VOLUME 2

THE PRACTICE OF CLOUD SYSTEM ADMINISTRATION

DESIGNING AND OPERATING LARGE DISTRIBUTED SYSTEMS

THOMAS A. LIMONCELLI • STRATA R. CHALUP • CHRISTINA J. HOGAN

The Practice of Cloud System Administration

Volume 2

This page intentionally left blank

The Practice of Cloud System Administration

Designing and Operating Large Distributed Systems

Volume 2

Thomas A. Limoncelli
Strata R. Chalup
Christina J. Hogan

Addison-Wesley
Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data
Limoncelli, Tom.
The practice of cloud system administration : designing and operating large distributed systems / Thomas A. Limoncelli, Strata R. Chalup, Christina J. Hogan.
volumes cm
Includes bibliographical references and index.
ISBN-13: 978-0-321-94318-7 (volume 2 : paperback)
ISBN-10: 0-321-94318-X (volume 2 : paperback)
1. Computer networks—Management. 2. Computer systems. 3. Cloud computing. 4. Electronic data processing—Distributed processing. I. Chalup, Strata R. II. Hogan, Christina J. III. Title.
TK5105.5.L529 2015
004.67'82068—dc23 2014024033

ISBN-13: 978-0-321-94318-7
ISBN-10: 0-321-94318-X
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, September 2014

Contents at a Glance

Contents

This page intentionally left blank

Preface

Which of the following statements are true?

1. The most reliable systems are built using cheap, unreliable components.
2. The techniques that Google uses to scale to billions of users follow the same patterns you can use to scale a system that handles hundreds of users.
3. The more risky a procedure is, the more you should do it.
4. Some of the most important software features are the ones that users never see.
5. You should pick random machines and power them off.
6. The code for every feature Facebook will announce in the next six months is probably in your browser already.
7. Updating software multiple times a day requires little human effort.
8. Being oncall doesn't have to be a stressful, painful experience.
9. You shouldn't monitor whether machines are up.
10. Operations and management can be conducted using the scientific principles of experimentation and evidence.
11. Google has rehearsed what it would do in case of a zombie attack.

All of these statements are true. By the time you finish reading this book, you'll know why.

This is a book about building and running cloud-based services on a large scale: internet-based services for millions or billions of users. That said, every day more and more enterprises are adopting these techniques. Therefore, this is a book for everyone.

The intended audience is system administrators and their managers. We do not assume a background in computer science, but we do assume experience with UNIX/Linux system administration, networking, and operating system concepts.

Our focus is on building and operating the services that make up the cloud, not a guide to using cloud-based services.

Cloud services must be available, fast, and secure. At cloud scale, this is a unique engineering feat. Therefore cloud-scale services are engineered differently than your typical enterprise service. Being available is important because the Internet is open 24 × 7 and has users in every time zone. Being fast is important because users are frustrated by slow services, so slow services lose out to faster rivals. Being secure is important because, as caretakers of other people's data, we are duty-bound (and legally responsible) to protect people's data.

These requirements are intermixed. If a site is not secure, by definition, it cannot be made reliable. If a site is not fast, it is not sufficiently available. If a site is down, by definition, it is not fast.

The most visible cloud-scale services are web sites. However, there is a huge ecosystem of invisible internet-accessible services that are not accessed with a browser. For example, smartphone apps use API calls to access cloud-based services.

For the remainder of this book we will tend to use the term "distributed computing" rather than "cloud computing." *Cloud computing* is a marketing term that means different things to different people. *Distributed computing* describes an architecture where applications and services are provided using many machines rather than one.

This is a book of fundamental principles and practices that are timeless. Therefore we don't make recommendations about which specific products or technologies to use. We could provide a comparison of the top five most popular web servers or NoSQL databases or continuous build systems. If we did, then this book would be out of date the moment it is published. Instead, we discuss the qualities one should look for when selecting such things. We provide a model to work from. This approach is intended to prepare you for a long career where technology changes over time but you are always prepared. We will, of course, illustrate our points with specific technologies and products, but not as an endorsement of those products and services.

This book is, at times, idealistic. This is deliberate. We set out to give the reader a vision of how things can be, what to strive for. We are here to raise the bar.

About This Book

The book is structured in two parts, Design and Operations.

Part I captures our thinking on the design of large, complex, cloud-based distributed computing systems. After the Introduction, we tackle each element of design from the bottom layers to the top. We cover distributed systems from the point of view of a system administrator, not a computer scientist. To operate a system, one must be able to understand its internals.

Part II describes how to run such systems. The first chapters cover the most fundamental issues. Later chapters delve into more esoteric technical activities, then high-level planning and strategy that tie together all of the above.

At the end is extra material including an assessment system for operations teams, a highly biased history of distributed computing, templates for forms mentioned in the text, recommended reading, and other reference material.

We're excited to present a new feature of our book series: our operational assessment system. This system consists of a series of assessments you can use to evaluate your operations and find areas of improvement. The assessment questions and "Look For" recommendations are found in Appendix A. Chapter 20 is the instruction manual.

Acknowledgments

This book wouldn't have been possible without the help and feedback we received from our community and people all over the world. The DevOps community was generous in its assistance.

First, we'd like to thank our spouses and families: Christine Polk, Mike Chalup, and Eliot and Joanna Lear. Your love and patience make all this possible.

If we have seen further, it is by standing on the shoulders of giants. Certain chapters relied heavily on support and advice from particular people: John Looney and Cian Synnott (Chapter 1); Marty Abbott and Michael Fisher (Chapter 5); Damon Edwards, Alex Honor, and Jez Humble (Chapters 9 and 10); John Allspaw (Chapter 12); Brent Chapman (Chapter 15); Caskey Dickson and Theo Schlossnagle (Chapters 16 and 17); Arun Kejariwal and Bruce Yan (Chapter 18); Benjamin Treynor Sloss (Chapter 19); and Geoff Halprin (Chapter 20 and Appendix A).

Thanks to Gene Kim for the "strategic" inspiration and encouragement.

Dozens of people helped us—some by supplying anecdotes, some by reviewing parts of or the entire book. The only fair way to thank them all is alphabetically and to apologize in advance to anyone we left out: Thomas Baden, George Beech, Raymond Blum, Kyle Brandt, Mark Burgess, Nick Craver, Geoff Dalgas, Robert P. J. Day, Patrick Debois, Bill Duane, Paul Evans, David Fullerton, Tom Geller, Peter Grace, Elizabeth Hamon Reid, Jim Hickstein, Zachary Hueras, Matt Jones, Jennifer Joy, Jimmy Kaplowitz, Daniel V. Klein, Steven Levine, Cory Lueninghoener, Shane Madden, Jim Maurer, Stephen McHenry, Dinah McNutt, Scott Hazen Mueller, Steve Murawski, Mohit Muthanna, Lenny Rachitsky, Amy Rich, Adele Shakal, Bart Silverstrim, Josh Simon, Joel Spolsky, Desiree Sylvester, Win Treese, Todd Underwood, Nicole Forsgren Velasquez, and Dave Zwieback.

Last but not least, thanks to everyone from Addison-Wesley. In particular, thanks to Debra Williams Cauley, for guiding us to Addison-Wesley and steering

us the entire way; Michael Thurston, for editing our earliest drafts and reshaping them to be much, much better; Kim Boedigheimer, who coordinated and assisted us calmly even when we were panicking; Lori Hughes, our LaTeX wizard; Julie Nahil, our production manager; Jill Hobbs, our copyeditor; and John Fuller and Mark Taub, for putting up with all our special requests!

Part I Design: Building It

Chapter 1: Designing in a Distributed World
Overview of how distributed systems are designed.
Chapter 2: Designing for Operations
Features software should have to enable smooth operations.
Chapter 3: Selecting a Service Platform
Physical and virtual machines, private and public clouds.
Chapter 4: Application Architectures
Building blocks for creating web and other applications.
Chapter 5: Design Patterns for Scaling
Building blocks for growing a service.
Chapter 6: Design Patterns for Resiliency
Building blocks for creating systems that survive failure.

Part II Operations: Running It

Chapter 7: Operations in a Distributed World
Overview of how distributed systems are run.
Chapter 8: DevOps Culture
Introduction to DevOps culture, its history and practices.
Chapter 9: Service Delivery: The Build Phase
How a service gets built and prepared for production.
Chapter 10: Service Delivery: The Deployment Phase
How a service is tested, approved, and put into production.
Chapter 11: Upgrading Live Services
How to upgrade services without downtime.
Chapter 12: Automation
Creating tools and automating operational work.
Chapter 13: Design Documents
Communicating designs and intentions in writing.
Chapter 14: Oncall
Handling exceptions.
Chapter 15: Disaster Preparedness
Making systems stronger through planning and practice.

This page intentionally left blank

About the Authors

Thomas A. Limoncelli is an internationally recognized author, speaker, and system administrator. During his seven years at Google NYC, he was an SRE for projects such as Blog Search, Ganeti, and various internal enterprise IT services. He now works as an SRE at Stack Exchange, Inc., home of ServerFault.com and StackOverflow.com. His first paid system administration job was as a student at Drew University in 1987, and he has since worked at small and large companies, including AT&T/Lucent Bell Labs. His best-known books include *Time Management for System Administrators* (O'Reilly) and *The Practice of System and Network Administration, Second Edition* (Addison-Wesley). His hobbies include grassroots activism, for which his work has been recognized at state and national levels. He lives in New Jersey.

Strata R. Chalup has been leading and managing complex IT projects for many years, serving in roles ranging from project manager to director of operations. Strata has authored numerous articles on management and working with teams and has applied her management skills on various volunteer boards, including BayLISA and SAGE. She started administering VAX Ultrix and Unisys UNIX in 1983 at MIT in Boston, and spent the dot-com years in Silicon Valley building internet services for clients like iPlanet and Palm. In 2007, she joined Tom and Christina to create the second edition of *The Practice of System and Network Administration* (Addison-Wesley). Her hobbies include working with new technologies, including Arduino and various 2D CAD/CAM devices, as well as being a master gardener. She lives in Santa Clara County, California.

Christina J. Hogan has twenty years of experience in system administration and network engineering, from Silicon Valley to Italy and Switzerland. She has gained experience in small startups, mid-sized tech companies, and large global corporations. She worked as a security consultant for many years and her customers included eBay, Silicon Graphics, and SystemExperts. In 2005 she and Tom

shared the SAGE Outstanding Achievement Award for their book *The Practice of System and Network Administration* (Addison-Wesley). She has a bachelor's degree in mathematics, a master's degree in computer science, a doctorate in aeronautical engineering, and a diploma in law. She also worked for six years as an aerodynamicist in a Formula 1 racing team and represented Ireland in the 1988 Chess Olympiad. She lives in Switzerland.

Introduction

The goal of this book is to help you build and run the best cloud-scale service possible. What is the ideal environment that we seek to create?

Business Objectives

Simply stated, the end result of our ideal environment is that business objectives are met. That may sound a little boring but actually it is quite exciting to work where the entire company is focused and working together on the same goals.

To achieve this, we must understand the business objectives and work backward to arrive at the system we should build.

Meeting business objectives means knowing what those objectives are, having a plan to achieve them, and working through the roadblocks along the way.

Well-defined business objectives are measurable, and such measurements can be collected in an automated fashion. A dashboard is automatically generated so everyone is aware of progress. This transparency enhances trust.

Here are some sample business objectives:

- Sell our products via a web site
- Provide service 99.99 percent of the time
- Process *x* million purchases per month, growing 10 percent monthly
- Introduce new features twice a week
- Fix major bugs within 24 hours

In our ideal environment, business and technical teams meet their objectives and project goals predictably and reliably. Because of this, both types of teams trust that other teams will meet their future objectives. As a result, teams can plan better. They can make more aggressive plans because there is confidence that external dependencies will not fail. This permits even more aggressive planning. Such an approach creates an upward spiral that accelerates progress throughout the company, benefiting everyone.

Ideal System Architecture

The ideal service is built on a solid architecture. It meets the requirements of the service today and provides an obvious path for growth as the system becomes more popular and receives more traffic. The system is resilient to failure. Rather than being surprised by failures and treating them as exceptions, the architecture accepts that hardware and software failures are a part of the physics of information technology (IT). As a result, the architecture includes redundancy and resiliency features that work around failures. Components fail but the system survives.

Each subsystem that makes up our service is itself a service. All subsystems are programmable via an application programming interface (API). Thus, the entire system is an ecosystem of interconnected subservices. This is called a service-oriented architecture (SOA). Because all these systems communicate over the same underlying protocol, there is uniformity in how they are managed. Because each subservice is loosely coupled to the others, all of these services can be independently scaled, upgraded, or replaced.

The geometry of the infrastructure is described electronically. This electronic description is read by IT automation systems, which then build the production environment without human intervention. Because of this automation, the entire infrastructure can be re-created elsewhere. Software engineers use the automation to make micro-versions of the environment for their personal use. Quality and test engineers use the automation to create environments for system tests.

This "infrastructure as code" can be achieved whether we use physical machines or virtual machines, and whether they are in datacenters we run or are hosted by a cloud provider. With virtual machines there is an obvious API available for spinning up a new machine. However, even with physical machines, the entire flow from bare metal to working system can be automated. In our ideal world the automation makes it possible to create environments using combinations of physical and virtual machines. Developers may build the environment out of virtual machines. The production environment might consist of a mixture of physical and virtual machines. The temporary and unexpected need for additional capacity may require extending the production environment into one or more cloud providers for some period of time.

Ideal Release Process

Our ideal environment has a smooth flow of code from development to operations.

Traditionally (not in our ideal environment) the sequence looks like this:

1. Developers check code into a repository.
2. Test engineers put the code through a number of tests.

3. If all the tests pass, the a release engineer builds the packages that will be used to deploy the software. Most of the files come from the source code repository, but some files may be needed from other sources such as a graphics department or documentation writers.
4. A test environment is created; without an "infrastructure as code" model, this may take weeks.
5. The packages are deployed into a test environment.
6. Test engineers perform further tests, focusing on the interaction between subsystems.
7. If all these tests succeed, the code is put into production.
8. System administrators upgrade systems while looking for failures.
9. If there are failures, the software is rolled back.

Doing these steps manually incurs a lot of risk, owing to the assumptions that the right people are available, that the steps are done the same way every time, that nobody makes mistakes, and that all the tasks are completed in time.

Mistakes, bugs, and errors happen, of course—and as a result defects are passed down the line to the next stage. When a mistake is discovered the flow of progress is reversed as the team members who were responsible for the previous stage are told to fix their problem. This means progress is halted and time is lost.

A typical response to a risky process is to do it as rarely as possible. Thus there is a temptation to do as few releases as possible. The result is "mega-releases" launched only a few times a year.

However, by batching up so many changes at once, we actually create more risk. How can we be sure thousands of changes, released simultaneously, will all work on the first try? We can't. Therefore we become even more recalcitrant toward and fearful of making changes. Soon change becomes nearly impossible and innovation comes to a halt.

Not so in our ideal environment.

In our ideal environment, we find automation that eliminates all manual steps in the software build, test, release, and deployment processes. The automation accurately and consistently performs tests that prevent defects from being passed to the next step. As a consequence, the flow of progress is in one direction: forward.

Rather than mega-releases, our ideal environment creates micro-releases. We reduce risk by doing many deployments, each with a few small changes. In fact, we might do 100 deployments per day.

1. When the developers check in code, a system detects this fact and triggers a series of automated tests. These tests verify basic code functionality.
2. If these tests pass, the process of building the packages is kicked off and runs in a completely automated fashion.

3. The successful creation of new packages triggers the creation of a test environment. Building a test environment used to be a long week of connecting cables and installing machines. But with infrastructure as code, the entire environment is created quickly with no human intervention.
4. When the test environment is complete, a series of automated tests are run.
5. On successful completion the new packages are rolled out to production. The roll-out is also automated but orderly and cautious.
6. Certain systems are upgraded first and the system watches for failures. Since the test environment was built with the same automation that built the production environment, there should be very few differences.
7. Seeing no failures, the new packages are rolled out to more and more systems until the entire production environment is upgraded.

In our ideal environment all problems are caught before they reach production. That is, roll-out is not a form of testing. Failure during a roll-out to production is essentially eliminated. However, if a failure does happen, it would be considered a serious issue warranting pausing new releases from going into production until a root causes analysis is completed. Tests are added to detect and prevent future occurrences of this failure. Thus, the system gets stronger over time.

Because of this automation, the traditional roles of release engineering, quality assurance, and deployment are practically unrecognizable from their roles at a traditional company. Hours of laborious manual toil are eliminated, leaving more time for improving the packaging system, improving the software quality, and refining the deployment process. In other words, people spend more time making improvements in how work is done rather than doing work itself.

A similar process is used for third-party software. Not all systems are homegrown or come with source code. Deploying third-party services and products follows a similar pattern of release, testing, deployment. However, because these products and services are developed externally, they require a slightly different process. New releases are likely to occur less frequently and we have less control over what is in each new release. The kind of testing these components require is usually related to features, compatibility, and integration.

Ideal Operations

Once the code is in production, operational objectives take precedence. The software is instrumented so that it can be monitored. Data is collected about how long it takes to process transactions from external users as well as from internal APIs. Other indicators such as memory usage are also monitored. This data is collected so that operational decisions can be made based on data, not guesses, luck, or

hope. The data is stored for many years so it may be used to predict the future capacity needs.

Measurements are used to detect internal problems while they are small, long before they result in a user-visible outage. We fix problems before they become outages. An actual outage is rare and would be investigated with great diligence. When problems are detected there is a process in place to make sure they are identified, worked on, and resolved quickly.

An automated system detects problems and alerts whoever is oncall. Our oncall schedule is a rotation constructed so that each shift typically receives a manageable number of alerts. At any given time one person is the primary oncall person and is first to receive any alerts. If that individual does not respond in time, a secondary person is alerted. The oncall schedule is prepared far enough in advance that people can plan vacations, recreational activities, and personal time.

There is a "playbook" of instructions on how to handle every alert that can be generated. Each type of alert is documented with a technical description of what is wrong, what the business impact is, and how to fix the issue. The playbook is continually improved. Whoever is oncall uses the playbook to fix the problem. If it proves insufficient, there is a well-defined escalation path, usually to the oncall person for the related subsystem. Developers also participate in the oncall rotation so they understand the operational pain points of the system they are building.

All failures have a corresponding countermeasure, whether it is manually or automatically activated. Countermeasures that are activated frequently are always automated. Our monitoring system detects overuse, as this may indicate a larger problem. The monitoring system collects internal indicator data used by engineers to reduce the failure rate as well as improve the countermeasure.

The less frequently a countermeasure is activated, the less confident we are that it will work the next time it is needed. Therefore infrequently activated countermeasures are periodically and automatically exercised by intentionally causing failures. Just as we require school children to practice fire drills so that everyone knows what to do in an emergency, so we practice fire drills with our operational practices. This way our team becomes experienced at implementing the countermeasures and is confident that they work. If a database failover process doesn't work due to an unexpected dependency, it is better to learn this during a live drill on Monday at 10 AM rather than during an outage at 4 AM on a Sunday morning. Again, we reduce risk by increasing repetition rather than shying away from it. The technical term for improving something through repetition is called "practice." We strongly believe that practice makes perfect.

Our ideal environment scales automatically. As more capacity is needed, additional capacity comes from internal or external cloud providers. Our dashboards indicate when re-architecting will be a better solution than simply allocating more RAM, disk, or CPU.

Scaling down is also automatic. When the system is overloaded or degraded, we never turn users away with a "503—Service Unavailable" error. Instead, the system automatically switches to algorithms that use less resources. Bandwidth fully utilized? Low-bandwidth versions of the service kick in, displaying fewer graphics or a more simplified user interface. Databases become corrupted? A read-only version of the service keeps most users satisfied.

Each feature of our service can be individually enabled or disabled. If a feature turns out to have negative consequences, such as security holes or unexpectedly bad performance, it can be disabled without deploying a different software release.

When a feature is revised, the new code does not eliminate the old functionality. The new behavior can be disabled to reveal the old behavior. This is particularly useful when rolling out a new user interface. If a release can produce both the old and new user interface, it can be enabled on a per-user basis. This enables us to get feedback from "early access" users. On the official release date, the new feature is enabled for successively larger and larger groups. If performance problems are found, the feature can easily be reverted or switched off entirely.

In our ideal environment there is excellent operational hygiene. Like brushing our teeth, we regularly do the things that preserve good operational health. We maintain clear and updated documentation for how to handle every countermeasure, process, and alert. Overactive alerts are fine-tuned, not ignored. Open bug counts are kept to a minimum. Outages are followed by the publication of a postmortem report with recommendations on how to improve the system in the future. Any "quick fix" is followed by a root causes analysis and the implementation of a long-term fix.

Most importantly, the developers and operations people do not think of themselves as two distinct teams. They are simply specializations within one large team. Some people write more code than others; some people do more operational projects than others. All share responsibility for maintaining high uptime. To that end, all members participate in the oncall (pager) rotation. Developers are most motivated to improve code that affects operations when they feel the pain of operations, too. Operations must understand the development process if they are to be able to constructively collaborate.

Now you know our vision of an ideal environment. The remainder of this book will explain how to create and run it.

This page intentionally left blank

Chapter 1

Designing in a Distributed World

> There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.
>
> —C.A.R. Hoare, The 1980 ACM Turing Award Lecture

How does Google Search work? How does your Facebook Timeline stay updated around the clock? How does Amazon scan an ever-growing catalog of items to tell you that people who bought this item also bought socks?

Is it magic? No, it's distributed computing.

This chapter is an overview of what is involved in designing services that use distributed computing techniques. These are the techniques all large web sites use to achieve their size, scale, speed, and reliability.

Distributed computing is the art of building large systems that divide the work over many machines. Contrast this with traditional computing systems where a single computer runs software that provides a service, or client–server computing where many machines remotely access a centralized service. In distributed computing there are typically hundreds or thousands of machines working together to provide a large service.

Distributed computing is different from traditional computing in many ways. Most of these differences are due to the sheer size of the system itself. Hundreds or thousands of computers may be involved. Millions of users may be served. Billions and sometimes trillions of queries may be processed.

Terms to Know

Server: Software that provides a function or application program interface (API). (Not a piece of hardware.)

Service: A user-visible system or product composed of many servers.

Machine: A virtual or physical machine.

QPS: Queries per second. Usually how many web hits or API calls received per second.

Traffic: A generic term for queries, API calls, or other requests sent to a server.

Performant: A system whose performance conforms to (meets or exceeds) the design requirements. A neologism from merging "performance" and "conformant."

Application Programming Interface (API): A protocol that governs how one server talks to another.

Speed is important. It is a competitive advantage for a service to be fast and responsive. Users consider a web site sluggish if replies do not come back in 200 ms or less. Network latency eats up most of that time, leaving little time for the service to compose the page itself.

In distributed systems, failure is normal. Hardware failures that are rare, when multiplied by thousands of machines, become common. Therefore failures are assumed, designs work around them, and software anticipates them. Failure is an expected part of the landscape.

Due to the sheer size of distributed systems, operations must be automated. It is inconceivable to manually do tasks that involve hundreds or thousands of machines. Automation becomes critical for preparation and deployment of software, regular operations, and handling failures.

1.1 Visibility at Scale

To manage a large distributed system, one must have visibility into the system. The ability to examine internal state—called **introspection**—is required to operate, debug, tune, and repair large systems.

In a traditional system, one could imagine an engineer who knows enough about the system to keep an eye on all the critical components or "just knows" what is wrong based on experience. In a large system, that level of visibility must be actively created by designing systems that draw out the information and make it visible. No person or team can manually keep tabs on all the parts.

Distributed systems, therefore, require components to generate copious logs that detail what happened in the system. These logs are then aggregated to a central location for collection, storage, and analysis. Systems may log information that is very high level, such as whenever a user makes a purchase, for each web query, or for every API call. Systems may log low-level information as well, such as the parameters of every function call in a critical piece of code.

Systems should export metrics. They should count interesting events, such as how many times a particular API was called, and make these counters accessible.

In many cases, special URLs can be used to view this internal state. For example, the Apache HTTP Web Server has a "server-status" page (`http://www.example.com/server-status/`).

In addition, components of distributed systems often appraise their own health and make this information visible. For example, a component may have a URL that outputs whether the system is ready (OK) to receive new requests. Receiving as output anything other than the byte "O" followed by the byte "K" (including no response at all) indicates that the system does not want to receive new requests. This information is used by load balancers to determine if the server is healthy and ready to receive traffic. The server sends negative replies when the server is starting up and is still initializing, and when it is shutting down and is no longer accepting new requests but is processing any requests that are still in flight.

1.2 The Importance of Simplicity

It is important that a design remain as simple as possible while still being able to meet the needs of the service. Systems grow and become more complex over time. Starting with a system that is already complex means starting at a disadvantage.

Providing competent operations requires holding a mental model of the system in one's head. As we work we imagine the system operating and use this mental model to track how it works and to debug it when it doesn't. The more complex the system, the more difficult it is to have an accurate mental model. An overly complex system results in a situation where no single person understands it all at any one time.

In *The Elements of Programming Style*, Kernighan and Plauger (1978) wrote:

> Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

The same is true for distributed systems. Every minute spent simplifying a design pays off time and time again when the system is in operation.

1.3 Composition

Distributed systems are composed of many smaller systems. In this section, we explore three fundamental composition patterns in detail:

- Load balancer with multiple backend replicas
- Server with multiple backends
- Server tree

1.3.1 Load Balancer with Multiple Backend Replicas

The first composition pattern is the load balancer with multiple backend replicas. As depicted in Figure 1.1, requests are sent to the load balancer server. For each request, it selects one **backend** and forwards the request there. The response comes back to the load balancer server, which in turn relays it to the original requester.

The backends are called **replicas** because they are all clones or replications of each other. A request sent to any replica should produce the same response.

The load balancer must always know which backends are alive and ready to accept requests. Load balancers send **health check** queries dozens of times each second and stop sending traffic to that backend if the health check fails. A health check is a simple query that should execute quickly and return whether the system should receive traffic.

Picking which backend to send a query to can be simple or complex. A simple method would be to alternate among the backends in a loop—a practice called **round-robin**. Some backends may be more powerful than others, however,

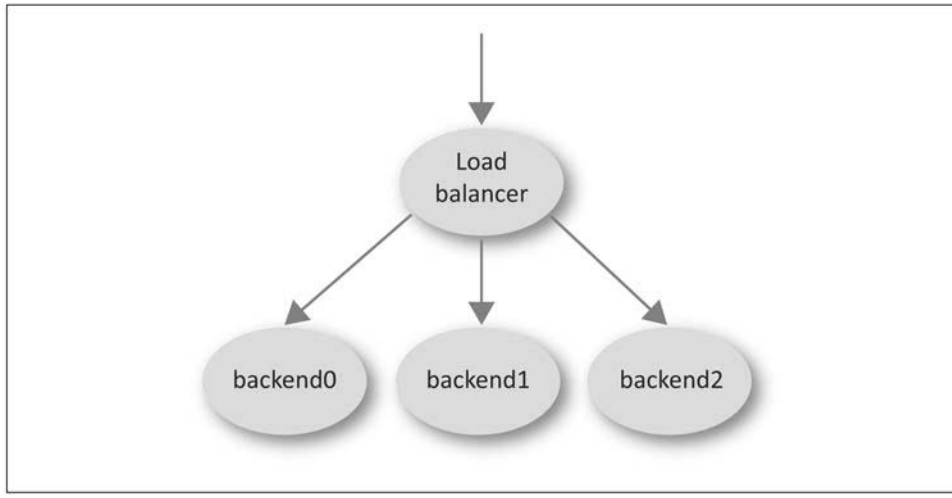

Figure 1.1: A load balancer with many replicas

and may be selected more often using a proportional round-robin scheme. More complex solutions include the **least loaded** scheme. In this approach, a load balancer tracks how loaded each backend is and always selects the least loaded one.

Selecting the least loaded backend sounds reasonable but a naive implementation can be a disaster. A backend may not show signs of being overloaded until long after it has actually become overloaded. This problem arises because it can be difficult to accurately measure how loaded a system is. If the load is a measurement of the number of connections recently sent to the server, this definition is blind to the fact that some connections may be long lasting while others may be quick. If the measurement is based on CPU utilization, this definition is blind to input/output (I/O) overload. Often a trailing average of the last 5 minutes of load is used. Trailing averages have a problem in that, as an average, they reflect the past, not the present. As a consequence, a sharp, sudden increase in load will not be reflected in the average for a while.

Imagine a load balancer with 10 backends. Each one is running at 80 percent load. A new backend is added. Because it is new, it has no load and, therefore, is the least loaded backend. A naive least loaded algorithm would send all traffic to this new backend; no traffic would be sent to the other 10 backends. All too quickly, the new backend would become absolutely swamped. There is no way a single backend could process the traffic previously handled by 10 backends. The use of trailing averages would mean the older backends would continue reporting artificially high loads for a few minutes while the new backend would be reporting an artificially low load.

With this scheme, the load balancer will believe that the new machine is less loaded than all the other machines for quite some time. In such a situation the machine may become so overloaded that it would crash and reboot, or a system administrator trying to rectify the situation might reboot it. When it returns to service, the cycle would start over again.

Such situations make the round-robin approach look pretty good. A less naive least loaded implementation would have some kind of control in place that would never send more than a certain number of requests to the same machine in a row. This is called a **slow start** algorithm.

Trouble with a Naive Least Loaded Algorithm

Without slow start, load balancers have been known to cause many problems. One famous example is what happened to the CNN.com web site on the day of the September 11, 2001, terrorist attacks. So many people tried to access CNN.com that the backends became overloaded. One crashed, and then crashed again after it came back up, because the naive least loaded algorithm

sent all traffic to it. When it was down, the other backends became overloaded and crashed. One at a time, each backend would get overloaded, crash, and become overloaded from again receiving all the traffic and crash again.

As a result the service was essentially unavailable as the system administrators rushed to figure out what was going on. In their defense, the web was new enough that no one had experience with handling sudden traffic surges like the one encountered on September 11.

The solution CNN used was to halt all the backends and boot them at the same time so they would all show zero load and receive equal amounts of traffic.

The CNN team later discovered that a few days prior, a software upgrade for their load balancer had arrived but had not yet been installed. The upgrade added a slow start mechanism.

1.3.2 Server with Multiple Backends

The next composition pattern is a server with multiple backends. The server receives a request, sends queries to many backend servers, and composes the final reply by combining those answers. This approach is typically used when the original query can easily be deconstructed into a number of independent queries that can be combined to form the final answer.

Figure 1.2a illustrates how a simple search engine processes a query with the help of multiple backends. The frontend receives the request. It relays the query to many backend servers. The spell checker replies with information so the search engine may suggest alternate spellings. The web and image search backends reply with a list of web sites and images related to the query. The advertisement server

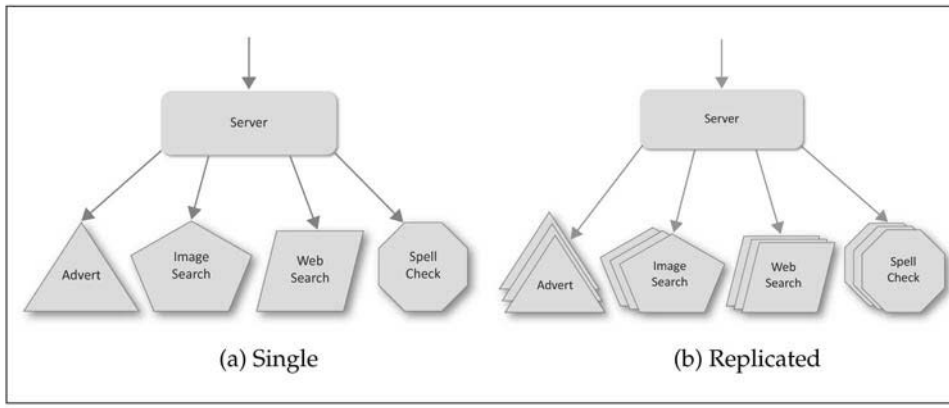

Figure 1.2: This service is composed of a server and many backends.

replies with advertisements relevant to the query. Once the replies are received, the frontend uses this information to construct the HTML that makes up the search results page for the user, which is then sent as the reply.

Figure 1.2b illustrates the same architecture with replicated, load-balanced, backends. The same principle applies but the system is able to scale and survive failures better.

This kind of composition has many advantages. The backends do their work in parallel. The reply does not have to wait for one backend process to complete before the next begins. The system is loosely coupled. One backend can fail and the page can still be constructed by filling in some default information or by leaving that area blank.

This pattern also permits some rather sophisticated latency management. Suppose this system is expected to return a result in 200 ms or less. If one of the backends is slow for some reason, the frontend doesn't have to wait for it. If it takes 10 ms to compose and send the resulting HTML, at 190 ms the frontend can give up on the slow backends and generate the page with the information it has. The ability to manage a latency time budget like that can be very powerful. For example, if the advertisement system is slow, search results can be displayed without any ads.

To be clear, the terms "frontend" and "backend" are a matter of perspective. The frontend sends requests to backends, which reply with a result. A server can be both a frontend and a backend. In the previous example, the server is the backend to the web browser but a frontend to the spell check server.

There are many variations on this pattern. Each backend can be replicated for increased capacity or resiliency. Caching may be done at various levels.

The term **fan out** refers to the fact that one query results in many new queries, one to each backend. The queries "fan out" to the individual backends and the replies **fan in** as they are set up to the frontend and combined into the final result.

Any fan in situation is at risk of having congestion problems. Often small queries may result in large responses. Therefore a small amount of bandwidth is used to fan out but there may not be enough bandwidth to sustain the fan in. This may result in congested network links and overloaded servers. It is easy to engineer the system to have the right amount of network and server capacity if the sizes of the queries and replies are consistent, or if there is an occasional large reply. The difficult situation is engineering the system when there are sudden, unpredictable bursts of large replies. Some network equipment is engineered specifically to deal with this situation by dynamically provisioning more buffer space to such bursts. Likewise, the backends can rate-limit themselves to avoid creating the situation in the first place. Lastly, the frontends can manage the congestion themselves by controlling the new queries they send out, by notifying the backends to slow down, or by implementing emergency measures to handle the flood better. The last option is discussed in Chapter 5.

1.3.3 Server Tree

The other fundamental composition pattern is the **server tree**. As Figure 1.3 illustrates, in this scheme a number of servers work cooperatively with one as the root of the tree, parent servers below it, and leaf servers at the bottom of the tree. (In computer science, trees are drawn upside-down.) Typically this pattern is used to access a large dataset or **corpus**. The corpus is larger than any one machine can hold; thus each leaf stores one fraction or **shard** of the whole.

To query the entire dataset, the root receives the original query and forwards it to the parents. The parents forward the query to the leaf servers, which search their parts of the corpus. Each leaf sends its findings to the parents, which sort and filter the results before forwarding them up to the root. The root then takes the response from all the parents, combines the results, and replies with the full answer.

Imagine you wanted to find out how many times George Washington was mentioned in an encyclopedia. You could read each volume in sequence and arrive at the answer. Alternatively, you could give each volume to a different person and have the various individuals search their volumes in parallel. The latter approach would complete the task much faster.

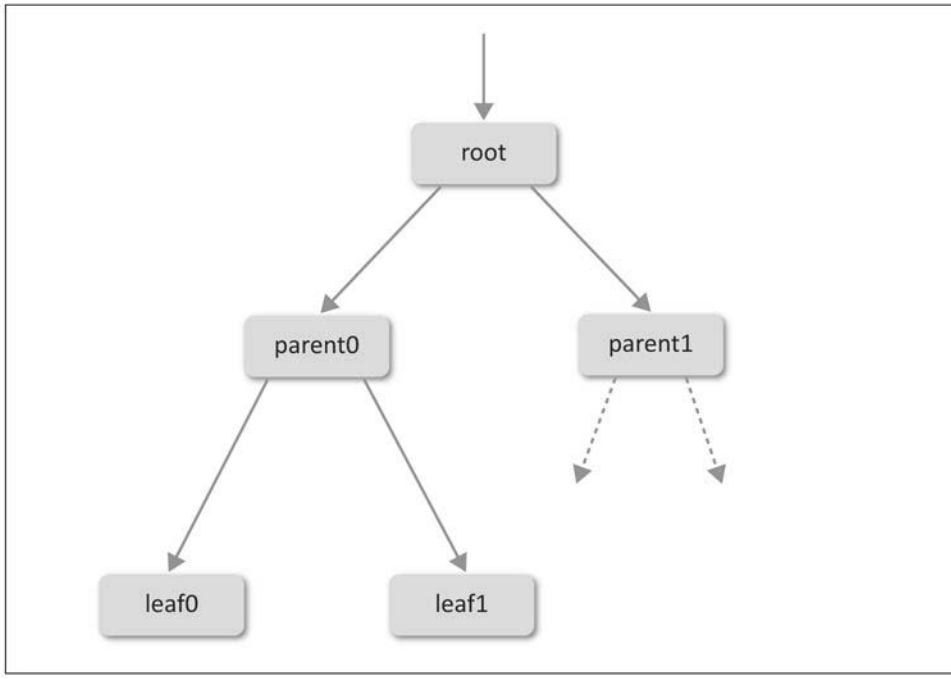

Figure 1.3: A server tree

The primary benefit of this pattern is that it permits parallel searching of a large corpus. Not only are the leaves searching their share of the corpus in parallel, but the sorting and ranking performed by the parents are also done in parallel.

For example, imagine a corpus of the text extracted from every book in the U.S. Library of Congress. This cannot fit in one computer, so instead the information is spread over hundreds or thousands of leaf machines. In addition to the leaf machines are the parents and the root. A search query would go to a root server, which in turn relays the query to all parents. Each parent repeats the query to all leaf nodes below it. Once the leaves have replied, the parent ranks and sorts the results by relevancy.

For example, a leaf may reply that all the words of the query exist in the same paragraph in one book, but for another book only some of the words exist (less relevant), or they exist but not in the same paragraph or page (even less relevant). If the query is for the best 50 answers, the parent can send the top 50 results to the root and drop the rest. The root then receives results from each parent and selects the best 50 of those to construct the reply.

This scheme also permits developers to work within a latency budget. If fast answers are more important than perfect answers, parents and roots do not have to wait for slow replies if the latency deadline is near.

Many variations of this pattern are possible. Redundant servers may exist with a load-balancing scheme to divide the work among them and route around failed servers. Expanding the number of leaf servers can give each leaf a smaller portion of the corpus to search, or each shard of corpus can be placed on multiple leaf servers to improve availability. Expanding the number of parents at each level increases the capacity to sort and rank results. There may be additional levels of parent servers, making the tree taller. The additional levels permit a wider fan-out, which is important for an extremely large corpus. The parents may provide a caching function to relieve pressure on the leaf servers; in this case more levels of parents may improve cache effectiveness. These techniques can also help mitigate congestion problems related to fan-in, as discussed in the previous section.

1.4 Distributed State

Large systems often store or process large amounts of state. State consists of data, such as a database, that is frequently updated. Contrast this with a corpus, which is relatively static or is updated only periodically when a new edition is published. For example, a system that searches the U.S. Library of Congress may receive a new corpus each week. By comparison, an email system is in constant churn with new data arriving constantly, current data being updated (email messages being marked as “read” or moved between folders), and data being deleted.

Distributed computing systems have many ways to deal with state. However, they all involve some kind of replication and sharding, which brings about problems of consistency, availability, and partitioning.

The easiest way to store state is to put it on one machine, as depicted in Figure 1.4. Unfortunately, that method reaches its limit quite quickly: an individual machine can store only a limited amount of state and if the one machine dies we lose access to 100 percent of the state. The machine has only a certain amount of processing power, which means the number of simultaneous reads and writes it can process is limited.

In distributed computing we store state by storing fractions or shards of the whole on individual machines. This way the amount of state we can store is limited only by the number of machines we can acquire. In addition, each shard is stored on multiple machines; thus a single machine failure does not lose access to any state. Each replica can process a certain number of queries per second, so we can design the system to process any number of simultaneous read and write requests by increasing the number of replicas. This is illustrated in Figure 1.5, where *N* QPS are received and distributed among three shards, each replicated three ways. As a result, on average one ninth of all queries reach a particular replica server.

Writes or requests that update state require all replicas to be updated. While this update process is happening, it is possible that some clients will read from stale replicas that have not yet been updated. Figure 1.6 illustrates how a write can be confounded by reads to an out-of-date cache. This will be discussed further in the next section.

In the most simple pattern, a root server receives requests to store or retrieve state. It determines which shard contains that part of the state and forwards the request to the appropriate leaf server. The reply then flows up the tree. This looks similar to the server tree pattern described in the previous section but there are two

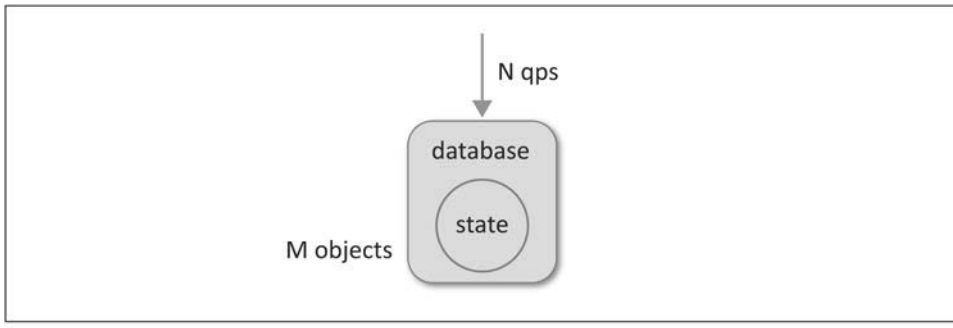

Figure 1.4: State kept in one location; not distributed computing

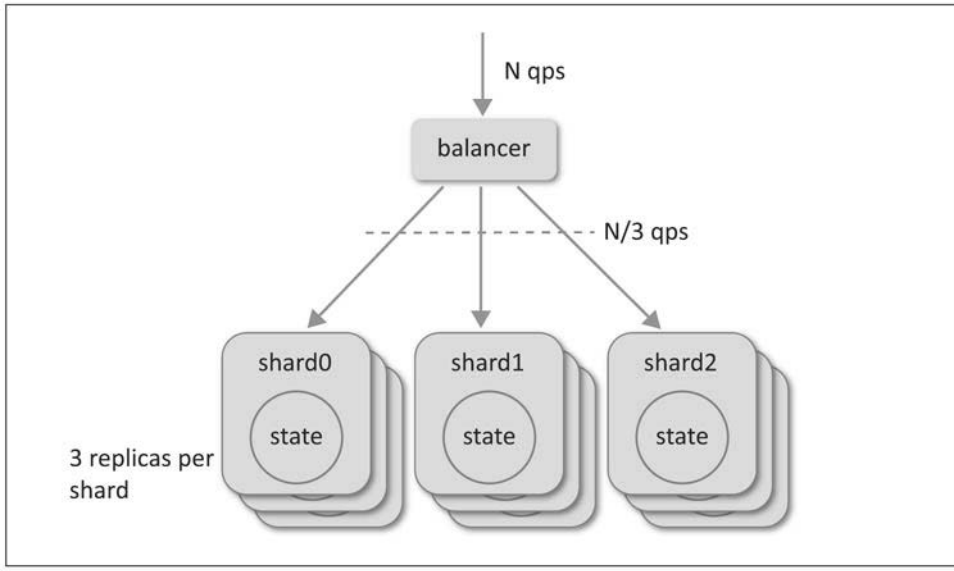

Figure 1.5: This distributed state is sharded and replicated.

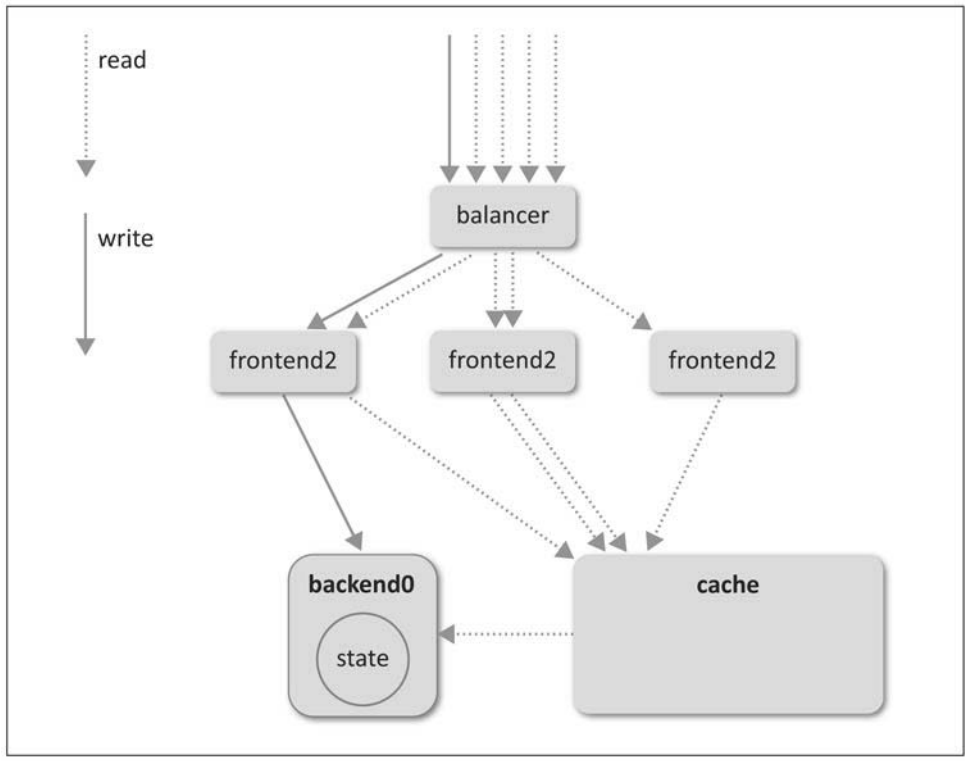

Figure 1.6: State updates using cached data lead to an inconsistent view.

differences. First, queries go to a single leaf instead of all leaves. Second, requests can be update (write) requests, not just read requests. Updates are more complex when a shard is stored on many replicas. When one shard is updated, all of the replicas must be updated, too. This may be done by having the root update all leaves or by the leaves communicating updates among themselves.

A variation of that pattern is more appropriate when large amounts of data are being transferred. In this case, the root replies with instructions on how to get the data rather than the data itself. The requestor then requests the data from the source directly.

For example, imagine a distributed file system with petabytes of data spread out over thousands of machines. Each file is split into gigabyte-sized chunks. Each chunk is stored on multiple machines for redundancy. This scheme also permits the creation of files larger than those that would fit on one machine. A master server tracks the list of files and identifies where their chunks are. If you are familiar with the UNIX file system, the master can be thought of as storing the inodes, or per-file lists of data blocks, and the other machine as storing the actual blocks of data. File system operations go through a master server that uses the inode-like information to determine which machines to involve in the operation.

Imagine that a large read request comes in. The master determines that the file has a few terabytes stored on one machine and a few terabytes stored on another machine. It could request the data from each machine and relay it to the system that made the request, but the master would quickly become overloaded while receiving and relaying huge chunks of data. Instead, it replies with a list of which machines have the data, and the requestor contacts those machines directly for the data. This way the master is not the middle man for those large data transfers. This situation is illustrated in Figure 1.7.

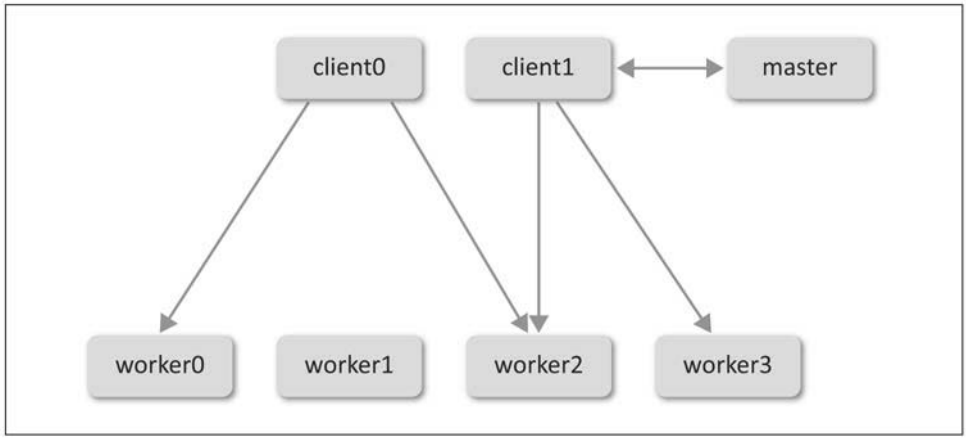

Figure 1.7: This master server delegates replies to other servers.

1.5 The CAP Principle

CAP stands for consistency, availability, and partition resistance. The CAP Principle states that it is not possible to build a distributed system that guarantees consistency, availability, and resistance to partitioning. Any one or two can be achieved but not all three simultaneously. When using such systems you must be aware of which are guaranteed.

1.5.1 Consistency

Consistency means that all nodes see the same data at the same time. If there are multiple replicas and there is an update being processed, all users see the update go live at the same time even if they are reading from different replicas. Systems that do not guarantee consistency may provide **eventual consistency**. For example, they may guarantee that any update will propagate to all replicas in a certain amount of time. Until that deadline is reached, some queries may receive the new data while others will receive older, out-of-date answers.

Perfect consistency is not always important. Imagine a social network that awards reputation points to users for positive actions. Your reputation point total is displayed anywhere your name is shown. The reputation database is replicated in the United States, Europe, and Asia. A user in Europe is awarded points and that change might take minutes to propagate to the United States and Asia replicas. This may be sufficient for such a system because an absolutely accurate reputation score is not essential. If a user in the United States and one in Asia were talking on the phone as one was awarded points, the other user would see the update seconds later and that would be okay. If the update took minutes due to network congestion or hours due to a network outage, the delay would still not be a terrible thing.

Now imagine a banking application built on this system. A person in the United States and another in Europe could coordinate their actions to withdraw money from the same account at the same time. The ATM that each person uses would query its nearest database replica, which would claim the money is available and may be withdrawn. If the updates propagated slowly enough, both people would have the cash before the bank realized the money was already gone.[1]

1.5.2 Availability

Availability is a guarantee that every request receives a response about whether it was successful or failed. In other words, it means that the system is up. For

1. The truth is that the global ATM system does not require database consistency. It can be defeated by leveraging network delays and outages. It is less expensive for banks to give out a limited amount of money when the ATM network is down than to have an unhappy customer stranded without cash. Fraudulent transactions are dealt with after the fact. Daily withdrawal limits prevent major fraud. Assessing overage fees is easier than implementing a globally consistant database.

example, using many replicas to store data such that clients always have access to at least one working replica guarantees availability.

The CAP Principle states that availability also guarantees that the system is able to report failure. For example, a system may detect that it is overloaded and reply to requests with an error code that means "try again later." Being told this immediately is more favorable than having to wait minutes or hours before one gives up.

1.5.3 Partition Tolerance

Partition tolerance means the system continues to operate despite arbitrary message loss or failure of part of the system. The simplest example of partition tolerance is when the system continues to operate even if the machines involved in providing the service lose the ability to communicate with each other due to a network link going down (see Figure 1.8).

Returning to our example of replicas, if the system is read-only it is easy to make the system partition tolerant, as the replicas do not need to communicate with each other. But consider the example of replicas containing state that is updated on one replica first, then copied to other replicas. If the replicas are unable to communicate with each other, the system fails to be able to guarantee updates will propagate within a certain amount of time, thus becoming a failed system.

Now consider a situation where two servers cooperate in a master–slave relationship. Both maintain a complete copy of the state and the slave takes over the master's role if the master fails, which is determined by a loss of heartbeat—that is,

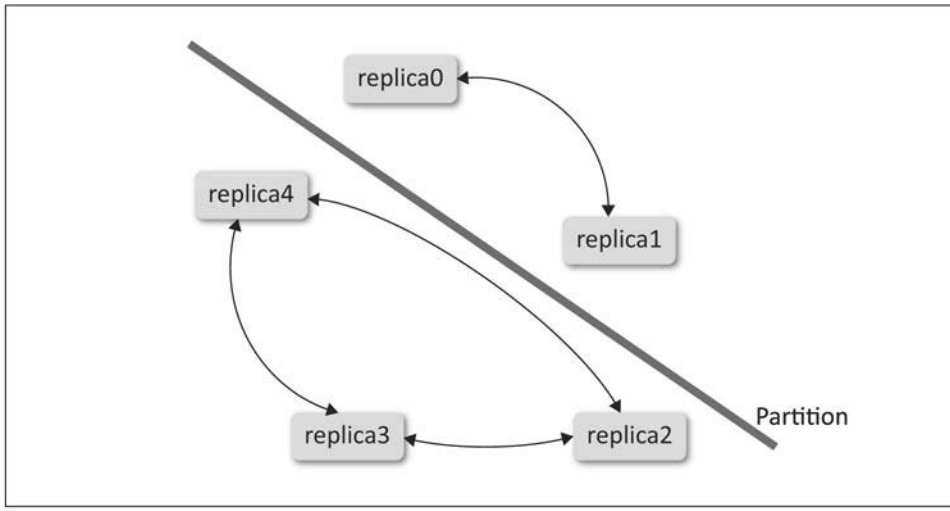

Figure 1.8: Nodes partitioned from each other

a periodic health check between two servers often done via a dedicated network. If the heartbeat network between the two is partitioned, the slave will promote itself to being the master, not knowing that the original master is up but unable to communicate on the heartbeat network. At this point there are two masters and the system breaks. This situation is called **split brain.**

Some special cases of partitioning exist. Packet loss is considered a temporary partitioning of the system as it applies to the CAP Principle. Another special case is the complete network outage. Even the most partition-tolerant system is unable to work in that situation.

The CAP Principle says that any one or two of the attributes are achievable in combination, but not all three. In 2002, Gilbert and Lynch published a formal proof of the original conjecture, rendering it a theorem. One can think of this as the third attribute being sacrificed to achieve the other two.

The CAP Principle is illustrated by the triangle in Figure 1.9. Traditional relational databases like Oracle, MySQL, and PostgreSQL are consistent and available (CA). They use transactions and other database techniques to assure that updates are atomic; they propagate completely or not at all. Thus they guarantee all users will see the same state at the same time. Newer storage systems such as Hbase,

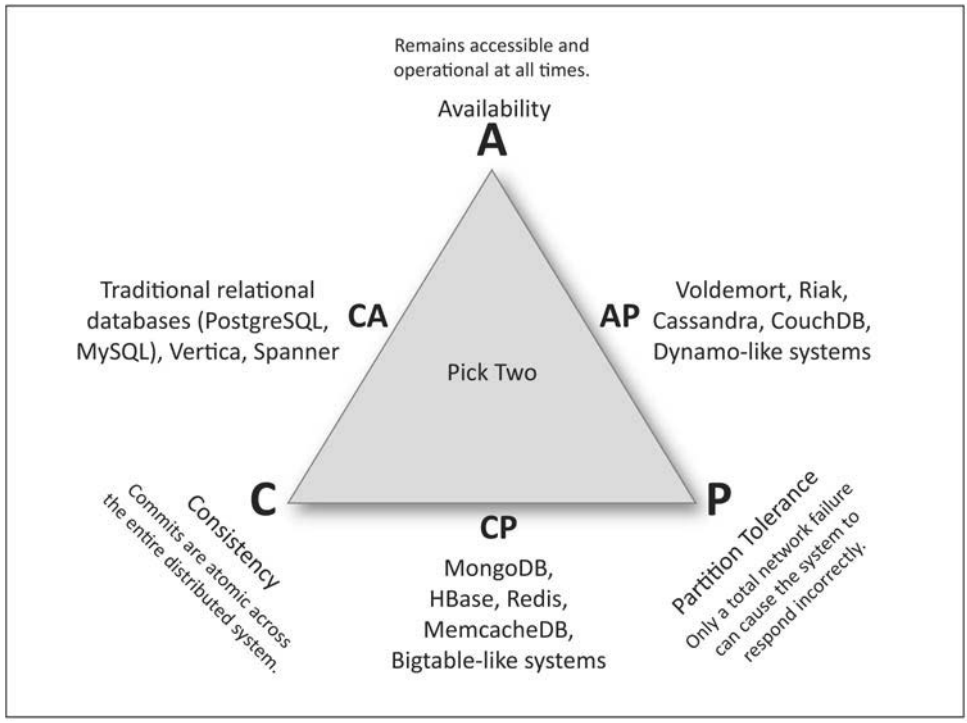

Figure 1.9: The CAP Principle

Redis, and Bigtable focus on consistency and partition tolerance (CP). When partitioned, they become read-only or refuse to respond to any requests rather than be inconsistent and permit some users to see old data while others see fresh data. Finally, systems such as Cassandra, Risk, and Dynamo focus on availability and partition tolerance (AP). They emphasize always being able to serve requests even if it means some clients receive outdated results. Such systems are often used in globally distributed networks where each replica talks to the others by less reliable media such as the Internet.

SQL and other relational databases use the term **ACID** to describe their side of the CAP triangle. ACID stands for Atomicity (transactions are "all or nothing"), Consistency (after each transaction the database is in a valid state), Isolation (concurrent transactions give the same results as if they were executed serially), and Durability (a committed transaction's data will not be lost in the event of a crash or other problem). Databases that provide weaker consistency models often refer to themselves as NoSQL and describe themselves as **BASE**: Basically Available Soft-state services with Eventual consistency.

1.6 Loosely Coupled Systems

Distributed systems are expected to be highly available, to last a long time, and to evolve and change without disruption. Entire subsystems are often replaced while the system is up and running.

To achieve this a distributed system uses **abstraction** to build a loosely coupled system. Abstraction means that each component provides an interface that is defined in a way that hides the implementation details. The system is loosely coupled if each component has little or no knowledge of the internals of the other components. As a result a subsystem can be replaced by one that provides the same abstract interface even if its implementation is completely different.

Take, for example, a spell check service. A good level of abstraction would be to take in text and return a description of which words are misspelled and a list of possible corrections for each one. A bad level of abstraction would simply provide access to a lexicon of words that the frontends could query for similar words. The reason the latter is not a good abstraction is that if an entirely new way to check spelling was invented, every frontend using the spell check service would need to be rewritten. Suppose this new version does not rely on a lexicon but instead applies an artificial intelligence technique called machine learning. With the good abstraction, no frontend would need to change; it would simply send the same kind of request to the new server. Users of the bad abstraction would not be so lucky.

For this and many other reasons, loosely coupled systems are easier to evolve and change over time.

Continuing our example, in preparation for the launch of the new spell check service both versions could be run in parallel. The load balancer that sits in front of the spell check system could be programmed to send all requests to both the old and new systems. Results from the old system would be sent to the users, but results from the new system would be collected and compared for quality control. At first the new system might not produce results that were as good, but over time it would be enhanced until its results were quantifiably better. At that point the new system would be put into production. To be cautious, perhaps only 1 percent of all queries would come through the new system—if no users complained, the new system would take a larger fraction. Eventually all responses would come from the new system and the old system could be decommissioned.

Other systems require more precision and accuracy than a spell check system. For example, there may be requirements that the new system be bug-for-bug compatible with the old system before it can offer new functionality. That is, the new system must reproduce not only the features but also the bugs from the old system. In this case the ability to send requests to both systems and compare results becomes critical to the operational task of deploying it.

Case Study: Emulation before Improvements

When Tom was at Cibernet, he was involved in a project to replace an older system. Because it was a financial system, the new system had to prove it was bug-for-bug compatible before it could be deployed.

The old system was built on obsolete, pre-web technology and had become so complex and calcified that it was impossible to add new features. The new system was built on newer, better technology and, being a cleaner design, was more easily able to accommodate new functionality. The systems were run in parallel and results were compared.

At that point engineers found a bug in the old system. Currency conversion was being done in a way that was non-standard and the results were slightly off. To make the results between the two systems comparable, the developers reverse-engineered the bug and emulated it in the new system.

Now the results in the old and new systems matched down to the penny. With the company having gained confidence in the new system's ability to be bug-for-bug compatible, it was activated as the primary system and the old system was disabled.

At this point, new features and improvements could be made to the system. The first improvement, unsurprisingly, was to remove the code that emulated the currency conversion bug.

1.7 Speed

So far we have elaborated on many of the considerations involved in designing large distributed systems. For web and other interactive services, one item may be the most important: speed. It takes time to get information, store information, compute and transform information, and transmit information. Nothing happens instantly.

An interactive system requires fast response times. Users tend to perceive anything faster than 200 ms to be instant. They also prefer fast over slow. Studies have documented sharp drops in revenue when delays as little as 50 ms were artificially added to web sites. Time is also important in batch and non-interactive systems where the total throughput must meet or exceed the incoming flow of work.

The general strategy for designing a system that is performant is to design a system using our best estimates of how quickly it will be able to process a request and then to build prototypes to test our assumptions. If we are wrong, we go back to step one; at least the next iteration will be informed by what we have learned. As we build the system, we are able to remeasure and adjust the design if we discover our estimates and prototypes have not guided us as well as we had hoped.

At the start of the design process we often create many designs, estimate how fast each will be, and eliminate the ones that are not fast enough. We do not automatically select the fastest design. The fastest design may be considerably more expensive than one that is sufficient.

How do we determine if a design is worth pursuing? Building a prototype is very time consuming. Much can be deduced with some simple estimating exercises. Pick a few common transactions and break them down into smaller steps, and then estimate how long each step will take.

Two of the biggest consumers of time are disk access and network delays.

Disk accesses are slow because they involve mechanical operations. To read a block of data from a disk requires the read arm to move to the right track; the platter must then spin until the desired block is under the read head. This process typically takes 10 ms. Compare this to reading the same amount of information from RAM, which takes 0.002 ms, which is 5,000 times faster. The arm and platters (known as a **spindle**) can process only one request at a time. However, once the head is on the right track, it can read many sequential blocks. Therefore reading two blocks is often nearly as fast as reading one block if the two blocks are adjacent. Solid-state drives (SSDs) do not have mechanical spinning platters and are much faster, though more expensive.

Network access is slow because it is limited by the speed of light. It takes approximately 75 ms for a packet to get from California to the Netherlands. About half of that journey time is due to the speed of light. Additional delays may be attributable to processing time on each router, the electronics that convert from

wired to fiber-optic communication and back, the time it takes to assemble and disassemble the packet on each end, and so on.

Two computers on the same network segment might seem as if they communicate instantly, but that is not really the case. Here the time scale is so small that other delays have a bigger factor. For example, when transmitting data over a local network, the first byte arrives quickly but the program receiving the data usually does not process it until the entire packet is received.

In many systems computation takes little time compared to the delays from network and disk operation. As a result you can often estimate how long a transaction will take if you simply know the distance from the user to the datacenter and the number of disk seeks required. Your estimate will often be good enough to throw away obviously bad designs.

To illustrate this, imagine you are building an email system that needs to be able to retrieve a message from the message storage system and display it within 300 ms. We will use the time approximations listed in Figure 1.10 to help us engineer the solution.

Jeff Dean, a Google Fellow, has popularized this chart of common numbers to aid in architectural and scaling decisions. As you can see, there are many orders of magnitude difference between certain options. These numbers improve every year. Updates can be found online.

Action	Typical Time	
L1 cache reference	0.5 ns	
Branch mispredict	5 ns	
L2 cache reference	7 ns	
Mutex lock/unlock	100 ns	
Main memory reference	100 ns	
Compress 1K bytes with Zippy	10,000 ns	(0.01 ms)
Send 2K bytes over 1 Gbps network	20,000 ns	(0.02 ms)
Read 1 MB sequentially from memory	250,000 ns	(0.25 ms)
Round trip within same datacenter	500,000 ns	(0.5 ms)
Read 1 MB from SSD	1,000,000 ns	(3 ms)
Disk seek	10,000,000 ns	(10 ms)
Read 1 MB sequentially from network	10,000,000 ns	(10 ms)
Read 1 MB sequentially from disk	30,000,000 ns	(30 ms)
Send packet from California to Netherlands to California	150,000,000 ns	(150 ms)

Figure 1.10: Numbers every engineer should know

First we follow the transaction from beginning to end. The request comes from a web browser that may be on another continent. The request must be authenticated, the database index is consulted to determine where to get the message text, the message text is retrieved, and finally the response is formatted and transmitted back to the user.

Now let's budget for the items we can't control. To send a packet between California and Europe typically takes 75 ms, and until physics lets us change the speed of light that won't change. Our 300 ms budget is reduced by 150 ms since we have to account for not only the time it takes for the request to be transmitted but also the reply. That's half our budget consumed by something we don't control.

We talk with the team that operates our authentication system and they recommend budgeting 3 ms for authentication.

Formatting the data takes very little time—less than the slop in our other estimates—so we can ignore it.

This leaves 147 ms for the message to be retrieved from storage. If a typical index lookup requires 3 disk seeks (10 ms each) and reads about 1 megabyte of information (30 ms), that is 60 ms. Reading the message itself might require 4 disk seeks and reading about 2 megabytes of information (100 ms). The total is 160 ms, which is more than our 147 ms remaining budget.

How Did We Know That?

How did we know that it will take 3 disk seeks to read the index? It requires knowledge of the inner workings of the UNIX file system: how files are looked up in a directory to find an inode and how inodes are used to look up the data blocks. This is why understanding the internals of the operating system you use is key to being able to design and operate distributed systems. The internals of UNIX and UNIX-like operating systems are well documented, thus giving them an advantage over other systems.

While disappointed that our design did not meet the design parameters, we are happy that disaster has been averted. Better to know now than to find out when it is too late.

It seems like 60 ms for an index lookup is a long time. We could improve that considerably. What if the index was held in RAM? Is this possible? Some quick calculations estimate that the lookup tree would have to be 3 levels deep to fan out to enough machines to span this much data. To go up and down the tree is 5 packets, or about 2.5 ms if they are all within the same datacenter. The new total ($150\ \text{ms} + 3\ \text{ms} + 2.5\ \text{ms} + 100\ \text{ms} = 255.5\ \text{ms}$) is less than our total 300 ms budget.

We would repeat this process for other requests that are time sensitive. For example, we send email messages less frequently than we read them, so the time to send an email message may not be considered time critical. In contrast, deleting a message happens almost as often reading messages. We might repeat this calculation for a few deletion methods to compare their efficiency.

One design might contact the server and delete the message from the storage system and the index. Another design might have the storage system simply mark the message as deleted in the index. This would be considerably faster but would require a new element that would reap messages marked for deletion and occasionally compact the index, removing any items marked as deleted.

Even faster response time can be achieved with an asynchronous design. That means the client sends requests to the server and quickly returns control to the user without waiting for the request to complete. The user perceives this system as faster even though the actual work is lagging. Asynchronous designs are more complex to implement. The server might queue the request rather than actually performing the action. Another process reads requests from the queue and performs them in the background. Alternatively, the client could simply send the request and check for the reply later, or allocate a thread or subprocess to wait for the reply.

All of these designs are viable but each offers different speed and complexity of implementation. With speed and cost estimates, backed by prototypes, the business decision of which to implement can be made.

1.8 Summary

Distributed computing is different from traditional computing in many ways. The scale is larger; there are many machines, each doing specialized tasks. Services are replicated to increase capacity. Hardware failure is not treated as an emergency or exception but as an expected part of the system. Thus the system works around failure.

Large systems are built through composition of smaller parts. We discussed three ways this composition is typically done: load balancer for many backend replicas, frontend with many different backends, and a server tree.

The load balancer divides traffic among many duplicate systems. The frontend with many different backends uses different backends in parallel, with each performing different processes. The server tree uses a tree configuration, with each tree level serving a different purpose.

Maintaining state in a distributed system is complex, whether it is a large database of constantly updated information or a few key bits to which many systems need constant access. The CAP Principle states that it is not possible to build a distributed system that guarantees consistency, availability, and resistance to partitioning simultaneously. At most two of the three can be achieved.

Systems are expected to evolve over time. To make this easier, the components are loosely coupled. Each embodies an abstraction of the service it provides, such that the internals can be replaced or improved without changing the abstraction. Thus, dependencies on the service do not need to change other than to benefit from new features.

Designing distributed systems requires an understanding of the time it takes various operations to run so that time-sensitive processes can be designed to meet their latency budget.

Exercises

1. What is distributed computing?
2. Describe the three major composition patterns in distributed computing.
3. What are the three patterns discussed for storing state?
4. Sometimes a master server does not reply with an answer but instead replies with where the answer can be found. What are the benefits of this method?
5. Section 1.4 describes a distributed file system, including an example of how reading terabytes of data would work. How would writing terabytes of data work?
6. Explain the CAP Principle. (If you think the CAP Principle is awesome, read "The Part-Time Parliament" (Lamport & Marzullo 1998) and "Paxos Made Simple" (Lamport 2001).)
7. What does it mean when a system is loosely coupled? What is the advantage of these systems?
8. Give examples of loosely and tightly coupled systems you have experience with. What makes them loosely or tightly coupled?
9. How do we estimate how fast a system will be able to process a request such as retrieving an email message?
10. In Section 1.7 three design ideas are presented for how to process email deletion requests. Estimate how long the request will take for deleting an email message for each of the three designs. First outline the steps each would take, then break each one into individual operations until estimates can be created.

This page intentionally left blank

Chapter 7

Operations in a Distributed World

> The rate at which organizations learn may soon become the only sustainable source of competitive advantage.
>
> —Peter Senge

Part I of this book discussed how to build distributed systems. Now we discuss how to run such systems.

The work done to keep a system running is called **operations**. More specifically, operations is the work done to keep a system running in a way that meets or exceeds operating parameters specified by a service level agreement (SLA). Operations includes all aspects of a service's life cycle: from initial launch to the final decommissioning and everything in between.

Operational work tends to focus on availability, speed and performance, security, capacity planning, and software/hardware upgrades. The failure to do any of these well results in a system that is unreliable. If a service is slow, users will assume it is broken. If a system is insecure, outsiders can take it down. Without proper capacity planning, it will become overloaded and fail. Upgrades, done badly, result in downtime. If upgrades aren't done at all, bugs will go unfixed. Because all of these activities ultimately affect the reliability of the system, Google calls its operations team Site Reliability Engineering (SRE). Many companies have followed suit.

Operations is a team sport. Operations is not done by a single person but rather by a team of people working together. For that reason much of what we describe will be processes and policies that help you work as a team, not as a group of individuals. In some companies, processes seem to be bureaucratic mazes that slow things down. As we describe here—and more important, in our professional experience—good processes are exactly what makes it possible to run very large

Terms to Know

Innovate: Doing (good) things we haven't done before.

Machine: A virtual or physical machine.

Oncall: Being available as first responder to an outage or alert.

Server: Software that provides a function or API. (Not a piece of hardware.)

Service: A user-visible system or product composed of one or more servers.

Soft launch: Launching a new service without publicly announcing it. This way traffic grows slowly as word of mouth spreads, which gives operations some cushion to fix problems or scale the system before too many people have seen it.

SRE: Site Reliability Engineer, the Google term for systems administrators who maintain live services.

Stakeholders: People and organizations that are seen as having an interest in a project's success.

computing systems. In other words, process is what makes it possible for teams to do the right thing, again and again.

This chapter starts with some operations management background, then discusses the operations service life cycle, and ends with a discussion of typical operations work strategies. All of these topics will be expanded upon in the chapters that follow.

7.1 Distributed Systems Operations

To understand distributed systems operations, one must first understand how it is different from typical enterprise IT. One must also understand the source of tension between operations and developers, and basic techniques for scaling operations.

7.1.1 SRE versus Traditional Enterprise IT

System administration is a continuum. On one end is a typical IT department, responsible for traditional desktop and client–server computing infrastructure, often called enterprise IT. On the other end is an SRE or similar team responsible for a distributed computing environment, often associated with web sites and other services. While this may be a broad generalization, it serves to illustrate some important differences.

SRE is different from an enterprise IT department because SREs tend to be focused on providing a single service or a well-defined set of services. A traditional enterprise IT department tends to have broad responsibility for desktop services,

back-office services, and everything in between ("everything with a power plug"). SRE's customers tend to be the product management of the service while IT customers are the end users themselves. This means SRE efforts are focused on a few select business metrics rather than being pulled in many directions by users, each of whom has his or her own priorities.

Another difference is in the attitude toward uptime. SREs maintain services that have demanding, 24×7 uptime requirements. This creates a focus on preventing problems rather than reacting to outages, and on performing complex but non-intrusive maintenance procedures. IT tends to be granted flexibility with respect to scheduling downtime and has SLAs that focus on how quickly service can be restored in the event of an outage. In the SRE view, downtime is something to be avoided and service should not stop while services are undergoing maintenance.

SREs tend to manage services that are constantly changing due to new software releases and additions to capacity. IT tends to run services that are upgraded rarely. Often IT services are built by external contractors who go away once the system is stable.

SREs maintain systems that are constantly being scaled to handle more traffic and larger workloads. Latency, or how fast a particular request takes to process, is managed as well as overall throughput. Efficiency becomes a concern because a little waste per machine becomes a big waste when there are hundreds or thousands of machines. In IT, systems are often built for environments that expect a modest increase in workload per year. In this case a workable strategy is to build the system large enough to handle the projected workload for the next few years, when the system is expected to be replaced.

As a result of these requirements, systems in SRE tend to be bespoke systems, built on platforms that are home-grown or integrated from open source or other third-party components. They are not "off the shelf" or turn key systems. They are actively managed, while IT systems may be unchanged from their initial delivery state. Because of these differences, distributed computing services are best managed by a separate team, with separate management, with bespoke operational and management practices.

While there are many such differences, recently IT departments have begun to see a demand for uptime and scalability similar to that seen in SRE environments. Therefore the management techniques from distributed computing are rapidly being adopted in the enterprise.

7.1.2 Change versus Stability

There is a tension between the desire for stability and the desire for change. Operations teams tend to favor stability; developers desire change. Consider how each group is evaluated during end-of-the-year performance reviews. A developer is praised for writing code that makes it into production. Changes that result in a

tangible difference to the service are rewarded above any other accomplishment. Therefore, developers want new releases pushed into production often. Operations, in contrast, is rewarded for achieving compliance with SLAs, most of which relate to uptime. Therefore stability is the priority.

A system starts at a baseline of stability. A change is then made. All changes have some kind of a destabilizing effect. Eventually the system becomes stable again, usually through some kind of intervention. This is called the **change-instability cycle**.

All software roll-outs affect stability. A change may introduce bugs, which are fixed through workarounds and new software releases. A release that introduces no new bugs still creates a destabilizing effect due to the process of shifting workloads away from machines about to be upgraded. Non-software changes also have a destabilizing effect. A network change may make the local network less stable while the change propagates throughout the network.

Because of the tension between the operational desire for stability and the developer desire for change, there must be mechanisms to reach a balance.

One strategy is to prioritize work that improves stability over work that adds new features. For example, bug fixes would have a higher priority than feature requests. With this approach, a major release introduces many new features, the next few releases focus on fixing bugs, and then a new major release starts the cycle over again. If engineering management is pressured to focus on new features and neglect bug fixes, the result is a system that slowly destabilizes until it spins out of control.

Another strategy is to align the goals of developers and operational staff. Both parties become responsible for SLA compliance as well as the velocity (rate of change) of the system. Both have a component of their annual review that is tied to SLA compliance and both have a portion tied to the on-time delivery of new features.

Organizations that have been the most successful at aligning goals like this have restructured themselves so that developers and operations work as one team. This is the premise of the DevOps movement, which will be described in Chapter 8.

Another strategy is to budget time for stability improvements and time for new features. Software engineering organizations usually have a way to estimate the size of a software request or the amount of time it is expected to take to complete. Each new release has a certain size or time budget; within that budget a certain amount of stability-improvement work is allocated. The case study at the end of Section 2.2.2 is an example of this approach. Similarly, this allocation can be achieved by assigning dedicated people to stability-related code changes.

The budget can also be based on an SLA. A certain amount of instability is expected each month, which is considered a budget. Each roll-out uses some of the budget, as do instability-related bugs. Developers can maximize the number

of roll-outs that can be done each month by dedicating effort to improve the code that causes this instability. This creates a positive feedback loop. An example of this is Google's Error Budgets, which are more fully explained in Section 19.4.

7.1.3 Defining SRE

The core practices of SRE were refined for more than 10 years at Google before being enumerated in public. In his keynote address at the first USENIX SREcon, Benjamin Treynor Sloss (2014), Vice President of Site Reliability Engineering at Google, listed them as follows:

Site Reliability Practices

1. Hire only coders.
2. Have an SLA for your service.
3. Measure and report performance against the SLA.
4. Use Error Budgets and gate launches on them.
5. Have a common staffing pool for SRE and Developers.
6. Have excess Ops work overflow to the Dev team.
7. Cap SRE operational load at 50 percent.
8. Share 5 percent of Ops work with the Dev team.
9. Oncall teams should have at least eight people at one location, or six people at each of multiple locations.
10. Aim for a maximum of two events per oncall shift.
11. Do a postmortem for every event.
12. Postmortems are blameless and focus on process and technology, not people.

The first principle for site reliability engineering is that SREs must be able to code. An SRE might not be a full-time software developer, but he or she should be able to solve nontrivial problems by writing code. When asked to do 30 iterations of a task, an SRE should do the first two, get bored, and automate the rest. An SRE must have enough software development experience to be able to communicate with developers on their level and have an appreciation for what developers do, and for what computers can and can't do.

When SREs and developers come from a common staffing pool, that means that projects are allocated a certain number of engineers; these engineers may be developers or SREs. The end result is that each SRE needed means one fewer developer in the team. Contrast this to the case at most companies where system administrators and developers are allocated from teams with separate budgets. Rationally a project wants to maximize the number of developers, since they write new features. The common staffing pool encourages the developers to create systems that can be operated efficiently so as to minimize the number of SREs needed.

Another way to encourage developers to write code that minimizes operational load is to require that excess operational work overflows to the developers. This practice discourages developers from taking shortcuts that create undue operational load. The developers would share any such burden. Likewise, by requiring developers to perform 5 percent of operational work, developers stay in tune with operational realities.

Within the SRE team, capping the operational load at 50 percent limits the amount of manual labor done. Manual labor has a lower return on investment than, for example, writing code to replace the need for such labor. This is discussed in Section 12.4.2, "Reducing Toil."

Many SRE practices relate to finding balance between the desire for change and the need for stability. The most important of these is the Google SRE practice called Error Budgets, explained in detail in Section 19.4.

Central to the Error Budget is the SLA. All services must have an SLA, which specifies how reliable the system is going to be. The SLA becomes the standard by which all work is ultimately measured. SLAs are discussed in Chapter 16.

Any outage or other major SLA-related event should be followed by the creation of a written postmortem that includes details of what happened, along with analysis and suggestions for how to prevent such a situation in the future. This report is shared within the company so that the entire organization can learn from the experience. Postmortems focus on the process and the technology, not finding who to blame. Postmortems are the topic of Section 14.3.2. The person who is oncall is responsible for responding to any SLA-related events and producing the postmortem report.

Oncall is not just a way to react to problems, but rather a way to reduce future problems. It must be done in a way that is not unsustainably stressful for those oncall, and it drives behaviors that encourage long-term fixes and problem prevention. Oncall teams are made up of at least eight members at one location, or six members at two locations. Teams of this size will be oncall often enough that their skills do not get stale, and their shifts can be short enough that each catches no more than two outage events. As a result, each member has enough time to follow through on each event, performing the required long-term solution. Managing oncall this way is the topic of Chapter 14.

Other companies have adopted the SRE job title for their system administrators who maintain live production services. Each company applies a different set of practices to the role. These are the practices that define SRE at Google and are core to its success.

7.1.4 Operations at Scale

Operations in distributed computing is operations at a large scale. Distributed computing involves hundreds and often thousands of computers working together. As a result, operations is different than traditional computing administration.

Manual processes do not scale. When tasks are manual, if there are twice as many tasks, there is twice as much human effort required. A system that is scaling to thousands of machines, servers, or processes, therefore, becomes untenable if a process involves manually manipulating things. In contrast, automation does scale. Code written once can be used thousands of times. Processes that involve many machines, processes, servers, or services should be automated. This idea applies to allocating machines, configuring operating systems, installing software, and watching for trouble. Automation is not a "nice to have" but a "must have." (Automation is the subject of Chapter 12.)

When operations is automated, system administration is more like an assembly line than a craft. The job of the system administrator changes from being the person who does the work to the person who maintains the robotics of an assembly line. Mass production techniques become viable and we can borrow operational practices from manufacturing. For example, by collecting measurements from every stage of production, we can apply statistical analysis that helps us improve system throughput. Manufacturing techniques such as **continuous improvement** are the basis for the Three Ways of DevOps. (See Section 8.2.)

Three categories of things are not automated: things that should be automated but have not been yet, things that are not worth automating, and human processes that can't be automated.

Tasks That Are Not Yet Automated

It takes time to create, test, and deploy automation, so there will always be things that are waiting to be automated. There is never enough time to automate everything, so we must prioritize and choose our methods wisely. (See Section 2.2.2 and Section 12.1.1.)

For processes that are not, or have not yet been, automated, creating procedural documentation, called a **playbook**, helps make the process repeatable and consistent. A good playbook makes it easier to automate the process in the future. Often the most difficult part of automating something is simply describing the process accurately. If a playbook does that, the actual coding is relatively easy.

Tasks That Are Not Worth Automating

Some things are not worth automating because they happen infrequently, they are too difficult to automate, or the process changes so often that automation is not possible. Automation is an investment in time and effort and the return on investment (ROI) does not always make automation viable.

Nevertheless, there are some common cases that are worth automating. Often when those are automated, the more rare cases (**edge cases**) can be consolidated or eliminated. In many situations, the newly automated common case provides such superior service that the edge-case customers will suddenly lose their need to be so unique.

Benefits of Automating the Common Case

At one company there were three ways that virtual machines were being provisioned. All three were manual processes, and customers often waited days until a system administrator was available to do the task. A project to automate provisioning was stalled because of the complexity of handling all three variations. Users of the two less common cases demanded that their provisioning process be different because they were (in their own eyes) unique and beautiful snowflakes. They had very serious justifications based on very serious (anecdotal) evidence and waved their hands vigorously to prove their point. To get the project moving, it was decided to automate just the most common case and promise the two edge cases would be added later.

This was much easier to implement than the original all-singing, all-dancing, provisioning system. With the initial automation, provisioning time was reduced to a few minutes and could happen without system administrator involvement. Provisioning could even happen at night and on weekends. At that point an amazing thing happened. The other two cases suddenly discovered that their uniqueness had vanished! They adopted the automated method. The system administrators never automated the two edge cases and the provisioning system remained uncomplicated and easy to maintain.

Tasks That Cannot Be Automated

Some tasks cannot be automated because they are human processes: maintaining your relationship with a stakeholder, managing the bidding process to make a large purchase, evaluating new technology, or negotiating within a team to assemble an oncall schedule. While they cannot be eliminated through automation, they can be streamlined:

- Many interactions with stakeholders can be eliminated through better documentation. Stakeholders can be more self-sufficient if provided with introductory documentation, user documentation, best practices recommendations, a style guide, and so on. If your service will be used by many other services or service teams, it becomes more important to have good documentation. Video instruction is also useful and does not require much effort if you simply make a video recording of presentations you already give.
- Some interactions with stakeholders can be eliminated by making common requests self-service. Rather than meeting individually with customers to understand future capacity requirements, their forecasts can be collected via a web user interface or an API. For example, if you provide a service to hundreds

of other teams, forecasting can be become a full-time job for a project manager; alternatively, it can be very little work with proper automation that integrates with the company's supply-chain management system.

- Evaluating new technology can be labor intensive, but if a common case is identified, the end-to-end process can be turned into an assembly-line process and optimized. For example, if hard drives are purchased by the thousand, it is wise to add a new model to the mix only periodically and only after a thorough evaluation. The evaluation process should be standardized and automated, and results stored automatically for analysis.
- Automation can replace or accelerate team processes. Creating the oncall schedule can evolve into a chaotic mess of negotiations between team members battling to take time off during an important holiday. Automation turns this into a self-service system that permits people to list their availability and that churns out an optimal schedule for the next few months. Thus, it solves the problem better and reduces stress.
- Meta-processes such as communication, status, and process tracking can be facilitated through online systems. As teams grow, just tracking the interaction and communication among all parties can become a burden. Automating that can eliminate hours of manual work for each person. For example, a web-based system that lets people see the status of their order as it works its way through approval processes eliminates the need for status reports, leaving people to deal with just exceptions and problems. If a process has many complex handoffs between teams, a system that provides a status dashboard and automatically notifies teams when hand-offs happen can reduce the need for legions of project managers.
- The best process optimization is elimination. A task that is eliminated does not need to be performed or maintained, nor will it have bugs or security flaws. For example, if production machines run three different operating systems, narrowing that number down to two eliminates a lot of work. If you provide a service to other service teams and require a lengthy approval process for each new team, it may be better to streamline the approval process by automatically approving certain kinds of users.

7.2 Service Life Cycle

Operations is responsible for the entire **service life cycle**: launch, maintenance (both regular and emergency), upgrades, and decommissioning. Each phase has unique requirements, so you'll need a strategy for managing each phase differently.

The stages of the life cycle are:

- **Service Launch:** Launching a service the first time. The service is brought to life, initial customers use it, and problems that were not discovered prior to the launch are discovered and remedied. (Section 7.2.1)
- **Emergency Tasks:** Handling exceptional or unexpected events. This includes handling outages and, more importantly, detecting and fixing conditions that precipitate outages. (Chapter 14)
- **Nonemergency Tasks:** Performing all manual work required as part of the normally functioning system. This may include periodic (weekly or monthly) maintenance tasks (for example, preparation for monthly billing events) as well as processing requests from users (for example, requests to enable the service for use by another internal service or team). (Section 7.3)
- **Upgrades:** Deploying new software releases and hardware platforms. The better we can do this, the more aggressively the company can try new things and innovate. Each new software release is built and tested before deployment. Tests include system tests, done by developers, as well as user acceptance tests (UAT), done by operations. UAT might include tests to verify there are no **performance regressions** (unexpected declines in performance). Vulnerability assessments are done to detect security issues. New hardware must go through a **hardware qualification** to test for compatibility, performance regressions, and any changes in operational processes. (Section 10.2)
- **Decommissioning:** Turning off a service. It is the opposite of a service launch: removing the remaining users, turning off the service, removing references to the service from any related service configurations, giving back any resources, archiving old data, and erasing or scrubbing data from any hardware before it is repurposed, sold, or disposed. (Section 7.2.2)
- **Project Work:** Performing tasks large enough to require the allocation of dedicated resources and planning. While not directly part of the service life cycle, along the way tasks will arise that are larger than others. Examples include fixing a repeating but intermittent failure, working with stakeholders on roadmaps and plans for the product's future, moving the service to a new datacenter, and scaling the service in new ways. (Section 7.3)

Most of the life-cycle stages listed here are covered in detail elsewhere in this book. Service launches and decommissioning are covered in detail next.

7.2.1 Service Launches

Nothing is more embarrassing than the failed public launch of a new service. Often we see a new service launch that is so successful that it receives too much traffic, becomes overloaded, and goes down. This is ironic but not funny.

Each time we launch a new service, we learn something new. If we launch new services rarely, then remembering those lessons until the next launch is difficult. Therefore, if launches are rare, we should maintain a checklist of things to do and record the things you should remember to do next time. As the checklist grows with each launch, we become better at launching services.

If we launch new services frequently, then there are probably many people doing the launches. Some will be less experienced than others. In this case we should maintain a checklist to share our experience. Every addition increases our **organizational memory**, the collection of knowledge within our organization, thereby making the organization smarter.

A common problem is that other teams may not realize that planning a launch requires effort. They may not allocate time for this effort and surprise operations teams at or near the launch date. These teams are unaware of all the potential pitfalls and problems that the checklist is intended to prevent. For this reason the launch checklist should be something mentioned frequently in documentation, socialized among product managers, and made easy to access. The best-case scenario occurs when a service team comes to operations wishing to launch something and has been using the checklist as a guide throughout development. Such a team has "done their homework"; they have been working on the items in the checklist in parallel as the product was being developed. This does not happen by accident; the checklist must be available, be advertised, and become part of the company culture.

A simple strategy is to create a checklist of actions that need to be completed prior to launch. A more sophisticated strategy is for the checklist to be a series of questions that are audited by a Launch Readiness Engineer (LRE) or a Launch Committee.

Here is a sample launch readiness review checklist:

Sample Launch Readiness Review Survey

The purpose of this document is to gather information to be evaluated by a Launch Readiness Engineer (LRE) when approving the launch of a new service. Please complete the survey prior to meeting with your LRE.

- General Launch Information:
  - What is the service name?
  - When is the launch date/time?
  - Is this a soft or hard launch?
- Architecture:
  - Describe the system architecture. Link to architecture documents if possible.
  - How does the failover work in the event of single-machine, rack, and datacenter failure?
  - How is the system designed to scale under normal conditions?

- Capacity:
  - What is the expected initial volume of users and QPS?
  - How was this number arrived at? (Link to load tests and reports.)
  - What is expected to happen if the initial volume is 2× expected? 5×? (Link to emergency capacity documents.)
  - What is the expected external (internet) bandwidth usage?
  - What are the requirements for network and storage after 1, 3, and 12 months? (Link to confirmation documents from the network and storage teams capacity planner.)
- Dependencies:
  - Which systems does this depend on? (Link to dependency/data flow diagram.)
  - Which RPC limits are in place with these dependencies? (Link to limits and confirmation from external groups they can handle the traffic.)
  - What will happen if these RPC limits are exceeded ?
  - For each dependency, list the ticket number where this new service's use of the dependency (and QPS rate) was requested and positively acknowledged.
- Monitoring:
  - Are all subsystems monitored? Describe the monitoring strategy and document what is monitored.
  - Does a dashboard exist for all major subsystems?
  - Do metrics dashboards exist? Are they in business, not technical, terms?
  - Was the number of "false alarm" alerts in the last month less than x?
  - Is the number of alerts received in a typical week less than x?
- Documentation:
  - Does a playbook exist and include entries for all operational tasks and alerts?
  - Have an LRE review each entry for accuracy and completeness.
  - Is the number of open documentation-related bugs less than x?
- Oncall:
  - Is the oncall schedule complete for the next n months?
  - Is the oncall schedule arranged such that each shift is likely to get fewer than x alerts?
- Disaster Preparedness:
  - What is the plan in case first-day usage is 10 times greater than expected?
  - Do backups work and have restores been tested?
- Operational Hygiene:
  - Are "spammy alerts" adjusted or corrected in a timely manner?

  - Are bugs filed to raise visibility of issues—even minor annoyances or issues with commonly known workarounds?
  - Do stability-related bugs take priority over new features?
  - Is a system in place to assure that the number of open bugs is kept low?
- Approvals:
  - Has marketing approved all logos, verbiage, and URL formats?
  - Has the security team audited and approved the service?
  - Has a privacy audit been completed and all issues remediated?

Because a launch is complex, with many moving parts, we recommend that a single person (the **launch lead**) take a leadership or coordinator role. If the developer and operations teams are very separate, one person from each might be selected to represent each team.

The launch lead then works through the checklist, delegating work, filing bugs for any omissions, and tracking all issues until launch is approved and executed. The launch lead may also be responsible for coordinating post-launch problem resolution.

Case Study: Self-Service Launches at Google

Google launches so many services that it needed a way to make the launch process streamlined and able to be initiated independently by a team. In addition to providing APIs and portals for the technical parts, the Launch Readiness Review (LRR) made the launch process itself self-service.

The LRR included a checklist and instructions on how to achieve each item. An SRE engineer was assigned to shepherd the team through the process and hold them to some very high standards.

Some checklist items were technical—for example, making sure that the Google load balancing system was used properly. Other items were cautionary, to prevent a launch team from repeating other teams' past mistakes. For example, one team had a failed launch because it received 10 times more users than expected. There was no plan for how to handle this situation. The LRR checklist required teams to create a plan to handle this situation and demonstrate that it had been tested ahead of time.

Other checklist items were business related. Marketing, legal, and other departments were required to sign off on the launch. Each department had its own checklist. The SRE team made the service visible externally only after verifying that all of those sign-offs were complete.

7.2.2 Service Decommissioning

Decommissioning (or just "decomm"), or turning off a service, involves three major phases: removal of users, deallocation of resources, and disposal of resources.

Removing users is often a product management task. Usually it involves making the users aware that they must move. Sometimes it is a technical issue of moving them to another service. User data may need to be moved or archived.

Resource deallocation can cover many aspects. There may be DNS entries to be removed, machines to power off, database connections to be disabled, and so on. Usually there are complex dependencies involved. Often nothing can begin until the last user is off the service; certain resources cannot be deallocated before others, and so on. For example, typically a DNS entry is not removed until the machine is no longer in use. Network connections must remain in place if deallocating other services depends on network connectivity.

Resource disposal includes securely erasing disks and other media and disposing of all hardware. The hardware may be repurposed, sold, or scrapped.

If decommissioning is done incorrectly or items are missed, resources will remain allocated. A checklist, that is added to over time, will help assure decommissioning is done completely and the tasks are done in the right order.

7.3 Organizing Strategy for Operational Teams

An operational team needs to get work done. Therefore teams need a strategy that assures that all incoming work is received, scheduled, and completed. Broadly speaking, there are three sources of operational work and these work items fall into three categories. To understand how to best organize a team, first you must understand these sources and categories.

The three sources of work are life-cycle management, interacting with stakeholders, and process improvement and automation. Life-cycle management is the operational work involved in running the service. Interacting with stakeholders refers to both maintaining the relationship with people who use and depend on the service, and prioritizing and fulfilling their requests. Process improvement and automation is work inspired by the business desire for continuous improvement.

No matter the source, this work tends to fall into one of these three broad categories:

- **Emergency Issues:** Outages, and issues that indicate a pending outage that can be prevented, and emergency requests from other teams. Usually initiated by an alert sent by the monitoring system via SMS or pager. (Chapter 14)

- **Normal Requests:** Process work (repeatable processes that have not yet been automated), non-urgent trouble reports, informational questions, and initial consulting that results in larger projects. Usually initiated by a request ticket system. (Section 14.1.3)
- **Project Work:** Small and large projects that evolve the system. Managed with whatever project management style the team selects. (Section 12.4.2)

To assure that all sources and categories of work receive attention, we recommend this simple organizing principle: people should always be working on projects, with exceptions made to assure that emergency issues receive immediate attention and non-project customer requests are triaged and worked in a timely manner.

More specifically, at any given moment, the highest priority for one person on the team should be responding to emergencies, the highest priority for one other person on the team should be responding to normal requests, and the rest of the team should be focused on project work.

This is counter to the way operations teams often work: everyone running from emergency to emergency with no time for project work. If there is no effort dedicated to improving the situation, the team will simply run from emergency to emergency until they are burned out.

Major improvements come from project work. Project work requires concentration and focus. If you are constantly being interrupted with emergency issues and requests, you will not be able to get projects done. If an entire team is focused on emergencies and requests, nobody is working on projects.

It can be tempting to organize an operations team into three subteams, each focusing on one source of work or one category of work. Either of these approaches will create silos of responsibility. Process improvement is best done by the people involved in the process, not by observers.

To implement our recommended strategy, all members of the team focus on project work as their main priority. However, team members take turns being responsible for emergency issues as they arise. This responsibility is called **oncall**. Likewise, team members take turns being responsible for normal requests from other teams. This responsibility is called **ticket duty**.

It is common that oncall duty and ticket duty are scheduled in a rotation. For example, a team of eight people may use an eight-week cycle. Each person is assigned a week where he or she is on call: expected to respond to alerts, spending any remaining time on projects. Each person is also assigned a different week where he or she is on ticket duty: expected to focus on triaging and responding to request tickets first, working on other projects only if there is remaining time. This gives team members six weeks out of the cycle that can be focused on project work.

Limiting each rotation to a specific person makes for smoother handoffs to the next shift. In such a case, there are two people doing the handoff rather than a large operations team meeting. If more than 25 percent of a team needs to be dedicated to ticket duty and oncall, there is a serious problem with firefighting and a lack of automation.

The team manager should be part of the operational rotation. This practice ensures the manager is aware of the operational load and firefighting that goes on. It also ensures that nontechnical managers don't accidentally get hired into the operations organization.

Teams may combine oncall and ticket duty into one position if the amount of work in those categories is sufficiently small. Some teams may need to designate multiple people to fill each role.

Project work is best done in small teams. Solo projects can damage a team by making members feel disconnected or by permitting individuals to work without constructive feedback. Designs are better with at least some peer review. Without feedback, members may end up working on projects they feel are important but have marginal benefit. Conversely, large teams often get stalled by lack of consensus. In their case, focusing on shipping quickly overcomes many of these problems. It helps by making progress visible to the project members, the wider team, and management. Course corrections are easier to make when feedback is frequent.

The Agile methodology, discussed in Section 8.6, is an effective way to organize project work.

Meta-work

There is also meta-work: meetings, status reports, company functions. These generally eat into project time and should be minimized. For advice, see Chapter 11, "Eliminating Time Wasters," in the book *Time Management for System Administrators* by Limoncelli (2005).

7.3.1 Team Member Day Types

Now that we have established an organizing principle for the team's work, each team member can organize his or her work based on what kind of day it is: a project-focused day, an oncall day, or a ticket duty day.

Project-Focused Days

Most days should be project days for operational staff. Specifically, most days should be spent developing software that automates or optimizes aspects of the team's responsibilities. Non-software projects include shepherding a new launch or working with stakeholders on requirements for future releases.

Organizing the work of a team through a single bug tracking system has the benefit of reducing time spent checking different systems for status. Bug tracking systems provide an easy way for people to prioritize and track their work. On a typical project day the staff member starts by checking the bug tracking system to review the bugs assigned to him or her, or possibly to review unassigned issues of higher priority the team member might need to take on.

Software development in operations tends to mirror the Agile methodology: rather than making large, sudden changes, many small projects evolve the system over time. Chapter 12 will discuss automation and software engineering topics in more detail.

Projects that do not involve software development may involve technical work. Moving a service to a new datacenter is highly technical work that cannot be automated because it happens infrequently.

Operations staff tend not to physically touch hardware not just because of the heavy use of virtual machines, but also because even physical machines are located in datacenters that are located far away. Datacenter technicians act as **remote hands**, applying physical changes when needed.

Oncall Days

Oncall days are spent working on projects until an alert is received, usually by SMS, text message, or pager.

Once an alert is received, the issue is worked until it is resolved. Often there are multiple solutions to a problem, usually including one that will fix the problem quickly but temporarily and others that are long-term fixes. Generally the quick fix is employed because returning the service to normal operating parameters is paramount.

Once the alert is resolved, a number of other tasks should always be done. The alert should be categorized and annotated in some form of electronic alert journal so that trends may be discovered. If a quick fix was employed, a bug should be filed requesting a longer-term fix. The oncall person may take some time to update the playbook entry for this alert, thereby building organizational memory. If there was a user-visible outage or an SLA violation, a postmortem report should be written. An investigation should be conducted to ascertain the root cause of the problem. Writing a postmortem report, filing bugs, and root causes identification are all ways that we raise the visibility of issues so that they get attention. Otherwise, we will continually muddle through ad hoc workarounds and nothing will ever get better. Postmortem reports (possibly redacted for technical content) can be shared with the user community to build confidence in the service.

The benefit of having a specific person assigned to oncall duty at any given time is that it enables the rest of the team to remain focused on project work. Studies have found that the key to software developer productivity is to have long periods

of uninterrupted time. That said, if a major crisis appears, the oncall person will pull people away from their projects to assist.

If oncall shifts are too long, the oncall person will be overloaded with follow-up work. If the shifts are too close together, there will not be time to complete the follow-up work. Many great ideas for new projects and improvements are first imagined while servicing alerts. Between oncall shifts people should have enough time to pursue such projects.

Chapter 14 will discuss oncall in greater detail.

Ticket Duty Days

Ticket duty days are spent working on requests from customers. Here the customers are the internal users of the service, such as other service teams that use your service's API. These are not tickets from external users. Those items should be handled by customer support representatives.

While oncall is expected to have very fast reaction time, tickets generally have an expected response time measured in days.

Typical tickets may consist of questions about the service, which can lead to some consulting on how to use the service. They may also be requests for activation of a service, reports of problems or difficulties people are experiencing, and so forth. Sometimes tickets are created by automated systems. For example, a monitoring system may detect a situation that is not so urgent that it needs immediate response and may open a ticket instead.

Some long-running tickets left from the previous shift may need follow-up. Often there is a policy that if we are waiting for a reply from the customer, every three days the customer will be politely "poked" to make sure the issue is not forgotten. If the customer is waiting for follow-up from us, there may be a policy that urgent tickets will have a status update posted daily, with longer stretches of time for other priorities.

If a ticket will not be completed by the end of a shift, its status should be included in the shift report so that the next person can pick up where the previous person left off.

By dedicating a person to ticket duty, that individual can be more focused while responding to tickets. All tickets can be triaged and prioritized. There is more time to categorize tickets so that trends can be spotted. Efficiencies can be realized by batching up similar tickets to be done in a row. More importantly, by dedicating a person to tickets, that individual should have time to go deeper into each ticket: to update documentation and playbooks along the way, to deep-dive into bugs rather than find superficial workarounds, to fix complex broken processes. Ticket duty should not be a chore, but rather should be part of the strategy to reduce the overall work faced by the team.

Every operations team should have a goal of eliminating the need for people to open tickets with them, similar to how there should always be a goal to automate

manual processes. A ticket requesting information is an indication that documentation should be improved. It is best to respond to the question by adding the requested information to the service's FAQ or other user documentation and then directing the user to that document. Requests for service activation, allocations, or configuration changes indicate an opportunity to create a web-based portal or API to make such requests obsolete. Any ticket created by an automated system should have a corresponding playbook entry that explains how to process it, with a link to the bug ID requesting that the automation be improved to eliminate the need to open such tickets.

At the end of oncall and ticket duty shifts, it is common for the person to email out a shift report to the entire team. This report should mention any trends noticed and any advice or status information to be passed on to the next person. The oncall end-of-shift report should also include a log of which alerts were received and what was done in response.

When you are oncall or doing ticket duty, that is your main project. Other project work that is accomplished, if any, is a bonus. Management should not expect other projects to get done, nor should people be penalized for having the proper focus. When people end their oncall or ticket duty time, they should not complain that they weren't able to get any project work done; their project, so to speak, was ticket duty.

7.3.2 Other Strategies

There are many other ways to organize the work of a team. The team can rotate though projects focused on a particular goal or subsystem, it can focus on reducing toil, or special days can be set aside for reducing technical debt.

Focus or Theme

One can pick a category of issues to focus on for a month or two, changing themes periodically or when the current theme is complete. For example, at the start of a theme, a number of security-related issues can be selected and everyone commit to focusing on them until they are complete. Once these items are complete, the next theme begins. Some common themes include monitoring, a particular service or subservice, or automating a particular task.

If the team cohesion was low, this can help everyone feel as if they are working as a team again. It can also enhance productivity: if everyone has familiarized themselves with the same part of the code base, everyone can do a better job of helping each other.

Introducing a theme can also provide a certain amount of motivation. If the team is looking forward to the next theme (because it is more interesting, novel, or fun), they will be motivated to meet the goals of the current theme so they can start the next one.

Toil Reduction

Toil is manual work that is particularly exhausting. If a team calculates the number of hours spent on toil versus normal project work, that ratio should be as low as possible. Management may set a threshold such that if it goes above 50 percent, the team pauses all new features and works to solve the big problems that are the source of so much toil. (See Section 12.4.2.)

Fix-It Days

A day (or series of days) can be set aside to reduce technical debt. **Technical debt** is the accumulation of small unfinished amounts of work. By themselves, these bits and pieces are not urgent, but the accumulation of them starts to become a problem. For example, a Documentation Fix-It Day would involve everyone stopping all other work to focus on bugs related to documentation that needs to be improved. Alternatively, a Fix-It Week might be declared to focus on bringing all monitoring configurations up to a particular standard.

Often teams turn fix-its into a game. For example, at the start a list of tasks (or bugs) is published. Prizes are given out to the people who resolve the most bugs. If done company-wide, teams may receive T-shirts for participating and/or prizes for completing the most tasks.

7.4 Virtual Office

Many operations teams work from home rather than an office. Since work is virtual, with remote hands touching hardware when needed, we can work from anywhere. Therefore, it is common to work from anywhere. When necessary, the team meets in chat rooms or other virtual meeting spaces rather than physical meeting rooms. When teams work this way, communication must be more intentional because you don't just happen to see each other in the office.

It is good to have a policy that anyone who is not working from the office takes responsibility for staying in touch with the team. They should clearly and periodically communicate their status. In turn, the entire team should take responsibility for making sure remote workers do not feel isolated. Everyone should know what their team members are working on and take the time to include everyone in discussions. There are many tools that can help achieve this.

7.4.1 Communication Mechanisms

Chat rooms are commonly used for staying in touch throughout the day. Chat room transcripts should be stored and accessible so people can read what they may have missed. There are many chat room "bots" (software robots that join the

chat room and provide services) that can provide transcription services, pass messages to offline members, announce when oncall shifts change, and broadcast any alerts generated by the monitoring system. Some bots provide entertainment: At Google, a bot keeps track of who has received the most virtual high-fives. At Stack Exchange, a bot notices if anyone types the phrase "not my fault" and responds by selecting a random person from the room and announcing this person has been randomly designated to be blamed.

Higher-bandwidth communication systems include voice and video systems as well as screen sharing applications. The higher the bandwidth, the better the fidelity of communication that can be achieved. Text-chat is not good at conveying emotions, while voice and video can. Always switch to higher-fidelity communication systems when conveying emotions is more important, especially when an intense or heated debate starts.

The communication medium with the highest fidelity is the in-person meeting. Virtual teams greatly benefit from periodic in-person meetings. Everyone travels to the same place for a few days of meetings that focus on long-term planning, team building, and other issues that cannot be solved online.

7.4.2 Communication Policies

Many teams establish a communication agreement that clarifies which methods will be used in which situations. For example, a common agreement is that chat rooms will be the primary communication channel but only for ephemeral discussions. If a decision is made in the chat room or an announcement needs to be made, it will be broadcast via email. Email is for information that needs to carry across oncall shifts or day boundaries. Announcements with lasting effects, such as major policies or design decisions, need to be recorded in the team wiki or other document system (and the creation of said document needs to be announced via email). Establishing this chat–email–document paradigm can go a long way in reducing communication problems.

7.5 Summary

Operations is different from typical enterprise IT because it is focused on a particular service or group of services and because it has more demanding uptime requirements.

There is a tension between the operations team's desire for stability and the developers' desire to get new code into production. There are many ways to reach a balance. Most ways involve aligning goals by sharing responsibility for both uptime and velocity of new features.

Operations in distributed computing is done at a large scale. Processes that have to be done manually do not scale. Constant process improvement and automation are essential.

Operations is responsible for the life cycle of a service: launch, maintenance, upgrades, and decommissioning. Maintenance tasks include emergency and nonemergency response. In addition, related projects maintain and evolve the service.

Launches, decommissioning of services, and other tasks that are done infrequently require an attention to detail that is best assured by use of checklists. Checklists ensure that lessons learned in the past are carried forward.

The most productive use of time for operational staff is time spent automating and optimizing processes. This should be their primary responsibility. In addition, two other kinds of work require attention. Emergency tasks need fast response. Nonemergency requests need to be managed such that they are prioritized and worked in a timely manner. To make sure all these things happen, at any given time one person on the operations team should be focused on responding to emergencies; another should be assigned to prioritizing and working on nonemergency requests. When team members take turns addressing these responsibilities, they receive the dedicated resources required to assure they happen correctly by sharing the responsibility across the team. People also avoid burning out.

Operations teams generally work far from the actual machines that run their services. Since they operate the service remotely, they can work from anywhere there is a network connection. Therefore teams often work from different places, collaborating and communicating in a chat room or other virtual office. Many tools are available to enable this type of organizational structure. In such an environment, it becomes important to change the communication medium based on the type of communication required. Chat rooms are sufficient for general communication but voice and video are more appropriate for more intense discussions. Email is more appropriate when a record of the communication is required, or if it is important to reach people who are not currently online.

Exercises

1. What is operations? What are its major areas of responsibilities?
2. How does operations in distributed computing differ from traditional desktop support or enterprise client–server support?
3. Describe the service life cycle as it relates to a service you have experience with.
4. Section 7.1.2 discusses the change-instability cycle. Draw a series of graphs where the x-axis is time and the y-axis is the measure of stability. Each graph should represent two months of project time.

Each Monday, a major software release that introduces instability (9 bugs) is rolled out. On Tuesday through Friday, the team has an opportunity to roll out a "bug-fix" release, each of which fixes three bugs. Graph these scenarios:

(a) No bug-fix releases
(b) Two bug-fix releases after every major release
(c) Three bug-fix releases after every major release
(d) Four bug-fix releases after every major release
(e) No bug-fix release after odd releases, five bug-fix releases after even releases

5. What do you observe about the graphs from Exercise 4?
6. For a service you provide or have experience with, who are the stakeholders? Which interactions did you or your team have with them?
7. What are some of the ways operations work can be organized? How does this compare to how your current team is organized?
8. For a service you are involved with, give examples of work whose source is life-cycle management, interacting with stakeholders, and process improvement and automation.
9. For a service you are involved with, give examples of emergency issues, normal requests, and project work.

This page intentionally left blank

This page intentionally left blank

Index