nessa linha, que possa fazer sentido.

Se houver passos particulares para tratar uma interrupção que consuma tempo ou seja intrincada, mas não exija seus privilégios para ser tratada, considere usar uma política para devolver a solicitação a quem a fez. Por exemplo, se as pessoas precisarem doar recursos computacionais, prepare um código, uma mudança de configuração ou um passo semelhante e, então, instrua o cliente a executar esse passo e enviá-lo para sua revisão. Lembre-se de que, se o cliente quiser que uma determinada tarefa seja executada, ele deverá estar preparado para investir um pouco de esforço a fim de conseguir o que quer.

Uma ressalva às soluções anteriores é que você deve encontrar um equilíbrio entre respeito pelo cliente e respeito a si mesmo. Seu princípio orientador na definição de uma estratégia para lidar com solicitações de clientes é que a solicitação deve ser significativa, racional e oferecer todas as informações e complementos necessários para que a solicitação seja tratada. Em troca, sua resposta deverá ser útil e oportuna.

¹ Veja a Wikipédia: Flow (psychology) (Fluxo [psicologia]), [http://en.wikipedia.org/wiki/Flow_\(psychology\)](http://en.wikipedia.org/wiki/Flow_(psychology)).

² Veja http://en.wikipedia.org/wiki/Running_the_gauntlet.

CAPÍTULO 30

Incluindo um SRE para se recuperar de uma sobrecarga operacional

Escrito por Randall Bosetti

Editado por Diane Bates

Dividir uniformemente seu tempo entre trabalho em projeto e tarefas operacionais reativas é uma política-padrão das equipes de SRE do Google. Na prática, às vezes, esse equilíbrio pode ser perturbado por meses devido a um aumento no volume de tickets diários. Uma carga pesada de tarefas operacionais é especialmente perigosa porque a equipe de SRE pode ficar estressada ou ser incapaz de fazer progressos em trabalho de projeto. Quando uma equipe precisa alocar uma quantidade desproporcional de tempo para resolver tickets em vez de gastar tempo para melhorar o serviço, a escalabilidade e a confiabilidade sofrem.

Uma maneira de aliviar essa carga é transferir temporariamente um SRE para a equipe sobrecarregada. Depois de ser incluído em uma equipe, o SRE se concentrará em aperfeiçoar suas práticas, em vez de simplesmente ajudar essa equipe a esvaziar a fila de tickets. O SRE observará a rotina diária da equipe e fará recomendações para melhorar suas práticas. Essa consultoria dá à equipe uma nova perspectiva sobre suas rotinas, que os membros da equipe sozinhos não conseguem ver.

Ao usar essa abordagem, não é necessário transferir mais de um engenheiro. Dois SREs não necessariamente produzirão melhores resultados e podem, na verdade, causar problemas se a equipe reagir de forma defensiva.

Se você está iniciando sua primeira equipe de SRE, a abordagem apresentada neste capítulo ajudará a evitar transformá-la em uma equipe de operações

com foco exclusivo em um rodízio para solucionar tickets. Se você decidir incluir a si mesmo ou alguém que responda a você em uma equipe, reserve tempo para rever as práticas e a filosofia de SRE na Introdução de Ben Treynor Sloss (Capítulo 1) e o material sobre monitoração no Capítulo 6.

As próximas seções apresentam diretrizes ao SRE que será incluído em uma equipe.

Fase 1: conheça o serviço e saiba qual é o contexto

Seu trabalho enquanto estiver incluído na equipe é articular por que os processos e os hábitos contribuem ou trabalham contra a escalabilidade do serviço. Lembre a equipe de que mais tickets não exigem mais SREs: a meta do modelo de SRE é introduzir mais seres humanos somente se mais complexidade for adicionada ao sistema. Em vez disso, procure chamar a atenção para o modo como hábitos de trabalho saudáveis reduzem o tempo gasto em tickets. Fazer isso é tão importante quanto apontar as oportunidades perdidas para automação ou simplificação do serviço.

Modo operacional versus escala não linear

O termo *modo operacional* refere-se a um determinado método de manter um serviço operando. Vários itens relacionados à operação aumentam com o tamanho do serviço. Por exemplo, um serviço precisa de um modo de aumentar o número de VMs (Virtual Machines, ou Máquinas Virtuais) configuradas à medida que se expande. Uma equipe em modo operacional responde aumentando o número de administradores que gerenciam essas VM. A SRE, por sua vez, tem como foco escrever um software ou eliminar preocupações com escalabilidade de modo que o número de pessoas necessário para operar um serviço não aumente como uma função da carga do serviço.

As equipes que se deixam entrar em modo operacional podem estar convencidas de que a escala não importa para elas (“meu serviço é minúsculo”). Acompanhe uma sessão de plantão a fim de determinar se a avaliação é verdadeira, pois o elemento escala afetará a sua estratégia.

Se o serviço principal for importante para o negócio, mas é realmente

minúsculo (o que implica menos recursos ou baixa complexidade), coloque mais foco nas maneiras pelas quais a abordagem atual da equipe impeça uma melhoria na confiabilidade do serviço. Lembre-se de que seu trabalho é fazer o serviço funcionar, e não proteger a equipe de desenvolvimento contra alertas.

Por outro lado, se o serviço está apenas começando, concentre-se em maneiras de preparar a equipe para um crescimento explosivo. Um serviço com 100 requisições/segundo pode se transformar em um serviço de 10k requisições/segundo em um ano.

Identifique as principais causas de estresse

As equipes de SRE às vezes entram em modo operacional porque se concentram em como tratar emergências rapidamente, em vez de colocar o foco na redução do número de emergências. Estar em modo operacional por padrão geralmente ocorre em resposta a uma pressão intensa, real *ou imaginada*. Depois de ter conhecido suficientemente o serviço a ponto de fazer perguntas difíceis sobre o seu design e a sua implantação, invista um tempo atribuindo prioridades às várias interrupções de serviço, de acordo com o seu impacto nos níveis de estresse da equipe. Tenha em mente que, por causa da perspectiva e da história da equipe, alguns problemas pequenos ou interrupções de serviço podem gerar uma quantidade desmesurada de estresse.

Identifique os fatores que causam estresse

Após identificar os maiores problemas de uma equipe, passe para as emergências à espera de acontecer. Às vezes, emergências prementes surgem na forma de um novo subsistema que não tenha sido projetado para ser autoadministrado. Outras causas incluem:

Lacunas no conhecimento

Em equipes grandes, as pessoas podem se especializar demais, sem que haja consequências imediatas. Quando uma pessoa se especializa, ela corre o risco de não ter um conhecimento amplo necessário para dar suporte ao plantão, ou isso permite que membros da equipe ignorem as partes críticas

do sistema pelo qual eles são responsáveis.

Serviços desenvolvidos pela SRE cuja importância aumenta silenciosamente

Esses serviços muitas vezes não recebem a mesma atenção cuidadosa que os lançamentos de novas funcionalidades, pois são menores em escala e são implicitamente endossados por, no mínimo, um SRE.

Forte dependência da “próxima grande solução”

Às vezes, as pessoas podem ignorar certos problemas por meses porque acreditam que a nova solução no horizonte invalidará as correções temporárias.

Alertas comuns que não são diagnosticados pela equipe de desenvolvimento nem pelos SREs

Alertas desse tipo são frequentemente classificados como *transientes*, mas distraem seus colegas de equipe, que não corrigirão os verdadeiros problemas. Investigue esses alertas de forma completa ou corrija as regras que os geram.

Qualquer serviço que seja o assunto de reclamações de seus clientes e não tenha um SLI/SLO/SLA formal

Veja o Capítulo 4, que apresenta uma discussão sobre SLIs, SLOs e SLAs.

Qualquer serviço com um plano de capacidade que seja efetivamente do tipo: “Adicione mais servidores: nossos servidores estavam ficando sem memória ontem à noite”

Planos de capacidade devem estar suficientemente voltados ao futuro. Se o modelo de seu sistema prevê que os servidores precisam de 2 GB, um teste de carga que passe no curto prazo (revelando 1,99 GB em sua última execução) não necessariamente significa que a capacidade de seu sistema esteja em boa forma.

Postmortems que tenham como ações apenas fazer rollback das mudanças específicas que provocaram uma interrupção no serviço

Por exemplo, “Mude o timeout de streaming de volta para 60 segundos” em vez de “Descubra por que, às vezes, são necessários 60 segundos para

buscar o primeiro megabyte de nossos vídeos promocionais”.

Qualquer componente crítico para servir às requisições, para o qual os SREs existentes respondam às perguntas dizendo: “Não sabemos nada sobre ele; a equipe de desenvolvimento é responsável por esse componente”

Para oferecer um suporte de plantão aceitável a um componente, você deve, no mínimo, saber quais são as consequências quando ele falha e a urgência necessária para corrigir seus problemas.

Fase 2: compartilhando o contexto

Após definir o escopo das dinâmicas e dos pontos problemáticos da equipe, prepare o terreno para as melhorias por meio das melhores práticas, como postmortems, e pela identificação das causas das tarefas penosas (toil work) e da melhor forma de abordá-las.

Escreva um bom postmortem para a equipe

Os postmortems oferecem muitos insights para o raciocínio coletivo de uma equipe. Postmortems conduzidos por equipes não saudáveis muitas vezes não são eficazes. Alguns membros da equipe podem considerar os postmortems punitivos, ou até mesmo inúteis. Embora você possa se sentir tentado a revisar os postmortems arquivados e deixar comentários para melhorias, fazer isso não ajudará a equipe. Pelo contrário, o exercício poderá colocá-la na defensiva.

Em vez de tentar corrigir erros anteriores, assuma a responsabilidade pelo próximo postmortem. *Haverá* uma interrupção de serviço enquanto você estiver incluído na equipe. Se você não for a pessoa que está de plantão, junte-se ao SRE de plantão para escrever um ótimo postmortem, sem culpados. Esse documento é uma oportunidade para mostrar como uma mudança em direção ao modelo de SRE beneficia a equipe, fazendo com que as correções de bug sejam mais permanentes. Correções de bug mais permanentes reduzem o impacto das interrupções de serviço no tempo dos membros da equipe.

Conforme mencionamos, você poderá se deparar com respostas como “Por

que eu?”. Essa resposta é especialmente provável quando uma equipe acredita que o processo de postmortem seja uma retaliação. Essa atitude resulta da aderência à Teoria das Maçãs Ruins: o sistema está funcionando bem e, se nos livrarmos de todas as maçãs ruins e de seus erros, o sistema continuará bem. A Teoria das Maçãs Ruins pode ser demonstrada como falsa, conforme mostrado por evidências [Dek14] em várias disciplinas, incluindo a segurança em aviação. Você deve apontar essa falsidade. A forma mais eficiente de escrever um postmortem é dizer: “Erros são inevitáveis em qualquer sistema com várias interações sutis. Você estava de plantão, e eu confio em você para tomar as decisões corretas com as informações corretas. Gostaria que você escrevesse o que estava pensando em cada instante no tempo, para que possamos descobrir em que ponto o sistema confundiu você e quando as exigências cognitivas eram altas demais”.

Classifique os fatores que causam estresse de acordo com o tipo
Há dois tipos de fatores que causam estresse nesse modelo simplificado por questões de conveniência:

- Alguns fatores não deveriam existir. Eles causam o que é comumente chamado de tarefas operacionais ou penosas (veja o Capítulo 5).
- Outros fatores que causam estresse e/ou fúria são, na verdade, parte do trabalho. Qualquer que seja o caso, a equipe precisa desenvolver ferramentas para controlar o estresse.

Classifique as causas de estresse da equipe em tarefas penosas ou não. Quando terminar, apresente a lista à equipe e explique claramente por que cada causa de estresse é uma tarefa que deve ser automatizada ou um overhead aceitável para operar o serviço.

Fase 3: conduzindo as mudanças

A saúde de uma equipe é um processo. Assim, não é algo que você possa resolver com um esforço heroico. Para garantir que a equipe possa administrar a si mesma, você pode ajudá-la a construir um bom modelo mental para um envolvimento ideal da SRE.



Seres humanos são muito bons em homeostase, portanto *concentre-se em criar (ou restaurar) as condições iniciais corretas e ensinar o pequeno conjunto de princípios necessário para fazer escolhas saudáveis.*

Comece pelo básico

Equipes lutando contra a distinção entre o modelo de SRE e o modelo tradicional de operações geralmente são incapazes de articular *por que* determinados aspectos do código, dos processos ou da cultura da equipe as incomodam. Em vez de tentar abordar cada um desses problemas ponto a ponto, trabalhe partindo dos princípios apresentados nos Capítulos de 1 a 6.

Sua primeira meta para a equipe deve ser escrever um SLO (Service Level Objective, ou Objetivo de Nível de Serviço), caso ainda não haja um. O SLO é importante porque oferece uma medida quantitativa do impacto das interrupções de serviço, além de mostrar a importância da mudança em um processo. Um SLO provavelmente é a alavanca única mais importante para fazer uma equipe passar de tarefas operacionais reativas para um foco em SRE saudável e de longo prazo. Se não houver esse consenso, nenhum outro conselho deste capítulo será útil. *Se você se vir em uma equipe sem SLOs, leia antes o Capítulo 4 e então reúna os líderes técnicos e a gerência em uma sala e comece a coordenar essa tarefa.*

Consiga ajuda para eliminar as causas de estresse

Talvez você sinta uma forte urgência para simplesmente corrigir os problemas que identificar. Por favor, resista a esse impulso de corrigir esses problemas você mesmo, pois fazer isso incentivará a ideia de que “fazer mudanças cabe a outras pessoas”. Em vez disso, siga os passos a seguir:

1. Encontre uma tarefa útil que possa ser realizada por um membro da equipe.
2. Explique claramente como essa tarefa trata um problema do postmortem *de modo permanente*. Mesmo equipes saudáveis poderão gerar uma lista de ações com uma visão restrita.
3. Sirva como revisor das mudanças de código e das revisões de documentação.

4. Repita o processo para dois ou três problemas.

Ao identificar um problema adicional, coloque-o em um relatório de bug ou em um documento para a equipe consultar. Fazer isso serve ao propósito duplo de distribuir informações e incentivar os membros da equipe a escrever documentos (que, com frequência, são a primeira vítima da pressão do prazo). Sempre explique seu raciocínio e enfatize que uma boa documentação garante que a equipe não repetirá antigos erros em um contexto um pouco diferente.

Explique o seu raciocínio

Depois que a equipe recuperar sua velocidade e dominar o básico sobre as alterações sugeridas, prossiga enfrentando as decisões cotidianas que originalmente levaram à sobrecarga operacional. Prepare-se, pois essa tarefa poderá ser desafiadora. Se estiver com sorte, o desafio será na linha de “Explique o porquê. Agora. No meio da reunião de produção semanal”.

Se não estiver com sorte, ninguém exigirá uma explicação. Ignore totalmente esse problema apenas explicando todas as suas decisões, independentemente de alguém pedir uma explicação. Refira-se ao básico que sustenta suas sugestões. Fazer isso ajuda a criar o modelo mental da equipe. *Depois que deixar a equipe, ela deverá ser capaz de prever quais seriam seus comentários acerca de um design ou de uma lista de mudanças.* Se você não explicar o seu raciocínio, ou fizer isso de forma precária, há um risco de a equipe simplesmente repetir aquele comportamento que deixou a desejar; portanto, seja explícito.

Exemplos de uma explicação completa sobre a sua decisão:

- “Não estou retornando para a última versão porque os testes estão ruins. Estou fazendo isso porque a provisão para erros que definimos para as versões se esgotou.”
- “É necessário fazer rollback das versões de modo seguro porque o nosso SLO está apertado. Atender a esse SLO exige que o tempo médio de recuperação seja baixo; portanto, fazer um diagnóstico detalhado antes de um rollback não é realista.”

Exemplos de uma explicação insuficiente para a sua decisão:

- “Não acho que fazer todos os servidores gerarem sua configuração de roteamento seja seguro, pois não podemos vê-la.”

Essa decisão provavelmente está correta, mas o raciocínio é precário (ou está explicado de forma precária). A equipe não é capaz de ler a sua mente, portanto é muito provável que vá repetir o raciocínio precário observado. No lugar disso, tente “[...] não é seguro porque um bug nesse código pode provocar uma falha correlacionada em todo o serviço e o código adicional é uma fonte de bugs que poderia deixar um rollback lento”.

- “A automação deve desistir se encontrar uma implantação conflitante.”

Como no exemplo anterior, essa explicação provavelmente está correta, mas é insuficiente. Em seu lugar, tente “[...] porque estamos fazendo a suposição simplificada de que todas as mudanças passam pela automação e algo claramente violou essa regra. Se isso ocorrer com frequência, devemos identificar e eliminar as causas de uma mudança desorganizada”.

Faça perguntas inteligentes

Perguntas inteligentes não são perguntas longas. Ao conversar com a equipe de SRE, procure fazer perguntas de um modo que incentive as pessoas a pensar nos princípios básicos. É particularmente importante que você modele esse comportamento porque, por definição, uma equipe em modo operacional rejeitará esse tipo de raciocínio desde o princípio. Após ter investido um tempo explicando seu raciocínio para várias perguntas sobre a política, essa prática reforçará a compreensão da equipe acerca da filosofia da SRE.

Exemplos de perguntas inteligentes:

- “Vejo que o alerta TaskFailures dispara com frequência, mas os engenheiros de plantão geralmente não fazem nada para responder a ele. Como isso afeta o SLO?”
- “Esse procedimento de ativação parece bem complicado. Você sabe por que há tantos arquivos de configuração para atualizar ao criar uma nova instância do serviço?”

Contraexemplos de perguntas inteligentes:

- “Qual é o problema dessas versões antigas e desatualizadas?”
- “Por que o Frobnitzer faz tantas tarefas?”

Conclusão

Seguir os princípios apresentados neste capítulo proporciona a uma equipe de SRE o seguinte:

- Uma perspectiva técnica, possivelmente quantitativa, sobre por que ela deve mudar.
- Um exemplo robusto de como deve ser o aspecto das mudanças.
- Uma explicação lógica para muitas das “sabedorias populares” usadas pela SRE.
- Os princípios básicos necessários para tratar situações novas de forma escalável.

Sua última tarefa é escrever um relatório após o fato consumado. Esse relatório deve reiterar sua perspectiva, os exemplos e as explicações. Ele também deve apresentar algumas ações para a equipe a fim de garantir que ela exercitará o que você ensinou. Você pode organizar o relatório como um postvitam¹, explicando as decisões críticas em cada passo que levaram ao sucesso.

A maior parte de seu envolvimento agora está completa. Depois que sua função como SRE incluído terminar, você deverá permanecer disponível para revisões de design e de código. Fique de olho na equipe nos próximos meses para confirmar que, aos poucos, ela esteja melhorando seu planejamento de capacidade, a resposta a emergências e os processos de rollout.

¹ Em oposição a um postmortem.

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 tem efeitos sobre como nos comunicamos e colaboramos.

Para começar, há uma diversidade enorme no que a SRE faz, e como o fazemos. Temos equipes de infraestrutura, de serviços e de produtos horizontais. Temos relacionamentos com equipes de desenvolvimento de produtos, que variam de equipes muitas vezes maiores que a nossa, equipes que têm aproximadamente o mesmo tamanho de suas contrapartidas e situações em que *somos* a equipe de desenvolvimento de produto. As equipes de SRE são compostas de pessoas com habilidades em engenharia de sistemas e arquitetura (veja [Hix15b]), habilidades em engenharia de software e em gerenciamento de projetos, instintos de liderança, experiências anteriores em todo tipo de mercado (veja o Capítulo 33), e assim por diante. Não temos apenas um modelo, e identificamos várias configurações que funcionam; essa flexibilidade é adequada à nossa natureza, em última instância, pragmática.

Também é verdade que a SRE não é uma organização de comando e controle. Em geral, devemos obediência a pelo menos dois mestres: nas equipes de SRE para serviços ou infraestrutura, trabalhamos muito próximos às equipes de desenvolvimento de produto correspondentes que trabalham nesses serviços ou infraestrutura; também trabalhamos, obviamente, no contexto da SRE em geral. O relacionamento com os serviços é muito forte, pois somos considerados responsáveis pelo desempenho desses sistemas, mas, apesar desse relacionamento, as linhas às quais respondemos estão na SRE como um

todo. Atualmente, passamos mais tempo dando suporte aos nossos serviços individuais do que a trabalhos em produção; porém, nossa cultura e nossos valores compartilhados produzem abordagens fortemente homogêneas aos problemas. Isso ocorre por design.¹

Os dois fatos anteriores têm levado a organização SRE a determinadas direções quando se trata de duas dimensões fundamentais acerca 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 pela produção, eles também devem fluir por uma equipe de SRE – dados sobre projetos, o estado dos serviços, da produção e dos indivíduos. Para a máxima eficiência de uma equipe, os dados devem fluir de maneiras confiáveis, de uma parte interessada para outra. Uma forma de imaginar esse fluxo é pensar na interface que uma equipe de SRE deve apresentar a outras equipes como se fosse uma API. Assim como uma API, um bom design é fundamental para uma operação eficiente, e se a API estiver incorreta, poderá ser difícil corrigi-la depois.

A metáfora de API como um contrato também é relevante para a colaboração, tanto entre as equipes de SRE quanto entre a SRE e as equipes de desenvolvimento de produtos – sempre deve haver progresso em um ambiente de mudanças contínuas. Para isso, nossa colaboração se parece com a colaboração em qualquer outra empresa que se move rapidamente. A diferença está na combinação de habilidades em engenharia de software, expertise em engenharia de sistemas e na sabedoria adquirida com a experiência em produção que a SRE ostenta nessa colaboração. Os melhores designs e as melhores implementações resultam das preocupações conjuntas de produção e produto, tratadas em uma atmosfera de respeito mútuo. Esta é a promessa da SRE: uma organização responsável pela confiabilidade, com as mesmas habilidades das equipes de desenvolvimento de produtos, melhorará os sistemas de forma considerável. 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 conduzir reuniões de forma eficaz seja abundante [Kra08], é difícil encontrar alguém que tenha sorte o bastante para ter tido *apenas* reuniões úteis e eficientes. Isso é igualmente verdadeiro para a SRE.

No entanto, há um tipo de reunião que fazemos, chamada de *reunião de produção* (production meeting), que é mais útil que a média. As reuniões de produção são um tipo especial de reunião, em que uma equipe de SRE cuidadosamente se coordena – e coordena com seus convidados – acerca do estado do(s) serviço(s) sob sua responsabilidade, de modo a aumentar o conhecimento geral entre todos os interessados e melhorar a operação desse(s) serviço(s). Em geral, essas reuniões são *orientadas a serviços*; elas não são diretamente sobre atualizações do status dos indivíduos. A meta é que todos saiam da reunião com uma ideia sobre o que está acontecendo – a *mesma* ideia. O outro objetivo importante das reuniões de produção é aperfeiçoar nossos serviços levando para eles a sabedoria adquirida em produção. Isso significa que conversamos em detalhes sobre o desempenho operacional do serviço e relacionamos esse desempenho ao design, à configuração ou à implementação, além de fazermos recomendações sobre como corrigir os problemas. Associar o desempenho do serviço às decisões de design em uma reunião regular representa um ciclo de feedback extremamente eficaz.

Nossas reuniões de produção costumam ocorrer semanalmente; dada a antipatia da SRE por reuniões sem sentido, essa parece ser a frequência correta: tempo para permitir que materiais relevantes se acumulem o suficiente para que a reunião valha a pena, e ao mesmo tempo não tão frequente a ponto de as pessoas encontrarem uma desculpa para não participar. Essas reuniões geralmente duram de 30 a 60 minutos. Se durarem menos, provavelmente você estará reduzindo algo desnecessariamente, ou é provável que deva aumentar seu portfólio de serviços. Se durarem mais, provavelmente você estará se atolando em detalhes, ou há muito que conversar e você deve fragmentar a equipe ou o conjunto de serviços.

Assim como em qualquer outra reunião, a reunião de produção deve ter alguém para conduzi-la. Muitas equipes de SRE fazem rodízio dessa função entre os vários membros da equipe, o que tem a vantagem de fazer com que

todos sintam que têm participação no serviço e um senso de responsabilidade sobre os problemas. É verdade que nem todos têm níveis iguais de habilidade para conduzir a reunião, mas o valor da responsabilidade do grupo é tão importante que ficar temporariamente abaixo do ideal é uma negociação que vale a pena. Além do mais, essa é uma oportunidade para desenvolver habilidades de liderança, o que é muito útil em situações de coordenação de incidentes comumente enfrentadas pela SRE.

Nos casos em que duas equipes de SRE fizerem uma reunião por vídeo, e uma das equipes for bem maior que a outra, temos observado uma dinâmica interessante em ação. Recomendamos colocar o líder da reunião do lado *menor* da chamada, por padrão. O lado maior naturalmente tenderá a ficar mais tranquilo e alguns dos efeitos ruins de tamanhos de equipe desiguais (que se tornarão piores pelos atrasos inerentes à videoconferência) melhorarão.² Não temos a menor ideia se essa técnica tem qualquer base científica, mas ela tende a funcionar.

Agenda

Há muitas maneiras de conduzir uma reunião de produção, atestando a diversidade daquilo pelo qual a SRE é responsável e como o fazemos. Para isso, não é apropriado ser prescritivo acerca de como conduzir uma dessas reuniões. No entanto, uma agenda-padrão (veja o Apêndice F, que apresenta um exemplo) pode ter um aspecto como este:

Mudanças que estão por vir em produção

Reuniões para acompanhamento das mudanças são bem conhecidas no mercado e, de fato, reuniões inteiras muitas vezes são dedicadas a interromper as mudanças. Entretanto, em nosso ambiente de produção, geralmente temos como padrão permitir as mudanças, o que exige monitorar o conjunto útil de propriedades dessas mudanças: horário de início, duração, efeitos esperados, e assim por diante. Isso representa a visibilidade do horizonte no curto prazo.

Métricas

Uma das principais maneiras de conduzir uma discussão orientada a

serviços é conversar sobre as métricas principais dos sistemas em questão; veja o Capítulo 4. Mesmo que os sistemas não tenham falhado drasticamente naquela semana, é muito comum estar em uma posição em que você está observando um aumento de carga gradual (ou muito acentuado!) ao longo do ano. Monitorar como os valores para a sua latência, a utilização de CPU etc. se alteram com o tempo é extremamente importante para desenvolver uma noção do desempenho geral de um sistema.

Algumas equipes monitoram o uso de recursos e a eficiência, que também são um indicador útil de mudanças mais lentas, talvez mais insidiosas, no sistema.

Interrupções de serviço

Esse item aborda problemas aproximadamente do tamanho de um postmortem, e é uma oportunidade indispensável de aprendizado. Uma boa análise de postmortem, conforme discutida no Capítulo 15, deve sempre manter a equipe dinâmica.

Eventos de paging

São os pages de seu sistema de monitoração, relacionados a problemas que *podem* fazer com que um postmortem valha a pena, mas, com frequência, não. Em qualquer evento, enquanto a parte referente a Interrupções de Serviço observa o quadro maior de uma interrupção, essa seção observa a visão tática: a lista de pages, quem recebeu o page, o que aconteceu então, e assim por diante. Há duas perguntas implícitas nessa seção: esse alerta deveria ter gerado um page da forma como o fez, e ele deveria realmente ter gerado um page? Se a resposta à última pergunta for não, elimine esses pages para os quais não há uma ação a ser disparada.

Eventos que não geram pages

Esse conjunto contém três itens:

- *Um problema que provavelmente deveria ter gerado um page, mas não o fez.* Nesses casos, você deve provavelmente corrigir a monitoração de modo que esses eventos gerem um page. Com frequência, você se deparará com esse caso enquanto estiver tentando corrigir outro

problema, ou ele estará relacionado a uma métrica que você esteja monitorando, mas para a qual não tinha um alerta.

- *Um problema que não gera pages, mas exige atenção*, como dados de baixo impacto corrompidos ou lentidão em alguma dimensão do sistema, que seja visível aos usuários. Monitorar tarefas operacionais reativas também é apropriado nesse caso.
- *Um problema que não gera pages e não exige atenção*. Esses alertas devem ser removidos, pois criam ruídos extras que distraem os engenheiros dos problemas que realmente merecem atenção.

Lista de ações prévias

As discussões detalhadas anteriores muitas vezes levam a ações que a SRE deve executar – corrigir isso, monitorar aquilo, desenvolver um subsistema para fazer mais aquilo. Acompanhe essas melhorias, assim como elas seriam acompanhadas em qualquer outra reunião: atribua listas de ações às pessoas e monitore seu progresso. Ter um item explícito na agenda que englobe tudo, se não houver outros itens, é uma boa ideia. Uma entrega consistente também é uma ótima forma de desenvolver credibilidade e confiança. Não importa como essa entrega é feita, desde que ela seja feita.

Participação

A participação é compulsória para todos os membros da equipe de SRE em questão. Isso é particularmente verdadeiro se sua equipe estiver espalhada em vários países e/ou fusos horários, pois essa é sua principal oportunidade de interagir como um grupo.

Os principais stakeholders também devem participar dessa reunião. Qualquer equipe de desenvolvimento de produtos parceira que você possa ter também deve participar. Algumas equipes de SRE dividem suas reuniões de modo que questões ligadas apenas à SRE sejam discutidas na primeira metade; não há problemas com essa prática, desde que todos, conforme afirmamos antes, saiam com a mesma ideia do que está acontecendo. Ocassionalmente, representantes de outras equipes de SRE podem comparecer, em particular se houver algum problema maior a ser discutido que envolva diferentes equipes,

mas, em geral, a equipe de SRE em questão e mais outras equipes importantes devem participar. Se seu relacionamento for tal que você não possa convidar seus parceiros em desenvolvimento de produtos, será preciso rever esse relacionamento: talvez o primeiro passo seja convidar um representante dessa equipe ou encontrar um intermediário de confiança para fazer a comunicação ou modelar interações saudáveis. Há muitos motivos pelos quais as equipes não se dão bem, e uma variedade de textos sobre como resolver esse problema: essa informação também é aplicável às equipes de SRE, mas é importante que o objetivo final de ter um ciclo de feedback das operações seja atingido; do contrário, uma boa parte do valor de ter uma equipe de SRE será perdida.

Ocasionalmente, você terá equipes demais ou participantes ocupados, porém de extrema importância, para convidar. Há uma série de técnicas que você pode usar para lidar com essas situações:

- Serviços menos ativos podem ter um único representante da equipe de desenvolvimento de produto ou pode haver apenas um comprometimento dessa equipe no sentido de ler e comentar as minutas da agenda.
- Se a equipe de desenvolvimento de produção for muito grande, nomeie um subconjunto de representantes.
- Participantes ocupados, porém muito importantes, podem oferecer feedback e/ou orientação com antecedência aos indivíduos ou usar as técnicas de agenda previamente preenchida (descritas a seguir).

A maior parte das estratégias de reunião que discutimos pertence ao senso comum, com um traço orientado a serviços. Um passo único para deixar as reuniões mais eficientes e inclusivas é usar as funcionalidades de colaboração em tempo real do Google Docs. Muitas equipes de SRE têm um documento desse tipo, com um endereço bem conhecido, que qualquer pessoa da engenharia possa acessar. Ter um documento assim permite duas práticas excelentes:

- Preencher previamente a agenda com ideias, comentários e informações “de baixo para cima”.
- Preparar a agenda em paralelo e com antecedência é realmente eficiente.

Utilize totalmente as funcionalidades de colaboração entre várias pessoas, permitidas pelo produto. Não há nada como ver um responsável pela reunião digitar uma frase e, então, ver outra pessoa fornecer um link para o material-fonte entre colchetes depois que ele termina de digitar e, em seguida, ver outra pessoa ainda corrigir a ortografia e a gramática da frase original. Uma colaboração como essa agiliza as tarefas e faz com que mais pessoas se sintam responsáveis por uma parte do que a equipe faz.

Colaboração na SRE

Obviamente, o Google é uma empresa multinacional. Por causa dos componentes de resposta a emergências e rodízio de pager em sua função, temos muitos bons motivos de negócio para sermos uma empresa distribuída, separada pelo menos por alguns fusos horários. O impacto prático dessa distribuição é que temos definições muito fluidas de “equipe”, quando comparadas, por exemplo, com a equipe média de desenvolvimento de produtos. Temos equipes locais, a equipe do site, equipes em mais de um continente, equipes virtuais de vários tamanhos e coerência, e tudo que estiver entre elas. Isso cria uma mistura empolgantemente caótica de responsabilidades, habilidades e oportunidades. Muitas das mesmas dinâmicas poderiam ser esperadas em qualquer empresa suficientemente grande (embora possam ser particularmente intensas em empresas de tecnologia). Considerando que a maior parte das colaborações locais não enfrenta nenhum obstáculo em particular, o caso interessante de colaboração ocorre entre equipes, entre sites, em uma equipe virtual e em casos semelhantes.

Esse padrão de distribuição também oferece a base para o modo como as equipes de SRE tendem a se organizar. Como nossa *razão de ser* é agregar valor por meio do domínio técnico, e esse domínio tende a ser difícil, tentamos, desse modo, encontrar uma maneira de ter um domínio sobre algum subconjunto relacionado dos sistemas ou infraestruturas a fim de reduzir a carga cognitiva. A especialização é uma forma de atingir esse objetivo, isto é, a equipe X trabalha somente no produto Y. A especialização é boa porque leva a maiores chances de ter um domínio técnico melhor, mas

também é ruim, pois resulta na formação de silos e no desconhecimento do panorama mais amplo. Tentamos ter uma composição sólida da equipe para definir a que ela dará suporte – e, acima de tudo, a que ela não dará –, mas nem sempre temos sucesso.

Composição da equipe

Temos uma ampla variedade de conjuntos de habilidades em SRE, que vai de engenharia de sistemas a engenharia de software, passando por organização e gerenciamento. O único ponto que podemos afirmar sobre a colaboração é que suas chances de uma colaboração bem-sucedida – e, na verdade, praticamente para tudo o mais – melhorarão se houver mais diversidade em sua equipe. Há muitas evidências que sugerem que equipes diversificadas simplesmente são equipes melhores [Nel14]. Ter uma equipe diversificada implica prestar atenção em particular à comunicação, aos vieses cognitivos e assim por diante, que não poderemos discutir 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 ou PgM)³. Algumas pessoas trabalham melhor quando essas funções têm responsabilidades bem definidas: a principal vantagem disso é que elas podem tomar decisões rápidas e seguras no escopo. Outras trabalham melhor em um ambiente mais fluido, com responsabilidades que mudam conforme a negociação da dinâmica. Em geral, quanto mais fluida a equipe, mais desenvolvida ela será em termos de capacidade dos indivíduos, e mais capaz ela será de se adaptar a novas situações – porém, à custa de ter que se comunicar mais e com mais frequência, pois menos contexto poderá ser pressuposto.

Independentemente de quão bem essas funções estejam definidas, em um nível básico, o líder técnico é responsável pela direção técnica da equipe, e pode conduzir essa função de várias maneiras – pode fazer de tudo, desde comentar cuidadosamente o código de todos, fazer apresentações trimestrais sobre a direção ou fazer com que a equipe chegue a um consenso. No Google, os TLs podem fazer quase tudo que caiba a um cargo de gerente, pois 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 o responsável por tudo que não for tratado por outra pessoa. Ótimos TLs, SRMs e TPMs têm um conjunto completo de habilidades e podem alegremente ajudar a organizar um projeto, comentar um documento de design ou escrever código, conforme for necessário.

Técnicas para trabalhar de modo eficiente

Há várias maneiras de fazer engenharia de modo eficiente na SRE.

Em geral, projetos de uma única pessoa falham, a menos que essa pessoa seja particularmente habilidosa ou o problema seja simples. Para realizar algo significativo, você precisa basicamente de várias pessoas. Portanto, precisará também de boas habilidades de colaboração. Novamente, muitos materiais foram escritos sobre esse assunto, e boa parte dessa literatura se aplica à SRE.

Em geral, um bom trabalho de SRE exige excelentes habilidades de comunicação quando você estiver trabalhando fora das fronteiras de sua equipe puramente local. Para colaborações fora de seu prédio, trabalhar de modo eficiente em fusos horários distintos implica uma ótima comunicação por escrito ou muitas viagens para proporcionar uma experiência pessoal, que podem ser postergadas, mas, em última instância, são necessárias para ter um relacionamento de alta qualidade. Mesmo que você seja um ótimo redator, com o tempo, acabará reduzido a apenas um endereço de email, até se fazer presente novamente em carne e osso.

Estudo de caso sobre colaboração em SRE: o Viceroy

Um exemplo de colaboração bem-sucedida entre SREs é um projeto chamado Viceroy, que é um framework para painel de monitoração e um serviço. A arquitetura organizacional atual da SRE pode fazer com que as equipes acabem produzindo várias cópias um pouco diferentes do mesmo trabalho; por diversos motivos, os frameworks de painéis de monitoração foram um terreno particularmente fértil para duplicação de trabalho.⁴

Os incentivos que levaram ao problema sério de lixo composto de muitas

porções abandonadas, em estado latente, de frameworks de monitoração espalhados por aí eram bem simples: cada equipe era recompensada por desenvolver sua própria solução, trabalhar fora das fronteiras da equipe era difícil e a infraestrutura que tendia a ser oferecida à SRE como um todo geralmente se assemelhava mais a um kit de ferramentas do que a um produto. Esse ambiente incentivava engenheiros individuais a usar o kit de ferramentas para criar outra solução tapa-buracos, em vez de corrigir o problema para o maior número possível de pessoas (um esforço que, desse modo, exigiria muito mais tempo).

O surgimento do Viceroy

O Viceroy era diferente. Ele teve início em 2012, quando algumas equipes estavam considerando a forma de passar para o Monarch – o novo sistema de monitoração do Google. A SRE é profundamente conservadora no que diz respeito aos sistemas de monitoração, portanto o Monarch, de certo modo, ironicamente demorou um bom tempo para ter impulso na SRE em relação a outras equipes que não eram de SRE. Porém, ninguém poderia argumentar que nosso sistema de monitoração legado – o Borgmon (veja o Capítulo 10) – não tivesse espaço para melhorias. Por exemplo, nossos consoles eram inconvenientes, pois usavam um sistema de templating HTML personalizado que era um caso especial, cheio de situações extremas inusitadas e difícil de testar. Naquela época, o Monarch havia amadurecido o suficiente para ser aceito, em princípio, como substituto para o sistema legado e, assim, estava sendo adotado por mais e mais equipes no Google, mas o fato é que ainda tínhamos um problema com os consoles.

Aqueles de nós que tentaram usar o Monarch para os nossos serviços logo descobriram que ele deixava a desejar no suporte a consoles por dois motivos principais:

- Os consoles eram fáceis de configurar para um serviço de pequeno porte, mas não escalavam bem para serviços com consoles complexos.
- Também não havia suporte para o sistema de monitoração legado, dificultando bastante a transição para o Monarch.

Como, naquela época, não havia nenhuma alternativa viável para implantar o

Monarch dessa maneira, uma série de projetos específicos para cada equipe foi iniciada. Como poucas atitudes no sentido de coordenar soluções de desenvolvimento ou até mesmo um acompanhamento entre grupos diferentes tivessem sido tomadas (um problema que, desde então, foi corrigido), acabamos duplicando esforços novamente. Várias equipes do Spanner, do Ads Frontend e de uma variedade de outros serviços investiram seus próprios esforços (um exemplo digno de nota se chamava Consoles++) no curso de 12 a 18 meses e, em algum momento, a sanidade prevaleceu quando os engenheiros de todas essas equipes acordaram e descobriram os respectivos esforços. Eles decidiram fazer o que era sensato e juntar forças a fim de criar uma solução genérica para toda a SRE. Assim, o projeto Viceroy nasceu em meados de 2012.

No início de 2013, o Viceroy havia começado a despertar o interesse das equipes que ainda precisavam deixar o sistema legado, mas estavam com medo de colocar um pé na água. Obviamente, as equipes com projetos de monitoração maiores já existentes tinham menos incentivos para passar para o novo sistema: era difícil para essas equipes pensar em deixar de lado o custo baixo de manutenção de sua solução atual que, basicamente, funcionava bem, e trocar por algo relativamente novo e não comprovado, que exigiria muito esforço para fazer funcionar. A enorme diversidade dos requisitos contribuiu com a relutância dessas equipes, apesar de todos os projetos de console de monitoração compartilharem dois requisitos principais, notadamente:

- Suporte a painéis de controle complexos e organizados
- Suporte tanto ao Monarch quanto ao sistema de monitoração legado

Cada projeto *também* tinha seu próprio conjunto de requisitos técnicos, que dependia da preferência ou da experiência do autor. Por exemplo:

- Várias fontes de dados além dos sistemas de monitoração principais
- Definição de consoles usando configuração *versus* layout HTML explícito
- Sem JavaScript *versus* adoção completa de JavaScript com AJAX
- Uso exclusivo de conteúdos estáticos, para que o navegador pudesse fazer cache dos consoles

Embora alguns desses requisitos fossem mais convincentes do que outros, em geral, eles dificultaram os esforços de combinação das soluções. Com efeito, embora a equipe do Consoles++ estivesse interessada em ver como seu projeto se comparava ao Viceroy, sua análise inicial na primeira metade de 2013 determinou que as diferenças fundamentais entre os dois projetos eram significativas o suficiente para impedir uma integração. A maior dificuldade estava no fato de o Viceroy, por design, não usar muito JavaScript, enquanto o Consoles++ havia sido escrito, em sua maior parte, em JavaScript. Houve uma centelha de esperança, porém, pois os dois sistemas tinham realmente uma série de semelhanças subjacentes:

- Usavam sintaxes semelhantes para renderização de template HTML.
- Compartilhavam uma série de metas de longo prazo, que nenhuma das equipes havia começado a abordar. Por exemplo, os dois sistemas queriam fazer cache dos dados de monitoração e oferecer suporte a um pipeline periódico offline para gerar dados que o console pudesse usar, mas que eram custosos do ponto de vista de processamento para serem gerados por demanda.

Acabamos suspendendo a discussão sobre o console unificado por um tempo. No entanto, no final de 2013, tanto o Consoles++ quanto o Viceroy haviam se desenvolvido de forma significativa. Suas diferenças técnicas tinham diminuído, pois o Viceroy havia começado a usar JavaScript para apresentar seus gráficos de monitoração. As duas equipes se reuniram e descobriram que a integração seria muito mais simples, agora que ela se reduzia a servir dados ao Consoles++ a partir do servidor do Viceroy. Os primeiros protótipos integrados foram concluídos no início de 2014, e provaram que os sistemas poderiam funcionar bem em conjunto. As duas equipes se sentiram à vontade para se comprometer com um esforço conjunto naquela época, e como o Viceroy já havia consolidado a sua marca como uma solução comum de monitoração, o projeto combinado preservou o nome Viceroy. Desenvolver todas as funcionalidades exigiu alguns trimestres, mas no final de 2014 o sistema combinado estava completo.

Reunir forças proporcionou grandes vantagens:

- O Viceroy recebeu muitas fontes de dados e os clientes JavaScript para

acessá-los.

- A compilação de JavaScript foi reescrita para tratar módulos separados que pudessem ser incluídos seletivamente. Isso é essencial para escalar o sistema para qualquer número de equipes com seu próprio código JavaScript.
- O Consoles++ se beneficiou das muitas melhorias feitas ativamente no Viceroy, como a adição de seu cache e do pipeline de dados em background.
- Em geral, a velocidade de desenvolvimento em *uma* solução foi muito maior que a soma das velocidades de todos os desenvolvimentos dos projetos duplicados.

Em última análise, a visão comum de futuro foi o fator essencial para combinar os projetos. As duas equipes viram a importância de expandir sua equipe de desenvolvimento e se beneficiaram com as contribuições uma da outra. O ímpeto foi tal que, no final de 2014, o Viceroy foi oficialmente declarado como a solução geral de monitoração para toda a SRE. Talvez, de modo característico para o Google, essa declaração não exigia que todas as equipes adotassem o Viceroy: em vez disso, é recomendável que as equipes o utilizem, em vez de escrever outro console de monitoração.

Desafios

Embora tenha sido, em última instância, um sucesso, o Viceroy não deixou de passar por dificuldades, e muitas delas surgiram devido à natureza do projeto de envolver mais de uma localidade.

Depois que a equipe ampliada do Viceroy foi definida, a coordenação inicial entre membros de equipes remotas se provou ser difícil. Logo que conhecemos as pessoas, sinais sutis na escrita e na fala podem ser erroneamente interpretados, pois os estilos de comunicação variam substancialmente de pessoa para pessoa. No início do projeto, os membros da equipe que não estavam em Mountain View também deixaram de participar das discussões improvisadas ao lado do bebedouro, que ocorriam com frequência imediatamente antes e depois das reuniões (embora a comunicação, desde então, tenha melhorado de forma considerável).

Embora a equipe nuclear do Viceroy permanecesse razoavelmente consistente, a equipe ampliada de colaboradores era bem dinâmica. Os colaboradores tinham outras responsabilidades que mudavam com o tempo e, assim, muitos eram capazes de dedicar entre um a três meses ao projeto. Desse modo, o pool de desenvolvedores que colaboravam, o qual era inherentemente maior que a equipe nuclear do Viceroy, era caracterizado por um volume significativo de rotação.

Acrescentar novas pessoas ao projeto exigia o treinamento de cada colaborador sobre o design e a estrutura geral do sistema, o que demandava certo tempo. Por outro lado, quando um SRE contribuía com a funcionalidade nuclear do Viceroy e depois retornava à sua própria equipe, ele passava a ser um expert local para o sistema. Essa disseminação não prevista de experts locais em Viceroy resultou em mais uso e adoção.

À medida que as pessoas se juntavam e saíam da equipe, percebemos que as contribuições casuais eram tanto úteis quanto custosas. O custo principal era a diluição da responsabilidade: depois que as funcionalidades eram entregues e a pessoa saía, essas funcionalidades ficavam sem suporte com o passar do tempo e, em geral, eram descartadas.

Além do mais, o escopo do projeto Viceroy aumentou com o tempo. Ele tinha metas ambiciosas no lançamento, porém o *escopo* inicial era limitado. Porém, à medida que o escopo aumentou, lutamos para entregar as funcionalidades essenciais a tempo, e tivemos que melhorar o gerenciamento do projeto, além de definir uma direção mais clara para garantir que o projeto permanecesse nos trilhos.

Por fim, a equipe do Viceroy achou difícil ser totalmente responsável por um componente que havia recebido contribuições significativas (e determinantes) de localidades distribuídas. Mesmo com a melhor disposição do mundo, as pessoas, em geral, acabam adotando o caminho da menor resistência e discutem problemas ou tomam decisões localmente, sem o envolvimento dos responsáveis remotos, o que pode gerar conflitos.

Recomendações

Você só deve desenvolver projetos envolvendo mais de uma localidade

quando for necessário, mas, muitas vezes, há bons motivos para isso. O custo de trabalhar em localidades diferentes é uma latência mais alta para as ações e mais necessidade de comunicação; a vantagem é ter um throughput mais elevado – se você dominar o mecanismo da forma correta. O projeto em um único local também pode ter a desvantagem de ninguém fora desse local saber o que você está fazendo, portanto há custos em ambas as abordagens.

Colaboradores motivados são valiosos, mas nem todos são igualmente valiosos. Garanta que os colaboradores do projeto estejam realmente comprometidos, e que não estejam se juntando a ele devido a alguma meta nebulosa de atualização própria (querendo ganhar um ponto ao associar seus nomes a um projeto de destaque, querendo programar em um novo projeto empolgante sem se comprometer com a sua manutenção). Colaboradores com uma meta específica a ser alcançada geralmente estarão mais motivados e darão uma manutenção melhor às suas contribuições.

À medida que os projetos se desenvolvem, em geral, eles crescem, e você nem sempre estará na posição feliz de ter pessoas em sua equipe local para contribuir com eles. Portanto, pense com cuidado na estrutura do projeto. Os líderes do projeto são importantes: eles proporcionam uma visão de longo prazo para o projeto e garantem que todo o trabalho esteja alinhado com essa visão e tenha prioridades atribuídas corretamente. Você também precisa ter uma maneira consensual de tomar decisões, e deve otimizar especificamente para que mais decisões sejam tomadas localmente se houver um alto nível de concordância e de confiança.

A estratégia-padrão de “dividir e conquistar” se aplica a projetos que envolvam localidades diferentes; você reduz os custos de comunicação principalmente separando o projeto no máximo de componentes possíveis de tamanhos razoáveis, e tentando garantir que cada componente possa ser atribuído a um grupo pequeno, de preferência em uma localidade. Divida esses componentes entre as subequipes do projeto e defina entregas e prazos claros. (Tente não deixar que a lei de Conway desfigure demais a forma natural do software.)⁵

Uma meta para uma equipe de projeto funciona melhor quando é orientada a oferecer alguma funcionalidade ou resolver algum problema. Essa abordagem

garante que os indivíduos que trabalhem em um componente saibam o que é esperado deles e que seu trabalho só estará concluído quando esse componente estiver totalmente integrado e for utilizado no projeto principal.

Obviamente, as melhores práticas usuais de engenharia se aplicam em projetos colaborativos: cada componente deve ter documentos de design e revisões com a equipe. Dessa maneira, todos na equipe terão a oportunidade de estar cientes das mudanças, além da chance de influenciar e melhorar os designs. Anotar por escrito é uma das principais técnicas que você tem para reduzir a distância física e/ou lógica – utilize-a.

Padrões são importantes. Diretrizes de estilo de programação são um bom ponto de partida, mas geralmente elas são bem táticas e, portanto, servem apenas como ponto de partida para definir normas para a equipe. Sempre que houver um debate em torno da escolha a ser feita acerca de um problema, discuta de forma completa com a equipe, mas com um tempo limite rigoroso. Em seguida, escolha uma solução, documente-a e prossiga. Se não puder chegar a um acordo, será necessário escolher algum árbitro que todos respeitem e, novamente, seguir em frente. Com o tempo, você reunirá uma coleção dessas melhores práticas, que ajudarão novas pessoas a se preparar.

Em última instância, não há substitutos para uma interação pessoal, embora parte da interação face a face possa ser postergada por um bom uso de conversa por voz e por uma boa comunicação por escrito. Se puder, faça com que os líderes do projeto conheçam o restante da equipe pessoalmente. Se o tempo e o orçamento permitirem, organize uma reunião da equipe para que todos os membros possam interagir pessoalmente. Uma reunião também oferece uma ótima oportunidade para discutir designs e metas. Em situações em que a neutralidade seja importante, é vantajoso conduzir reuniões de equipe em um local neutro para que nenhuma localidade em particular tenha a “vantagem de estar em casa”.

Por fim, utilize o estilo de gerenciamento de projeto que seja mais adequado ao projeto em seu estado atual. Mesmo projetos com metas ambiciosas começarão pequenos, portanto o overhead deverá ser igualmente baixo. À medida que o projeto crescer, será apropriado adaptar e mudar a forma de gerenciá-lo. Dado um crescimento suficiente, um gerenciamento completo de

projeto será necessário.

Colaboração fora da SRE

Conforme sugerimos (e será discutido no Capítulo 32), a colaboração entre a organização de desenvolvimento de produtos e a SRE é melhor quando ocorre precocemente na fase de design, e o ideal é que ocorra antes que um commit de qualquer linha de código seja feito. Os SREs estão em uma posição melhor para fazer recomendações sobre a arquitetura e o comportamento do software que possam ser bem difíceis de readaptar (se não for impossível). Ter essa voz presente na sala quando um novo sistema é projetado é melhor para todos. Falando de modo geral, usamos o processo OKR (Objectives & Key Results, ou Objetivos & Resultados Essenciais) [Kla12] para o acompanhamento dessa tarefa. Para algumas equipes de serviços, uma colaboração desse tipo é a parte principal do que elas fazem – acompanhar novos designs, fazer recomendações, ajudar a implementá-las e fazer um acompanhamento até a produção.

Estudo de caso: migração do DFP para F1

Projetos grandes de migração de serviços existentes são bem comuns no Google. Exemplos típicos incluem portar componentes de serviços para uma nova tecnologia ou atualizar componentes para que tratem um novo formato de dados. Com a introdução recente das tecnologias de bancos de dados capazes de escalar a um nível global, como o Spanner [Cor12] e o F1 [Shu13], o Google assumiu uma série de projetos de migração de larga escala envolvendo bancos de dados. Um desses projetos foi a migração do banco de dados principal do DoubleClick for Publishers (DFP)⁶, do MySQL para o F1. Em particular, alguns dos autores deste capítulo eram responsáveis por uma parte do sistema servidor (mostrada na Figura 31.1) que extraía e processava continuamente dados do banco de dados, a fim de gerar um conjunto de arquivos indexados que eram então carregados e servidos ao mundo. Esse sistema estava distribuído em vários datacenters e utilizava aproximadamente 1.000 CPUs e 8 TB de RAM para indexar 100 TB de dados diariamente.

A migração não era trivial: além de migrar para uma nova tecnologia, o esquema do banco de dados era significativamente refatorado e simplificado graças à capacidade do F1 de armazenar e indexar dados do buffer de protocolo em colunas de tabelas. A meta era migrar o sistema processador para que ele pudesse gerar uma saída perfeitamente idêntica àquela do sistema existente. Isso nos permitiria deixar o sistema servidor inalterado e realizar, do ponto de vista do usuário, uma migração tranquila. Como uma restrição adicional, o produto exigia que concluíssemos uma migração ao vivo, sem qualquer perturbação no serviço aos nossos usuários, em nenhum instante. Para conseguir isso, a equipe de desenvolvimento do produto e a equipe de SRE começaram a trabalhar de maneira próxima, desde o início, a fim de desenvolver um novo serviço de indexação.

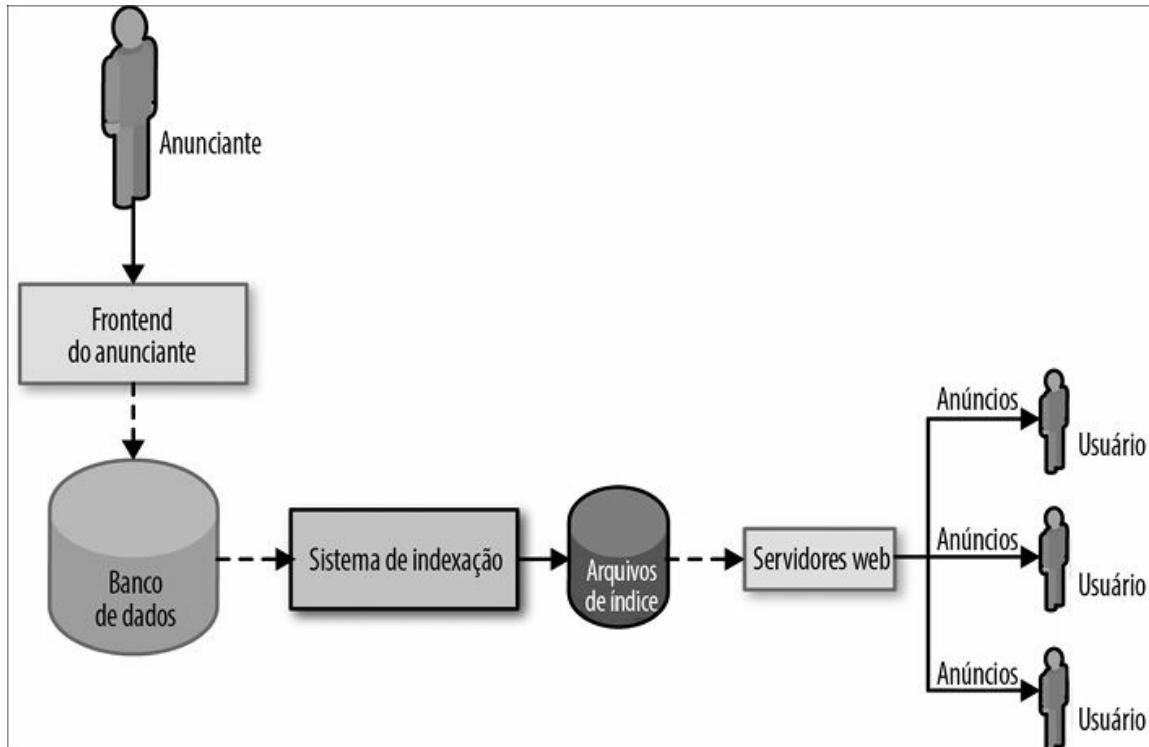


Figura 31.1 – Um sistema genérico para servir anúncios.

Por ser seus principais desenvolvedores, as equipes de desenvolvimento de produto geralmente têm mais familiaridade com a BL (Business Logic, ou Lógica de Negócios) do software, além de estar em contato mais próximo com os Gerentes de Produto e com o componente propriamente dito de “necessidades do negócio” dos produtos. Por outro lado, as equipes de SRE

geralmente têm mais expertise no que diz respeito aos componentes da infraestrutura do software (por exemplo, bibliotecas para conversar com sistemas distribuídos de armazenagem ou bancos de dados), pois os SREs com frequência reutilizam os mesmos blocos de construção em diferentes serviços, conhecendo as diversas ressalvas e nuances que permitem que o software execute de forma escalável e confiável no tempo.

Desde o início do projeto de migração, o desenvolvimento de produtos e a SRE sabiam que precisariam colaborar de forma mais próxima ainda, conduzindo reuniões semanais a fim de se sincronizarem quanto ao progresso do projeto. Nesse caso em particular, as mudanças na BL eram parcialmente dependentes das mudanças na infraestrutura. Por esse motivo, o projeto teve início com o design da nova infraestrutura; os SREs, que tinham um amplo conhecimento do domínio de extração e processamento de dados em escala, conduziram o design das mudanças de infraestrutura. Isso envolveu o design de como extrair as várias tabelas do F1, como filtrar e reunir os dados, como extrair somente os dados que mudaram (em oposição ao banco de dados completo), como sustentar a perda de algumas das máquinas sem impactar o serviço, como garantir que o uso de recursos aumentasse linearmente com a quantidade de dados extraídos, o planejamento de capacidade e muitos outros aspectos semelhantes. A nova infraestrutura proposta era semelhante à de outros serviços que já extraíam e processavam dados do F1. Assim, poderíamos ter certeza da robustez da solução e reutilizar partes da monitoração e das ferramentas.

Antes de prosseguir com o desenvolvimento dessa nova infraestrutura, dois SREs geraram um documento de design detalhado. Em seguida, tanto a equipe de desenvolvimento de produtos quanto a de SRE revisaram completamente o documento, ajustando a solução para que tratasse alguns casos extremos e, posteriormente, chegaram a um acordo sobre um plano de design. Um plano como esse claramente identificava quais tipos de mudanças a nova infraestrutura traria para a BL. Por exemplo, projetamos a nova infraestrutura para extrair apenas os dados alterados, em vez de extrair repetidamente todo o banco de dados; a BL tinha que levar em conta essa nova abordagem. Com antecedência, definimos as novas interfaces entre a infraestrutura e a BL, e fazer isso permitiu que a equipe de desenvolvimento

de produto trabalhava de modo independente nas mudanças da BL. De modo semelhante, a equipe de desenvolvimento de produto mantinha a SRE informada das mudanças na BL. Nos pontos em que elas interagiam (por exemplo, mudanças na BL dependentes da infraestrutura), essa estrutura de coordenação nos permitia saber que as alterações estavam ocorrendo e também tratá-las de modo rápido e correto.

Nas fases posteriores do projeto, os SREs começaram a implantar o novo serviço em um ambiente de teste que se assemelhava ao futuro ambiente de produção final do projeto. Esse passo foi essencial para avaliar o comportamento esperado do serviço – em particular, o desempenho e a utilização de recursos – ao mesmo tempo que o desenvolvimento da BL ainda estava em andamento. A equipe de desenvolvimento de produto usou esse ambiente de teste para realizar a validação do novo serviço: o índice dos anúncios gerados pelo serviço antigo (executando em produção) deveria coincidir perfeitamente com o índice gerado pelo novo serviço (executando no ambiente de teste). Conforme suspeitávamos, o processo de validação revelou discrepâncias entre o serviço antigo e o novo (por causa de alguns casos extremos no novo formato de dados), que a equipe de desenvolvimento de produto foi capaz de resolver iterativamente; para cada anúncio, eles depuraram a causa da diferença e corrigiram a BL que havia gerado a saída ruim. Nesse ínterim, a equipe de SRE começou a preparar o ambiente de produção: alocando os recursos necessários em um datacenter diferente, configurando processos e regras de monitoração e treinando os engenheiros designados para estar de plantão no serviço. A equipe de SRE também definiu um processo básico de entrega de versão que incluía validação – uma tarefa geralmente concluída pela equipe de desenvolvimento de produto ou pelos Engenheiros de Release, mas que, nesse caso específico, foi feita pelos SREs para agilizar a migração.

Quando o serviço estava pronto, os SREs preparam um plano de rollout, em conjunto com a equipe de desenvolvimento de produto, e lançaram o novo serviço. O lançamento foi um grande sucesso e ocorreu tranquilamente, sem qualquer impacto visível aos usuários.

Conclusão

Considerando a natureza globalmente distribuída das equipes de SRE, uma comunicação eficiente tem sido sempre uma alta prioridade nessa organização. Este capítulo discutiu as ferramentas e as técnicas que as equipes de SRE usam para manter relacionamentos eficientes em suas equipes e com as várias equipes parceiras.

A colaboração entre as equipes de SRE tem seus desafios, mas também tem ótimas recompensas em potencial, incluindo abordagens comuns a plataformas para resolução de problemas, o que nos permite focar na resolução de problemas mais difíceis.

¹ E, como todos sabemos, a cultura vence a estratégia o tempo todo: [Mer11].

² A equipe maior geralmente tende a convencer involuntariamente a equipe menor, é mais difícil controlar conversas paralelas que provoquem distrações etc.

³ N.T.: TL = Tech Lead, ou Líder Técnico; SRM = System Resource Manager, ou Gerente de Recursos do Sistema; PM = Project Manager, ou Gerente de Projeto; TPM = Technical Product Manager, ou Gerente Técnico de Produto; PgM = Program Manager, ou Gerente de Programa.

⁴ Nesse caso em particular, o caminho para o inferno, na verdade, foi pavimentado com JavaScript.

⁵ Isto é, o software tem a mesma estrutura que a estrutura das comunicações da organização que o produz – veja https://en.wikipedia.org/wiki/Conway%27s_law.

⁶ DoubleClick for Publishers é uma ferramenta para os anunciantes administrarem propagandas servidas em seus sites e em suas aplicações.

CAPÍTULO 32

O modelo de engajamento da SRE em evolução

Escrito por Acacio Cruz e Ashish Bhambhani

Editado por Betsy Beyer e Tim Harvey

Engajamento da SRE: o quê, como e por quê

Discutimos, na maior parte do restante deste livro, o que acontece quando a SRE já é responsável por um serviço. Poucos serviços iniciam seu ciclo de vida desfrutando do suporte da SRE, portanto deve haver um processo para avaliar um serviço, garantindo que ele mereça esse suporte, negociando como melhorar qualquer deficiência que impeça o suporte da SRE e realmente instituindo esse suporte. Chamamos a esse processo de integração (*onboarding*). Se você estiver em um ambiente cercado de muitos serviços existentes em estados variados de perfeição, sua equipe de SRE provavelmente trabalhará com uma fila de integrações com prioridades por um bom tempo, até que a equipe termine de tratar os alvos de maior valor.

Embora isso seja muito comum, além de ser uma maneira totalmente razoável de lidar com um ambiente que já seja um *fato consumado*, na verdade, há pelo menos duas melhores formas de levar a sabedoria adquirida em produção e o suporte da SRE aos serviços antigos e, igualmente, aos novos.

No primeiro caso, assim como em engenharia de software – em que quanto mais cedo um bug for encontrado, mais barato será corrigi-lo –, quanto antes a consultoria com a equipe de SRE ocorrer, melhor será o serviço e mais rápido ele sentirá as vantagens. Quando a SRE é envolvida durante as etapas iniciais de *design*, o tempo para integração será menor e o serviço será mais

confiável “de imediato”, geralmente porque não precisaremos gastar tempo para corrigir um design ou uma implementação que deixem a desejar.

Outra maneira, talvez a melhor, é fazer um curto-circuito no processo pelo qual sistemas especialmente criados, com muitas variações individuais, acabam “chegando” às portas da SRE. Ofereça ao desenvolvimento de produtos uma *plataforma* com uma infraestrutura validada pela SRE, sobre a qual eles possam desenvolver seus sistemas. Essa plataforma terá a dupla vantagem de ser confiável e escalável. Isso evita totalmente determinadas classes de problemas de carga cognitiva e, ao usar práticas voltadas a uma infraestrutura comum, permite que as equipes de desenvolvimento de produtos se concentrem em inovações na camada de aplicação, à qual, em sua maior parte, elas pertencem.

Nas próximas seções, investiremos um tempo analisando cada um desses modelos individualmente, começando pelo modelo “clássico”, que é o modelo orientado a PRR.

O modelo PRR

O passo inicial mais característico do engajamento da SRE é a PRR (Production Readiness Review, ou Revisão de Prontidão para Produção) – um processo que identifica as necessidades de confiabilidade de um serviço com base em seus detalhes específicos. Por meio da PRR, os SREs procuram aplicar o que aprenderam e vivenciaram para garantir a confiabilidade de um serviço que opere em produção. Uma PRR é considerada um pré-requisito para uma equipe de SRE aceitar a responsabilidade de administrar os aspectos de produção de um serviço.

A Figura 32.1 ilustra o ciclo de vida de um serviço típico. A PRR pode ser iniciada em qualquer ponto do ciclo de vida de um serviço, mas os estágios em que o engajamento da SRE é aplicado se expandiram com o passar do tempo. Este capítulo descreve o Modelo Simples de PRR (Simple PRR Model), e então discute como a sua modificação para o Modelo de Engajamento Estendido (Extended Engagement Model) e a estrutura de Frameworks e Plataforma de SRE (Frameworks and SRE Platform) permitem à SRE escalar seu processo de engajamento e o impacto.

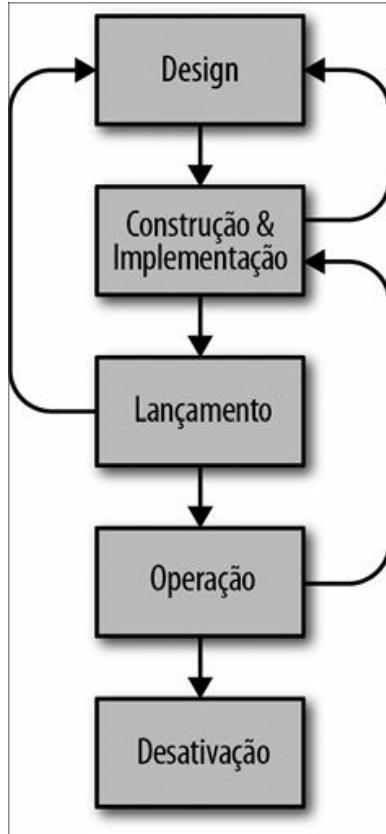


Figura 32.1 – Um ciclo de vida típico de um serviço.

O modelo de engajamento da SRE

A SRE procura ter responsabilidade de produção em serviços importantes para os quais ela possa fazer contribuições concretas à confiabilidade. A SRE se preocupa com vários aspectos de um serviço que, coletivamente, são chamados de *produção*. Esses aspectos incluem:

- Arquitetura do sistema e dependências entre serviços
- Instrumentação, métricas e monitoração
- Respostas a emergências
- Planejamento de capacidade
- Gerenciamento de mudanças
- Desempenho: disponibilidade, latência e eficiência

Quando os SREs se envolvem com um serviço, procuramos melhorá-lo ao longo de todos esses eixos, o que facilita administrar o serviço em produção.

Suporte alternativo

Nem todos os serviços do Google têm um engajamento próximo da SRE. Dois fatores influenciam nesse caso:

- Muitos serviços não precisam de alta confiabilidade e disponibilidade, portanto o suporte pode ser dado por outros meios.
- Por design, o número de equipes de desenvolvimento que exijam o suporte da SRE excede a capacidade disponível das equipes de SRE (veja o Capítulo 1).

Quando a SRE não é capaz de oferecer um suporte completo, ela apresentará outras opções para melhorias em produção (por exemplo, documentação e consultoria).

Documentação

Manuais de desenvolvimento estão disponíveis para tecnologias internas e clientes de sistemas amplamente utilizados. Os Manuais de Produção (Production Guide) do Google documentam as melhores práticas de produção para os serviços, conforme determinadas pelas experiências da SRE e, igualmente, pelas equipes de desenvolvimento. Os desenvolvedores podem implementar as soluções e as recomendações contidas nessa documentação a fim de aperfeiçoar seus serviços.

Consultoria

Os desenvolvedores também podem procurar consultoria junto à SRE para discutir serviços ou áreas de problema específicos. A equipe de LCE (Launch Coordination Engineering, ou Engenharia de Coordenação de Lançamentos) – veja o Capítulo 27 – investe a maior parte de seu tempo dando consultoria às equipes de desenvolvimento. As equipes de SRE que não estejam especificamente dedicadas às consultorias em lançamento também se envolvem em consultoria para as equipes de desenvolvimento.

Quando um novo serviço ou uma nova funcionalidade são implementados, os desenvolvedores geralmente consultam a SRE para receber conselhos sobre a preparação da fase de Lançamento. A consultoria para lançamento em geral envolve um ou dois SREs, que passam algumas horas estudando o design e a

implementação em nível geral. Os consultores da SRE então se reúnem com a equipe de desenvolvimento a fim de oferecer conselhos sobre áreas de risco que mereçam atenção e discutir padrões ou soluções bem conhecidos que possam ser incorporados a fim de melhorar o serviço em produção. Parte desses conselhos pode vir dos Manuais de Produção mencionados anteriormente.

As sessões de consultoria são necessariamente amplas em escopo, pois não é possível ter uma compreensão profunda de um dado sistema no tempo limitado disponível. Para algumas equipes de desenvolvimento, a consultoria não é suficiente:

- Serviços que cresceram em ordens de grandeza desde que foram lançados, e agora exigem mais tempo para serem compreendidos do que é viável por meio de documentação e consultoria;
- Serviços com os quais vários outros serviços passaram subsequentemente a contar, e que agora tratam mais tráfego de muitos clientes diferentes, de forma significativa.

Esses tipos de serviços podem ter crescido até o ponto em que começam a se deparar com dificuldades significativas em produção, ao mesmo tempo que passaram a ser importantes para os usuários. Em casos assim, um envolvimento de longo prazo da SRE se torna necessário para garantir que esses serviços sejam apropriadamente mantidos em produção à medida que crescem.

Revisões de Prontidão para Produção: Modelo Simples de PRR

Quando uma equipe de desenvolvimento pede que a SRE assuma o gerenciamento de um serviço em produção, a SRE avalia tanto a importância do serviço quanto a disponibilidade das equipes de SRE. Se o serviço merecer o suporte da SRE, e a equipe de SRE e de desenvolvimento chegarem a um acordo quanto ao número de pessoas para facilitar esse suporte, a SRE iniciará uma Revisão de Prontidão para Produção (Production Readiness Review) com a equipe de desenvolvimento.

Os objetivos da Revisão de Prontidão para Produção são:

- Verificar se um serviço atende aos padrões aceitos de configuração de produção e prontidão para operação, e se os proprietários do serviço estão preparados para trabalhar com a SRE e tirar proveito de seu expertise.
- Melhorar a confiabilidade do serviço em produção e minimizar o número e a severidade dos incidentes que possam ser esperados. Uma PRR tem como alvo todos os aspectos de produção com os quais a SRE se importa.

Após melhorias suficientes terem sido feitas e o serviço for considerado pronto para ter o suporte da SRE, uma equipe de SRE assumirá a responsabilidade pelo serviço em produção.

Isso nos leva ao processo de Revisão de Prontidão para Produção propriamente dito. Há três modelos de engajamento diferentes, porém relacionados (Modelo Simples de PRR [Simple PRR Model], Modelo de Engajamento Precoce [Early Engagement Model] e Frameworks e Plataforma de SRE [Frameworks and SRE Platform]), que serão discutidos um a um.

Inicialmente, descreveremos o Modelo Simples de PRR, que geralmente é voltado a um serviço que já tenha sido lançado e cuja responsabilidade será assumida por uma equipe de SRE. Uma PRR segue várias etapas, de modo muito semelhante ao ciclo de vida de um desenvolvimento, embora possa ocorrer independentemente, em paralelo com o ciclo de vida do desenvolvimento.

Engajamento

A liderança da SRE inicialmente decide qual equipe de SRE é uma boa escolha para assumir o serviço. Geralmente, de um a três SREs são selecionados ou se apresentam como voluntários para conduzir o processo de PRR. Esse pequeno grupo, então, inicia a discussão com a equipe de desenvolvimento. A discussão inclui questões como:

- Determinar um SLO/SLA para o serviço
- Planejar mudanças que possam, em potencial, desestruturar o design, mas que são necessárias para melhorar a confiabilidade
- Cronogramas de planejamento e treinamento

O objetivo é chegar a um acordo sobre o processo, os objetivos finais e os resultados necessários para que a equipe de SRE se envolva com a equipe de desenvolvimento e com o seu serviço.

Análise

A análise é o primeiro grande segmento de trabalho. Durante essa fase, os analistas da SRE conhecem o serviço e passam a analisá-lo quanto às deficiências em produção. Eles procuram avaliar a maturidade do serviço ao longo dos vários eixos de preocupação da SRE. Também analisam o design e a implementação do serviço a fim de verificar se esses seguem as melhores práticas de produção. Geralmente, a equipe de SRE define e mantém uma checklist de PRR explicitamente para a fase de Análise. A checklist é específica ao serviço e, em geral, é baseada na expertise do domínio, na experiência com sistemas relacionados ou semelhantes e nas melhores práticas do Manual de Produção. A equipe de SRE também pode consultar outras equipes que tenham mais experiência com determinados componentes ou dependências do serviço.

Alguns exemplos de itens da checklist incluem:

- As atualizações no serviço têm impacto em um grande percentual não razoável do sistema, de uma só vez?
- O serviço se conecta com a instância servidora apropriada de suas dependências? Por exemplo, as requisições do usuário final a um serviço não devem depender de um sistema projetado para um caso de uso de processamento em batch.
- O serviço solicita uma qualidade de serviço de rede suficientemente alta quando conversa com um serviço remoto crítico?
- O serviço informa erros a sistemas centrais de logging para análise? Ele informa todas as condições excepcionais que resultem em respostas degradadas ou falhas aos usuários finais?
- Todas as falhas de requisições visíveis aos usuários estão bem instrumentadas e monitoradas, com geração adequada de alertas configurada?

A checklist também pode incluir padrões operacionais e as melhores práticas seguidas por uma equipe de SRE específica. Por exemplo, uma configuração de serviço perfeitamente funcional que não siga o “padrão de ouro” de uma equipe de SRE pode ser refatorada para funcionar melhor com as ferramentas de SRE a fim de gerenciar as configurações de forma escalável. Os SREs também observam incidentes recentes e postmortems do serviço, assim como as tarefas de acompanhamento associadas a esses incidentes. Esse processo avalia as demandas da resposta a emergências para o serviço e a disponibilidade de controles operacionais bem definidos.

Melhorias e refatoração

A fase de Análise leva à identificação de melhorias recomendadas para o serviço. Essa fase seguinte ocorre desta maneira:

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 acordo é feito acerca de um plano de execução.
3. Tanto a equipe de SRE quanto a de desenvolvimento de produto participam e dão assistência uma à outra na refatoração de partes do serviço ou na implementação de funcionalidades adicionais.

Essa fase geralmente varia mais quanto à duração e ao volume de esforço. Quanto tempo e esforço essa fase envolverá depende da disponibilidade do tempo de engenharia para a refatoração, da maturidade e da complexidade do serviço no início da revisão e de uma variedade de outros fatores.

Treinamento

A responsabilidade em administrar um serviço em produção geralmente é assumida por uma equipe de SRE completa. Para garantir que a equipe esteja preparada, os analistas da SRE que conduzem a PRR assumem a responsabilidade pelo treinamento da equipe, que inclui a documentação necessária para dar suporte ao serviço. Geralmente, com a ajuda e a participação da equipe de desenvolvimento, esses engenheiros organizam uma série de sessões de treinamento e exercícios. O treinamento pode incluir:

- Visão geral do design
- Explorações profundas de vários fluxos de requisição no sistema
- Uma descrição da configuração de produção
- Exercícios práticos para vários aspectos da operação do sistema

Quando o treinamento estiver concluído, a equipe de SRE deverá estar preparada para administrar o serviço.

Integração à SRE

A fase de Treinamento habilita a integração do serviço pela equipe de SRE. Ela envolve uma transferência progressiva das responsabilidades de vários aspectos de produção do serviço, incluindo parte das operações, o processo de gerenciamento de mudanças, direitos de acesso, e assim por diante. A equipe de SRE continua a se concentrar nas várias áreas de produção mencionadas antes. Para completar a transição, a equipe de desenvolvimento deve estar disponível para dar suporte e conselhos à equipe de SRE por um período de tempo, enquanto ela se acomoda para administrar o serviço em produção. Esse relacionamento passa a ser a base para o trabalho contínuo entre as equipes.

Melhorias contínuas

Serviços ativos mudam continuamente em resposta a novas exigências e condições, incluindo solicitações de usuários por novas funcionalidades, evolução das dependências do sistema e upgrades de tecnologia, além de outros fatores. A equipe de SRE deve manter os padrões de confiabilidade do serviço diante dessas mudanças conduzindo melhorias contínuas. A equipe de SRE responsável naturalmente aprende mais sobre o serviço ao operá-lo, analisando novas mudanças, respondendo a incidentes e, em especial, quando conduz postmortems/análises de causas-raízes. Essa expertise é compartilhada com a equipe de desenvolvimento como sugestões e propostas para mudanças no serviço sempre que novas funcionalidades, componentes e dependências possam ser acrescentados ao serviço. As lições aprendidas ao administrar o serviço também colaboraram com as melhores práticas, que são documentadas no Manual de Produção e em outros lugares.

Engajamento com o Shakespeare

Inicialmente, os desenvolvedores do serviço Shakespeare eram responsáveis pelo produto, incluindo portar o pager para resposta a emergências. No entanto, com o uso crescente do serviço e o aumento de sua receita, o suporte da SRE tornou-se desejável. O produto já havia sido lançado, portanto a SRE conduziu uma Revisão de Prontidão para Produção. Um dos aspectos que eles identificaram foi que os painéis de controle não estavam incluindo totalmente algumas das métricas definidas no SLO, portanto isso precisaria ser corrigido. Após todos os problemas registrados terem sido corrigidos, a SRE assumiu a responsabilidade pelo pager do serviço, embora dois desenvolvedores também estivessem no rodízio de plantão. Os desenvolvedores participam da reunião semanal do plantão para discutir os problemas da última semana e o modo de tratar manutenções de larga escala que estejam por vir ou desativações de clusters. Além disso, planos futuros para o serviço agora são discutidos também com os SREs para garantir que novos lançamentos ocorram sem falhas (embora a Lei de Murphy sempre esteja procurando oportunidades para atrapalhar).

Evolução do Modelo Simples de PRR: Engajamento Precoce

Até agora, discutimos a Revisão de Prontidão para Produção conforme usada no Modelo Simples de PRR, limitado a serviços que já tenham entrado na fase de Lançamento. Há várias limitações e custos associados a esse modelo. Por exemplo:

- Uma comunicação adicional entre as equipes pode aumentar o overhead de algum processo para a equipe de desenvolvimento e a carga cognitiva para os analistas na SRE.
- Os analistas corretos na SRE devem estar disponíveis e ser capazes de administrar seu tempo e suas prioridades em relação aos engajamentos existentes.

- O trabalho feito pelos SREs deve ser altamente visível e suficientemente revisado pela equipe de desenvolvimento para garantir um compartilhamento de conhecimento eficiente. Os SREs devem essencialmente trabalhar como parte da equipe de desenvolvimento, em vez de ser uma unidade externa.

No entanto, as principais limitações do Modelo de PRR têm origem no fato de o serviço já ter sido lançado e estar servindo em escala, e o envolvimento da SRE começar muito tarde no ciclo de vida do desenvolvimento. Se a PRR tivesse ocorrido mais cedo no ciclo de vida do serviço, a oportunidade da SRE de remediar possíveis problemas do serviço aumentaria de forma significativa. Como resultado, o sucesso do engajamento da SRE e o futuro sucesso do próprio serviço provavelmente seriam maiores. As desvantagens resultantes podem representar um desafio significativo para o sucesso do engajamento da SRE e para o futuro sucesso do próprio serviço.

Candidatos a um Engajamento Precoce

O Modelo de Engajamento Precoce (Early Engagement Model) introduz a SRE mais cedo no ciclo de vida do desenvolvimento a fim de obter vantagens adicionais significativas. Aplicar o Modelo de Engajamento Precoce exige identificar a importância e/ou valor de um serviço ao negócio mais cedo no ciclo de vida do desenvolvimento e determinar se o serviço terá escala ou complexidade suficientes para se beneficiar com a expertise da SRE. Serviços aplicáveis geralmente têm as seguintes características:

- O serviço implementa uma funcionalidade significativa nova e fará parte de um sistema já existente, administrado pela SRE.
- O serviço é uma reescrita significativa ou uma alternativa a um sistema existente, voltado aos mesmos casos de uso.
- A equipe de desenvolvimento procurou o conselho da SRE ou abordou-a para assumir o lançamento.

O Modelo de Engajamento Precoce essencialmente faz com que os SREs mergulhem no processo de desenvolvimento. O foco da SRE permanece o mesmo, embora os meios para conseguir um serviço melhor em produção sejam diferentes. A SRE participa do Design e das fases seguintes e, em

algum momento, assume o serviço, em qualquer instante durante ou após a fase de Construção. Esse modelo é baseado em uma colaboração ativa entre as equipes de desenvolvimento e de SRE.

Vantagens do Modelo de Engajamento Precoce

Embora o Modelo de Engajamento Precoce implique certos riscos e desafios discutidos anteriormente, a expertise adicional e a colaboração da SRE durante todo o ciclo de vida do produto proporcionam vantagens significativas se comparadas a um envolvimento mais tardio no ciclo de vida do serviço.

Fase de Design

A colaboração da SRE durante a fase de Design pode impedir que uma variedade de problemas ou incidentes ocorra mais tarde em produção. Embora as decisões de design possam ser revertidas ou corrigidas posteriormente no ciclo de vida do desenvolvimento, essas mudanças têm um custo alto em termos de esforço e de complexidade. Os melhores incidentes em produção são aqueles que jamais ocorrem!

Ocasionalmente, negociações difíceis levam à seleção de um design abaixo do ideal. A participação na fase de Design significa que os SREs estão previamente cientes das negociações e fazem parte da decisão de escolher uma opção abaixo da ideal. O envolvimento precoce da SRE tem como meta minimizar discussões futuras sobre opções de design, depois que o serviço estiver em produção.

Construção e implementação

A fase de Construção trata aspectos da produção como instrumentação e métricas, controles operacionais e emergenciais, uso de recursos e eficiência. Durante essa fase, a SRE pode influenciar e melhorar a implementação recomendando bibliotecas e componentes específicos existentes ou ajudando a desenvolver determinados controles no sistema. A participação da SRE nessa fase ajuda a facilitar as operações no futuro e permite que ela tenha uma experiência operacional anterior ao lançamento.

Lançamento

A SRE também pode ajudar a implementar padrões de lançamento e controles amplamente utilizados. Por exemplo, a SRE pode ajudar a implementar uma configuração de “lançamento oculto” (dark launch), em que parte do tráfego de usuários existentes é enviada ao novo serviço, além de ser enviada ao serviço ativo em produção. As respostas desse novo serviço são “ocultas”, pois são descartadas e não são realmente apresentadas aos usuários. Práticas como lançamentos ocultos permitem que a equipe adquira um insight operacional, resolva problemas sem que haja impacto nos usuários existentes e reduza o risco de se deparar com problemas após o lançamento. Um lançamento tranquilo é extremamente conveniente para manter a carga operacional baixa e fazer o desenvolvimento prosseguir após o lançamento. As perturbações em torno do lançamento podem facilmente resultar em mudanças emergenciais no código-fonte e na produção, e desestruturar o trabalho da equipe de desenvolvimento para funcionalidades futuras.

Pós-lançamento

Ter um sistema estável no momento do lançamento em geral leva a menos prioridades conflitantes para a equipe de desenvolvimento no que diz respeito a escolher entre melhorar a confiabilidade do serviço *versus* adicionar novas funcionalidades. Nas fases posteriores do serviço, as lições das fases anteriores podem dar uma base melhor para a refatoração e para uma revisão do design.

Com um envolvimento estendido, a equipe de SRE pode estar pronta para assumir o novo serviço muito mais cedo do que seria possível com um Modelo Simples de PRR. Um envolvimento mais longo e mais próximo entre as equipes de SRE e de desenvolvimento também cria um relacionamento de colaboração que pode ser sustentado no longo prazo. Um relacionamento positivo entre as equipes promove um sentimento mútuo de solidariedade e ajuda a SRE a assumir a responsabilidade pela produção.

Desligando-se de um serviço

Às vezes, um serviço não justifica um gerenciamento completo pela equipe de SRE – essa constatação pode ser feita após o lançamento, ou a SRE pode

se envolver com um serviço, mas jamais assumi-lo oficialmente. Esse é um resultado positivo, pois significa que o serviço foi planejado para ser confiável e ter baixa manutenção, e desse modo pode permanecer com a equipe de desenvolvimento.

Também é possível que a SRE se envolva cedo com um serviço que falhe em atender aos níveis de uso projetados. Em casos como esse, o esforço da SRE simplesmente faz parte do risco geral do negócio associado a novos projetos, e representa um pequeno custo associado ao sucesso dos projetos que atendam à escala esperada. A equipe de SRE pode receber uma nova atribuição, e as lições aprendidas podem ser incorporadas no processo de engajamento.

Desenvolvimento de serviços em evolução: frameworks e plataforma de SRE

O Modelo de Engajamento Precoce representou um grande passo na evolução do engajamento da SRE para além do Modelo Simples de PRR, que se aplicava somente a serviços que já haviam sido lançados. No entanto, ainda havia progressos a serem feitos para escalar o engajamento da SRE para o próximo nível ao fazer o design visando à confiabilidade.

Lições aprendidas

Com o tempo, o modelo de engajamento da SRE descrito até agora gerou vários padrões distintos:

- A integração de cada serviço exigia dois ou três SREs e, geralmente, durava dois ou três trimestres. O tempo de antecedência para uma PRR era relativamente alto (trimestres antes). O nível de esforço exigido era proporcional ao número de serviços em análise, e era limitado pelo número insuficiente de SREs disponíveis para conduzir as PRRs. Essas condições levaram à serialização para assumir a responsabilidade pelos serviços e a uma atribuição rigorosa de prioridade aos serviços.
- Devido às diferentes práticas de software entre os serviços, cada funcionalidade em produção era implementada de modo distinto. Para

atender aos padrões orientados a PRR, as funcionalidades geralmente tinham que ser reimplementadas especificamente para cada serviço ou, no melhor caso, uma vez para cada pequeno subconjunto de serviços que compartilhava o código. Essas reimplementações eram um desperdício de esforço de engenharia. Um exemplo canônico é a implementação de frameworks de logging funcionalmente semelhantes, de forma repetida e na mesma linguagem, pois diferentes serviços não implementavam a mesma estrutura de código.

- Uma análise dos problemas e das interrupções de serviço comuns revelou determinados padrões, mas não havia nenhuma maneira fácil de replicar as correções e as melhorias entre os serviços. Exemplos típicos incluíam situações de sobrecarga dos serviços e hot-spotting de dados.
- As contribuições da engenharia de software da SRE frequentemente eram locais ao serviço. Assim, desenvolver soluções genéricas para serem reutilizadas era difícil. Como consequência, não havia um modo fácil de implementar as novas lições aprendidas pelas equipes individuais de SRE e as melhores práticas nos serviços que já tivessem sido integrados à SRE.

Fatores externos que afetam a SRE

Fatores externos tradicionalmente exerceram pressão na organização de SRE e em seus recursos de várias maneiras.

O Google cada vez mais segue as tendências de mercado em direção aos microsserviços.¹ Como resultado, tanto o número de solicitações para suporte de SRE quanto a cardinalidade dos serviços a serem tratados têm aumentado. Como cada serviço tem um custo operacional básico fixo, mesmo serviços simples exigem mais funcionários. Os microsserviços também implicam uma expectativa de tempo prévio menor para implantação, que não era possível com o modelo anterior de PRR (que exigia um tempo prévio de meses).

Contratar SREs experientes e qualificados é difícil e custoso. Apesar do enorme esforço do departamento de recrutamento, nunca há SREs suficientes para dar suporte a todos os serviços que necessitem de sua expertise. Depois que os SREs são contratados, seu treinamento também é um processo mais demorado que um processo típico para os engenheiros de desenvolvimento.

Por fim, a organização de SRE é responsável por atender às necessidades do número elevado e crescente de equipes de desenvolvimento que ainda não desfruta diretamente do suporte da SRE. Isso exige estender o modelo de suporte da SRE para muito além do conceito e do modelo de engajamento originais.

Em direção a uma solução estrutural: Frameworks

Para responder de modo eficiente a essas condições, tornou-se necessário desenvolver um modelo que abrangesse os princípios a seguir:

Melhores práticas traduzidas em código

A capacidade de transformar o que funciona bem em produção em código, de modo que os serviços possam simplesmente utilizar esse código e estar “prontos para produção”, por design.

Soluções reutilizáveis

Implementações de técnicas comuns e facilmente compartilháveis, usadas para atenuar os problemas de escalabilidade e de confiabilidade.

Uma plataforma comum de produção com uma superfície de controle comum

Conjuntos uniformes de interfaces para instalações em produção, conjuntos uniformes de controles operacionais, além de monitoração, logging e configuração uniformes para todos os serviços.

Automação mais fácil e sistemas mais inteligentes

Uma superfície comum de controle que permita ter automação e sistemas mais inteligentes em um nível que não era possível antes. Por exemplo, os SREs podem receber prontamente uma única visão das informações relevantes de uma interrupção de serviço, em vez de coletar manualmente e analisar dados, em sua maioria, brutos, de fontes dispare (logs, dados de monitoração, e assim por diante).

Com base nesses princípios, um conjunto de frameworks de plataforma e de serviços com suporte da SRE foi criado, um para cada ambiente ao qual damos suporte (Java, C++, Go). Os serviços criados com base nesses frameworks compartilham implementações concebidas para funcionar com a

plataforma à qual a SRE dá suporte, e são mantidos tanto pela equipe de SRE quanto pela equipe de desenvolvimento. A principal mudança proporcionada pelos frameworks foi permitir que as equipes de desenvolvimento de produtos fizessem o design das aplicações usando a solução de framework desenvolvida e abençoada pela SRE, em oposição a readaptar a aplicação conforme as especificações feitas posteriormente pela SRE, ou readaptar mais SREs para dar suporte a um serviço notadamente diferente de outros serviços do Google.

Uma aplicação geralmente é composta de uma lógica de negócios que, por sua vez, depende de vários componentes da infraestrutura. As preocupações da SRE com a produção estão concentradas, em sua maioria, nas partes de um serviço relacionadas à infraestrutura. Os frameworks de serviços implementam um código de infraestrutura de forma padronizada e tratam várias das preocupações de produção. Cada preocupação está encapsulada em um ou mais módulos de framework, em que cada um oferece uma solução coesa para um domínio de problema ou uma dependência da infraestrutura. Os módulos dos frameworks tratam as diversas preocupações da SRE mencionadas antes, como:

- Instrumentação e métricas
- Logging de requisições
- Sistemas de controle que envolvam tráfego e gerenciamento de carga

A SRE desenvolve módulos de framework para implementar soluções canônicas para a área de produção com a qual ela está preocupada. Como resultado, as equipes de desenvolvimento podem se concentrar na lógica de negócios, pois o framework já cuida do uso correto da infraestrutura.

Um framework essencialmente é uma implementação prescritiva para usar um conjunto de componentes de software e uma forma canônica de combinar esses componentes. O framework também pode expor funcionalidades que controlam vários componentes de forma coesa. Por exemplo, um framework pode oferecer o seguinte:

- Lógica de negócios organizada como componentes semânticos bem definidos, que podem ser referenciados usando termos padrões

- Dimensões padrões para instrumentação de monitoração
- Um formato-padrão para logs de requisições para depuração
- Um formato de configuração padrão para administrar o descarte de carga
- Capacidade de um único servidor e determinação de “sobrecarga” que possam usar uma medida semanticamente consistente para feedback de vários sistemas de controle

Os frameworks oferecem vários ganhos prévios em termos de consistência e de eficiência. Eles livram os desenvolvedores de ter que unir e configurar componentes individuais de forma *ad hoc* a um serviço específico, de maneiras sempre levemente incompatíveis e que, então, precisam ser manualmente revisadas pelos SREs. Os frameworks apresentam uma solução única e reutilizável para as preocupações de produção aos serviços, o que significa que os usuários dos frameworks acabam com a mesma implementação comum e com diferenças mínimas de configuração.

O Google trata várias linguagens importantes para o desenvolvimento de aplicações, e os frameworks são implementados em todas essas linguagens. Embora diferentes implementações do framework (por exemplo, em C++ *versus* Java) não possam compartilhar códigos, a meta é expor uma API, um comportamento, uma configuração e controles iguais para uma funcionalidade idêntica. Assim, as equipes de desenvolvimento podem escolher a plataforma de linguagem que seja adequada às suas necessidades e experiência, enquanto os SREs podem esperar o mesmo comportamento familiar em produção, além de terem ferramentas padrões para administrar o serviço.

Novo serviço e vantagens do gerenciamento

A abordagem estrutural encontrada nos frameworks de serviço, além de uma plataforma e uma superfície de controle comuns em produção, oferece uma série de novas vantagens.

Overhead operacional significativamente menor

Uma plataforma de produção construída com base nos frameworks, com convenções fortes, reduziu significativamente o overhead operacional, pelos

seguintes motivos:

- Aceita testes robustos de conformidade para estrutura de códigos, dependências, testes, diretrizes para estilo de programação, e assim por diante. Essa funcionalidade também melhora a privacidade dos dados de usuários, os testes e a conformidade com a segurança.
- Oferece implantação de serviços, monitoração e automação embutidos para todos os serviços.
- Facilita o gerenciamento de um grande número de serviços, especialmente de microsserviços que estão aumentando cada vez mais.
- Permite implantações muito mais rápidas: uma ideia pode evoluir gradualmente até se transformar em um sistema totalmente implantado, com qualidade de produção no nível de SRE, em questão de dias!

Suporte universal por design

O constante crescimento do número de serviços no Google implica que a maioria desses serviços pode não justificar o engajamento da SRE nem ser mantida por ela. Independentemente disso, os serviços que não recebam suporte total da SRE podem ser construídos de modo a usar as funcionalidades de produção desenvolvidas e mantidas pelos SREs. Essa prática efetivamente acaba com a barreira de contratar funcionários para a SRE. Permitir que padrões e ferramentas de produção com suporte da SRE sejam usados por todas as equipes melhora a qualidade geral do serviço no Google. Além do mais, todos os serviços implementados com frameworks se beneficiam automaticamente das melhorias feitas com o passar do tempo nos módulos dos frameworks.

Engajamentos mais rápidos, com menos overhead

A abordagem com frameworks resulta em uma execução mais rápida de PRR, pois podemos contar com:

- Funcionalidades de serviços embutidos como parte da implementação dos frameworks
- Integração mais rápida dos serviços à SRE (geralmente, acompanhada por um único SRE durante um trimestre)

- Menos carga cognitiva para as equipes de SRE que administram os serviços desenvolvidos com os frameworks

Essas propriedades permitem às equipes de SRE reduzir os esforços de avaliação e de qualificação para a integração de serviços à SRE, ao mesmo tempo que mantêm um nível alto de exigência quanto à qualidade do serviço em produção.

Um novo modelo de engajamento baseado em responsabilidade compartilhada

O modelo original de engajamento da SRE apresentava somente duas opções: um suporte completo da SRE ou aproximadamente nenhum engajamento por parte dela.²

Uma plataforma de produção com uma estrutura de serviços, convenções e infraestrutura de software comuns possibilitou que uma equipe de SRE oferecesse suporte para a infraestrutura da “plataforma”, enquanto as equipes de desenvolvimento oferecem suporte de plantão para problemas funcionais com o serviço – isto é, bugs no código da aplicação. Nesse modelo, os SREs assumem a responsabilidade pelo desenvolvimento e pela manutenção de grandes partes da infraestrutura de software dos serviços, particularmente dos sistemas de controle, como descarte de carga, sobrecarga, automação, gerenciamento de tráfego, logging e monitoração.

Esse modelo representa um afastamento significativo do modo como o gerenciamento de serviço havia sido originalmente concebido, de duas maneiras: implica um novo modelo de relacionamento para a interação entre as equipes de SRE e de desenvolvimento e um novo modelo de provisão de funcionários para o gerenciamento de serviços com suporte da SRE.³

Conclusão

A confiabilidade do serviço pode ser melhorada por meio do engajamento da SRE, em um processo que inclua revisões sistemáticas e melhoria de seus aspectos de produção. Uma abordagem inicial sistemática desse tipo adotada pela SRE do Google – a Revisão Simples de Prontidão para Produção – deu grandes passos em direção à padronização do modelo de engajamento da

SRE, mas era aplicável somente a serviços que já haviam entrado na fase de Lançamento.

Com o tempo, a SRE estendeu e aperfeiçoou esse modelo. O Modelo de Engajamento Precoce envolveu a SRE mais cedo no ciclo de vida de desenvolvimento para que houvesse um “design voltado à confiabilidade”. À medida que a demanda pela expertise da SRE continuava a aumentar, a necessidade de um modelo de engajamento mais escalável tornou-se cada vez mais evidente. Frameworks para serviços em produção foram desenvolvidos para atender a essa demanda: padrões de código baseados nas melhores práticas de produção foram definidos e encapsulados nos frameworks, de modo que o uso deles passou a ser uma maneira relativamente simples, recomendada e consistente de desenvolver serviços prontos para a produção.

Todos os três modelos de engajamento descritos continuam em prática no Google. No entanto, a adoção de frameworks está se tornando uma influência predominante no desenvolvimento de serviços prontos para a produção no Google, além de estar expandindo profundamente a contribuição da SRE, reduzindo o overhead do gerenciamento dos serviços e melhorando a qualidade do serviço básico na empresa.

¹ Veja a página da Wikipédia sobre microsserviços em <http://en.wikipedia.org/wiki/Microservices>.

² Ocasionalmente, havia envolvimento das equipes de SRE para consultoria em alguns serviços não integrados a ela, mas as consultorias eram uma abordagem de melhor esforço, limitadas em número e em escopo.

³ O novo modelo de gerenciamento de serviços altera o modelo de provisão de funcionários na SRE de duas maneiras: (1) como muitas tecnologias de serviços são comuns, o número de SREs necessário por serviço pode ser reduzido; (2) permite a criação de plataformas de produção com separação de preocupações, divididas entre suporte à plataforma de produção (feito pelos SREs) e suporte à lógica de negócios específica de um serviço, que permanece com a equipe de desenvolvimento. Essas equipes de plataforma têm funcionários com base na necessidade de manutenção da plataforma, e não no número de serviços, e podem ser compartilhadas entre diferentes produtos.

PARTE V

Conclusões

Após termos abordado muitos aspectos que dizem respeito ao modo como a SRE funciona no Google, e como os princípios e as práticas que desenvolvemos podem ser aplicados a outras empresas em nossa área, agora parece ser apropriado nos voltarmos ao Capítulo 33, *Lições aprendidas de outros mercados*, para analisar as práticas de SRE comparadas a outros mercados em que a confiabilidade é extremamente importante.

Por último, Benjamin Lutch, VP do Google para Site Reliability Engineering (Engenharia de Confiabilidade de Sites), escreve sobre a evolução da SRE no curso de sua carreira em sua conclusão, analisando a SRE através das lentes de algumas observações sobre a indústria de aviação.

CAPÍTULO 33

Lições aprendidas com outros mercados

Escrito por Jennifer Petoff

Editado por Betsy Beyer

Um mergulho profundo na cultura e nas práticas de SRE no Google leva naturalmente à questão de como outros mercados administram seus negócios para ter confiabilidade. Compilar este livro sobre a SRE do Google criou uma oportunidade para conversar com vários engenheiros dessa empresa sobre suas experiências de trabalho anteriores em uma variedade de outras áreas que exigem alta confiabilidade a fim de abordar as seguintes perguntas comparativas:

- Os princípios usados na Site Reliability Engineering (Engenharia de Confiabilidade de Sites) também são importantes fora do Google, ou outros mercados enfrentam os requisitos de alta confiabilidade de formas notadamente diferentes?
- Se outros mercados também são aderentes aos princípios de SRE, como esses princípios se manifestam?
- Quais são as semelhanças e as diferenças na implementação desses princípios nos diferentes mercados?
- Quais fatores determinam as semelhanças e as diferenças de implementação?
- O que o Google e o mercado de tecnologia podem aprender com essas comparações?

Uma série de princípios fundamentais da Site Reliability Engineering no Google será discutida neste capítulo. Para simplificar nossa comparação

sobre as melhores práticas em outros mercados, separamos esses conceitos em quatro temas:

- Preparo e testes para desastres
- Cultura de postmortem
- Automação e overhead operacional reduzido
- Tomada de decisão estruturada e racional

Este capítulo apresenta os mercados dos quais traçamos o perfil e os veteranos desses mercados que entrevistamos. Definiremos os temas essenciais para a SRE, discutiremos como esses temas são implementados no Google e daremos exemplos de como esses princípios se revelam em outros mercados para fazermos uma comparação. Concluímos com alguns insights e uma discussão sobre os padrões e antipadrões que identificamos.

Conheça os veteranos do mercado

Peter Dahl é um Principal Engineer (Engenheiro-chefe) no Google. Anteriormente, trabalhou como contratado em defesa para vários sistemas de alta confiabilidade, incluindo GPSs para muitos veículos aéreos e com rodas e sistemas de orientação por inércia. As consequências de um lapso em confiabilidade em sistemas como esses incluem mau funcionamento ou perda do veículo, além das consequências financeiras associadas a essa falha.

Mike Doherty é Site Reliability Engineer (Engenheiro de Confiabilidade de Sites) no Google. Trabalhou como salva-vidas e treinador de salva-vidas durante uma década no Canadá. A confiabilidade é absolutamente essencial por natureza nessa área, pois vidas estão em risco todos os dias.

Erik Gross atualmente é engenheiro de software no Google. Antes de se juntar à empresa, passou sete anos fazendo design de algoritmos e códigos para lasers e sistemas usados para realizar cirurgias oculares refrativas (por exemplo, LASIK). É uma área que exige alta confiabilidade, com muita coisa em jogo, em que várias lições relevantes para a confiabilidade diante de regulamentações governamentais e risco a seres humanos foram aprendidas à medida que a tecnologia recebia a aprovação do FDA, foi gradualmente melhorada e, por fim, se tornou presente em todos os lugares.

Gus Hartmann e Kevin Greer têm experiência no mercado de telecomunicações, incluindo a manutenção do sistema de resposta a emergências E911.¹ Atualmente, Kevin é engenheiro de software na equipe do Google Chrome e Gus é engenheiro de sistemas na equipe de Corporate Engineering (Engenharia Corporativa) do Google. As expectativas dos usuários no mercado de telecomunicações exigem alta confiabilidade. As implicações de um lapso no serviço variam da inconveniência ao usuário por causa de uma interrupção no sistema até fatalidades caso o E911 fique fora do ar.

Ron Heiby é Technical Program Manager (Gerente Técnico de Programa) para a Site Reliability Engineering no Google. Ron tem experiência no desenvolvimento de celulares, dispositivos médicos e no mercado automotivo. Em algumas ocasiões, trabalhou em componentes de interface nesses mercados (por exemplo, em um dispositivo para permitir que leituras de EKG² em ambulâncias fossem transmitidas por meio da rede telefônica digital sem fios). Nesses mercados, o impacto de um problema de confiabilidade pode variar de danos aos negócios como consequência de recalls de equipamentos até um impacto indireto em vidas e na saúde (por exemplo, pessoas que não conseguem obter a atenção médica necessária se o EKG não puder ser informado ao hospital).

Adrian Hilton é Launch Coordination Engineer (Engenheiro de Coordenação de Lançamentos) no Google. Anteriormente, trabalhou com aeronaves militares britânicas e norte-americanas, aviação naval e em sistemas de gerenciamento de armazenagens de aeronaves, além de ter atuado em sistemas britânicos de sinalização ferroviária. A confiabilidade é crucial nessa área, pois o impacto de incidentes varia da perda de muitos milhões de dólares em equipamento a lesões e fatalidades.

Eddie Kennedy é gerente de projetos na equipe de Global Customer Experience (Experiência Global de Clientes) no Google e engenheiro mecânico de formação. Eddie passou seis anos trabalhando como engenheiro de processos no Six Sigma Black Belt em uma instalação de fábrica que produzia diamantes sintéticos. Essa indústria é caracterizada por um foco incansável em segurança, pois as exigências de extremos de temperatura e de

pressão do processo impõem um alto nível de perigo aos funcionários em seu dia a dia.

John Li atualmente é Site Reliability Engineer no Google. Trabalhou antes como administrador de sistemas e desenvolvedor de software em uma empresa de trading proprietário no mercado financeiro. Problemas de confiabilidade no setor financeiro são levados muito a sério, pois podem levar a graves consequências fiscais.

Dan Sheridan é Site Reliability Engineer no Google. Antes de se juntar à empresa, trabalhou como consultor de segurança na indústria nuclear civil na Grã-Bretanha. A confiabilidade é importante na indústria nuclear, pois um incidente pode ter repercussões sérias: as interrupções de serviço podem implicar milhões por dia em receita perdida, enquanto os riscos aos funcionários e à comunidade são mais perigosos ainda, exigindo tolerância zero a falhas. A infraestrutura nuclear é projetada com uma série de proteções contra falhas que interrompem as operações antes que um incidente de qualquer magnitude ocorra.

Jeff Stevenson atualmente é gerente de operações de hardware no Google. Tem experiência anterior como engenheiro nuclear na marinha dos Estados Unidos em um submarino. O que está em jogo por conta da confiabilidade na marinha nuclear tem um alto valor – problemas que surjam em caso de incidentes variam de equipamentos danificados a um impacto ambiental de longa duração ou, ainda, uma potencial perda de vidas.

Matthew Toia é Site Reliability Manager (Gerente de Confiabilidade de Sites), com foco em sistemas de armazenagem. Antes do Google, trabalhou no desenvolvimento de softwares e na implantação de sistemas de software para controle de tráfego aéreo. Efeitos de incidentes nesse mercado variam de inconveniências aos passageiros e às companhias aéreas (por exemplo, voos atrasados, aviões desviados) à potencial perda de vidas em caso de acidente. A defesa em profundidade é uma estratégia essencial para evitar falhas catastróficas.

Agora que você conheceu nossos experts e tem um conhecimento geral dos motivos pelos quais a confiabilidade é importante nas respectivas áreas em que essas pessoas atuaram antes, exploraremos os quatro temas fundamentais

associados à confiabilidade.

Preparo e testes para desastre

“Esperança não é uma estratégia.” Esse grito de guerra da equipe de SRE do Google sintetiza o que queremos dizer com preparo e testes para desastre. A cultura de SRE envolve estar sempre vigilante e ser constantemente questionador: O que poderia dar errado? Qual atitude podemos tomar para tratar esses problemas antes que eles levem a uma interrupção no serviço ou à perda de dados? Nosso treinamento de DiRT (Disaster and Recovery Testing, Testes para Desastre e Recuperação) anual procura abordar essas perguntas de frente [Kri12]. Nos exercícios de DiRT, os SREs levam os sistemas de produção ao limite e infligem interrupções de serviço verdadeiras a fim de:

- Garantir que os sistemas reajam do modo como achamos que o farão
- Determinar pontos fracos inesperados
- Descobrir maneiras de deixar os sistemas mais robustos para evitar interrupções de serviço descontroladas

Várias estratégias em outros mercados para testar a prontidão para desastres e garantir o preparo foram reveladas a partir de nossas conversas. Essas estratégias incluem:

- Foco organizacional incansável em segurança
- Atenção aos detalhes
- Capacidade de alternância
- Simulações e treinamentos ao vivo
- Treinamento e certificação
- Foco obsessivo na coleta de requisitos detalhados e no design
- Defesa em profundidade

Foco organizacional incansável em segurança

Esse princípio é particularmente importante em um contexto de engenharia industrial. De acordo com Eddie Kennedy, que trabalhou em chão de fábrica, onde os funcionários enfrentam ameaças à segurança, “toda reunião de

gerência começava com uma discussão sobre segurança". A indústria manufatureira se prepara para o inesperado criando processos altamente definidos, rigorosamente seguidos em todos os níveis da empresa. É crucial que todos os funcionários levem a segurança a sério e sintam que têm autoridade para falar se e quando houver algo errado. No caso dos mercados de energia nuclear, aeronaves militares e sinalização ferroviária, os padrões de segurança para software estão bem detalhados (por exemplo, UK Defence Standard [Padrão de Defesa Britânico] 00-56, IEC 61508, IEC513, US DO-178B/C e DO-254) e os níveis de confiabilidade para esses sistemas estão claramente definidos (por exemplo, SIL [Safety Integrity Level, ou Nível de Integridade de Segurança] 1–4)³, com o objetivo de especificar abordagens aceitáveis para entregar um produto.

Atenção aos detalhes

Do tempo que passou na marinha dos Estados Unidos, Jeff Stevenson se lembra de ter uma consciência aguda de como uma falta de zelo na execução de pequenas tarefas (por exemplo, manutenção de óleo lubrificante) poderia levar a falhas importantes no submarino. Um pequeno erro ou deixar de notar um detalhe podem ter grandes efeitos. Os sistemas são altamente interconectados, portanto um acidente em uma área pode causar impactos em vários componentes relacionados. A marinha nuclear tem como foco a manutenção rotineira para garantir que pequenos problemas não se transformem em uma bola de neve.

Capacidade de alternância

A utilização dos sistemas no mercado de telecomunicações pode ser altamente imprevisível. A capacidade absoluta pode ser consumida por eventos imprevisíveis, como desastres naturais, assim como eventos grandes e previsíveis, como os Jogos Olímpicos. De acordo com Gus Hartmann, o mercado lida com esses incidentes implantando capacidade de alternância na forma de um SOW (Switch On Wheels, ou Central Telefônica Sobre Rodas), que é uma central telefônica móvel. Essa capacidade em excesso pode ser implantada em uma emergência ou em antecipação a um evento conhecido que provavelmente sobrecarregará o sistema. Problemas de capacidade

também podem surgir de algo inesperado, de questões que nada têm a ver com a capacidade absoluta. Por exemplo, quando o número de telefone privado de uma celebridade vazou em 2005 e milhares de fãs tentaram ligar para ela ao mesmo tempo, o sistema de telecomunicações exibiu sintomas semelhantes aos de um DDoS ou de um erro massivo de roteamento.

Simulações e treinamentos ao vivo

Os testes de Disaster Recovery (Recuperação de Desastres) do Google têm muito em comum com as simulações e treinamentos ao vivo, que são um foco essencial de muitos mercados consolidados que pesquisamos. As possíveis consequências de uma interrupção no sistema determinam se usar uma simulação ou um treinamento ao vivo é apropriado. Por exemplo, Matthew Toia destaca que a indústria de aviação não pode realizar um teste ao vivo “em produção” sem colocar equipamentos e passageiros em risco. Em vez disso, eles empregam simuladores extremamente realistas, com feeds de dados ao vivo, em que as salas de controle e os equipamentos são modelados até os mínimos detalhes para garantir uma experiência realista, sem colocar pessoas reais em risco. Gus Hartmann informa que o mercado de telecomunicações geralmente se concentra em treinamentos ao vivo, com foco para sobrevivência a furacões e outras emergências climáticas. Uma modelagem como essa levou à criação de instalações à prova de efeitos climáticos, com geradores no prédio, capazes de durar mais que uma tempestade.

A marinha nuclear dos Estados Unidos usa uma mistura de exercícios de raciocínios do tipo “e se” com treinamentos ao vivo. De acordo com Jeff Stevenson, os treinamentos ao vivo envolvem “realmente provocar falhas reais em algo, mas com parâmetros de controle. Os treinamentos reais são feitos religiosamente todas as semanas, em dois ou três dias da semana”. Para a marinha nuclear, exercícios envolvendo raciocínios são úteis, mas não são suficientes para se preparar para os verdadeiros incidentes. As respostas devem ser praticadas para que não sejam esquecidas.

De acordo com Mike Doherty, os salva-vidas enfrentam exercícios de treinamento para desastres que se assemelham mais à experiência de um

“comprador misterioso” (mystery shopper). Geralmente, o gerente de um local trabalha com uma criança ou um salva-vidas incógnito em um treinamento para encenar um afogamento simulado. Esses cenários são conduzidos de modo a serem realistas ao máximo; assim, os salva-vidas não são capazes de diferenciar entre emergências reais e encenadas.

Treinamento e certificação

Nossas entrevistas sugerem que treinamento e certificação são particularmente importantes quando há vidas em jogo. Por exemplo, Mike Doherty descreveu como os salva-vidas concluem um treinamento rigoroso para obter uma certificação, além de passar por um processo periódico de recertificação. Os cursos incluem componentes de boa forma (por exemplo, um salva-vidas deve ser capaz de segurar alguém mais pesado que ele mesmo, com os ombros fora da água), componentes técnicos, como primeiros socorros e reanimação cardiopulmonar (CPR) e elementos operacionais (por exemplo, se um salva-vidas entrar na água, como os outros membros da equipe devem responder?). Toda instalação também tem um treinamento específico para esse lugar, pois atuar como salva-vidas em uma piscina é significativamente diferente de fazê-lo em uma praia, à beira de um lago ou do oceano.

Foco na coleta de requisitos detalhados e no design

Alguns dos engenheiros que entrevistamos discutiram a importância da coleta de requisitos detalhados e dos documentos de design. Essa prática foi particularmente importante no trabalho com dispositivos médicos. Em muitos desses casos, o uso ou a manutenção do equipamento não está no escopo dos designers do produto. Assim, os requisitos de uso e de manutenção devem ser coletados de outras fontes.

Por exemplo, de acordo com Erik Gross, as máquinas para cirurgia ocular a laser são projetadas para serem tão seguras quanto possível. Desse modo, solicitar requisitos de cirurgiões que realmente utilizem essas máquinas e dos técnicos responsáveis pela sua manutenção é particularmente importante. Em outro exemplo, o ex-contratado da área de defesa, Peter Dahl, descreveu uma

cultura de design bastante detalhada, em que criar um novo sistema de defesa comumente implicava um ano todo de design, seguido de apenas três semanas escrevendo o código para concretizá-lo. Esses dois exemplos são significativamente diferentes da cultura de lançamentos e iterações do Google, que promove uma taxa muito maior de mudanças com um risco calculado. Outros mercados (por exemplo, o mercado hospitalar e o militar, conforme discutimos antes) têm pressões, apetites para riscos e requisitos muito diferentes, e seus processos são baseados, em grande parte, nessas circunstâncias.

Defesa em profundidade e em largura

Na indústria de energia nuclear, a defesa em profundidade é um elemento essencial para estar preparado [IAEA12]. Os reatores nucleares apresentam redundância em todos os sistemas e implementam uma metodologia de design que obriga a existência de sistemas alternativos por trás dos sistemas principais, em caso de falhas. O sistema é projetado com várias camadas de proteção, incluindo uma última barreira física contra vazamentos radioativos em torno da própria planta. A defesa em profundidade é particularmente importante na indústria nuclear por causa da tolerância zero a falhas e incidentes.

Cultura de postmortem

CAPA⁴ (Corrective And Preventative Action, ou Ação Corretiva e Preventiva) é um conceito bem conhecido para melhorar a confiabilidade, que se concentra na investigação sistemática das causas-raízes dos problemas ou riscos identificados, a fim de evitar a recorrência. Esse princípio está incorporado na forte cultura de SRE de postmortems sem acusações. Quando algo dá errado (e dada a escala, a complexidade e a rápida taxa de mudanças no Google, algo inevitavelmente *dará* errado), é importante avaliar o seguinte:

- O que aconteceu
- A eficiência da resposta

- O que faríamos de forma diferente na próxima vez
- Quais atitudes serão tomadas para garantir que um incidente em particular não ocorra novamente

Esse exercício é feito sem apontar o dedo para acusar qualquer indivíduo. Em vez de atribuir culpas, é muito mais importante descobrir *o que* deu errado e, como uma empresa, de que modo lutaremos para garantir que isso não ocorra novamente. Insistir em *quem* pode ter causado a interrupção no serviço é contraproducente. Os postmortems são conduzidos após os incidentes e são publicados para as equipes de SRE para que todos possam se beneficiar das lições aprendidas.

Nossas entrevistas revelaram que muitos mercados têm uma versão de postmortem (embora muitos não usem esse nome específico, por motivos óbvios). A *motivação* por trás desses exercícios parece ser o principal diferenciador entre as práticas do mercado.

Muitos mercados são altamente regulamentados e são considerados responsáveis por autoridades governamentais específicas quando algo dá errado. Essa regulamentação está especialmente enraizada quando o que está em jogo em caso de falha é muito importante (por exemplo, há vidas em risco). Agências governamentais relevantes incluem FCC (telecomunicações), FAA (aviação), OSHA (indústrias manufatureiras e químicas), FDA (dispositivos médicos) e as várias National Competent Authorities (Autoridades Nacionais Competentes) na União Europeia.⁵ Os mercados de energia nuclear e transporte também são altamente regulamentados.

Considerações de segurança são outro fator motivador por trás dos postmortems. Nas indústrias manufatureiras e químicas, o risco de lesões ou mortes está sempre presente devido à natureza das condições necessárias para gerar o produto final (altas temperaturas, pressão, toxicidade e corrosividade, para nomear algumas). Por exemplo, a Alcoa tem uma cultura de segurança notável. Paul O'Neill, ex-CEO, exigia que os funcionários o notificassem em 24 horas acerca de qualquer lesão que provocasse a perda de um dia de trabalho de um funcionário. Ele até mesmo distribuía seu número de telefone particular aos funcionários da fábrica para que eles pudessem alertá-lo

pessoalmente sobre preocupações com segurança.⁶

O valor do que está em jogo é tão alto nas indústrias manufatureiras e químicas que mesmo casos de “quase acidente” – quando um dado evento poderia ter provocado danos sérios, mas não o fez – são cuidadosamente analisados. Esses cenários funcionam como um tipo de postmortem preventivo. De acordo com VM Brasseur, em uma palestra dada no YAPC NA de 2015, “Há muitos casos de ‘quase acidente’ em praticamente todos os desastres e crises de negócio e, geralmente, eles são ignorados quando ocorrem. Um erro latente, somado a uma condição favorável, é igual a ter algo que não funcionará exatamente como você planejou” [Bra15]. “Quase acidentes” são efetivamente desastres à espera de acontecer. Por exemplo, cenários em que um funcionário não siga o procedimento de operação padrão, um funcionário que salte para sair do caminho no último segundo a fim de evitar um respingo ou algo derramado na escada e que não foi limpado – todos esses casos representam situações de “quase acidente” e são oportunidades para aprender e melhorar. Na próxima vez, o funcionário e a empresa poderão não ter tanta sorte assim. O CHIRP (Confidential Reporting Programme for Aviation and Maritime, ou Programa de Relatórios Confidenciais para Aviação e Marinha) do Reino Unido procura elevar o nível de conscientização acerca de incidentes como esses no mercado oferecendo um ponto central de relatórios em que funcionários da aviação e da marinha possam informar casos de “quase acidentes” confidencialmente. Relatórios e análises desses quase acidentes são então publicados em boletins periódicos.

Os salva-vidas têm uma cultura profundamente enraizada de análises pós-incidentes e planejamento de ações. Mike Doherty brinca dizendo que “Se os pés de um salva-vidas entrarem na água, haverá papelada para preencher!”. Um relatório detalhado é necessário após qualquer incidente na piscina ou na praia. No caso de incidentes sérios, a equipe coletivamente analisa o incidente de ponta a ponta, discutindo o que deu certo e o que deu errado. Mudanças operacionais são então feitas com base nessas constatações, e um treinamento muitas vezes é agendado para ajudar as pessoas a desenvolver confiança em torno de sua capacidade de tratar um incidente semelhante no futuro. Nos casos de incidentes particularmente chocantes ou traumáticos, um

conselheiro é conduzido ao local para ajudar os funcionários a lidar com as consequências psicológicas. Os salva-vidas poderiam estar bem preparados para o que aconteceu na prática, mas podem *achar* que não fizeram um trabalho apropriado. De modo semelhante no Google, os salva-vidas adotam uma cultura de análise de incidentes sem apontar culpados. Incidentes são caóticos, e muitos fatores contribuem com qualquer incidente em particular. Nessa área, colocar a culpa em um único indivíduo não ajuda.

Automatizando tarefas repetitivas e o overhead operacional

Em sua essência, os Site Reliability Engineers (Engenheiros de Confiabilidade de Sites) do Google são engenheiros de software com baixa tolerância para tarefas reativas e repetitivas. Evitar repetir uma operação que não agregue valor a um serviço é algo que está fortemente enraizado em nossa cultura. Se uma tarefa puder ser automatizada, por que você operaria um sistema com base em tarefas repetitivas, com pouco valor? A automação reduz o overhead operacional e faz com que nossos engenheiros tenham mais tempo para avaliar e melhorar os serviços aos quais eles dão suporte, de forma proativa.

Os mercados que pesquisamos eram diversificados no que diz respeito a como e por que adotavam a automação. Determinados mercados confiam mais em seres humanos do que em máquinas. Durante o período de atuação de nosso veterano do mercado, a marinha nuclear dos Estados Unidos evitava a automação em favor de uma série de passos combinados e de procedimentos administrativos. Por exemplo, de acordo com Jeff Stevenson, operar uma válvula exigia um operador, um supervisor e um membro da tripulação no telefone, além do engenheiro de máquinas responsável por monitorar a resposta à ação realizada. Essas operações eram muito manuais por causa da preocupação de que um sistema automatizado pudesse não identificar um problema que um ser humano definitivamente perceberia. As operações em um submarino são governadas por uma cadeia confiável de decisões humanas – uma *série* de pessoas, e não apenas um indivíduo. A marinha nuclear também estava preocupada com o fato de a automação e os

computadores se moverem muito rapidamente a ponto de serem capazes de cometer um erro grande e irreparável. Ao lidar com reatores nucleares, uma abordagem metódica lenta e contínua é mais importante que realizar uma tarefa rapidamente.

De acordo com John Li, o mercado de trading proprietário tornou-se cada vez mais cauteloso em sua aplicação de automação nos últimos anos. A experiência tem mostrado que uma automação configurada incorretamente pode infligir danos significativos e provocar uma perda financeira enorme em um período de tempo muito curto. Por exemplo, em 2012, a Knight Capital Group se deparou com um “glitch de software” que resultou na perda de 440 milhões de dólares em apenas algumas horas.⁷ De modo semelhante, em 2010, o mercado de ações norte-americano vivenciou um Flash Crash (Crash Instantâneo) que, em última instância, foi atribuído a um trader desonesto tentando manipular o mercado por meios automatizados. Embora o mercado tenha sido rápido para se recuperar, o Flash Crash resultou em uma perda da ordem de trilhões de dólares em apenas *30 minutos*.⁸ Os computadores podem executar tarefas muito rapidamente, mas a velocidade pode ser um fator negativo se essas tarefas forem configuradas incorretamente.

Em oposição, algumas empresas adotam a automação exatamente *porque* os computadores agem mais rapidamente que as pessoas. De acordo com Eddie Kennedy, eficiência e economia monetária são fundamentais na indústria manufatureira, e a automação oferece um meio de realizar tarefas de modo mais eficiente e com menor custo. Além do mais, a automação geralmente é mais confiável e repetível que o trabalho conduzido manualmente por seres humanos, o que significa que ela gera padrões de mais alta qualidade e tolerância mais rigorosa. Dan Sheridan discutiu a automação conforme implantada na indústria nuclear britânica. Nesse caso, uma regra geral determina que se uma planta tiver que responder a uma dada situação em menos de 30 minutos, essa resposta deve ser automatizada.

Segundo a experiência de Matt Toia, a indústria de aviação aplica a automação de forma seletiva. Por exemplo, um failover operacional é realizado automaticamente, mas quando se trata de outras tarefas em particular, o mercado confia na automação somente quando ela é conferida

por um ser humano. Embora o mercado empregue uma boa dose de monitoração automática, as implementações de sistemas de controle de tráfego aéreo devem ser inspecionadas manualmente por seres humanos.

De acordo com Erik Gross, a automação tem sido bem eficiente para reduzir erros de usuários em cirurgias oculares a laser. Antes da cirurgia LASIK ser realizada, o médico avaliava o paciente usando um teste ocular refrativo. Originalmente, o médico digitava os números e pressionava um botão, e o laser fazia o seu trabalho corrigindo a visão do paciente. No entanto, erros na inserção dos dados poderiam ser um grande problema. Esse processo também implicava a possibilidade de misturar dados de pacientes ou confundir os números referentes ao olho esquerdo e ao olho direito.

A automação atualmente reduz bastante as chances de seres humanos cometerem um erro que tenha impacto na visão de alguém. Uma verificação de sanidade computadorizada dos dados inseridos manualmente foi a primeira melhoria principal da automação: se um operador humano inserir medidas que estejam fora de um intervalo esperado, a automação sinaliza esse caso como incomum, de forma imediata e explícita. Outras melhorias de automação vieram depois desse desenvolvimento: agora, a íris é fotografada durante o teste ocular refrativo preliminar. Na hora de realizar a cirurgia, a íris do paciente é automaticamente conferida com a íris da foto, eliminando assim a possibilidade de confundir os dados dos pacientes. Quando essa solução automatizada foi implementada, toda uma classe de erros médicos desapareceu.

Tomada de decisão estruturada e racional

No Google em geral, e na Site Reliability Engineering em particular, os dados são cruciais. A equipe aspira a tomar decisões de forma estruturada e racional, garantindo que:

- Haja concordância prévia no que diz respeito à base da decisão, em vez de isso ser justificado após o fato consumado
- As informações de entrada para a decisão sejam claras
- Qualquer suposição tenha sido explicitamente apresentada

- Decisões orientadas a dados vençam sobre decisões baseadas em sentimentos, palpites ou a opinião do funcionário mais experiente da sala

A SRE do Google trabalha com a suposição básica de que todos na equipe:

- Têm os melhores interesses dos usuários de um serviço em seu coração
- Podem descobrir como proceder com base nos dados disponíveis

As decisões devem ser fundamentadas, em vez de serem prescritivas, e devem ser tomadas sem acatar opiniões pessoais – mesmo da pessoa mais experiente da sala, que Eric Schmidt e Jonathan Rosenberg chamam de “HiPPO”, que quer dizer “Highest-Paid Person’s Opinion” (Opinião da Pessoa Mais Bem Paga) [Sch14].

A tomada de decisão em mercados diferentes varia bastante. Aprendemos que alguns mercados utilizam uma abordagem do tipo *se não estiver quebrado, não conserte... nunca*. Mercados que tenham sistemas cujo design tenha exigido muito planejamento e esforço muitas vezes são caracterizados por uma relutância a mudanças na tecnologia subjacente. Por exemplo, a indústria de telecomunicações ainda utiliza centrais telefônicas de longa distância implementadas nos anos 80. Por que eles dependem de uma tecnologia desenvolvida algumas décadas atrás? Essas centrais telefônicas “são basicamente à prova de balas e são extremamente redundantes”, de acordo com Gus Hartmann. Conforme informado por Dan Sheridan, a indústria nuclear, de modo semelhante, é lenta para mudanças. Todas as decisões são sustentadas pela ideia de que *se está funcionando agora, não mude*.

Muitos mercados têm um foco intenso em manuais e procedimentos, e não em uma resolução aberta de problemas. Todo cenário humanamente concebível é capturado em uma checklist ou “na pasta”. Quando algo dá errado, esse recurso é a fonte de autoridade sobre como reagir. Essa abordagem prescritiva funciona em mercados que evoluem e se desenvolvem de forma relativamente lenta, pois os cenários que poderiam dar errado não estão em constante evolução por causa de atualizações ou mudanças no sistema. Essa abordagem também é comum em mercados em que o nível de habilidade dos funcionários possa ser limitado, e a melhor maneira de garantir que as pessoas responderão apropriadamente em uma emergência é fornecer um conjunto simples e claro de instruções.

Outros mercados também adotam uma abordagem clara, orientada a dados, para a tomada de decisões. Segundo a experiência de Eddie Kennedy, ambientes de pesquisa e de manufatura são caracterizados por uma cultura rigorosa de experimentação que conta intensamente com formulação e teste de hipóteses. Esses mercados conduzem regularmente experimentos controlados para garantir que uma dada mudança produza o resultado esperado em um nível estatisticamente significativo, e que nada inesperado ocorra. As mudanças são implementadas somente quando os dados gerados pelo experimento dão suporte à decisão.

Por fim, alguns mercados, como o de trading proprietário, dividem a tomada de decisões para administrar melhor os riscos. De acordo com John Li, esse mercado tem uma equipe de fiscalização separada dos traders para garantir que não haja riscos indevidos no esforço de obter lucros. A equipe de fiscalização é responsável por monitorar eventos básicos e interromper a negociação caso eles saiam do controle. Se houver alguma anormalidade no sistema, a primeira resposta da equipe de fiscalização é desativá-lo. Conforme diz John Li, “Se não estivermos negociando, não estaremos perdendo dinheiro. Tampouco estaremos ganhando dinheiro, mas, pelo menos, não estaremos perdendo”. Somente a equipe de fiscalização pode restaurar o sistema, apesar do quanto excruciantemente possa parecer essa demora aos traders, que estão perdendo uma oportunidade potencialmente rentável.

Conclusões

Muitos dos princípios que são essenciais à Site Reliability Engineering (Engenharia de Confiabilidade de Sites) do Google são evidentes em uma grande variedade de mercados. As lições já aprendidas nos mercados bem consolidados provavelmente inspiraram algumas das práticas em uso no Google atualmente.

Uma conclusão importante de nossa pesquisa em outros mercados foi que, em muitas partes do nosso negócio de software, o Google tem um apetite maior por velocidade do que aqueles que atuam na maioria dos outros mercados. A capacidade de se mover ou mudar rapidamente deve ser avaliada em relação às diferentes implicações de uma falha. Nas indústrias nucleares,

de aviação e médica, por exemplo, pessoas poderão sofrer lesões ou até mesmo morrer caso haja uma interrupção de serviço ou uma falha. Quando as apostas são altas, uma abordagem conservadora para conseguir uma alta confiabilidade é justificada.

No Google, andamos constantemente na corda bamba entre as expectativas dos usuários para alta confiabilidade *versus* um foco preciso em mudanças rápidas e inovações. Embora o Google seja extremamente sério a respeito da confiabilidade, devemos adaptar nossas abordagens à nossa alta taxa de mudanças. Conforme discutimos em capítulos anteriores, muitos de nossos negócios de software, como o Search, tomam decisões conscientes para definir o que é ser “confiável o suficiente”.

O Google tem essa flexibilidade na maioria dos produtos de software e serviços, que operam em um ambiente em que vidas não estão diretamente em risco caso algo dê errado. Assim, somos capazes de usar ferramentas como provisão para erros (veja a seção “Motivação para provisão de erros”) como um meio de “financiar” uma cultura de inovação, com riscos calculados. Em essência, o Google adaptou princípios conhecidos de confiabilidade que, em muitos casos, foram desenvolvidos e aperfeiçoados em outros mercados, para criar sua própria cultura única de confiabilidade: uma cultura que trate uma equação complicada, a qual equilibra escala, complexidade e velocidade com alta confiabilidade.

¹ E911 (Enhanced 911, ou 911 Melhorado): linha de resposta a emergências nos Estados Unidos que tira proveito de dados de localização.

² Leituras de eletrocardiograma: <https://en.wikipedia.org/wiki/Electrocardiography>.

³ https://en.wikipedia.org/wiki/Safety_integrity_level

⁴ https://en.wikipedia.org/wiki/Corrective_and_preventive_action

⁵ https://en.wikipedia.org/wiki/Competent_authority

⁶ <http://ehstoday.com/safety/nsc-2013-oneill-exemplifies-safety-leadership>

⁷ Veja “FACTS, Section B” (FATOS, Seção B) para uma discussão sobre a Knight e o software Power Peg em [Sec13].

⁸ “Regulators blame computer algorithm for stock market ‘flash crash’” (Reguladores culpam algoritmo de computador pelo ‘flash crash’ no mercado de ações), Computerworld, <http://www.computerworld.com/article/2516076/financial-it/regulators-blame-computer-algorithm-for-stock-market--flash-crash-.html>.

CAPÍTULO 34

Conclusão

Escrito por Benjamin Lutch¹

Editado por Betsy Beyer

Li este livro com muito orgulho. Desde a época em que comecei a trabalhar na Excite, no início dos anos 90, quando minha equipe era uma espécie de grupo de SRE neandertal chamado de “Operações de Software”, passei minha carreira às voltas com o processo de desenvolver sistemas. À luz de minhas experiências ao longo dos anos no mercado de tecnologia, é incrível ver como a ideia de SRE criou raízes no Google e evoluiu tão rapidamente. A SRE cresceu, partindo de algumas centenas de engenheiros, quando me juntei ao Google em 2006, para mais de mil pessoas atualmente, espalhadas em uma dúzia de localidades e executando o que eu penso ser a infraestrutura computacional mais interessante do planeta.

Então, o que permitiu que a organização de SRE no Google evoluísse na última década de modo a manter essa infraestrutura gigantesca de forma inteligente, eficiente e escalável? Acho que o segredo para o incrível sucesso da SRE está na natureza dos princípios pelos quais ela trabalha.

As equipes de SRE são criadas de modo que nossos engenheiros dividam seu tempo entre dois tipos igualmente importantes de tarefas. Os SREs participam de turnos de plantão, o que implica colocar a mão na massa nos sistemas, observar em que pontos e como esses sistemas falham e compreender desafios (por exemplo, a melhor maneira de escalá-los). Contudo, também temos tempo para refletir e decidir o que desenvolver para fazer com que esses sistemas sejam mais fáceis de administrar. Em essência, temos o prazer de assumir as duas funções: de piloto e de engenheiro/designer. Nossas experiências em operar uma infraestrutura computacional gigantesca estão presentes nos códigos propriamente ditos e

são empacotadas na forma de um produto discreto.

Essas soluções são, então, facilmente utilizáveis por outras equipes de SRE e, em última instância, por qualquer pessoa no Google (ou até mesmo fora dele... pense no Google Cloud!) que queira usá-las ou melhorá-las com base na experiência que acumulamos e nos sistemas que desenvolvemos.

Ao abordar a criação de uma equipe ou de um sistema, o ideal é que sua base seja um conjunto de regras ou de axiomas genérico o suficiente para ser imediatamente útil, mas que permaneça relevante no futuro. Muito do que Ben Treynor Sloss apresentou na introdução deste livro representa exatamente isto: um conjunto de responsabilidades flexível, à prova de futuro em sua maior parte, que permaneça exato dez anos após ter sido concebido, apesar das mudanças e do crescimento da infraestrutura do Google e da equipe de SRE.

À medida que a SRE cresceu, percebemos duas dinâmicas diferentes em ação. A primeira é a natureza consistente das responsabilidades principais e das preocupações da SRE com o passar do tempo: nossos sistemas podem ser mil vezes maiores ou mais rápidos, mas, em última instância, ainda precisam permanecer confiáveis, flexíveis e fáceis de administrar em uma emergência, bem monitorados e com um planejamento de capacidade. Ao mesmo tempo, as atividades típicas assumidas pela SRE evoluem por questões de necessidade, à medida que os serviços do Google e as competências da SRE amadurecem. Por exemplo, o que já foi uma meta do tipo “desenvolver um painel de controle para vinte máquinas” agora pode ser “automatizar a descoberta, desenvolver painéis de controle e gerar alertas para uma frota de dezenas de milhares de máquinas”.

Para aqueles que não estiveram nas trincheiras da SRE na última década, uma analogia entre o modo como a SRE pensa nos sistemas complexos e a indústria de aviação aborda os voos das aeronaves é conveniente para conceituar a evolução e o amadurecimento da SRE com o tempo. Embora o que está em jogo em caso de falha nos dois mercados seja muito diferente, determinadas semelhanças essenciais se mantêm.

Suponha que você quisesse voar entre duas cidades cem anos atrás. Seu avião provavelmente tinha um único motor (dois, se estivesse com sorte), um pouco

de carga e um piloto. O piloto também assumia a função de mecânico e, possivelmente, atuava como carregador e descarregador. O cockpit tinha espaço para o piloto e, se você estivesse com sorte, um copiloto/navegador. Seu pequeno avião balançaria por uma pista em tempo bom e, se tudo corresse bem, você subiria lentamente em direção aos céus; em algum momento, pousaria em outra cidade, talvez a algumas centenas de quilômetros de distância. Uma falha em qualquer sistema do avião seria catastrófica, e já ouvimos falar de casos em que um piloto teve que sair do cockpit para fazer reparos em pleno voo! Os sistemas que alimentavam o cockpit eram essenciais, simples e frágeis e, muito provavelmente, não eram redundantes.

Vamos avançar rapidamente cem anos, até um enorme 747 parado na pista. Centenas de passageiros ocupam os dois andares, enquanto toneladas de carga são carregadas simultaneamente no compartimento abaixo. O avião está repleto de sistemas confiáveis e redundantes. É um modelo de segurança e de confiabilidade; na verdade, você está realmente mais seguro no ar do que em terra em um carro. Seu avião decolará a partir de uma linha pontilhada na pista em um continente e pousará tranquilamente em uma linha pontilhada em outra pista a dez mil quilômetros de distância, no horário correto – minutos antes do horário de pouso previsto. Mas dê uma olhada no cockpit, e o que você vê? Apenas dois pilotos novamente!

Como é que todos os demais elementos da experiência de voo – segurança, capacidade, velocidade e confiabilidade – escalaram tão bem, mas ainda há apenas dois pilotos? A resposta a essa pergunta é um grande paralelo ao modo como o Google aborda os sistemas enormes e incrivelmente complexos operados pela SRE. As interfaces aos sistemas de operação dos aviões foram bem planejadas e são acessíveis o suficiente, de modo que aprender a pilotá-los em condições normais não é uma tarefa inviável. Além disso, essas interfaces também oferecem flexibilidade suficiente, as pessoas que as operam são bem treinadas e as respostas às emergências são robustas e rápidas. O cockpit foi projetado por pessoas que entendem de sistemas complexos e sabem como apresentá-los aos seres humanos de uma forma que seja tanto consumível quanto escalável. Os sistemas subjacentes ao cockpit têm as mesmas propriedades discutidas neste livro: disponibilidade,

otimização de desempenho, gerenciamento de mudanças, monitoração e alertas, planejamento de capacidade e resposta a emergências.

Em última instância, a meta da SRE é seguir um curso semelhante. Uma equipe de SRE deve ser tão compacta quanto possível e deve operar em um alto nível de abstração, contando com muitos sistemas de backup e proteções contra falhas, além de APIs bem planejadas para se comunicar com os sistemas. Ao mesmo tempo, a equipe de SRE também deve ter um conhecimento abrangente dos sistemas – como eles funcionam, como falham e como responder às falhas – resultante da operação diária desses sistemas.

¹ Vice-presidente de Site Reliability Engineering (Engenharia de Confiabilidade de Sites) do Google, Inc.

APÊNDICE A

Tabela de disponibilidade

A disponibilidade geralmente é calculada com base no tempo que um serviço permaneceu indisponível em certo período de tempo. Supondo que não haja nenhum downtime planejado, a Tabela A.1 mostra a duração permitida de downtime para alcançar um dado nível de disponibilidade.

Tabela A.1 – Tabela de disponibilidade

Nível de disponibilidade	Intervalo de indisponibilidade permitido					
	por ano	por trimestre	por mês	por semana	por dia	por hora
90%	36,5 dias	9 dias	3 dias	16,8 horas	2,4 horas	6 minutos
95%	18,25 dias	4,5 dias	1,5 dia	8,4 horas	1,2 hora	3 minutos
99%	3,65 dias	21,6 horas	7,2 horas	1,68 hora	14,4 minutos	36 segundos
99,5%	1,83 dia	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 minuto	3,6 segundos
99,95%	4,38 horas	1,08 hora	21,6 minutos	5,04 minutos	43,2 segundos	1,8 segundo
99,99%	52,6 minutos	12,96 minutos	4,32 minutos	60,5 segundos	8,64 segundos	0,36 segundo
99,999%	5,26	1,30	25,9	6,05	0,87	0,04

	minutos	minuto	segundos	segundos	segundo	segundo
--	---------	--------	----------	----------	---------	---------

Usar uma métrica de indisponibilidade agregada (isto é, “X% de todas as operações falharam”) é mais útil do que focar na duração das interrupções de serviços que possam estar parcialmente disponíveis – por exemplo, devido ao fato de haver várias réplicas, em que apenas algumas estão indisponíveis – e para serviços cujas cargas variem no curso de um dia ou de uma semana, em vez de permanecerem constantes.

Veja as Equações 3.1 e 3.2, no Capítulo 3, para ver os cálculos.

APÊNDICE B

Um conjunto de melhores práticas para serviços em produção

Escrito por Ben Treynor Sloss

Editado por Betsy Beyer

Falhe de forma saudável

Faça a sanitização e a validação de entradas de configuração e responda a entradas implausíveis continuando a operar no estado anterior e alertando para a recepção de entradas ruins. Entradas ruins geralmente se enquadram em uma destas categorias:

Dados incorretos

Valide a sintaxe e, se for possível, também a semântica. Tome cuidado com dados vazios e parciais ou dados truncados (por exemplo, gere um alerta se a configuração for $N\%$ menor que a versão anterior).

Dados atrasados

Podem invalidar dados atuais por causa de timeouts. Gere um alerta bem antes do instante em que se espera que os dados expirem.

Falhe de modo que a função seja preservada, possivelmente à custa de ser demasiadamente permissivo ou simplista. Percebemos que, em geral, é mais seguro que os sistemas continuem funcionando com suas configurações anteriores e esperem a aprovação de uma pessoa antes que dados novos, talvez inválidos, sejam usados.

Exemplos

Em 2005, o sistema global de distribuição de carga e latência do Google

recebeu um arquivo de entrada de DNS vazio como resultado de permissões de arquivos. Esse sistema aceitou o arquivo vazio e serviu NXDOMAIN durante seis minutos para todas as propriedades do Google. Em resposta, o sistema atualmente realiza uma série de verificações de sanidade em novas configurações, incluindo confirmar a presença de IPs virtuais para *google.com*, e continua servindo as entradas de DNS anteriores até receber um novo arquivo que passe pelas verificações de entrada.

Em 2009, dados incorretos (porém válidos) levaram o Google a marcar toda a Web como contendo um 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. Verificar se houve alterações drásticas em tamanho de arquivos e se a configuração está fazendo a correspondência de sites que provavelmente não conteriam malwares teria evitado que esses tipos de problema atingissem a produção.

Rollouts progressivos

Rollouts não emergenciais *devem* ocorrer em fases. Tanto mudanças de configuração quanto de binários introduzem riscos, e eles podem ser atenuados aplicando a mudança a pequenas frações do tráfego e da capacidade a cada vez. O tamanho de seu serviço ou rollout, assim como o seu perfil de risco, servirão de base para calcular os percentuais da capacidade de produção para os quais o rollout será feito e o período de tempo apropriado entre as fases. Também é uma boa ideia executar as diferentes fases em regiões geográficas distintas a fim de detectar problemas relacionados a ciclos diurnos de tráfego e diferenças de tráfego entre regiões geográficas.

Os rollouts devem ser supervisionados. Para garantir que nada inesperado ocorra durante o rollout, ele deve ser monitorado, seja pelo engenheiro que está executando a etapa do rollout, seja – de preferência – por um sistema de monitoração comprovadamente confiável. Se um comportamento inesperado for detectado, faça um rollback antes e depois se preocupe com o diagnóstico

a fim de minimizar o Tempo Médio para Recuperação (Mean Time to Recovery).

Defina SLOs como um usuário

Avalie a disponibilidade e o desempenho em termos daquilo que é importante para um usuário final. Veja o Capítulo 4, que apresenta outras discussões sobre esse assunto.

Exemplo

Mensurar as taxas de erro e a latência no cliente do Gmail, em vez de fazê-lo no servidor, resultou em uma redução significativa de nossa avaliação da disponibilidade do Gmail e provocou mudanças tanto no código do cliente quanto no do servidor. Como resultado, o Gmail passou de aproximadamente 99,0% de disponibilidade para mais de 99,9% em alguns anos.

Provisões de erro

Crie um equilíbrio entre a confiabilidade e o ritmo de inovações usando provisões de erro (erro budgets – veja a seção “Motivação para provisão de erros”), que definem o nível aceitável de falhas para um serviço durante certo período de tempo; com frequência, usamos um mês. Uma provisão é simplesmente 1 menos o SLO de um serviço; por exemplo, um serviço com uma meta de disponibilidade de 99,99% tem uma “provisão” de 0,01% de indisponibilidade. Desde que o serviço não tenha gastado sua provisão para erros no mês com a taxa de erros em background, somado a qualquer downtime, a equipe de desenvolvimento é livre (dentro do razoável) para lançar novas funcionalidades, atualizações, e assim por diante.

Se a provisão para erros se esgotar, o serviço interromperá as mudanças (exceto correções urgentes de segurança e de bugs que tratem qualquer causa relacionada ao aumento de erros), até que haja folga novamente na provisão ou o mês expire. Para serviços maduros, com um SLO acima de 99,99%, uma reinicialização trimestral, em vez de mensal, da provisão é apropriada, pois o tempo de downtime permitido é baixo.

Provisões para erros eliminam a tensão estrutural que, de outro modo, se desenvolveria entre as equipes de SRE e de desenvolvimento de produtos, oferecendo-lhes um mecanismo comum, orientado a dados, para avaliar os riscos de lançamentos. Elas também proporcionam às equipes de SRE e de desenvolvimento de produtos uma meta comum para desenvolver práticas e tecnologia que permitam inovações mais rápidas e mais lançamentos sem “estourar a provisão”.

Monitoração

A monitoração pode ter apenas três tipos de saída:

Pages

Um ser humano deve fazer algo *agora*.

Tickets

Um ser humano deve fazer algo em alguns dias.

Logging

Ninguém precisa olhar para essa saída imediatamente, mas ela estará disponível para análise mais tarde, se for necessário.

Se um evento for importante o suficiente para exigir a atenção de um ser humano, ele deverá *exigir* uma atenção imediata (isto é, será um page) ou deverá ser tratado como um bug e inserido em seu sistema de monitoração de bugs. Colocar alertas em emails e esperar que alguém vá ler todos eles e notar aqueles que são importantes é o equivalente moral de enviá-los por meio de um pipe a `/dev/null`: eles serão ignorados. A história mostra que essa estratégia é atraente, porém incômoda, pois pode funcionar por um tempo, mas depende de uma vigilância humana perpétua, e a interrupção de serviço inevitável será, desse modo, mais grave quando acontecer.

Postmortems

Os postmortems (veja o Capítulo 15) não devem apontar culpados e devem se concentrar no processo e na tecnologia, e não nas pessoas. Suponha que as

pessoas envolvidas em um incidente sejam inteligentes, assim como bem-intencionadas, e estavam fazendo as melhores escolhas que podiam, considerando as informações disponíveis na ocasião. Segue-se daí que não podemos “consertar” as pessoas; em vez disso, devemos corrigir o ambiente: por exemplo, melhorar o design do sistema para evitar classes inteiras de problemas, fazer com que as informações apropriadas estejam facilmente disponíveis e validar automaticamente as decisões operacionais para dificultar colocar os sistemas em estados perigosos.

Planejamento de capacidade

Faça um provisionamento para tratar simultaneamente interrupções de serviço planejadas e não planejadas, sem deixar que a experiência de usuário se torne inaceitável; isso resulta em uma configuração “ $N + 2$ ”, em que o tráfego de pico possa ser tratado por N instâncias (possivelmente em modo degradado), enquanto as duas maiores instâncias estiverem indisponíveis:

- Valide as previsões de demanda em relação à realidade até que elas sejam consistentemente correspondentes. Uma divergência implica previsões instáveis, provisionamento ineficiente e o risco de uma falta de capacidade.
- Utilize testes de carga em vez de contar com a tradição para definir a razão entre recursos e capacidade: um cluster com X máquinas podia tratar Y consultas por segundo há três meses, mas ele ainda pode fazer isso, considerando as mudanças no sistema?
- Não confunda a carga do primeiro dia com a carga em estado estável. Os lançamentos, com frequência, atraem mais tráfego, ao mesmo tempo que, em especial, é o momento em que você vai querer que o produto mostre o seu melhor lado. Veja o Capítulo 27 e o Apêndice E.

Sobrecargas e falhas

Os serviços devem produzir resultados razoáveis, mas abaixo do ideal, se estiverem sobrecarregados. Por exemplo, o Google Search fará pesquisas em uma fração menor do índice, e parará de servir funcionalidades como Instant

para continuar a oferecer resultados de pesquisa web de boa qualidade quando estiver sobrecarregado. A SRE para Search testa clusters de pesquisa web além de sua capacidade definida para garantir que tenham um desempenho aceitável quando estiverem sobrecarregados por causa de tráfego.

Para as ocasiões em que a carga é alta o suficiente para que mesmas respostas degradadas sejam muito custosas para todas as consultas, faça um descarte elegante da carga usando enfileiramentos bem comportados e timeouts dinâmicos; veja o Capítulo 21. Outras técnicas incluem responder a requisições após um atraso significativo (“tarpitting”) e escolher um subconjunto consistente de clientes para receber erros, preservando uma boa experiência de usuário para os demais.

As retentativas podem amplificar taxas baixas de erro, gerando níveis mais altos de tráfego, o que pode resultar em falhas em cascata (veja o Capítulo 22). Responda a falhas em cascata descartando uma fração do tráfego (incluindo retentativas!) de upstream do sistema, depois que a carga total exceder a capacidade total.

Todo cliente que faça uma RPC deve implementar um backoff exponencial (com jitter) para as retentativas, a fim de evitar a amplificação de erros. Clientes móveis são especialmente problemáticos, pois pode haver milhões deles; atualizar seus códigos para corrigir um comportamento exige um período de tempo significativo – possivelmente, de semanas – e é necessário que os usuários instalem as atualizações.

Equipes de SRE

As equipes de SRE devem gastar não mais do que 50% de seu tempo em tarefas operacionais (veja o Capítulo 5); o excesso de tarefas operacionais deve ser direcionado à equipe de desenvolvimento de produtos. Muitos serviços também incluem os desenvolvedores de produtos no rodízio de plantão e no tratamento de tickets, mesmo que, no momento, não haja sobrecarga. Isso incentiva a projetar sistemas que minimizem ou eliminem as tarefas operacionais penosas, além de garantir que os desenvolvedores de produto estejam em contato com o lado operacional do serviço. Uma reunião

regular de produção entre as equipes de SRE e de desenvolvimento (veja o Capítulo 31) também é útil.

Percebemos que pelo menos oito pessoas devem fazer parte da equipe de plantão a fim de evitar o cansaço e permitir um número de funcionários sustentável e baixa rotatividade de pessoal. De preferência, esses engenheiros de plantão devem estar em duas localidades bem separadas geograficamente (por exemplo, na Califórnia e na Irlanda) para oferecer uma melhor qualidade de vida, evitando pages durante a noite; nesse caso, seis pessoas em cada localidade é o tamanho mínimo da equipe.

Espere lidar com não mais de dois eventos por turno de plantão (por exemplo, a cada 12 horas): é preciso ter tempo para responder e corrigir interrupções de serviço, iniciar o postmortem e registrar os bugs resultantes. Eventos mais frequentes podem degradar a qualidade da resposta e sugerir que há algo errado com o design do sistema (ou pelo menos com um de seus designs), a sensibilidade da monitoração e a resposta a bugs de postmortem.

Ironicamente, se você implementar essas melhores práticas, a equipe de SRE poderá, em algum momento, acabar ficando fora de forma para responder a incidentes devido à sua falta de frequência, fazendo com que haja uma interrupção de serviço mais longa quando essa deveria ser rápida. Exercite o tratamento de interrupções de serviço hipotéticas (veja a seção “Interpretando papéis em situações de desastre”) de forma rotineira e melhore sua documentação para tratamento de incidentes nesse processo.

APÊNDICE C

Exemplo de documento de estado do incidente

Sobrecarga no Shakespeare por causa de novo soneto: 21-10-2015

Informações sobre gerenciamento de incidentes: <http://incident-management-cheat-sheet>

(O líder de comunicação deve manter o resumo atualizado.)

Resumo: O serviço de pesquisa Shakespeare apresenta falha em cascata porque um soneto recentemente descoberto não está no índice de pesquisa.

Status: ativo, incidente #465

Postagem(ns) para o coordenador: #shakespeare no IRC

Hierarquia da coordenação (*todos que respondem*)

- Coordenadora atual do incidente: jennifer
 - Líder de operações: docbrown
 - Líder de planejamento: jennifer
 - Líder de comunicação: jennifer
- Próximo coordenador do incidente: *a ser definido*

(Atualize pelo menos a cada quatro horas e na transferência da função de Líder de Comunicação.)

Status detalhado (última atualização em 21-10-2015 15:28 UTC por jennifer)

Critérios de saída:

- Novo soneto acrescentado ao *corpus* de pesquisa do Shakespeare

PENDENTE

- Dentro dos SLOs de disponibilidade (99,99%) e de latência (percentil de 99% < 100 ms) por mais de 30 minutos PENDENTE

Lista de PENDENTES e bugs registrados:

- Executar job MapReduce para reindexar o *corpus* do Shakespeare FEITO
- Emprestar recursos de emergência para acrescentar capacidade extra FEITO
- Habilitar o capacitor de fluxo para distribuir a carga entre os clusters (Bug 5554823) PENDENTE

Linha do tempo do incidente (*mais recente antes: horários em UTC*)

- 21-10-2015 15:28 UTC jennifer
 - Aumentando globalmente a capacidade de servir em 2x
- 21-10-2015 15:21 UTC jennifer
 - Direcionando todo o tráfego para cluster de sacrifício USA-2 e drenando o tráfego de outros clusters para que possam se recuperar da falha em cascata enquanto ativam mais tarefas
 - Job de indexação MapReduce concluído, esperando a replicação do Bigtable em todos os clusters
- 21-10-2015 15:10 UTC martym
 - Adicionando novo soneto ao *corpus* do Shakespeare e iniciando o MapReduce de indexação
- 21-10-2015 15:04 UTC martym
 - Obtenção do texto do soneto recém-descoberto da lista de discussão *shakespeare-discuss@*
- 21-10-2015 15:01 UTC docbrown
 - Incidente declarado por causa de uma falha em cascata
- 21-10-2015 14:55 UTC docbrown
 - Chuva de pages, ManyHttp500s em todos os clusters

APÊNDICE D

Exemplo de postmortem

Postmortem do novo soneto de Shakespeare (incidente #465)

Data: 21-10-2015

Autores: jennifer, martym, agoogler

Status: Completo, lista de ações em progresso

Resumo: Shakespeare Search inativo por 66 minutos durante um período de interesse muito alto em Shakespeare por causa da descoberta de um novo soneto.

Impacto:¹ Estimativa de 1,21B de consultas perdidas, sem impacto na receita.

Causas-raízes:² Falha em cascata devido à combinação de carga excepcionalmente alta e um vazamento de recurso (resource leak) quando as pesquisas falharam por causa de termos ausentes no *corpus* do Shakespeare. O soneto recém-descoberto usava uma palavra que não havia aparecido antes em nenhuma das obras de Shakespeare, que, por acaso, era o termo pesquisado pelos usuários. Em circunstâncias normais, a taxa de falhas em tarefas devido ao vazamento de recursos é baixa o suficiente a ponto de passar despercebida.

Disparo: Bug latente disparado por causa do aumento súbito de tráfego.

Solução: O tráfego foi direcionado para o cluster sacrificial e uma capacidade dez vezes maior foi adicionada para atenuar a falha em cascata. Índice atualizado implantado, resolvendo a interação com o bug latente. Capacidade extra mantida até que o surto de interesse do público no novo soneto passe. Vazamento de recurso identificado e correção implantada.

Detecção: O Borgmon detectou um alto nível de HTTP 500 e gerou page para o plantão.

Lista de ações:³

Ação	Tipo	Responsável	Bug
Atualizar o manual com instruções para responder a uma falha em cascata	atenuação	jennifer	n/a FEITO
Usar o capacitor de fluxo para distribuir a carga entre os clusters	prevenção	martym	Bug 5554823 PENDENTE
Agendar teste de falha em cascata para o próximo DiRT	processo	docbrown	n/a PENDENTE
Investigar a execução contínua do MR de indexação/fusão	prevenção	jennifer	Bug 5554824 PENDENTE
Associar o vazamento de descritor de arquivos ao subsistema de classificação de pesquisa	prevenção	agoogler	Bug 5554825 FEITO
Acrescentar capacidades de descarte de carga na pesquisa do Shakespeare	prevenção	agoogler	Bug 5554826 PENDENTE
Criar testes de regressão para garantir que os servidores respondam de forma saudável a “queries of death” (consultas mortais)	prevenção	clarac	Bug 5554827 PENDENTE
Implantar subsistema de classificação de pesquisa atualizado em produção	prevenção	jennifer	n/a FEITO
Congelar a produção até 20-11-2015 por causa do esgotamento da provisão para erros, ou buscar exceção por causa de circunstâncias grotescas, inacreditáveis, bizarras e sem precedentes	outro	docbrown	n/a PENDENTE

Lições aprendidas

O que deu certo

- A monitoração rapidamente nos alertou acerca da alta taxa (que quase alcançou 100%) de HTTP 500
- *Corpus* do Shakespeare atualizado foi rapidamente distribuído a todos os clusters

O que deu errado

- Estamos sem prática para responder a uma falha em cascata
- Esgotamos nossa provisão para erros relativa à disponibilidade (em várias ordens de grandeza) por causa do surto excepcional de tráfego que essencialmente resultou em falhas

Em que pontos tivemos sorte⁴

- A lista de discussão dos fãs de Shakespeare tinha uma cópia do novo soneto disponível.
- Logs do servidor tinham stack traces que apontavam para o esgotamento de descritores de arquivos como a causa da falha.
- Uma query-of-death foi resolvida implantando um novo índice contendo o termo de pesquisa popular.

Linha do tempo⁵

21-10-2015 (*todos os horários em UTC*)

- 14:51 Notícias informam que um novo soneto de Shakespeare foi descoberto no porta-luvas de um Delorean.
- 14:53 Tráfego para pesquisa no Shakespeare aumenta em 88x após postagem em */r/shakespeare* apontar o sistema de pesquisa Shakespeare como o lugar para encontrar o novo soneto (exceto que não o tínhamos ainda)
- 14:54 INTERRUPÇÃO NO SERVIÇO TEM INÍCIO — backends da pesquisa começam a se desestruturar com a carga

- 14:55 docbrown recebe uma enxurrada de pages, ManyHttp500s de todos os clusters
- 14:57 Todo o tráfego da pesquisa Shakespeare em falha: veja http://monitor/shakespeare?end_time=20151021T145700
- 14:58 docbrown começa a investigar, percebe que a taxa de falhas de backend é muito alta
- 15:01 INCIDENTE COMEÇA docbrown declara o incidente #465 por causa de uma falha em cascata, coordenação em #shakespeare, nomeia jennifer como coordenadora do incidente
- 15:02 Alguém envia coincidentemente um email para *shakespeare-discuss@* sobre a descoberta do soneto, que por acaso está no início da caixa de entrada de martym
- 15:03 jennifer notifica a lista *shakespeare-incidents@* acerca do incidente
- 15:04 martym encontra o texto do novo soneto e procura a documentação sobre atualização do *corpus*
- 15:06 docbrown percebe que os sintomas da falha são idênticos em todas as tarefas de todos os clusters, investigando a causa com base nos logs da aplicação
- 15:07 martym encontra a documentação, começa o trabalho de preparação para a atualização do *corpus*
- 15:10 martym acrescenta o soneto às obras conhecidas de Shakespeare e inicia o job de indexação
- 15:12 docbrown entra em contato com clarac & agoogler (da equipe de desenvolvimento do Shakespeare) para ajudar a analisar a base de código para possíveis causas
- 15:18 clarac encontra um forte suspeito nos logs que apontam para o esgotamento de descriptores de arquivo, confirma no código que há um vazamento caso o termo não esteja no *corpus* em que é pesquisado
- 15:20 O job MapReduce de indexação de martym é concluído
- 15:21 jennifer e docbrown decidem aumentar o número de instâncias o suficiente para diminuir a carga de modo que as instâncias sejam capazes

de fazer um trabalho apreciável antes de morrer e serem reiniciadas

- 15:23 docbrown faz a distribuição de carga de todo o tráfego para o cluster USA-2, permitindo que o número de instâncias aumente em outros clusters sem que os servidores falhem de imediato
- 15:25 marty começa a replicar o novo índice em todos os clusters
- 15:28 docbrown começa a dobrar o número de instâncias
- 15:32 jennifer altera a distribuição de carga para aumentar o tráfego aos clusters não sacrificiais
- 15:33 tarefas em clusters não sacrificiais começam a falhar com os mesmos sintomas de antes
- 15:34 descoberto um erro de ordem de grandeza em cálculos feitos no quadro-branco para o aumento do número de instâncias
- 15:36 jennifer reverte a distribuição de carga para sacrificar novamente o cluster USA-2 em preparação para aumento adicional global de número de instâncias em 5x (para um total de 10x em relação à capacidade inicial)
- 15:36 INTERRUPÇÃO DE SERVIÇO ATENUADA, índice atualizado replicado em todos os clusters
- 15:39 docbrown inicia segunda onda de aumento de número de instâncias para 10x a capacidade inicial
- 15:41 jennifer restaura a distribuição de carga em todos os clusters para 1% do tráfego
- 15:43 taxas de HTTP 500 nos clusters não sacrificiais em taxas nominais, falhas intermitentes de tarefas em níveis baixos
- 15:45 jennifer distribui 10% do tráfego entre os clusters não sacrificiais
- 15:47 taxa de HTTP 500 nos clusters não sacrificiais permanece dentro do SLO, nenhuma falha de tarefas observada
- 15:50 30% do tráfego distribuído entre os clusters não sacrificiais
- 15:55 50% do tráfego distribuído entre os clusters não sacrificiais
- 16:00 FIM DA INTERRUPÇÃO DE SERVIÇO, todo o tráfego distribuído em todos os clusters

- 16:30 FIM DO INCIDENTE, atingido o critério de saída de desempenho nominal de 30 minutos

Informações para suporte⁶

- Painel de monitoração, [http://monitor/shakespeare?
end_time=20151021T160000&duration=7200](http://monitor/shakespeare?end_time=20151021T160000&duration=7200)
-

1 Impacto são os efeitos nos usuários, na receita etc.

2 Uma explicação das circunstâncias em que esse incidente ocorreu. Com frequência, usar uma técnica como os Cinco Porquês [Ohn88] é útil para entender os fatores que contribuíram com o incidente.

3 Ações “reflexivas” com frequência acabam sendo muito extremas ou custosas para implementar, e talvez seja necessária uma avaliação para redefinir seu escopo em um contexto maior. Há um risco de haver uma otimização exagerada para um problema em particular, com o acréscimo de monitoração/alertas específicos, quando mecanismos confiáveis como testes de unidade poderiam capturar o problema muito mais cedo no processo de desenvolvimento.

4 Esta seção é realmente para os “quase acidentes”; por exemplo, “O goat teleporter estava disponível para uso emergencial com outros animais, apesar da falta de certificação”.

5 Um “screenplay” do incidente; use a linha do tempo do incidente contida no documento de Gerenciamento do Incidente (Incident Management) para começar a preencher a linha do tempo do postmortem; em seguida, acrescente outras entradas relevantes.

6 Informações úteis, links, logs, capturas de tela, gráficos, logs de IRC, logs de IM etc.

APÊNDICE E

Checklist da coordenação de lançamentos

Esta é a Checklist de Coordenação de Lançamento (Launch Coordination Checklist) original do Google, que data aproximadamente de 2005, levemente reduzida por questões de concisão:

Arquitetura

- Esboço da arquitetura, tipos de servidores, tipos de requisições dos clientes
- Requisições de clientes geradas por código

Máquinas e datacenters

- Máquinas e largura de banda, datacenters, redundância N+2, QoS de rede
- Novos nomes de domínio, distribuição de carga no DNS

Estimativas de volume, capacidade e desempenho

- Estimativas de tráfego HTTP e de largura de banda, “picos” no lançamento, combinação de tráfego, seis meses depois
- Testes de carga, testes fim a fim, capacidade por datacenter com latência máxima
- Impacto em outros serviços com os quais nos preocupamos mais
- Capacidade de armazenagem

Confiabilidade do sistema e failover

- O que acontece quando:
 - Máquinas morrem, um rack falha ou um cluster sai do ar

- A rede falha entre dois datacenters
- Para cada tipo de servidor que conversa com outros servidores (seus backends):
 - Como detectar quando os backends morrem e o que fazer nesse caso
 - Como terminar ou reiniciar sem afetar clientes ou usuários
 - Comportamentos de distribuição de carga, limitação de taxas, timeout, retentativas e tratamento de erros
- Backup/restauração de dados, recuperação de desastres

Monitoração e gerenciamento de servidores

- Monitoração de estado interno, monitoração de comportamento fim a fim, gerenciamento de alertas
- Monitoração da monitoração
- Alertas e logs financeiramente importantes
- Dicas para executar servidores no ambiente de cluster
- Não provoque falhas em servidores de emails enviando alertas via email a si mesmo no código de seu próprio servidor

Segurança

- Revisão de design de segurança, auditoria em códigos de segurança, riscos de spam, autenticação, SSL
- Visibilidade de pré-lançamento/controle de acesso, vários tipos de listas-negras

Automação e tarefas manuais

- Métodos e controle de mudanças para atualizar servidores, dados e configurações
- Processo de release, builds repetíveis, canários com tráfego real, rollouts de teste

Questões ligadas ao crescimento

- Capacidade extra, crescimento em dez vezes, alertas de crescimento

- Gargalos para escalabilidade, escala linear, escala com hardware, alterações necessárias
- Caching, fragmentação/refragmentação de dados

Dependências externas

- Sistemas de terceiros, monitoração, rede, volume de tráfego, picos de lançamento
- Degradação elegante, como evitar um uso accidental exagerado de serviços de terceiros
- Integração tranquila com parceiros de syndication (sindicação), sistemas de emails, serviços no Google

Cronogramas e planejamento de rollouts

- Prazos fixos, eventos externos, segundas e sextas-feiras
- Procedimentos padrões de operação para esse serviço, para outros serviços

APÊNDICE F

Exemplo de minutas de reunião de produção

Data: 23-10-2015

Participantes: agoogler, clarac, docbrown, jennifer, martym

Anúncios:

- Interrupção de serviço importante (#465), acabou com a provisão para erros

Revisão da lista anterior de ações

- Certificar Goat Teleporter para uso com gado (bug 1011101)
 - Não linearidades em aceleração de massa agora previsíveis, devemos ser capazes de ter um alvo preciso em alguns dias

Análise da interrupção de serviço

- Novo soneto (interrupção de serviço 465)
 - 1,21B de consultas perdidas devido a falha em cascata após interação entre bug latente (vazamento de descritores de arquivo em pesquisas sem resultado) + não ter o novo soneto no *corpus* + volume de tráfego sem precedentes & inesperado
 - Bug de vazamento de descritores de arquivo corrigido (bug 5554825) e implantado em produção
 - Observando o uso do capacitor de fluxo para distribuição de carga (bug 5554823) e usando descarte de carga (bug 5554826) para evitar recorrência

- Provisão para erros referente à disponibilidade aniquilada; atualizações em produção congeladas por um mês, a menos que docbrown possa conseguir uma exceção argumentando que o evento foi bizarro & imprevisível (mas o consenso é que a exceção é improvável)

Eventos que geraram pages

- AnnotationConsistencyTooEventual: gerou 5 pages esta semana, provavelmente devido a atrasos de replicação entre regiões, entre os Bigtables.
 - Investigação ainda em andamento, veja o bug 4821600
 - Nenhuma correção esperada para logo, aumentaremos o limite de consistência aceitável a fim de reduzir alertas que não disparem ações

Eventos que não geraram pages

- Nenhum

Monitoração de mudanças e/ou silêncios

- AnnotationConsistencyTooEventual, limite aceitável de atraso elevado de 60s para 180s, veja o bug 4821600; PENDENTE(martym).

Mudanças planejadas em produção

- Cluster USA-1 será colocado offline para manutenção entre 29-10-2015 e 02-11-2015.
 - Nenhuma resposta necessária, tráfego será automaticamente encaminhado para outros clusters da região.

Recursos

- Recursos emprestados para responder ao incidente do novo soneto, instâncias adicionais de servidores serão reduzidas e recursos serão devolvidos na próxima semana
- Utilização em 60% de CPU, 75% de RAM, 44% de disco (subindo de 40%, 70%, 40% na semana passada)

Métricas essenciais do serviço

- OK latência do percentil 99: meta de SLO 88 ms < 100 ms [últimos 30 dias]
- Disponibilidade RUIM: meta de SLO 86,95% < 99,99% [últimos 30 dias]

Discussão / Atualizações sobre projetos

- Lançamento do projeto Molière em duas semanas.

Nova lista de ações

- PENDENTE(martym): elevar limite de AnnotationConsistencyTooEventual.
- PENDENTE(docbrown): retornar número de instâncias ao normal e devolver 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” (A prática e o futuro da Engenharia de Release: uma mesa-redonda com três Engenheiros de Release, <http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=434819>), *IEEE Software*, vol. 32, nº 2 (março/abril de 2015), pp. 42-49.
- [Agu10] M. K. Aguilera, “Stumbling over Consensus Research: Misunderstandings and Issues” (Tropeçando na pesquisa com o consenso: interpretações errôneas e problemas, <http://dl.acm.org/citation.cfm?id=2172342>), em *Replication*, Lecture Notes in Computer Science 5959, 2010.
- [All10] J. Allspaw e J. Robbins, *Web Operations: Keeping the Data on Time* (Operações web: mantendo os dados atualizados): O'Reilly, 2010.
- [All12] J. Allspaw, “Blameless PostMortems and a Just Culture” (Postmortems sem culpados e uma cultura justa, <https://codeascraft.com/2012/05/22/blameless-postmortems/>), postagem de blog, 2012.
- [All15] J. Allspaw, “Trade-Offs Under Pressure: Heuristics and Observations of Teams Resolving Internet Service Outages” (Negociações sob pressão: heurística e observações de equipes resolvendo interrupções de serviço na Internet, <http://lup.lub.lu.se/student-papers/record/8084520/file/8084521.pdf>), tese de mestrado, Universidade de Lund, 2015.
- [Ana07] S. Anantharaju, “Automating web application security testing” (Automatizando testes de segurança de aplicações web,

<https://security.googleblog.com/2007/07/automating-web-application-security.html>), postagem de blog, julho de 2007.

- [Ana13] R. Ananatharayan et al., “Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams” (Photon: junção escalável e tolerante a falhas de streams de dados contínuos, <https://research.google.com/pubs/pub41318.html>), em *SIGMOD '13*, 2013.
- [And05] A. Andrieux, K. Czajkowski, A. Dan, et al., “Web Services Agreement Specification (WS-Agreement)” (Especificação de acordo entre web services [WS-Agreement], <http://www.ogf.org/documents/GFD.107.pdf>), setembro de 2005.
- [Bai13] P. Bailis e A. Ghodsi, “Eventual Consistency Today: Limitations, Extensions, and Beyond” (Consistência eventual nos dias de hoje: limitações, extensões e além, <http://dl.acm.org/citation.cfm?id=2462076>), em *ACM Queue*, vol. 11, nº 3, 2013.
- [Bai83] L. Bainbridge, “Ironies of Automation” (Ironias da automação, [http://dx.doi.org/10.1016/0005-1098\(83\)90046-8](http://dx.doi.org/10.1016/0005-1098(83)90046-8)), em *Automatica*, vol. 19, nº 6, novembro de 1983.
- [Bak11] J. Baker et al., “Megastore: Providing Scalable, Highly Available Storage for Interactive Services” (Megastore: oferecendo armazenagem escalável, altamente disponível, para serviços interativos, <https://research.google.com/pubs/pub36971.html>), em *Proceedings of the Conference on Innovative Data System Research* (Anais da Conferência de Pesquisa em Sistemas de Dados Inovadores), 2011.
- [Bar11] L. A. Barroso, “Warehouse-Scale Computing: Entering the Teenage Decade” (Computação em escala de armazém de dados: entrando na década da adolescência, <http://dl.acm.org/citation.cfm?id=2019527>), palestra no 38th Annual Symposium on Computer Architecture (38º Simpósio Anual em Arquitetura de Computadores), vídeo disponível online, 2011.
- [Bar13] L. A. Barroso, J. Clidaras e U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2^a edição (O datacenter como um computador: uma introdução ao design de

máquinas em escala de armazém de dados, <https://research.google.com/pubs/pub41606.html>), Morgan & Claypool, 2013.

[Ben12] C. Bennett e A. Tseitlin, “Chaos Monkey Released Into The Wild” (Chaos Monkey à solta, <http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html>), postagem de blog, julho de 2012.

[Bla14] M. Bland, “Goto Fail, Heartbleed, and Unit Testing Culture” (Falha goto, Heartbleed e a cultura de testes de unidade, <http://martinfowler.com/articles/testing-culture.html>), postagem de blog, junho de 2014.

[Boc15] L. Bock, *Work Rules!* (O trabalho é demais!), Twelve Books, 2015.

[Bol11] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters e P. Li, “Paxos Replicated State Machines as the Basis of a High-Performance Data Store” (Máquinas de estado replicadas com Paxos como base para um repositório de dados de alto desempenho, https://www.usenix.org/legacy/event/nsdi11/tech/full_papers/Bolosky.pdf), em *Proc. NSDI 2011*, 2011.

[Boy13] P. G. Boysen, “Just Culture: A Foundation for Balanced Accountability and Patient Safety” (Cultura justa: uma base para responsabilidade equilibrada e segurança dos pacientes, <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3776518/>), em *The Ochsner Journal*, outono de 2013.

[Bra15] VM Brasseur, “Failure: Why it happens & How to benefit from it” (Falha: por que ela acontece & como se beneficiar dela, <https://www.youtube.com/DLn4fZsZsKM&t=29m05s>), YAPC 2015.

[Bre01] E. Brewer, “Lessons From Giant-Scale Services” (Lições de serviços em escala gigantesca, <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=939450>), em *IEEE Internet Computing*, vol. 5, nº 4, julho/agosto de 2001.

[Bre12] E. Brewer, “CAP Twelve Years Later: How the “Rules” Have Changed” (CAP doze anos depois: como as “regras” mudaram,

<http://Computerlore.ieee.org/xpl/articleDetails.jsp?arnumber=6133253>), em *Computer*, vol. 45, nº 2, fevereiro de 2012.

- [Bro15] M. Brooker, “Exponential Backoff and Jitter” (Backoff exponencial e jitter, <https://www.awsarchitectureblog.com/2015/03/backoff.html>), em *AWS Architecture Blog*, março de 2015.
- [Bro95] F. P. Brooks Jr., “No Silver Bullet – Essence and Accidents of Software Engineering” (Sem solução mágica – essência e acidentes da engenharia de software), em *The Mythical Man-Month*, Boston: Addison-Wesley, 1995, pp. 180-186.
- [Bru09] J. Brutlag, “Speed Matters” (Velocidade é importante, <http://googleresearch.blogspot.com/2009/06/speed-matters.html>), em *Google Research Blog*, junho de 2009.
- [Bul80] G. M. Bull, *The Dartmouth Time-sharing System* (O sistema de compartilhamento de tempo Dartmouth): Ellis Horwood, 1980.
- [Bur99] M. Burgess, *Princípios de administração de redes e sistemas*: LTC, 2006.
- [Bur06] M. Burrows, “The Chubby Lock Service for Loosely-Coupled Distributed Systems” (O serviço de lock Chubby para sistemas distribuídos com baixo acoplamento, <https://research.google.com/archive/chubby.html>), em *OSDI '06: Seventh Symposium on Operating System Design and Implementation* (OSDI 2006: 7º Simpósio de Design e Implementação de Sistemas Operacionais), novembro de 2006.
- [Bur16] B. Burns, B. Grant, D. Oppenheimer, E. Brewer e J. Wilkes, “Borg, Omega, and Kubernetes” (Borg, Omega e Kubernetes, <http://dl.acm.org/citation.cfm?id=2898444>), em *ACM Queue*, vol. 14, nº 1, 2016.
- [Cas99] M. Castro e B. Liskov, “Practical Byzantine Fault Tolerance” (Tolerância a falhas bizantinas na prática, <http://www.pmg.lcs.mit.edu/papers/osdi99.pdf>), 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” (FlumeJava: pipelines de dados paralelos, simples e eficientes, <http://research.google.com/pubs/pub35650.html>), em *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Conferência da ACM SIGPLAN de Design e Implementação de Linguagens de Programação), 2010.

[Cha96] T. D. Chandra e S. Toueg, “Unreliable Failure Detectors for Reliable Distributed Systems” (Detectores de falha não confiáveis para sistemas distribuídos confiáveis, <http://dl.acm.org/citation.cfm?id=226647>), em *J. ACM*, 1996.

[Cha07] T. Chandra, R. Griesemer e J. Redstone, “Paxos Made Live – An Engineering Perspective” (Dando vida ao Paxos – uma perspectiva da engenharia, http://research.google.com/archive/paxos_made_live.html), em *PODC ’07: 26th ACM Symposium on Principles of Distributed Computing* (PODC 2007: 26º Simpósio de Princípios de Processamento Distribuído da ACM), 2007.

[Cha06] F. Chang et al., “Bigtable: A Distributed Storage System for Structured Data” (Bigtable: um sistema de armazenagem distribuído para dados estruturados, <https://research.google.com/archive/bigtable.html>), em *OSDI ’06: Seventh Symposium on Operating System Design and Implementation* (OSDI 2006: 7º Simpósio de Design e Implementação de Sistemas Operacionais), novembro de 2006.

[Chr09] G. P. Chrousous, “Stress and Disorders of the Stress System” (Estresse e problemas do sistema de estresse, <http://www.ncbi.nlm.nih.gov/pubmed/19488073>), em *Nature Reviews Endocrinology*, vol 5., nº 7, 2009.

[Clos53] C. Clos, “A Study of Non-Blocking Switching Networks” (Um estudo de redes de comutação não bloqueantes, <http://dx.doi.org/10.1002/j.1538-7305.1953.tb01433.x>), em *Bell System Technical Journal*, vol. 32, nº 2, 1953.

[Con15] C. Contavalli, W. van der Gaast, D. Lawrence e W. Kumari, “Client

Subnet in DNS Queries” (Sub-rede de clientes em consultas de DNS, <https://tools.ietf.org/html/draft-vandergaast-edns-client-subnet>), IETF Internet-Draft, 2015.

- [Con63] M. E. Conway, “Design of a Separable Transition-Diagram Compiler” (Design de um compilador de diagrama de transições separável, <http://dl.acm.org/citation.cfm?id=366704>), em *Commun. ACM* 6, 7 (julho de 1963), 396-408.
- [Con96] P. Conway, “Preservation in the Digital World” (Preservação no mundo digital, <http://www.clir.org/pubs/reports/conway2/index.html>), artigo publicado pelo Council on Library and Information Resources (Conselho para Recursos de Biblioteca e Informações), 1996.
- [Coo00] R. I. Cook, “How Complex Systems Fail” (Como sistemas complexos falham, <http://web.mit.edu/2.75/resources/random/How%20Complex%20Systems%20Fail.pdf>), em *Web Operations*: O'Reilly, 2010.
- [Cor12] J. C. Corbett et al., “Spanner: Google's Globally-Distributed Database” (Spanner: banco de dados globalmente distribuído do Google, <https://research.google.com/archive/spanner.html>), em *OSDI '12: Tenth Symposium on Operating System Design and Implementation* (OSDI 2012: 10º Simpósio de Design e Implementação de Sistemas Operacionais), outubro de 2012.
- [Cra10] J. Cranmer, “Visualizing code coverage” (Visualizando a cobertura de código, <https://quetzalcoatal.blogspot.com.br/2010/03/visualizing-code-coverage.html>), postagem de blog, março de 2010.
- [Dea13] J. Dean e L. A. Barroso, “The Tail at Scale” (A cauda em escala, <http://research.google.com/pubs/pub40801.html>), em *Communications of the ACM*, vol. 56, 2013.
- [Dea04] J. Dean e S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters” (MapReduce: processamento de dados simplificado em clusters de grande porte, <https://research.google.com/archive/mapreduce.html>), em *OSDI'04: Sixth Symposium on Operating System Design and Implementation* (OSDI 2004: 6º Simpósio de Design e Implementação de Sistemas Operacionais), outubro de 2004.

Symposium on Operating System Design and Implementation (OSDI 2014: 6º Simpósio de Design e Implementação de Sistemas Operacionais), dezembro de 2004.

- [Dea07] J. Dean, “Software Engineering Advice from Building Large-Scale Distributed Systems” (Conselho de engenharia de software para desenvolver sistemas distribuídos de larga escala, <https://static.googleusercontent.com/media/research.google.com/en//people/295-talk.pdf>), aula no curso CS297 de Stanford, primavera de 2007.
- [Dek02] S. Dekker, “Reconstructing human contributions to accidents: the new view on error and performance” (Revendo contribuições humanas a acidentes: a nova visão sobre erros e desempenho, <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.411.4985&rep=rep1&type=pdf>), em *Journal of Safety Research*, vol. 33, nº 3, 2002.
- [Dek14] S. Dekker, *The Field Guide to Understanding “Human Error”* (O guia de campo para entender o “erro humano”), 3ª edição: Ashgate, 2014.
- [Dic14] C. Dickson, “How Embracing Continuous Release Reduced Change Complexity” (Como a adoção de releases contínuos reduziu a complexidade das mudanças, <http://usenix.org/conference/ures14west/summit-program/presentation/dickson>), apresentação na USENIX Release Engineering Summit West 2014 (Encontro de Engenharia de Release da USENIX), vídeo disponível online.
- [Dur05] J. Durmer e D. Dinges, “Neurocognitive Consequences of Sleep Deprivation” (Consequências neurocognitivas da privação de sono, <http://www.ncbi.nlm.nih.gov/pubmed/15798944>), em *Seminars in Neurology*, vol. 25, no. 1, 2005.
- [Eis16] D. E. Eisenbud et al., “Maglev: A Fast and Reliable Software Network Load Balancer” (Maglev: um software rápido e confiável para distribuição de carga em rede, <https://research.google.com/pubs/pub44824.html>), em *NSDI ’16: 13th*

USENIX Symposium on Networked Systems Design and Implementation (NSDI 2016: 13º Simpósio de Design e Implementação de Sistemas em Rede da USENIX), março de 2016.

- [Ere03] J. R. Erenkrantz, “Release Management Within Open Source Projects” (Gerenciamento de releases em projetos de código aberto, <http://www.erenkrantz.com/Geeks/Research/Publications/ReleaseManagement.pdf>), em *Proceedings of the 3rd Workshop on Open Source Software Engineering* (Anais do 3º Workshop em Engenharia de Software de Código Aberto), Portland, Oregon, maio de 2003.
- [Fis85] M. J. Fischer, N. A. Lynch e M. S. Paterson, “Impossibility of Distributed Consensus with One Faulty Process” (Impossibilidade de consenso distribuído com um processo em falha, <http://dl.acm.org/citation.cfm?id=214121>), *J. ACM*, 1985.
- [Fit12] B. W. Fitzpatrick e B. Collins-Sussman, *Team Geek: A Software Developer’s Guide to Working Well with Others* (Equipe geek: um guia para desenvolvedores de software trabalharem bem com outros): O’Reilly, 2012.
- [Flo94] S. Floyd e V. Jacobson, “The Synchronization of Periodic Routing Messages” (A sincronização de mensagens periódicas de roteamento, <http://dl.acm.org/citation.cfm?id=187045>), em *IEEE/ACM Transactions on Networking*, vol. 2, nº 2, abril de 1994, pp. 122-136.
- [For10] D. Ford et al, “Availability in Globally Distributed Storage Systems” (Disponibilidade em sistemas de armazenagem globalmente distribuídos, <http://research.google.com/pubs/pub36737.html>), em *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (Anais do 9º Simpósio de Design e Implementação de Sistemas Operacionais da USENIX), 2010.
- [Fox99] A. Fox e E. A. Brewer, “Harvest, Yield, and Scalable Tolerant Systems” (Colheita, produção e sistemas tolerantes escaláveis, http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=798396), em *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*

(Anais do 7º Workshop de Tópicos em Evidência sobre Sistemas Operacionais), Rio Rico, Arizona, março de 1999.

[Fow08] M. Fowler, “GUI Architectures” (Arquiteturas de GUI, <http://martinfowler.com/eaaDev/uiArchs.html>), postagem de blog, 2006.

[Gal78] J. Gall, *SYSTEMANTICS: How Systems Really Work and How They Fail* (SYSTEMANTICS: como os sistemas realmente funcionam e como falham), 1ª ed., Pocket, 1977.

[Gal03] J. Gall, *The Systems Bible: The Beginner’s Guide to Systems Large and Small* (A bíblia dos sistemas: guia para sistemas de grande e pequeno porte para iniciantes), 3ª ed., General Systemantics Press/Liberty, 2003.

[Gaw09] A. Gawande, *The Checklist Manifesto: How to Get Things Right* (O manifesto de checklist: como fazer corretamente): Henry Holt and Company, 2009.

[Ghe03] S. Ghemawat, H. Gobioff e S-T. Leung, “The Google File System” (O sistema de arquivos do Google, <https://research.google.com/archive/gfs.html>), em *19th ACM Symposium on Operating Systems Principles* (19º Simpósio de Princípios de Sistemas Operacionais da ACM), outubro de 2003.

[Gil02] S. Gilbert e N. Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services” (Conjectura de Brewer e a viabilidade de web services consistentes, disponíveis e tolerantes a partições, <http://dl.acm.org/citation.cfm?id=564601>), em *ACM SIGACT News*, vol. 33, nº 2, 2002.

[Gla02] R. Glass, *Facts and Fallacies of Software Engineering* (Fatos e falácias sobre engenharia de software), Addison-Wesley Professional, 2002.

[Gol14] W. Golab et al., “Eventually Consistent: Not What You Were Expecting?” (Eventualmente consistente: não era o que você estava esperando?, <http://dl.acm.org/citation.cfm?id=2582994>), em *ACM Queue*, vol. 12, nº 1, 2014.

[Gra09] P. Graham, “Maker’s Schedule, Manager’s Schedule” (Cronograma

de quem faz, cronograma do gerente, <http://paulgraham.com/makersschedule.html>), postagem de blog, julho de 2009.

[Gup15] A. Gupta e J. Shute, “High-Availability at Massive Scale: Building Google’s Data Infrastructure for Ads” (Alta disponibilidade em escala gigantesca: construindo a infraestrutura de dados do Google para o Ads, <https://research.google.com/pubs/pub44686.html>), em *Workshop on Business Intelligence for the Real Time Enterprise* (Workshop em Inteligência de Negócios para a Empresa de Tempo Real), 2015.

[Ham07] J. Hamilton, “On Designing and Deploying Internet-Scale Services” (Sobre design e implantação de serviços em escala de Internet, <https://www.usenix.org/legacy/event/lisa07/tech/hamilton.html>), em *Proceedings of the 21st Large Installation System Administration Conference* (Anais da 21ª Conferência de Administração de Instalações de Sistemas de Grande Porte), novembro de 2007.

[Han94] S. Hanks, T. Li, D. Farinacci e P. Traina, “Generic Routing Encapsulation over IPv4 networks” (Encapsulamento genérico de roteamento sobre redes IPv4, <https://tools.ietf.org/html/rfc1702>), *IETF Informational RFC*, 1994.

[Hic11] M. Hickins, “Tape Rescues Google in Lost Email Scare” (Fitas salvam Google no susto da perda de emails, <http://blogs.wsj.com/digits/2011/03/01/tape-rescues-google-in-lost-email-scare/>), em *Digits, Wall Street Journal*, 1 de março de 2011.

[Hix15a] D. Hixson, “Capacity Planning” (Planejamento de capacidade, <https://www.usenix.org/publications/login/feb15/capacity-planning>), em *;login;*, vol. 40, nº 1, fevereiro de 2015.

[Hix15b] D. Hixson, “The Systems Engineering Side of Site Reliability Engineering” (O lado da engenharia de sistemas da Engenharia de Confiabilidade de Sites, <https://www.usenix.org/publications/login/june15/hixson>), em *;login;*, vol. 40, nº 3, junho de 2015.

- [Hod13] J. Hodges, “Notes on Distributed Systems for Young Bloods” (Observações sobre sistemas distribuídos para iniciantes, <https://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods/>), postagem de blog, 14 de janeiro 2013.
- [Hol14] L. Holmwood, “Applying Cardiac Alarm Management Techniques to Your On-Call” (Aplicando técnicas de gerenciamento de alarme cardíaco em seu plantão, <http://fractio.nl/2014/08/26/cardiac-alarms-and-ops/>), postagem de blog, 26 de agosto de 2014.
- [Hum06] J. Humble, C. Read, D. North, “The Deployment Production Line” (A linha de produção de implantação), em *Proceedings of the IEEE Agile Conference* (Anais da Conferência de Agile do IEEE), julho de 2006.
- [Hum10] J. Humble e D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* (Entrega contínua: releases de software confiáveis por meio de automação de construção, teste e implantação): Addison-Wesley, 2010.
- [Hun10] P. Hunt, M. Konar, F. P. Junqueira e B. Reed, “ZooKeeper: Wait-free coordination for Internet-scale systems” (ZooKeeper: coordenação sem espera para sistemas em escala de internet, https://www.usenix.org/legacy/events/atc10/tech/full_papers/Hunt.pdf), em USENIX ATC, 2010.
- [IAEA12] International Atomic Energy Agency (Agência Internacional de Energia Atômica), “Safety of Nuclear Power Plants: Design, SSR-2/1” (Segurança em plantas de energia nuclear: design, SSR-2/1, http://www-pub.iaea.org/MTCD/publications/PDF/Pub1534_web.pdf), 2012.
- [Jai13] S. Jain et al., “B4: Experience with a Globally-Deployed Software Defined WAN” (B4: experiência com uma WAN globalmente implantada definida por software, <https://research.google.com/pubs/pub41761.html>), em SIGCOMM ’13.
- [Jon15] C. Jones, T. Underwood e S. Nukala, “Hiring Site Reliability Engineers” (Contratando Engenheiros de Confiabilidade de Sites, <https://www.usenix.org/publications/login/june15/hiring-site-reliability->

engineers), em ;login:, vol. 40, nº 3, junho de 2015.

[Jun07] F. Junqueira, Y. Mao e K. Marzullo, “Classic Paxos vs. Fast Paxos: Caveat Emptor” (Classic Paxos *versus* Fast Paxos: por conta e risco de quem usa, <http://dl.acm.org/citation.cfm?id=1323158>), em *Proc. HotDep '07*, 2007.

[Jun11] F. P. Junqueira, B. C. Reid e M. Serafini, “Zab: High-performance broadcast for primary-backup systems.” (Zab: broadcast de alto desempenho para sistemas de backup principal, http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5958223&tag=1), em *Dependable Systems & Networks (DSN)*, 2011 IEEE/IFIP 41st International Conference (Sistemas & redes confiáveis (DSN), 41ª Conferência Internacional do IEEE/IFIP de 2011) em 27 de junho de 2011: 245-256.

[Kah11] D. Kahneman, *Thinking, Fast and Slow* (Pensando de forma rápida e lenta): Farrar, Straus e Giroux, 2011.

[Kar97] D. Karger et al., “Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web” (Hashing consistente e árvores aleatórias: protocolos de caching distribuído para aliviar hot spots na World Wide Web, <http://dl.acm.org/citation.cfm?id=258660>), em *Proc. STOC '97*, 29th annual ACM symposium on theory of computing (29º Simpósio Anual de Teoria de Computação da ACM), 1997.

[Kem11] C. Kemper, “Build in the Cloud: How the Build System Works” (Build na nuvem: como o sistema de build funciona, <https://google-engtools.blogspot.com.br/2011/08/build-in-cloud-how-build-system-works.html>), postagem de blog no *Google Engineering Tools*, agosto de 2011.

[Ken12] S. Kendrick, “What Takes Us Down?” (O que nos derruba?, <http://usenix.org/publications/login/october-2012-volume-37-number-5/what-takes-us-down>), em ;login:, vol. 37, nº 5, outubro de 2012.

[Kinc09] Kincaid, Jason. “T-Mobile Sidekick Disaster: Danger’s Servers

Crashed, And They Don't Have A Backup.” (Desastre no Sidekick da T-Mobile: servidores do Danger falharam e eles não têm backup) *Techcrunch*. s.n., 10 de outubro de 2009. Web. 20 de janeiro de 2015, <http://techcrunch.com/2009/10/10/t-mobile-sidekick-disaster-microsofts-servers-crashed-and-they-dont-have-a-backup>.

[Kin15] K. Kingsbury, “The trouble with timestamps” (O problema com os timestamps, <http://www.aphyr.com/posts/299-the-trouble-with-timestamps>), postagem de blog, 2013.

[Kir08] J. Kirsch e Y. Amir, “Paxos for System Builders: An Overview” (Paxos para desenvolvedores de sistemas: uma visão geral, <http://dl.acm.org/citation.cfm?id=1529979>), em *Proc. LADIS '08*, 2008.

[Kla12] R. Klau, “How Google Sets Goals: OKRs” (Como o Google define metas: OKRs, <https://library.gv.com/how-google-sets-goals-okrs-a1f69b0b72c7>), postagem de blog, outubro de 2012.

[Kle06] D. V. Klein, “A Forensic Analysis of a Distributed Two-Stage Web-Based Spam Attack” (Uma análise forense de um ataque de spam distribuído de dois estágios baseado em web, https://www.usenix.org/legacy/event/lisa06/tech/klein_klein_html/index.html em *Proceedings of the 20th Large Installation System Administration Conference* (Anais da 20ª Conferência de Administração de Instalações de Sistemas de Grande Porte), dezembro de 2006.

[Kle14] D. V. Klein, D. M. Betser e M. G. Monroe, “Making Push On Green a Reality” (Fazendo do Push On Green uma realidade, <https://www.usenix.org/publications/login/october-2014-vol-39-no-5/making-push-green-reality>), em *:login:*, vol. 39, nº 5, outubro de 2014.

[Kra08] T. Krattenmaker, “Make Every Meeting Matter” (Faça toda reunião valer a pena, <https://hbr.org/2008/02/make-every-meeting-matter>), em *Harvard Business Review*, 27 de fevereiro de 2008.

[Kre12] J. Kreps, “Getting Real About Distributed System Reliability” (Levando a sério a confiabilidade de sistemas distribuídos, <http://blog.empathybox.com/post/19574936361/getting-real-about->

distributed-system-reliability), postagem de blog, 19 de março de 2012.

[Kri12] K. Krishan, “Weathering The Unexpected” (Vencendo o inesperado, <http://dl.acm.org/citation.cfm?id=2366332>), em *Communications of the ACM*, vol. 55, nº 11, novembro de 2012.

[Kum15] A. Kumar et al., “BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing” (BwE: alocação de banda flexível e hierárquica para processamento distribuído em WAN, <https://research.google.com/pubs/pub43838.html>), em *SIGCOMM '15*.

[Lam98] L. Lamport, “The Part-Time Parliament” (O Part-Time Parliament, <http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf>), em *ACM Transactions on Computer Systems* 16, 2, maio de 1998.

[Lam01] L. Lamport, “Paxos Made Simple” (Paxos simplificado, <http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf>), em *ACM SIGACT News* 121, dezembro de 2001.

[Lam06] L. Lamport, “Fast Paxos” (O Fast Paxos, <https://www.microsoft.com/en-us/research/publication/fast-paxos/>), em *Distributed Computing* 19.2, outubro de 2006.

[Lim14] T. A. Limoncelli, S. R. Chalup e C. J. Hogan, *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems, volume 2* (A prática da administração de sistemas em nuvem: projetando e operando sistemas distribuídos de grande porte), Addison-Wesley, 2014.

[Loo10] J. Loomis, “How to Make Failure Beautiful: The Art and Science of Postmortems” (Como deixar uma falha bonita: a arte e a ciência dos postmortems), em *Web Operations*: O'Reilly, 2010.

[Lu15] H. Lu et al, “Existential Consistency: Measuring and Understanding Consistency at Facebook” (Consistência existencial: mensurando e entendendo a consistência no Facebook, <http://sigops.org/sosp/sosp15/current/2015-Monterey/printable/240-lu.pdf>), em *SOSP '15*, 2015.

- [Mao08] Y. Mao, F. P. Junqueira e K. Marzullo, “Mencius: Building Efficient Replicated State Machines for WANs” (Mencius: criando máquinas de estado eficientes e replicadas para WANs, https://www.usenix.org/legacy/events/osdi08/tech/full_papers/mao/mao.pdf) em OSDI ’08, 2008.
- [Mas43] A. H. Maslow, “A Theory of Human Motivation” (Uma teoria sobre a motivação humana), em *Psychological Review* 50(4), 1943.
- [Mau15] B. Maurer, “Fail at Scale” (Falha em escala, <http://dl.acm.org/citation.cfm?id=2839461>), em *ACM Queue*, vol. 13, nº 12, 2015.
- [May09] M. Mayer, “*This site may harm your computer* on every search result?!?!” (Este site pode prejudicar o seu computador em todo resultado de pesquisa?!?!, <https://googleblog.blogspot.com.br/2009/01/this-site-may-harm-your-computer-on.html>), postagem de blog, janeiro de 2009.
- [McI86] M. D. McIlroy, “A Research Unix Reader: Annotated Excerpts from the Programmer’s Manual, 1971-1986” (Um leitor de pesquisas sobre Unix: excertos com anotações do Manual do Programador, 1971-1986 <http://www.cs.dartmouth.edu/~doug/reader.pdf>).
- [McN13] D. McNutt, “Maintaining Consistency in a Massively Parallel Environment” (Mantendo a consistência em um ambiente intensamente paralelo, <https://www.usenix.org/conference/ucms13/summit-program/presentation/mcnutt>), apresentação no USENIX Configuration Management Summit 2013 (Encontro de Gerenciamento de Configuração da USENIX 2013), vídeo disponível online.
- [McN14a] D. McNutt, “Accelerating the Path from Dev to DevOps” (Acelerando o caminho, do desenvolvimento ao DevOps, https://www.usenix.org/system/files/login/articles/05_mcnutt.pdf), em *:login:*, vol. 39, nº 2, abril de 2014.
- [McN14b] D. McNutt, “The 10 Commandments of Release Engineering” (Os dez mandamentos da Engenharia de Release, https://www.youtube.com/watch?v=RNMJYV_UsQ8), apresentação no 2nd

International Workshop on Release Engineering 2014 (2º Workshop Internacional em Engenharia de Release de 2014), abril de 2014.

[McN14c] D. McNutt, “Distributing Software in a Massively Parallel Environment” (Distribuindo software em um ambiente intensamente paralelo, <https://www.usenix.org/conference/lisa14/conference-program/presentation/mcnutt>), apresentação no USENIX LISA 2014, vídeo disponível online.

[Mic03] Microsoft TechNet, “What is SNMP?” (O que é SNMP?), última modificação em 28 de março de 2003, <https://technet.microsoft.com/en-us/library/cc776379%28v=ws.10%29.aspx>.

[Mea08] D. Meadows, *Thinking in Systems* (Pensando em sistemas): Chelsea Green, 2008.

[Men07] P. Menage, “Adding Generic Process Containers to the Linux Kernel” (Acrescentando contêineres de processos genéricos ao kernel do Linux, <https://www.kernel.org/doc/ols/2007/ols2007v2-pages-45-58.pdf>), em *Proc. of Ottawa Linux Symposium* (Anais do Simpósio de Linux em Ottawa), 2007.

[Mer11] N. Merchant, “Culture Trumps Strategy, Every Time” (A cultura vence a estratégia, sempre, <https://hbr.org/2011/03/culture-trumps-strategy-every>), em *Harvard Business Review*, 22 de março de 2011.

[Moc87] P. Mockapetris, “Domain Names – Implementation and Specification” (Nomes de domínio – implementação e especificação, <https://tools.ietf.org/html/rfc1035>), *IETF Internet Standard*, 1987.

[Mol86] C. Moler, “Matrix Computation on Distributed Memory Multiprocessors” (Processamento matricial em multiprocessadores com memória distribuída), em *Hypercube Multiprocessors 1986*, 1987.

[Mor12a] I. Moraru, D. G. Andersen e M. Kaminsky, “Egalitarian Paxos” (O Egalitarian Paxos, <http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-12-108.pdf>), *Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-12-108* (Relatório Técnico do Laboratório de Dados Paralelos da Universidade Carnegie-Mellon CMU-PDL-12-108), 2012.

- [Mor14] I. Moraru, D. G. Andersen e M. Kaminsky, “Paxos Quorum Leases: Fast Reads Without Sacrificing Writes” (Leases de quórum do Paxos: leituras rápidas sem sacrificar as escritas, <http://dl.acm.org/citation.cfm?id=2671001>), in *Proc. SOCC ’14*, 2014.
- [Mor12b] J. D. Morgenthaler, M. Gridnev, R. Sauciuc e S. Bhansali, “Searching for Build Debt: Experiences Managing Technical Debt at Google” (Procurando débitos técnicos: experiências no gerenciamento de débitos técnicos no Google, <https://research.google.com/pubs/pub37755.html>), em *Proceedings of the 3rd Int’l Workshop on Managing Technical Debt* (Anais do 3º Workshop Internacional de Gerenciamento de Débitos Técnicos), 2012.
- [Nar12] C. Narla e D. Salas, “Hermetic Servers” (Servidores herméticos, <http://googletesting.blogspot.com/2012/10/hermetic-servers.html>), postagem de blog, 2012.
- [Nel14] B. Nelson, “The Data on Diversity” (Os dados na diversidade, <http://dl.acm.org/citation.cfm?id=2684442.2597886>), em *Communications of the ACM*, vol. 57, 2014.
- [Nic12] K. Nichols e V. Jacobson, “Controlling Queue Delay” (Controlando atrasos em filas, <http://dl.acm.org/citation.cfm?id=2209336>), em *ACM Queue*, vol. 10, nº 5, 2012.
- [Oco12] P. O’Connor e A. Kleyner, *Practical Reliability Engineering* (Engenharia de confiabilidade na prática), 5ª edição: Wiley, 2012.
- [Ohn88] T. Ohno, *Toyota Production System: Beyond Large-Scale Production* (Sistema de produção na Toyota: além da produção em larga escala): Productivity Press, 1988.
- [Ong14] D. Ongaro e J. Ousterhout, “In Search of an Understandable Consensus Algorithm (Extended Version)” (Em busca de um algoritmo de consenso comprehensível (versão estendida), <https://ramcloud.stanford.edu/raft.pdf>).
- [Pen10] D. Peng e F. Dabek, “Large-scale Incremental Processing Using Distributed Transactions and Notifications” (Processamento incremental de

larga escala usando transações e notificações distribuídas, (<https://research.google.com/pubs/pub36726.html>), em *Proc. of the 9th USENIX Symposium on Operating System Design and Implementation* (Anais do 9º Simpósio de Design e Implementação de Sistemas Operacionais da USENIX), novembro de 2010.

[Per99] C. Perrow, *Normal Accidents: Living with High-Risk Technologies* (Acidentes normais: convivendo com tecnologias de alto risco), Princeton University Press, 1999.

[Per07] A. R. Perry, “Engineering Reliability into Web Sites: Google SRE” (Engenharia de Confiabilidade em sites web: a SRE do Google, <https://research.google.com/pubs/pub32583.html>), em *Proc. of LinuxWorld 2007* (Anais do LinuxWorld de 2007), 2007.

[Pik05] R. Pike, S. Dorward, R. Griesemer, S. Quinlan, “Interpreting the Data: Parallel Analysis with Sawzall” (Interpretando os dados: análise paralela com o Sawzall, <https://research.google.com/archive/sawzall.html>), em *Scientific Programming Journal* vol. 13, nº 4, 2005.

[Pot16] R. Potvin e J. Levenberg, “The Motivation for a Monolithic Codebase: Why Google stores billions of lines of code in a single repository” (A motivação para uma base de código monolítica: por que o Google armazena bilhões de linhas de código em um único repositório), em *Communications of the ACM*, julho de 2016. Vídeo disponível no YouTube (<https://www.youtube.com/watch?v=W71BTkUbdqE>).

[Roo04] J. J. Rooney e L. N. Vanden Heuvel, “Root Cause Analysis for Beginners” (Análise de causas-raízes para iniciantes, <http://asq.org/quality-progress/2004/07/quality-tools/root-cause-analysis-for-beginners.html>), em *Quality Progress*, julho de 2004.

[Sai39] A. de Saint-Exupéry, *Terre des Hommes* (Paris: Le Livre de Poche, 1939, tradução de Lewis Galantière como *Wind, Sand and Stars*.

[Sam14] R. R. Sambasivan, R. Fonseca, I. Shafer e G. R. Ganger, “So, You Want To Trace Your Distributed System? Key Design Insights from Years of Practical Experience” (Então, você quer monitorar seu sistema

distribuído? Insights de design essenciais baseados em anos de experiência prática, http://pdl.cmu.edu/PDL-FTP/SelfStar/CMU-PDL-14-102_abs.shtml), Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-14-102 (Relatório Técnico do Laboratório de Dados Paralelos da Universidade Carnegie-Mellon CMU-PDL-14-102), 2014.

- [San11] N. Santos e A. Schiper, “Tuning Paxos for High-Throughput with Batching and Pipelining” (Ajustando o Paxos para alto throughput com batching e pipelining, http://rd.springer.com/chapter/10.1007%2F978-3-642-25959-3_11), em *13th Int’l Conf. on Distributed Computing and Networking* (13ª Conferência Internacional de Processamento Distribuído e Rede), 2012.
- [Sar97] N. B. Sarter, D. D. Woods e C. E. Billings, “Automation Surprises” (Surpresas da automação), em *Handbook of Human Factors & Ergonomics*, 2ª edição, G. Salvendy (ed.), Wiley, 1997.
- [Sch14] E. Schmidt, J. Rosenberg e A. Eagle, *Como o Google funciona*: Editora Intrínseca, 2015.
- [Sch15] B. Schwartz, “The Factors That Impact Availability, Visualized” (Os fatores que impactam a disponibilidade, visualizados, <https://www.vividcortex.com/blog/the-factors-that-impact-availability-visualized>), postagem de blog, 21 dezembro de 2015.
- [Sch90] F. B. Schneider, “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial” (Implementando serviços tolerantes a falhas usando a abordagem de máquina de estados: um tutorial, <http://dl.acm.org/citation.cfm?id=98167>), em *ACM Computing Surveys*, vol. 22, nº 4, 1990.
- [Sec13] Securities and Exchange Commission, “Order In the Matter of Knight Capital Americas LLC” (Decisão na questão da Knight Capital Americas LLC, <https://www.sec.gov/litigation/admin/2013/34-70694.pdf>), arquivo 3-15570, 2013.
- [Sha00] G. Shao, F. Berman e R. Wolski, “Master/Slave Computing on the Grid” (Processamento master/slave na Grid,

<http://www.cs.ucsb.edu/~rich/publications/shao-hcw.pdf>), em *Heterogeneous Computing Workshop* (Workshop de Computação Heterogênea), 2000.

[Shu13] J. Shute et al., “F1: A Distributed SQL Database That Scales” (F1: um banco de dados SQL distribuído que escala, <https://research.google.com/pubs/pub41344.html>), em *Proc. VLDB 2013*, 2013.

[Sig10] B. H. Sigelman et al., “Dapper, a Large-Scale Distributed Systems Tracing Infrastructure” (Dapper, uma infraestrutura de monitoração de sistemas distribuídos de larga escala, <https://research.google.com/pubs/pub36356.html>), Google Technical Report (Relatório Técnico do Google), 2010.

[Sin15] A. Singh et al., “Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google’s Datacenter Network” (Surgimento do Jupiter: uma década de topologias clos e controle centralizado na rede de datacenters do Google, <https://research.google.com/pubs/pub43837.html>), em *SIGCOMM ’15*.

[Skel13] M. Skelton, “Operability can Improve if Developers Write a Draft Run Book” (Operações podem melhorar se os desenvolvedores escreverem o rascunho de um manual de execução, <http://blog.softwareoperability.com/2013/10/16/operability-can-improve-if-developers-write-a-draft-run-book/>), postagem de blog, 16 de outubro de 2013.

[Slo11] B. Treynor Sloss, “Gmail back soon for everyone” (Gmail de volta para todos em breve, <http://gmailblog.blogspot.com/2011/02/gmail-back-soon-for-everyone.html>), postagem de blog, 28 de fevereiro de 2011.

[Tat99] S. Tatham, “How to Report Bugs Effectively” (Como relatar bugs de modo eficiente, <http://www.chiark.greenend.org.uk/~sgtatham/bugs.html>), 1999.

[Ver15] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune e J. Wilkes, “Large-scale cluster management at Google with Borg”

(Gerenciamento de clusters de larga escala no Google com o Borg, <https://research.google.com/pubs/pub43438.html>), em *Proceedings of the European Conference on Computer Systems* (Anais da Conferência Europeia de Sistemas de Computação), 2015.

- [Wal89] D. R. Wallace e R. U. Fujii, “Software Verification and Validation: An Overview” (Verificação e validação de software: uma visão geral, http://www-usr.inf.ufsm.br/~ceretta/papers/fujii89_software_vv.pdf), *IEEE Software*, vol. 6, nº 3 (maio de 1989), pp. 10, 17.
- [War14] R. Ward e B. Beyer, “BeyondCorp: A New Approach to Enterprise Security” (BeyondCorp: uma nova abordagem para segurança corporativa, <https://www.usenix.org/publications/login/dec14/ward>), em ;login:, vol. 39, nº 6, dezembro de 2014.
- [Whi12] J. A. Whittaker, J. Arbon e J. Carollo, *How Google Tests Software* (Como o Google testa softwares): Addison-Wesley, 2012.
- [Woo96] A. Wood, “Predicting Software Reliability” (Prevendo a confiabilidade do software, <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=544240>), em *Computer*, vol. 29, nº 11, 1996.
- [Wri12a] H. K. Wright, “Release Engineering Processes, Their Faults and Failures” (Processos de Engenharia de Release, suas faltas e falhas, <http://www.hyrumwright.org/papers/dissertation.pdf>), (seção 7.2.2.2) Tese de doutorado, Universidade do Texas em Austin, 2012.
- [Wri12b] H. K. Wright e D. E. Perry, “Release Engineering Practices and Pitfalls” (Práticas e armadilhas da Engenharia de Release, <http://www.hyrumwright.org/papers/icse2012.pdf>), em *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)* (Anais da 34ª Conferência Internacional de Engenharia de Software de 2012). (IEEE, 2012), pp. 1281-1284.
- [Wri13] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, Z. Wan, “Large-Scale Automated Refactoring Using ClangMR” (Refatoração automatizada em larga escala usando ClangMR, <http://static.googleusercontent.com/media/research.google.com/en/us/pubs/>

em *Proceedings of the 29th International Conference on Software Maintenance (ICSM '13)* (Anais da 29^a Conferência Internacional de Manutenção de Software em 2013), (IEEE, 2013), pp. 548-551.

[Zoo14] Projeto ZooKeeper (Fundação Apache), “ZooKeeper Recipes and Solutions” (Receitas e soluções com ZooKeeper, <http://zookeeper.apache.org/doc/trunk/recipes.html>), na documentação do ZooKeeper 3.4, 2014.

Sobre os autores

Betsy Beyer é Redatora Técnica (Technical Writer) do Google em Nova York, especializada em Site Reliability Engineering (Engenharia de Confiabilidade de Sites). Anteriormente, escreveu documentação para as Equipes de Operações de Datacenter e de Hardware no Google em Mountain View e para os datacenters globalmente distribuídos. Antes de se mudar para Nova York, Betsy dava palestras sobre escrita técnica na Universidade de Stanford. Em sua jornada para a carreira atual, Betsy estudou Relações Internacionais e Literatura Inglesa e graduou-se por Stanford e Tulane.

Chris Jones é Site Reliability Engineer (Engenheiro de Confiabilidade de Sites) do Google App Engine: um produto de plataforma como serviço na nuvem, que serve a mais de 28 bilhões de requisições por dia. Residente em San Francisco, anteriormente foi responsável por cuidar das estatísticas de anúncios, armazém de dados (data warehousing) e sistemas de suporte a clientes no Google. No passado, Chris trabalhou com TI acadêmica, analisou dados para campanhas políticas e se envolveu com alguns hackings leves no kernel do BSD; graduou-se em Engenharia de Computação, Economia e Políticas Tecnológicas durante esse processo. Também é um engenheiro profissional certificado.

Jennifer Petoff é Gerente de Programa (Program Manager) na equipe de Site Reliability Engineering do Google e reside em Dublin, na Irlanda. Já gerenciou projetos globais de grande porte envolvendo domínios amplos que incluíam pesquisa científica, engenharia, recursos humanos e operações em publicidade. Jennifer juntou-se ao Google após ter passado oito anos na indústria química. Tem doutorado em Química pela Universidade de Stanford, além de bacharelado em Química e graduação em Psicologia pela Universidade de Rochester.

Niall Murphy lidera a equipe de Site Reliability Engineering do Ads do Google na Irlanda. Está envolvido com o mercado de internet há

aproximadamente vinte anos e, atualmente, é presidente do INEX – o peering hub da Irlanda. É autor e coautor de vários artigos técnicos e/ou livros, incluindo *IPv6 Network Administration* (Administração de rede IPv6) da O'Reilly, e de uma série de RFCs. Atualmente, está escrevendo uma história da internet na Irlanda em parceria com outro autor, e tem graduação em Ciência da Computação, Matemática e Estudos de Poesia, o que, certamente, é algum tipo de erro. Mora em Dublin com sua esposa e dois filhos.

Colofão

O animal na capa de *Site Reliability Engineering* é o lagarto-monitor ornamentado: um réptil nativo da África Ocidental e Central. Até 1997, era considerado uma subespécie do lagarto-do-nilo (*Varanus niloticus*), mas, atualmente, é classificado como uma forma polimórfica do *Varanus stellatus* e do *Varanus niloticus* por causa dos diferentes padrões de sua pele. Também ocupa regiões menos extensas que o lagarto-do-nilo, preferindo um *habitat* de florestas tropicais baixas.

Os lagartos-monitores ornamentados são lagartos grandes, capazes de atingir de 1,8 m a 2,1 m. Têm cores mais brilhantes que os lagartos-do-nilo, com uma pele verde-azeitona mais escura e menos faixas de pontos amarelos brilhantes, que se estendem dos ombros até a cauda. Como todos os lagartos-monitores, esse animal tem um corpo musculoso robusto, garras afiadas e uma cabeça alongada. Suas narinas ficam no alto de seus focinhos, permitindo que permaneçam por certo tempo na água. São excelentes nadadores e escaladores, o que lhes permite se sustentar com uma dieta à base de peixes, sapos, ovos, insetos e pequenos mamíferos.

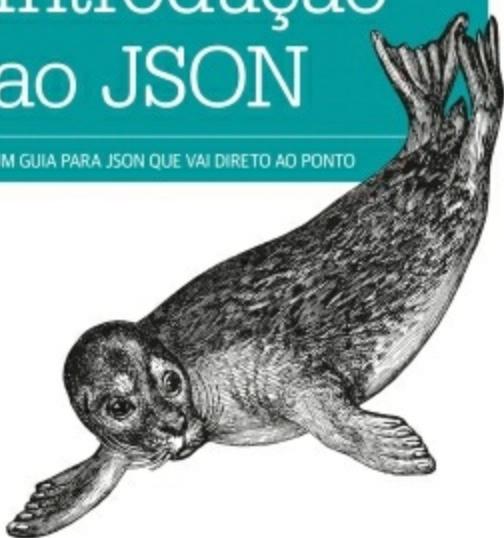
Os lagartos-monitores com frequência são usados como animais de estimação, embora exijam bastante cuidado e não sejam adequados para iniciantes. Podem ser perigosos quando se sentem ameaçados (dando chicotadas com suas caudas poderosas, arranhando ou mordendo), mas é possível domesticá-los até certo ponto, com um tratamento regular, ensinando-lhes a associar a presença de seus donos com o recebimento de comida.

Muitos dos animais nas capas dos livros da O'Reilly estão ameaçados; todos eles são importantes para o mundo. Para saber mais sobre como você pode ajudar, acesse animals.oreilly.com.

O'REILLY®

Introdução ao JSON

UM GUIA PARA JSON QUE VAI DIRETO AO PONTO



novatec

Lindsay Bassett

Introdução ao JSON

Bassett, Lindsay

9788575227459

152 páginas

[Compre agora e leia](#)

O que é JSON (JavaScript Object Notation, ou Notação de objetos JavaScript) e como podemos colocá-lo para funcionar? Este guia conciso ajudará os profissionais ocupados de TI a estarem rapidamente preparados para usar esse formato popular para intercâmbio de dados e possibilita uma profunda compreensão sobre o funcionamento do JSON. A autora Lindsay Bassett começa com uma visão geral da sintaxe, dos tipos de dados, da formatação e das questões relacionadas à segurança com o JSON antes de explorar as diversas maneiras como esse formato pode ser usado atualmente. Entre Web APIs, bibliotecas desenvolvidas na linguagem do lado servidor, bancos de dados NoSQL e frameworks do lado cliente, o JSON se destacou como uma alternativa viável ao XML para a troca de dados entre plataformas diferentes. Se você tem alguma experiência com programação e uma compreensão básica de HTML e de JavaScript, este livro é ideal para você. •Aprenda por que a sintaxe do JSON representa dados

por meio de pares nome-valor. •Explore os tipos de dados JSON, incluindo objetos, strings, números e arrays. •Descubra como é possível combater problemas comuns de segurança. •Saiba como o JSON Schema pode verificar se os dados estão formatados corretamente. •Analise o relacionamento entre navegadores, web APIs e o JSON. •Entenda como os servidores web podem tanto solicitar quanto criar dados. •Descubra como a jQuery e outros frameworks do lado cliente usam o JSON. •Aprenda por que o banco de dados NoSQL CouchDB utiliza JSON para armazenar dados.

[Compre agora e leia](#)

O'REILLY®

Padrões para Kubernetes

Elementos reutilizáveis no design de aplicações
nativas de nuvem



novatec

Bilgin Ibryam
Roland Huß

Padrões para Kubernetes

Ibryam, Bilgin

9788575228159

272 páginas

[Compre agora e leia](#)

O modo como os desenvolvedores projetam, desenvolvem e executam software mudou significativamente com a evolução dos microserviços e dos contêineres. Essas arquiteturas modernas oferecem novas primitivas distribuídas que exigem um conjunto diferente de práticas, distinto daquele com o qual muitos desenvolvedores, líderes técnicos e arquitetos estão acostumados. Este guia apresenta padrões comuns e reutilizáveis, além de princípios para o design e a implementação de aplicações nativas de nuvem no Kubernetes. Cada padrão inclui uma descrição do problema e uma solução específica no Kubernetes. Todos os padrões acompanham e são demonstrados por exemplos concretos de código. Este livro é ideal para desenvolvedores e arquitetos que já tenham familiaridade com os conceitos básicos do Kubernetes, e que queiram aprender a solucionar desafios comuns no ambiente nativo de nuvem, usando padrões de projeto de uso comprovado. Você conhecerá as seguintes classes de padrões:

- Padrões básicos,

que incluem princípios e práticas essenciais para desenvolver aplicações nativas de nuvem com base em contêineres. • Padrões comportamentais, que exploram conceitos mais específicos para administrar contêineres e interações com a plataforma. • Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos. • Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes. • Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).

[Compre agora e leia](#)

CANDLESTICK

Um método para ampliar lucros na Bolsa de Valores



novatec

Carlos Alberto Debastiani

Candlestick

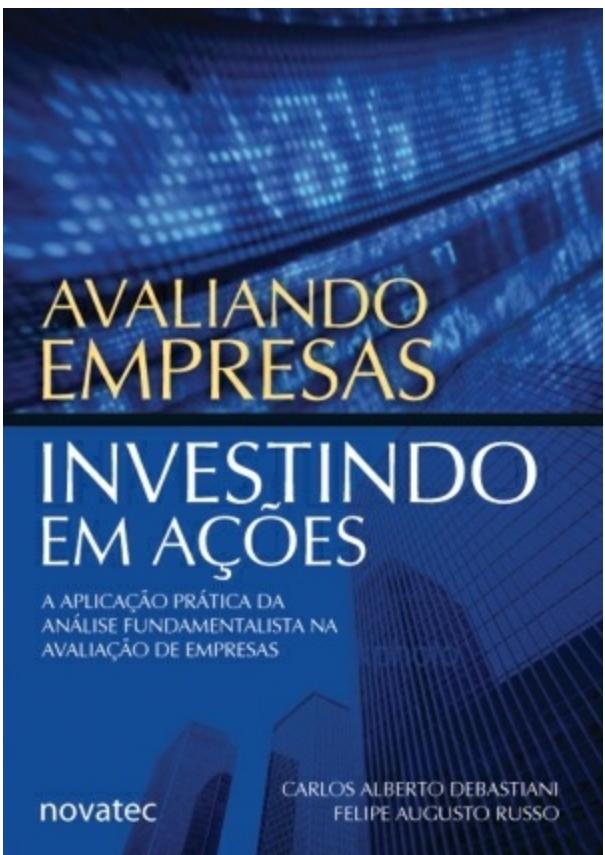
Debastiani, Carlos Alberto
9788575225943
200 páginas

[Compre agora e leia](#)

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos.

Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.

[Compre agora e leia](#)



Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto

9788575225974

224 páginas

[Compre agora e leia](#)

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.

[Compre agora e leia](#)

Marcos Abe

MANUAL DE ANÁLISE TÉCNICA

ESSÊNCIA E ESTRATÉGIAS AVANÇADAS

TUDO O QUE UM INVESTIDOR PRECISA SABER PARA
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE

novatec

Manual de Análise Técnica

Abe, Marcos

9788575227022

256 páginas

[Compre agora e leia](#)

Este livro aborda o tema Investimento em Ações de maneira inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá: - os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado; - identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos; - estruturar e proteger operações por meio do gerenciamento de capital. Destina-se a iniciantes na bolsa

de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.

[Compre agora e leia](#)