

VOLUME 2




THE PRACTICE OF CLOUD SYSTEM ADMINISTRATION

DESIGNING AND OPERATING
LARGE DISTRIBUTED SYSTEMS

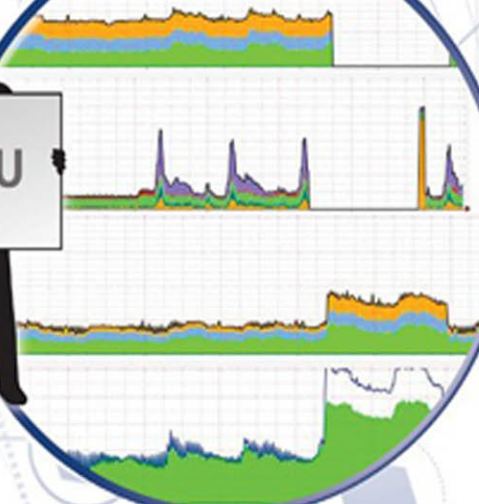
```
    subject = subject.String()
    change {
      notify in func(notifications map[string]bool) {
        for _, n := range notifications {
          n.Notify(state, n)
        }
      }
      switch status {
        case stCritical:
          notifyia.CritNotification()
        case stWarn:
          notifyia.WarnNotification()
      }
    }
  }
  if changed {
    n.nc ← true
  }
}

func (s *Scheduler) CheckAlertia want.Alertia {
  crits := s.CheckExpria, a.Crit, stCritical, nil
  w := s.CheckExpria, a.Warn, stWarning, crits
}

func (s *Scheduler) CheckExpria want.Alert, w want.Expr,
  if a == nil {
    return
  }
  results, err := s.Execute(s.cache, nil)
  if err != nil {
    log.Println(err)
    return
  }
  loop:
  for _, r := range results {
    if a.Spooled(r.Group) {
      continue
    }
    ak := AlertKey(a.Name, r.Group.String())
    for _, v := range ignore {
      if ak == v {
        continue loop
      }
    }
  }
}
```



YOU



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

DevOps and SRE Practices for
Web Services

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corp-sales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact intlcs@pearsoned.com.

Visit us on the Web: informit.com/aw

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

Copyright © 2015 Thomas A. Limoncelli, Virtual.NET Inc., Christina J. Lear née Hogan

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, 200 Old Tappan Road, Old Tappan, New Jersey 07675, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-94318-7

ISBN-10: 0-321-94318-X

3 17

Contents at a Glance

<i>Contents</i>	vii
<i>Preface</i>	xxiii
<i>About the Authors</i>	xxix
Introduction	1
Part I Design: Building It	7
Chapter 1 Designing in a Distributed World	9
Chapter 2 Designing for Operations	31
Chapter 3 Selecting a Service Platform	51
Chapter 4 Application Architectures	69
Chapter 5 Design Patterns for Scaling	95
Chapter 6 Design Patterns for Resiliency	119
Part II Operations: Running It	145
Chapter 7 Operations in a Distributed World	147
Chapter 8 DevOps Culture	171
Chapter 9 Service Delivery: The Build Phase	195
Chapter 10 Service Delivery: The Deployment Phase	211
Chapter 11 Upgrading Live Services	225
Chapter 12 Automation	243
Chapter 13 Design Documents	275
Chapter 14 Oncall	285
Chapter 15 Disaster Preparedness	307
Chapter 16 Monitoring Fundamentals	331

Chapter 17 Monitoring Architecture and Practice	345
Chapter 18 Capacity Planning	365
Chapter 19 Creating KPIs	387
Chapter 20 Operational Excellence	401
Epilogue	417
Part III Appendices	419
Appendix A Assessments	421
Appendix B The Origins and Future of Distributed Computing and Clouds	451
Appendix C Scaling Terminology and Concepts	475
Appendix D Templates and Examples	481
Appendix E Recommended Reading	487
<i>Bibliography</i>	491
<i>Index</i>	499

Contents

<i>Preface</i>	xxiii
<i>About the Authors</i>	xxix
Introduction	1
Part I Design: Building It	7
1 Designing in a Distributed World	9
1.1 Visibility at Scale	10
1.2 The Importance of Simplicity	11
1.3 Composition	12
1.3.1 Load Balancer with Multiple Backend Replicas	12
1.3.2 Server with Multiple Backends	14
1.3.3 Server Tree	16
1.4 Distributed State	17
1.5 The CAP Principle	21
1.5.1 Consistency	21
1.5.2 Availability	21
1.5.3 Partition Tolerance	22
1.6 Loosely Coupled Systems	24
1.7 Speed	26
1.8 Summary	29
Exercises	30

2	Designing for Operations	31
2.1	Operational Requirements	31
2.1.1	Configuration	33
2.1.2	Startup and Shutdown	34
2.1.3	Queue Draining	35
2.1.4	Software Upgrades	36
2.1.5	Backups and Restores	36
2.1.6	Redundancy	37
2.1.7	Replicated Databases	37
2.1.8	Hot Swaps	38
2.1.9	Toggles for Individual Features	39
2.1.10	Graceful Degradation	39
2.1.11	Access Controls and Rate Limits	40
2.1.12	Data Import Controls	41
2.1.13	Monitoring	42
2.1.14	Auditing	42
2.1.15	Debug Instrumentation	43
2.1.16	Exception Collection	43
2.1.17	Documentation for Operations	44
2.2	Implementing Design for Operations	45
2.2.1	Build Features in from the Beginning	45
2.2.2	Request Features as They Are Identified	46
2.2.3	Write the Features Yourself	47
2.2.4	Work with a Third-Party Vendor	48
2.3	Improving the Model	48
2.4	Summary	49
	Exercises	50
3	Selecting a Service Platform	51
3.1	Level of Service Abstraction	52
3.1.1	Infrastructure as a Service	52
3.1.2	Platform as a Service	54
3.1.3	Software as a Service	55
3.2	Type of Machine	56
3.2.1	Physical Machines	57
3.2.2	Virtual Machines	57
3.2.3	Containers	60

3.3	Level of Resource Sharing	62
3.3.1	Compliance	63
3.3.2	Privacy	63
3.3.3	Cost	63
3.3.4	Control	64
3.4	Colocation	65
3.5	Selection Strategies	66
3.6	Summary	68
	Exercises	68
4	Application Architectures	69
4.1	Single-Machine Web Server	70
4.2	Three-Tier Web Service	71
4.2.1	Load Balancer Types	72
4.2.2	Load Balancing Methods	74
4.2.3	Load Balancing with Shared State	75
4.2.4	User Identity	76
4.2.5	Scaling	76
4.3	Four-Tier Web Service	77
4.3.1	Frontends	78
4.3.2	Application Servers	79
4.3.3	Configuration Options	80
4.4	Reverse Proxy Service	80
4.5	Cloud-Scale Service	80
4.5.1	Global Load Balancer	81
4.5.2	Global Load Balancing Methods	82
4.5.3	Global Load Balancing with User-Specific Data	82
4.5.4	Internal Backbone	83
4.6	Message Bus Architectures	85
4.6.1	Message Bus Designs	86
4.6.2	Message Bus Reliability	87
4.6.3	Example 1: Link-Shortening Site	87
4.6.4	Example 2: Employee Human Resources Data Updates	89
4.7	Service-Oriented Architecture	90
4.7.1	Flexibility	91
4.7.2	Support	91
4.7.3	Best Practices	91

4.8	Summary	92
	Exercises	93
5	Design Patterns for Scaling	95
5.1	General Strategy	96
5.1.1	Identify Bottlenecks	96
5.1.2	Reengineer Components	97
5.1.3	Measure Results	97
5.1.4	Be Proactive	97
5.2	Scaling Up	98
5.3	The AKF Scaling Cube	99
5.3.1	<i>x</i> : Horizontal Duplication	99
5.3.2	<i>y</i> : Functional or Service Splits	101
5.3.3	<i>z</i> : Lookup-Oriented Split	102
5.3.4	Combinations	104
5.4	Caching	104
5.4.1	Cache Effectiveness	105
5.4.2	Cache Placement	106
5.4.3	Cache Persistence	106
5.4.4	Cache Replacement Algorithms	107
5.4.5	Cache Entry Invalidation	108
5.4.6	Cache Size	109
5.5	Data Sharding	110
5.6	Threading	112
5.7	Queueing	113
5.7.1	Benefits	113
5.7.2	Variations	113
5.8	Content Delivery Networks	114
5.9	Summary	116
	Exercises	116
6	Design Patterns for Resiliency	119
6.1	Software Resiliency Beats Hardware Reliability	120
6.2	Everything Malfunctions Eventually	121
6.2.1	MTBF in Distributed Systems	121
6.2.2	The Traditional Approach	122
6.2.3	The Distributed Computing Approach	123

6.3	Resiliency through Spare Capacity	124
6.3.1	How Much Spare Capacity	125
6.3.2	Load Sharing versus Hot Spares	126
6.4	Failure Domains	126
6.5	Software Failures	128
6.5.1	Software Crashes	128
6.5.2	Software Hangs	129
6.5.3	Query of Death	130
6.6	Physical Failures	131
6.6.1	Parts and Components	131
6.6.2	Machines	134
6.6.3	Load Balancers	134
6.6.4	Racks	136
6.6.5	Datacenters	137
6.7	Overload Failures	138
6.7.1	Traffic Surges	138
6.7.2	DoS and DDoS Attacks	140
6.7.3	Scraping Attacks	140
6.8	Human Error	141
6.9	Summary	142
	Exercises	143

Part II Operations: Running It 145

7	Operations in a Distributed World	147
7.1	Distributed Systems Operations	148
7.1.1	SRE versus Traditional Enterprise IT	148
7.1.2	Change versus Stability	149
7.1.3	Defining SRE	151
7.1.4	Operations at Scale	152
7.2	Service Life Cycle	155
7.2.1	Service Launches	156
7.2.2	Service Decommissioning	160
7.3	Organizing Strategy for Operational Teams	160
7.3.1	Team Member Day Types	162
7.3.2	Other Strategies	165

7.4	Virtual Office	166
7.4.1	Communication Mechanisms	166
7.4.2	Communication Policies	167
7.5	Summary	167
	Exercises	168
8	DevOps Culture	171
8.1	What Is DevOps?	172
8.1.1	The Traditional Approach	173
8.1.2	The DevOps Approach	175
8.2	The Three Ways of DevOps	176
8.2.1	The First Way: Workflow	176
8.2.2	The Second Way: Improve Feedback	177
8.2.3	The Third Way: Continual Experimentation and Learning	178
8.2.4	Small Batches Are Better	178
8.2.5	Adopting the Strategies	179
8.3	History of DevOps	180
8.3.1	Evolution	180
8.3.2	Site Reliability Engineering	181
8.4	DevOps Values and Principles	181
8.4.1	Relationships	182
8.4.2	Integration	182
8.4.3	Automation	182
8.4.4	Continuous Improvement	183
8.4.5	Common Nontechnical DevOps Practices	183
8.4.6	Common Technical DevOps Practices	184
8.4.7	Release Engineering DevOps Practices	186
8.5	Converting to DevOps	186
8.5.1	Getting Started	187
8.5.2	DevOps at the Business Level	187
8.6	Agile and Continuous Delivery	188
8.6.1	What Is Agile?	188
8.6.2	What Is Continuous Delivery?	189
8.7	Summary	192
	Exercises	193

9	Service Delivery: The Build Phase	195
9.1	Service Delivery Strategies	197
9.1.1	Pattern: Modern DevOps Methodology	197
9.1.2	Anti-pattern: Waterfall Methodology	199
9.2	The Virtuous Cycle of Quality	200
9.3	Build-Phase Steps	202
9.3.1	Develop	202
9.3.2	Commit	202
9.3.3	Build	203
9.3.4	Package	204
9.3.5	Register	204
9.4	Build Console	205
9.5	Continuous Integration	205
9.6	Packages as Handoff Interface	207
9.7	Summary	208
	Exercises	209
10	Service Delivery: The Deployment Phase	211
10.1	Deployment-Phase Steps	211
10.1.1	Promotion	212
10.1.2	Installation	212
10.1.3	Configuration	213
10.2	Testing and Approval	214
10.2.1	Testing	215
10.2.2	Approval	216
10.3	Operations Console	217
10.4	Infrastructure Automation Strategies	217
10.4.1	Preparing Physical Machines	217
10.4.2	Preparing Virtual Machines	218
10.4.3	Installing OS and Services	219
10.5	Continuous Delivery	221
10.6	Infrastructure as Code	221
10.7	Other Platform Services	222
10.8	Summary	222
	Exercises	223

11	Upgrading Live Services	225
11.1	Taking the Service Down for Upgrading	225
11.2	Rolling Upgrades	226
11.3	Canary	227
11.4	Phased Roll-outs	229
11.5	Proportional Shedding	230
11.6	Blue-Green Deployment	230
11.7	Toggling Features	230
11.8	Live Schema Changes	234
11.9	Live Code Changes	236
11.10	Continuous Deployment	236
11.11	Dealing with Failed Code Pushes	239
11.12	Release Atomicity	240
11.13	Summary	241
	Exercises	241
12	Automation	243
12.1	Approaches to Automation	244
12.1.1	The Left-Over Principle	245
12.1.2	The Compensatory Principle	246
12.1.3	The Complementarity Principle	247
12.1.4	Automation for System Administration	248
12.1.5	Lessons Learned	249
12.2	Tool Building versus Automation	250
12.2.1	Example: Auto Manufacturing	251
12.2.2	Example: Machine Configuration	251
12.2.3	Example: Account Creation	251
12.2.4	Tools Are Good, But Automation Is Better	252
12.3	Goals of Automation	252
12.4	Creating Automation	255
12.4.1	Making Time to Automate	256
12.4.2	Reducing Toil	257
12.4.3	Determining What to Automate First	257
12.5	How to Automate	258
12.6	Language Tools	258
12.6.1	Shell Scripting Languages	259
12.6.2	Scripting Languages	259

12.6.3	Compiled Languages	260
12.6.4	Configuration Management Languages	260
12.7	Software Engineering Tools and Techniques	262
12.7.1	Issue Tracking Systems	263
12.7.2	Version Control Systems	265
12.7.3	Software Packaging	266
12.7.4	Style Guides	266
12.7.5	Test-Driven Development	267
12.7.6	Code Reviews	268
12.7.7	Writing Just Enough Code	269
12.8	Multitenant Systems	270
12.9	Summary	271
	Exercises	272
13	Design Documents	275
13.1	Design Documents Overview	275
13.1.1	Documenting Changes and Rationale	276
13.1.2	Documentation as a Repository of Past Decisions	276
13.2	Design Document Anatomy	277
13.3	Template	279
13.4	Document Archive	279
13.5	Review Workflows	280
13.5.1	Reviewers and Approvers	281
13.5.2	Achieving Sign-off	281
13.6	Adopting Design Documents	282
13.7	Summary	283
	Exercises	284
14	Oncall	285
14.1	Designing Oncall	285
14.1.1	Start with the SLA	286
14.1.2	Oncall Roster	287
14.1.3	Onduty	288
14.1.4	Oncall Schedule Design	288
14.1.5	The Oncall Calendar	290
14.1.6	Oncall Frequency	291

14.1.7	Types of Notifications	292
14.1.8	After-Hours Maintenance Coordination	294
14.2	Being Oncall	294
14.2.1	Pre-shift Responsibilities	294
14.2.2	Regular Oncall Responsibilities	294
14.2.3	Alert Responsibilities	295
14.2.4	Observe, Orient, Decide, Act (OODA)	296
14.2.5	Oncall Playbook	297
14.2.6	Third-Party Escalation	298
14.2.7	End-of-Shift Responsibilities	299
14.3	Between Oncall Shifts	299
14.3.1	Long-Term Fixes	299
14.3.2	Postmortems	300
14.4	Periodic Review of Alerts	302
14.5	Being Paged Too Much	304
14.6	Summary	305
	Exercises	306
15	Disaster Preparedness	307
15.1	Mindset	308
15.1.1	Antifragile Systems	308
15.1.2	Reducing Risk	309
15.2	Individual Training: Wheel of Misfortune	311
15.3	Team Training: Fire Drills	312
15.3.1	Service Testing	313
15.3.2	Random Testing	314
15.4	Training for Organizations: Game Day/DiRT	315
15.4.1	Getting Started	316
15.4.2	Increasing Scope	317
15.4.3	Implementation and Logistics	318
15.4.4	Experiencing a DiRT Test	320
15.5	Incident Command System	323
15.5.1	How It Works: Public Safety Arena	325
15.5.2	How It Works: IT Operations Arena	326
15.5.3	Incident Action Plan	326
15.5.4	Best Practices	327
15.5.5	ICS Example	328

15.6	Summary	329
	Exercises	330
16	Monitoring Fundamentals	331
16.1	Overview	332
	16.1.1 Uses of Monitoring	333
	16.1.2 Service Management	334
16.2	Consumers of Monitoring Information	334
16.3	What to Monitor	336
16.4	Retention	338
16.5	Meta-monitoring	339
16.6	Logs	340
	16.6.1 Approach	341
	16.6.2 Timestamps	341
16.7	Summary	342
	Exercises	342
17	Monitoring Architecture and Practice	345
17.1	Sensing and Measurement	346
	17.1.1 Blackbox versus Whitebox Monitoring	346
	17.1.2 Direct versus Synthesized Measurements	347
	17.1.3 Rate versus Capability Monitoring	348
	17.1.4 Gauges versus Counters	348
17.2	Collection	350
	17.2.1 Push versus Pull	350
	17.2.2 Protocol Selection	351
	17.2.3 Server Component versus Agent versus Poller	352
	17.2.4 Central versus Regional Collectors	352
17.3	Analysis and Computation	353
17.4	Alerting and Escalation Manager	354
	17.4.1 Alerting, Escalation, and Acknowledgments	355
	17.4.2 Silence versus Inhibit	356
17.5	Visualization	358
	17.5.1 Percentiles	359
	17.5.2 Stack Ranking	360
	17.5.3 Histograms	361
17.6	Storage	362

17.7	Configuration	362
17.8	Summary	363
	Exercises	364
18	Capacity Planning	365
18.1	Standard Capacity Planning	366
18.1.1	Current Usage	368
18.1.2	Normal Growth	369
18.1.3	Planned Growth	369
18.1.4	Headroom	370
18.1.5	Resiliency	370
18.1.6	Timetable	371
18.2	Advanced Capacity Planning	371
18.2.1	Identifying Your Primary Resources	372
18.2.2	Knowing Your Capacity Limits	372
18.2.3	Identifying Your Core Drivers	373
18.2.4	Measuring Engagement	374
18.2.5	Analyzing the Data	375
18.2.6	Monitoring the Key Indicators	380
18.2.7	Delegating Capacity Planning	381
18.3	Resource Regression	381
18.4	Launching New Services	382
18.5	Reduce Provisioning Time	384
18.6	Summary	385
	Exercises	386
19	Creating KPIs	387
19.1	What Is a KPI?	388
19.2	Creating KPIs	389
19.2.1	Step 1: Envision the Ideal	390
19.2.2	Step 2: Quantify Distance to the Ideal	390
19.2.3	Step 3: Imagine How Behavior Will Change	390
19.2.4	Step 4: Revise and Select	391
19.2.5	Step 5: Deploy the KPI	392
19.3	Example KPI: Machine Allocation	393
19.3.1	The First Pass	393

19.3.2	The Second Pass	394
19.3.3	Evaluating the KPI	396
19.4	Case Study: Error Budget	396
19.4.1	Conflicting Goals	396
19.4.2	A Unified Goal	397
19.4.3	Everyone Benefits	398
19.5	Summary	399
	Exercises	399
20	Operational Excellence	401
20.1	What Does Operational Excellence Look Like?	401
20.2	How to Measure Greatness	402
20.3	Assessment Methodology	403
20.3.1	Operational Responsibilities	403
20.3.2	Assessment Levels	405
20.3.3	Assessment Questions and Look-For's	407
20.4	Service Assessments	407
20.4.1	Identifying What to Assess	408
20.4.2	Assessing Each Service	408
20.4.3	Comparing Results across Services	409
20.4.4	Acting on the Results	410
20.4.5	Assessment and Project Planning Frequencies	410
20.5	Organizational Assessments	411
20.6	Levels of Improvement	412
20.7	Getting Started	413
20.8	Summary	414
	Exercises	415
Epilogue		416
Part III	Appendices	419
A	Assessments	421
A.1	Regular Tasks (RT)	423
A.2	Emergency Response (ER)	426
A.3	Monitoring and Metrics (MM)	428

A.4	Capacity Planning (CP)	431
A.5	Change Management (CM)	433
A.6	New Product Introduction and Removal (NPI/NPR)	435
A.7	Service Deployment and Decommissioning (SDD)	437
A.8	Performance and Efficiency (PE)	439
A.9	Service Delivery: The Build Phase	442
A.10	Service Delivery: The Deployment Phase	444
A.11	Toil Reduction	446
A.12	Disaster Preparedness	448
B	The Origins and Future of Distributed Computing and Clouds	451
B.1	The Pre-Web Era (1985–1994)	452
	Availability Requirements	452
	Technology	453
	Scaling	454
	High Availability	454
	Costs	454
B.2	The First Web Era: The Bubble (1995–2000)	455
	Availability Requirements	455
	Technology	455
	Scaling	456
	High Availability	457
	Costs	459
B.3	The Dot-Bomb Era (2000–2003)	459
	Availability Requirements	460
	Technology	460
	High Availability	461
	Scaling	462
	Costs	464
B.4	The Second Web Era (2003–2010)	465
	Availability Requirements	465
	Technology	465
	High Availability	466
	Scaling	467
	Costs	468

B.5	The Cloud Computing Era (2010–present)	469
	Availability Requirements	469
	Costs	469
	Scaling and High Availability	471
	Technology	472
B.6	Conclusion	472
	Exercises	473
C	Scaling Terminology and Concepts	475
C.1	Constant, Linear, and Exponential Scaling	475
C.2	Big O Notation	476
C.3	Limitations of Big O Notation	478
D	Templates and Examples	481
D.1	Design Document Template	481
D.2	Design Document Example	482
D.3	Sample Postmortem Template	484
E	Recommended Reading	487
	<i>Bibliography</i>	491
	<i>Index</i>	499

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.

Chapter 16: Monitoring Fundamentals

Monitoring terminology and strategy.

Chapter 17: Monitoring Architecture and Practice

The components and practice of monitoring.

Chapter 18: Capacity Planning

Planning for and providing additional resources before we need them.

Chapter 19: Creating KPIs

Driving behavior scientifically through measurement and reflection.

Chapter 20: Operational Excellence

Strategies for constant improvement.

Epilogue

Some final thoughts.

Part III Appendices

Appendix A: Assessments

Appendix B: The Origins and Future of Distributed Computing and Clouds

Appendix C: Scaling Terminology and Concepts

Appendix D: Templates and Examples

Appendix E: Recommended Reading

Bibliography

Index

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 Stack-Overflow.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 home-grown 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.

Part I

Design: Building It

This page intentionally left blank

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,

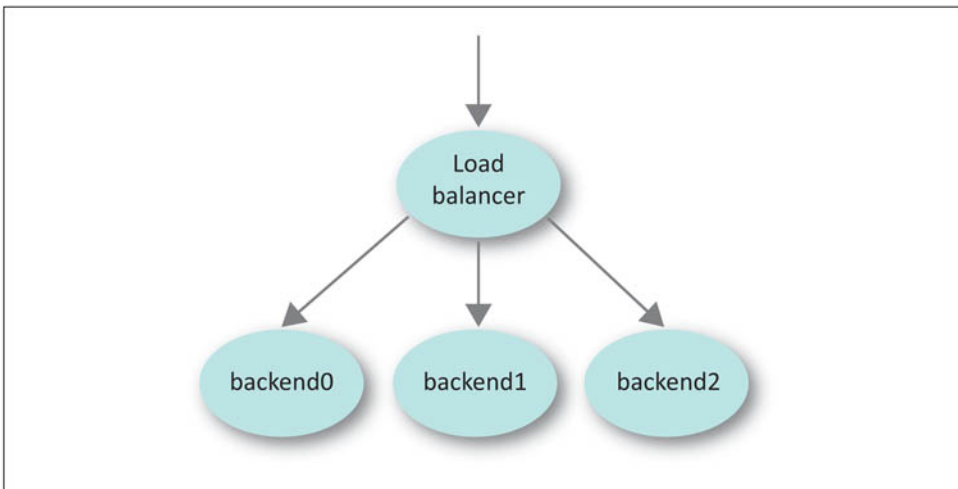


Figure 1.1: A load balancer with many replicas

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

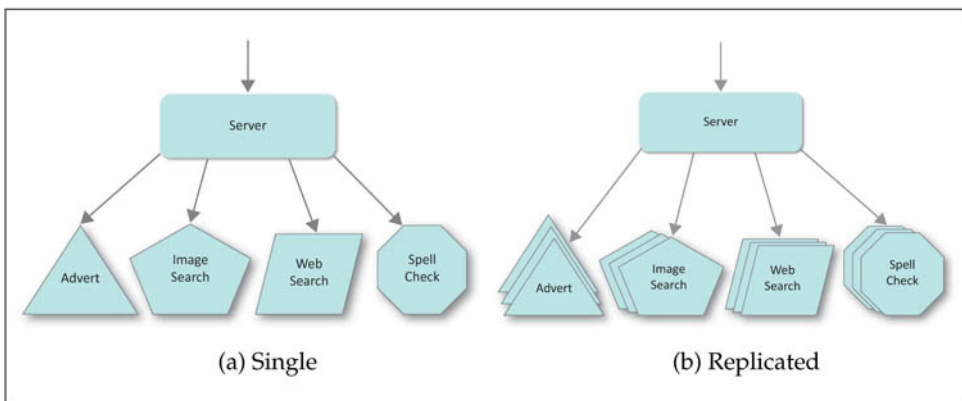


Figure 1.2: This service is composed of a server and many backends.

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.

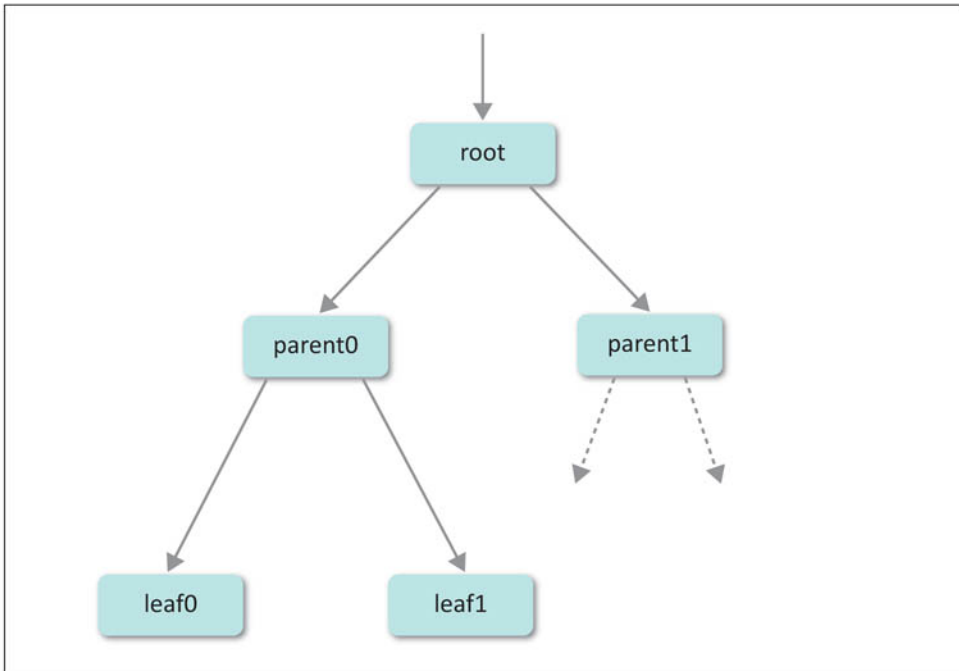


Figure 1.3: A server tree

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

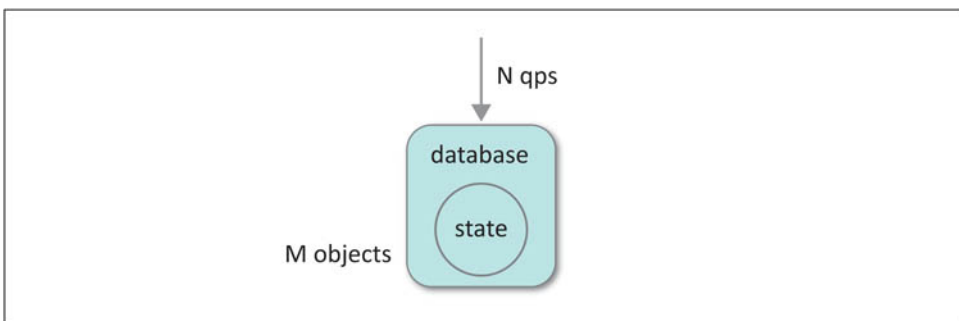


Figure 1.4: State kept in one location; not distributed computing

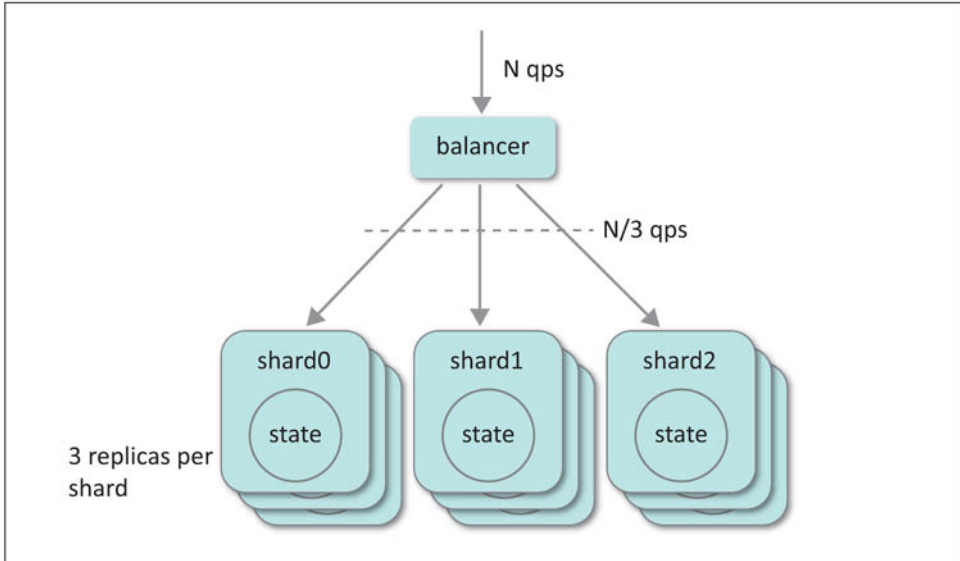


Figure 1.5: This distributed state is sharded and replicated.

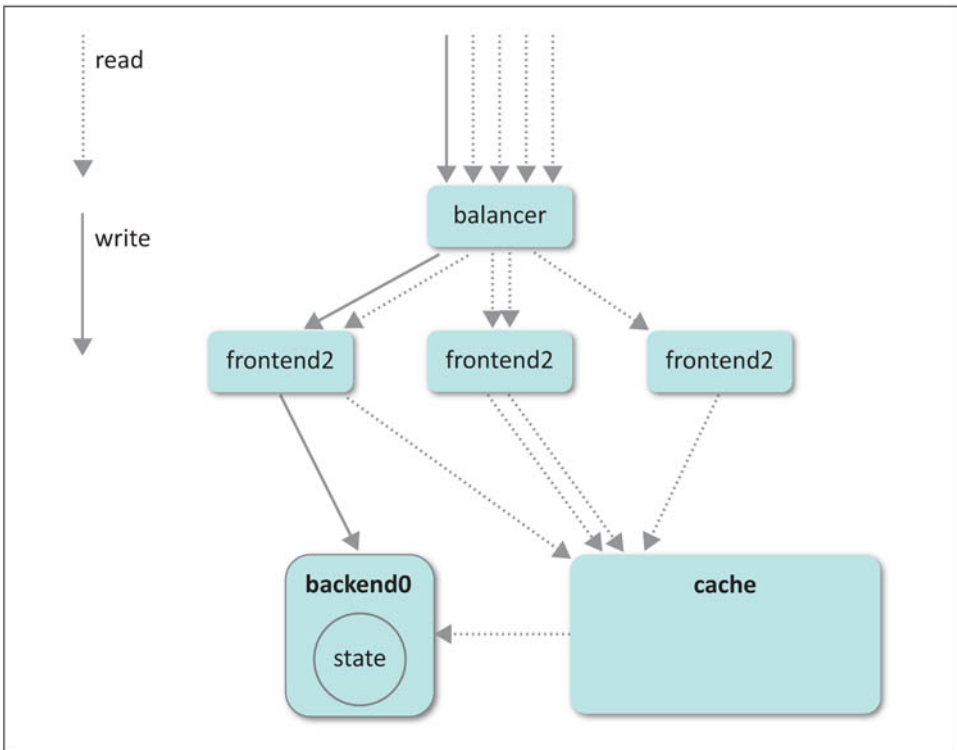


Figure 1.6: State updates using cached data lead to an inconsistent view.

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.

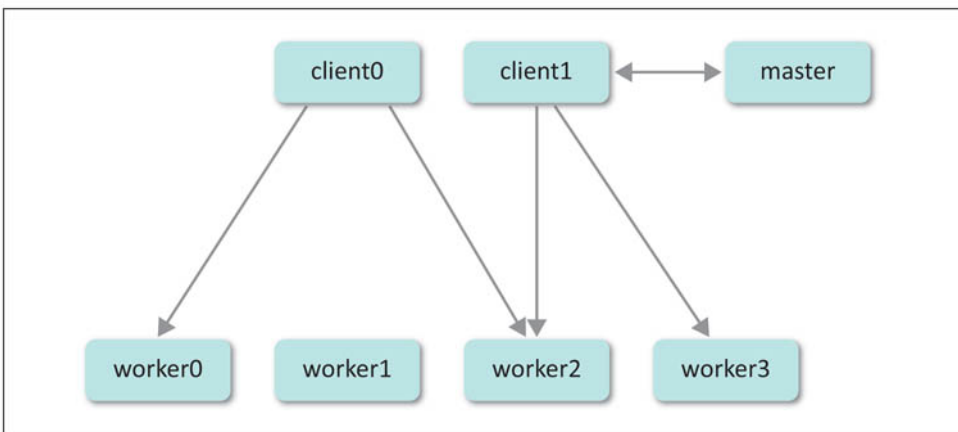


Figure 1.7: This master server delegates replies to other servers.

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 consistent 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,

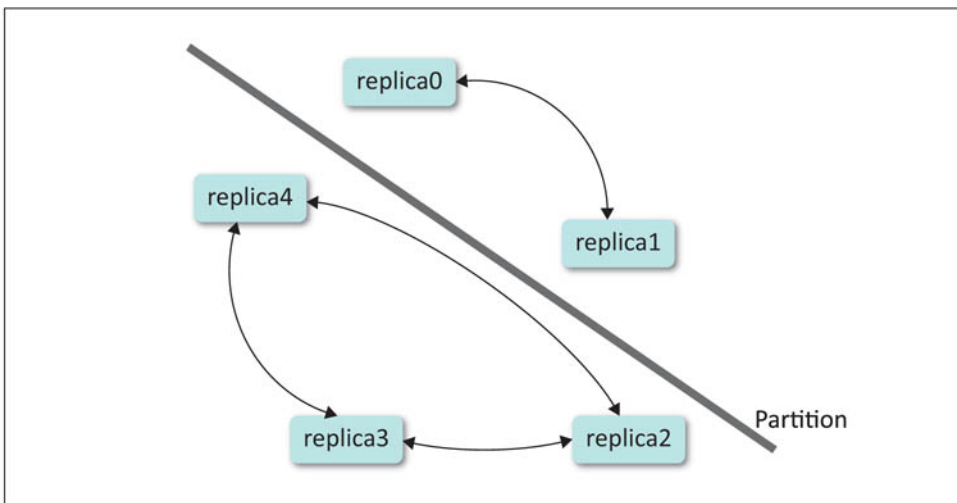


Figure 1.8: Nodes partitioned from each other

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,

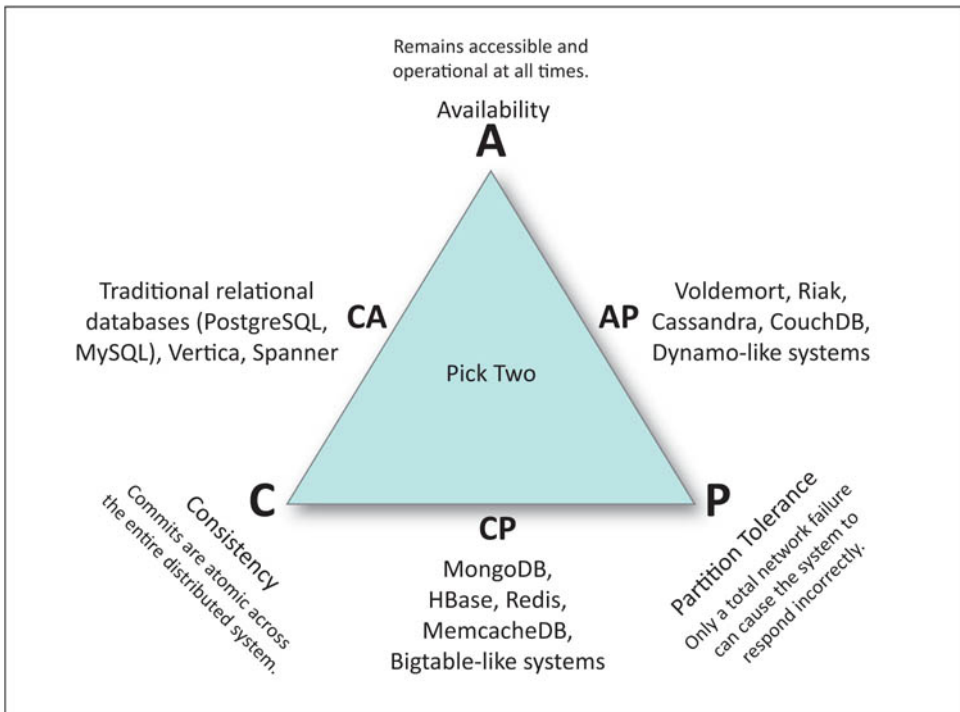


Figure 1.9: The CAP Principle

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, Riak, 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.