Develop Cloud-Native Applications

Cloud Architecture Patterns

O'REILLY®

Bill Wilder

Cloud Architecture Patterns

If your team is investigating ways to design applications for the cloud, this concise book introduces 11 architecture patterns that can help you take advantage of several cloud-platform services. You'll learn how each of these platform-agnostic patterns work, when they might be useful in the cloud, and what impact they'll have on your application architecture. You'll also see an example of each pattern applied to an application built with Windows Azure.

The patterns are organized into four major topics, such as scalability and eventual consistency, and primer chapters provide background on each topic. With the information in this book, you'll be able to make informed decisions for designing effective cloud-native applications.

Learn about architectural patterns for:

- **Scalability.** Discover the advantages of horizontal scaling. Patterns covered include Horizontally Scaling Compute, Queue-Centric Workflow, and Auto-Scaling
- **Eventual consistency.** Learn how to maintain data consistency across a distributed system. Patterns covered include MapReduce and Database Sharding
- **Multitenancy and commodity hardware.** Understand how they influence your applications. Patterns covered include Busy Signal and Node Failure
- **Network latency.** Learn how to deal with delays due to network latency. Patterns covered include Colocation, Direct-to-Storage and Multi-Site Deployment

Twitter: @oreillymedia
facebook.com/oreilly

US $24.99 CAN $26.99
ISBN: 978-1-449-31977-9

O'REILLY®
oreilly.com

Cloud Architecture Patterns

Bill Wilder

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Cloud Architecture Patterns

by Bill Wilder

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Rachel Roumeliotis
Production Editor: Holly Bauer

Proofreader: BIM Publishing Services
Indexer: BIM Publishing Services
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Elizabeth O'Connor, Rebecca Demarest

Revision History for the First Edition:

2012-09-20 First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449319779* for release details.

ISBN: 978-1-449-31977-9

[LSI]

Table of Contents

Preface

This book focuses on the development of *cloud-native applications*. A cloud-native application is architected to take advantage of specific engineering practices that have proven successful in some of the world's largest and most successful web properties. Many of these practices are unconventional, yet the need for unprecedented scalability and efficiency inspired development and drove adoption in the relatively small number of companies that truly needed them. After an approach has been adopted successfully enough times, it becomes a *pattern*. In this book, a pattern is an approach that can be duplicated to produce an expected outcome. Use of any of the patterns included in this book will impact the architecture of your application, some in small ways, some in large ways.

Historically, many of these patterns have been risky and expensive to implement, and it made sense for most companies to avoid them. That has changed. Cloud computing platforms now offer services that dramatically lower the risk and cost by shielding the application from most of the complexity. The desired benefit of using the pattern is the same, but the cost and complexity of realizing that benefit is lower. The majority of modern applications can now make practical use of these heretofore seldom used patterns.

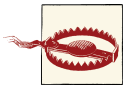
Cloud platform services simplify building cloud-native applications.

The architecture patterns described in this book were selected because they are useful for building cloud-native applications. None are *specific* to the cloud. All are *relevant* to the cloud.

Concisely stated, cloud-native applications leverage cloud-platform services to cost-efficiently and automatically allocate resources horizontally to match current needs, handle transient and hardware failures without downtime, and minimize network latency. These terms are explained throughout the book.

An application need not support millions of users to benefit from cloud-native patterns. There are benefits beyond scalability that are applicable to many web and mobile applications. These are also explored throughout the book.

The patterns assume the use of a cloud platform, though not any specific one. General expectations are outlined in *Scalability Primer* (Chapter 1).



This book will *not* help you move traditional applications to the cloud "as is."

Audience

This book is written for those involved in—or who wish to become involved in—conversations around software architecture, especially cloud architecture. The audience is not limited to those with "architect" in their job title. The material should be relevant to developers, CTOs, and CIOs; more technical testers, designers, analysts, product managers, and others who wish to understand the basic concepts.

For learning beyond the material in this book, paths will diverge. Some readers will not require information beyond what is provided in this book. For those going deeper, this book is just a starting point. Many references for further reading are provided in Appendix A.

Why This Book Exists

I have been studying cloud computing and the Windows Azure Platform since it was unveiled at the Microsoft Professional Developer's Conference (PDC) in 2008. I started the Boston Azure Cloud User Group in 2009 to accelerate my learning, I began writing and speaking on cloud topics, and then started consulting. I realized there were many technologists who had not been exposed to the interesting differences between the application-building techniques they'd been using for years and those used in creating cloud-native applications.

The most important conversations about the cloud are more about architecture than technology.

This is the book I wish I could have read myself when I was starting to learn about cloud and Azure, or even ten years ago when I was learning about scaling. Because such a book did not materialize on its own, I have written it. The principles, concepts, and patterns in this book are growing more important every day, making this book more relevant than ever.

Assumptions This Book Makes

This book assumes that the reader knows what the cloud is and has some familiarity with how cloud services can be used to build applications with Windows Azure, Amazon Web Services, Google App Engine, or similar public or private cloud platforms. The reader is not expected to be familiar with the concept of a cloud-native application and how cloud platform services can be used to build one.

This book is written to educate and inform. While this book will help the reader understand cloud architecture, it is not actually advising the use of any particular patterns. The goal of the book is to provide readers with enough information to make informed decisions.

This book focuses on concepts and patterns, and does not always directly discuss costs. Readers should consider the costs of using cloud platform services, as well as trade-offs in development effort. Get to know the pricing calculator for your cloud platform of choice.

This book includes patterns useful for architecting cloud-native applications. This book is not focused on *how to* (beyond what is needed to understand), but rather about *when* and *why* you might want to apply certain patterns, and then which features in Windows Azure you might find useful. This book intentionally does not delve into the detailed implementation level because there are many other resources for those needs, and that would distract from the real focus: architecture.

This book does not provide a comprehensive treatment of how to build cloud applications. The focus of the pattern chapters is on understanding each pattern in the context of its value in building cloud-native applications. Thus, not all facets are covered; emphasis is on the big picture. For example, in *Database Sharding Pattern* (Chapter 7), techniques such as optimizing queries and examining query plans are not discussed because they are no different in the cloud. Further, this book is not intended to guide development, but rather provide some options for architecture; some references are given pointing to more resources for realizing many of the patterns, but that is not otherwise intended to be part of this book.

Contents of This Book

There are two types of chapters in this book: primers and patterns.

Individual chapters include:

Scalability Primer (Chapter 1)
: This primer explains scalability with an emphasis on the key differences between vertical and horizontal scaling.

Horizontally Scaling Compute Pattern (Chapter 2)
: This fundamental pattern focuses on horizontally scaling compute nodes.

Queue-Centric Workflow Pattern (Chapter 3)
: This essential pattern for loose coupling focuses on asynchronous delivery of command requests sent from the user interface to a processing service. This pattern is a subset of the CQRS pattern.

Auto-Scaling Pattern (Chapter 4)
: This essential pattern for automating operations makes horizontal scaling more practical and cost-efficient.

Eventual Consistency Primer (Chapter 5)
: This primer introduces eventual consistency and explains some ways to use it.

MapReduce Pattern (Chapter 6)
: This pattern focuses on applying the MapReduce data processing pattern.

Database Sharding Pattern (Chapter 7)
: This advanced pattern focuses on horizontally scaling data through sharding.

Multitenancy and Commodity Hardware Primer (Chapter 8)
: This primer introduces multitenancy and commodity hardware and explains why they are used by cloud platforms.

Busy Signal Pattern (Chapter 9)
: This pattern focuses on how an application should respond when a cloud service responds to a programmatic request with a busy signal rather than success.

Node Failure Pattern (Chapter 10)
: This pattern focuses on how an application should respond when the compute node on which it is running shuts down or fails.

Network Latency Primer (Chapter 11)
: This basic primer explains network latency and why delays due to network latency matter.

Colocate Pattern (Chapter 12)
: This basic pattern focuses on avoiding unnecessary network latency.

Valet Key Pattern (Chapter 13)
: This pattern focuses on efficiently using cloud storage services with untrusted clients.

CDN Pattern (Chapter 14)
: This pattern focuses on reducing network latency for commonly accessed files through globally distributed edge caching.

Multisite Deployment Pattern (Chapter 15)
: This advanced pattern focuses on deploying a single application to more than one data center.

Appendix A
: The appendix contains a list of references for readers interested in additional material related to the primers and patterns presented in the book.

The primers exist to ensure that readers have the proper background to appreciate the pattern; primers precede the pattern chapters for which that background is needed. The patterns are the heart of the book and describe how to address specific challenges you are likely to encounter in the cloud.

Because individual patterns tend to impact multiple architectural concerns, these patterns defy placement into a clean hierarchy or taxonomy; instead, each pattern chapter includes an *Impact* section (listing the areas of architectural impact). Other sections include *Context* (when this pattern might be useful in the cloud); *Mechanics* (how the pattern works); an *Example* (which uses the Page of Photos sample application and Windows Azure); and finally a brief *Summary*. Also, many cross-chapter references are included to highlight where patterns overlap or can be used in tandem.

Although the *Example* section uses the Windows Azure platform, it is intended to be read as a core part of the chapter because a specific example of applying the pattern is discussed.

The book is intended to be vendor-neutral, with the exception that *Example* sections in pattern chapters necessarily use terminology and features specific to Windows Azure. Existing well-known names for concepts and patterns are used wherever possible. Some patterns and concepts did not have standard vendor-neutral names, so these are provided.

Building Page of Photos on Windows Azure

Each pattern chapter provides a general introduction to one cloud architecture pattern. After the general pattern is introduced, a specific use case with that pattern is described in more depth. This is intended to be a concrete example of applying that pattern to improve a cloud-native application. A single demonstration application called *Page of Photos* is used throughout the book.

The Page of Photos application, or PoP for short, is a simple web application that allows anyone to create an account and add photos to that account.

Each PoP account gets its own web address, which is the main web address followed by a folder name. For example, *http://www.pageofphotos.com/widaketi* displays photos under the folder name *widaketi*.

The PoP application was chosen because it is very simple to understand, while also allowing for enough complexity to illustrate the patterns without having sample application details get in the way.

This very basic introduction to PoP should get you started. Features are added to PoP in the *Example* section in each pattern chapter, always using Windows Azure capabilities, and always related to the general cloud pattern that is the focus of the chapter. By the end of the book, PoP will be a more complete, well-architected cloud-native application.

The PoP application was created as a concrete example for readers of this book and also as an exercise for double-checking some of the patterns. Look for it at *http://www.pageofphotos.com*.

Windows Azure is used for the PoP example, but the concepts apply as readily to Amazon Web Services and other cloud services platforms. I chose Windows Azure because that's where I have deep expertise and know it to be a rich and capable platform for cloud-native application development. It was a pragmatic choice.

Terminology

The book uses the terms *application* and *web application* broadly, even though *service*, *system*, and other terms may be just as applicable in some contexts. More specific terms are used as needed.

Conventions Used in This Book

The following typographical conventions are used in this book:

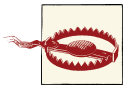
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

Constant width italic
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Cloud Architecture Patterns* by Bill Wilder (O'Reilly). Copyright 2012 Bill Wilder, 978-1-449-31977-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley

Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/cloud_architecture_patterns*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

This is a far better book than I could have possibly written myself because of the generous support of many talented people. I was thrilled that so many family members, friends, and professional colleagues (note: categories are not mutually exclusive!) were willing to spend their valuable time to help me create a better book. Roughly in order of appearance…

Joan Wortman (UX Specialist) was the first to review the earliest book drafts (which were painful to read). To my delight, Joan stayed with me, continuing to provide valuable, insightful comments though to the very last drafts. Joan was also really creative in brainstorming ideas for the illustrations in the book. Elizabeth O'Connor (majoring in Illustration at Mass College of Art) created the original versions of the beautiful illustrations in the book. Jason Haley was the second to review early (and still painful to

read) drafts. Later Jason was kind enough to sign on as the official technical editor, remarking at one point (with a straight face), "Oh, was that the same book?" I guess it got better over time. Rahul Rai (Microsoft) offered detailed technical feedback and suggestions, with insights relating to every area in the book. Nuno Godinho (Cloud Solution Architect – World Wide, Aditi) commented on early drafts and helped point out challenges with some confusing concepts. Michael Collier (Windows Azure National Architect, Neudesic) offered detailed comments and many suggestions in all chapters. Michael and Nuno are fellow Windows Azure MVPs. John Ahearn (a sublime entity) made every chapter in the book clearer and more pleasant to read, tirelessly reviewing chapters and providing detailed edits. John did not proofread the prior sentence, but if he did, I'm sure he would improve it. Richard Duggan is one of the smartest people I know, and also one of the funniest. I always looked forward to his comments since they were guaranteed to make the book better while making me laugh in the process. Mark Eisenberg (Fino Consulting) offered thought-provoking feedback that helped me see the topic more clearly and be more to the point. Jen Heney provided helpful comments and edits on the earliest chapters. Michael Stiefel (Reliable Software) provided pointed and insightful feedback that really challenged me to write a better book. Both Mark and Michael forced me to rethink my approach in multiple places. Edmond O'Connor (SS&C Technologies Inc.) offered many improvements where needed and confirmation where things were on the right track. Nazik Huq and George Babey have been helping me run the Boston Azure User Group for the past couple of years, and now their book comments have also helped me to write a better book. Also from the Boston Azure community is Nathan Pickett (KGS Buildings); Nate read the whole book, provided feedback on every chapter, and was one of the few who actually answered the annoying questions I posed in the text to reviewers. John Zablocki reviewed one of the chapters, as a last-minute request from me; John's feedback was both speedy and helpful. Don McNamara and William Gross (both from Geek Dinner) provided useful feedback, some good pushback, and even encouragement. Liam McNamara (a truly top-notch software professional, and my personal guide to the pubs of Dublin) read the whole manuscript late in the process and identified many (of my) errors and offered improved examples and clearer language. Will Wilder and Daniel Wilder proofread chapters and helped make sure the book made sense. Kevin Wilder and T.J. Wilder helped with data crunching to add context to the busy signal and network latency topics, proofreading, and assisted with writing the Page of Photos sample application. Many, many thanks to all of you for all of your valuable help, support, insights, and encouragement.

Special thanks to the team at O'Reilly, especially those I worked directly with: editor Rachel Roumeliotis (from inception to the end), production editor Holly Bauer, and copy editor Gillian McGarvey. Thanks also to the other staffers behind the scenes. And a special shout-out to Julie Lerman (who happens to live near the Long Trail in Vermont)

who changed my thinking about this book; originally I was thinking about a really short, self-published ebook, but Julie ended up introducing me to O'Reilly who liked my idea enough to sign me on. And here we are. By the way, the Preface for Julie's *Programming Entity Framework* book is a lot more entertaining than this one.

I know my Mom would be very proud of me for writing this book. She was always deeply interested in my software career and was always willing to listen to me babble on about the technological marvels I was creating. My Dad thinks it is pretty cool that I have written a book and is looking forward to seeing "his" name on the cover—finally, after all these years, naming a son after him has paid off (yes, I am Bill Jr.).

Most importantly of all, I am also deeply grateful to my wife Maura for encouraging me and making this possible. This book would simply not exist without her unflagging support.

CHAPTER 1

Scalability Primer

This primer explains scalability with an emphasis on the key differences between vertical and horizontal scaling.

Scaling is about allocating resources for an application and managing those resources efficiently to minimize contention. The user experience (UX) is negatively impacted when an application requires more resources than are available. The two primary approaches to scaling are *vertical scaling* and *horizontal scaling*. Vertical scaling is the simpler approach, though it is more limiting. Horizontal scaling is more complex, but can offer scales that far exceed those that are possible with vertical scaling. Horizontal scaling is the more cloud-native approach.

This chapter assumes we are scaling a distributed multi-tier web application, though the principles are also more generally applicable.

This chapter is not specific to the cloud except where explicitly stated.

Scalability Defined

The *scalability* of an application is a measure of the number of users it can effectively support at the same time. The point at which an application cannot handle additional users effectively is the limit of its scalability. Scalability reaches its limit when a critical hardware resource runs out, though scalability can sometimes be extended by providing additional hardware resources. The hardware resources needed by an application usually include CPU, memory, disk (capacity and throughput), and network bandwidth.

An application runs on multiple *nodes*, which have hardware resources. Application logic runs on *compute nodes* and data is stored on *data nodes*. There are other types of nodes, but these are the primary ones. A node might be part of a physical server (usually a virtual machine), a physical server, or even a cluster of servers, but the generic term *node* is useful when the underlying resource doesn't matter. Usually it doesn't matter.

In the public cloud, a compute node is most likely a virtual machine, while a data node provisioned through a cloud service is most likely a cluster of servers.

Application scale can be extended by providing additional hardware resources, as long as the application can effectively utilize those resources. The manner in which we add these resources defines which of two scaling approaches we take.

- To *vertically scale up* is to increase overall application capacity by increasing the resources within existing nodes.
- To *horizontally scale out* is to increase overall application capacity by adding nodes.

These scaling approaches are neither mutually exclusive nor all-or-nothing. Any application is capable of vertically scaling up, horizontally scaling out, neither, or both. For example, parts of an application might only vertically scale up, while other parts might also horizontally scale out.

Increasing Capacity of Roadways

Consider a roadway for automobile travel. If the roadway was unable to support the desired volume of traffic, we could improve matters in a number of possible ways. One improvement would be to upgrade the road materials ("the hardware") from a dirt road to pavement to support higher travel speeds. This is vertically scaling up; the cars and trucks ("the software") will be able to go faster. Alternatively, we could widen the road to multiple lanes. This is horizontally scaling out; more cars and trucks can drive in parallel. And of course we could both upgrade the road materials and add more lanes, combining scaling up with scaling out.

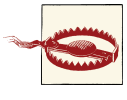
The horizontal and vertical scaling approaches apply to any resources, including both computation and data storage. Once either approach is implemented, scaling typically does not require changes to application logic. However, converting an application from vertical scaling to horizontal scaling usually requires significant changes.

Vertically Scaling Up

Vertically scaling up is also known simply as *vertical scaling* or *scaling up*. The main idea is to increase the capacity of individual nodes through hardware improvements. This might include adding memory, increasing the number of CPU cores, or other single-node changes.

Historically, this has been the most common approach to scaling due to its broad applicability, (often) low risk and complexity, and relatively modest cost of hardware improvements when compared to algorithmic improvements. Scaling up applies equally to standalone applications (such as desktop video editing, high-end video games, and mobile apps) and server-based applications (such as web applications, distributed multiplayer games, and mobile apps connected to backend services for heavy lifting such as for mapping and navigation).

Scaling up is limited by the utilizable capability of available hardware.



Vertical scaling can also refer to running multiple instances of software within a single machine. The architecture patterns in this book only consider vertical scaling as it relates to physical system resources.

There are no guarantees that sufficiently capable hardware exists or is affordable. And once you have the hardware, you are also limited by the extent to which your software is able to take advantage of the hardware.

Because hardware changes are involved, usually this approach involves downtime.

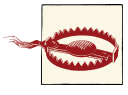
Horizontally Scaling Out

Horizontally scaling out, also known simply as *horizontal scaling* or *scaling out*, increases overall application capacity by adding entire nodes. Each additional node typically adds equivalent capacity, such as the same amount of memory and the same CPU.

The architectural challenges in vertical scaling differ from those in horizontal scaling; the focus shifts from maximizing the power of individual nodes to combining the power of many nodes. Horizontal scaling tends to be more complex than vertical scaling, and has a more fundamental influence on application architecture. Vertical scaling is often hardware- and infrastructure-focused—we "throw hardware at the problem"—whereas horizontal scaling is development- and architecture-focused. Depending on which scaling strategy is employed, the responsibility may fall to specialists in different departments, complicating matters for some companies.

Imperfect Terminology

To align with well-established industry practice, we use the term *horizontal scaling*, although *horizontal resource allocation* is more descriptive. The pattern is really about how resources are allocated and assembled; application scalability is simply one of many possible outcomes from the use of this pattern. Some specific benefits are listed later in the section that defines *cloud-native application*.



Parallel or multicore programming to fully leverage CPU cores within a single node should not be confused with using multiple nodes together. This book is concerned only with the latter.

Applications designed for horizontal scaling generally have nodes allocated to specific functions. For example, you may have web server nodes and invoicing service nodes. When we increase overall capacity by adding a node, we do so by adding a node for a specific function such as a web server or an invoicing service; we don't just "add a node" because node configuration is specific to the supported function.

When all the nodes supporting a specific function are configured identically—same hardware resources, same operating system, same function-specific software—we say these nodes are *homogeneous*.

Not all nodes in the application are homogeneous, just nodes within a function. While the web server nodes are homogeneous and the invoicing service nodes are homogeneous, the web server nodes don't need to be the same as the invoicing service nodes.

Horizontal scaling is more efficient with homogeneous nodes.

Horizontal scaling with homogeneous nodes is an important simplification. If the nodes are homogeneous, then basic round-robin load balancing works nicely, capacity planning is easier, and it is easier to write rules for auto-scaling. If nodes can be different, it becomes more complicated to efficiently distribute requests because more context is needed.

Within a specific type of node (such as a web server), nodes operate autonomously, independent of one another. One node does not need to communicate with other similar nodes in order to do its job. The degree to which nodes coordinate resources will limit efficiency.

An *autonomous node* does not know about other nodes of the same type.

Autonomy is important so that nodes can maintain their own efficiency regardless of what other nodes are doing.

Horizontal scaling is limited by the efficiency of added nodes. The best outcome is when each additional node adds the same incremental amount of usable capacity.

Describing Scalability

Descriptions of application scalability often simply reference the number of application users: "it scales to 100 users." A more rigorous description can be more meaningful. Consider the following definitions.

- *Concurrent users*: the number of users with activity within a specific time interval (such as ten minutes).
- *Response time*: the elapsed time between a user initiating a request (such as by clicking a button) and receiving the round-trip response.

Response time will vary somewhat from user to user. A meaningful statement can use the number of concurrent users and response time collectively as an indicator of overall system scalability.

Example: With 100 concurrent users, the response time will be under 2 seconds 60% of the time, 2-5 seconds 38% of the time, and 5 seconds or greater 2% of the time.

This is a good start, but not all application features have the same impact on system resources. A mix of features is being used: home page view, image upload, watching videos, searching, and so forth. Some features may be low impact (like a home page view), and others high impact (like image upload). An average usage mix may be 90% low impact and 10% high impact, but the mix may also vary over time.

An application may also have different types of users. For example, some users may be interacting directly with your web application through a web browser while others may be interacting indirectly through a native mobile phone application that accesses resources through programmatic interfaces (such as REST services). Other dimensions may be relevant, such as the user's location or the capabilities of the device they are using. Logging actual feature and resource usage will help improve this model over time.

The above measures can help in formulating scalability goals for your application or a more formal *service level agreement (SLA)* provided to paying users.

The Scale Unit

When scaling horizontally, we add homogeneous nodes, though possibly of multiple types. This is a predictable amount of capacity that ideally equates to specific application functionality that can be supported. For example, for every 100 users, we may need 2 web server nodes, one application service node, and 100 MB of disk space.

These combinations of resources that need to be scaled together are known as a *scale unit*. The scale unit is a useful modeling concept, such as with *Auto-Scaling Pattern* (Chapter 4).

For business analysis, scalability goals combined with resource needs organized by scale units are useful in developing cost projections.

Resource Contention Limits Scalability

Scalability problems are *resource contention* problems. It is not the number of concurrent users, per se, that limits scalability, but the competing demands on limited resources such as CPU, memory, and network bandwidth. There are not enough resources to go around for each user, and they have to be shared. This results in some users either being slowed down or blocked. These are referred to as *resource bottlenecks*.

For example, if we have high performing web and database servers, but a network connection that does not offer sufficient bandwidth to handle traffic needs, the resource bottleneck is the network connection. The application is limited by its inability to move data over the network quickly enough.

To scale beyond the current bottleneck, we need to either reduce demands on the resource or increase its capacity. To reduce a network bandwidth bottleneck, compressing the data before transmission may be a good approach.

Of course, eliminating the current bottleneck only reveals the next one. And so it goes.

Easing Resource Contention

There are two ways to ease contention for resources: don't use them up so fast, and add more of them.

An application can utilize resources more or less efficiently. Because scale is limited by resource contention, if you tune your application to more efficiently use resources that

could become bottlenecks, you will improve scalability. For example, tuning a database query can improve resource efficiency (not to mention performance). This efficiency allows us to process more transactions per second. Let's call these *algorithmic improvements*.

Efficiency often requires a trade-off. Compressing data will enable more efficient use of network bandwidth, but at the expense of CPU utilization and memory. Be sure that removing one resource bottleneck does not introduce another.

Another approach is to improve our hardware. We could upgrade our mobile device for more storage space. We could migrate our database to a more powerful server and benefit from faster CPU, more memory, and a larger and faster disk drive. *Moore's Law*, which simply states that computer hardware performance approximately doubles every couple of years, neatly captures why this is possible: hardware continuously improves. Let's call these *hardware improvements*.

Not only does hardware continuously improve year after year, but so does the price/performance ratio: our money goes further every year.

Algorithmic and hardware improvements can help us extend limits only to a certain point. With algorithmic improvements, we are limited by our cleverness in devising new ways to make better use of existing hardware resources. Algorithmic improvements may be expensive, risky, and time consuming to implement. Hardware improvements tend to be straightforward to implement, though ultimately will be limited by the capability of the hardware you are able to purchase. It could turn out that the hardware you need is prohibitively expensive or not available at all.

What happens when we can't think of any more algorithmic improvements and hardware improvements aren't coming fast enough? This depends on our scaling approach. We may be stuck if we are scaling vertically.

Scalability is a Business Concern

A speedy website is good for business. A Compuware analysis of 33 major retailers across 10 million home page views showed that a 1-second delay in page load time reduced conversions by 7%. Google observed that adding a 500-millisecond delay to page response time caused a 20% decrease in traffic, while Yahoo! observed a 400-millisecond delay caused a 5-9% decrease. Amazon.com reported that a 100-millisecond delay caused a 1% decrease in retail revenue. Google has started using website performance as a signal in its search engine rankings. (Sources for statistics are provided in Appendix A.)

There are many examples of companies that have improved customer satisfaction and increased revenue by speeding up their web applications, and even more examples of utter failure where a web application crashed because it simply was not equipped to handle an onslaught of traffic. Self-inflicted failures can happen, such as when large retailers advertise online sales for which they have not adequately prepared (this happens routinely on the Monday after Thanksgiving in the United States, a popular online shopping day known as Cyber Monday). Similar failures are associated with Super Bowl commercials.

Comparing Performance and Scalability

Discussions of web application speed (or "slowness") sometimes conflate two concepts: performance and scalability.

Performance is what an individual user experiences; scalability is how many users get to experience it.

Performance refers to the experience of an individual user. Servicing a single user request might involve data access, web server page generation, and the delivery of HTML and images over an Internet connection. Each of these steps takes time. Once the HTML and images are delivered to the user, a web browser still needs to assemble and render the page. The elapsed time necessary to complete all these steps limits overall performance. For interactive web applications, the most important of the performance-related measurements is response time.

Scalability refers to the number of users who have a positive experience. If the application sustains consistent performance for individual users as the number of concurrent users grows, it is scaling. For example, if the average response time is 1 second with 10 concurrent users, but the average response time climbs to 5 seconds with 100 concurrent users, then the application is not scaling. An application might scale well (handling many concurrent users with *consistent* performance), but not perform well (that consistent performance might be *slow*, whether with 100 concurrent users or just one). There is always a threshold at which scalability problems take hold; an application might perform well up to 100 concurrent users, and then degrade as the number of concurrent users increases beyond 100. In this last scenario, the application does not scale beyond 100 concurrent users.

Network latency can be an important performance factor influencing user experience. This is considered in more depth starting with *Network Latency Primer* (Chapter 11).

The Cloud-Native Application

This is a book for building *cloud-native applications*, so it is important that the term be defined clearly. First, we spell out the assumed characteristics of a *cloud platform*, which enables cloud-native applications. We then cover the expected characteristics of cloud-native applications that are built on such a platform using the patterns and ideas included in this book.

Cloud Platform Defined

The following characteristics of a cloud platform make cloud-native applications possible:

- Enabled by (the illusion of) infinite resources and limited by the maximum capacity of individual virtual machines, cloud scaling is horizontal.
- Enabled by a short-term resource rental model, cloud scaling releases resources as easily as they are added.
- Enabled by a metered pay-for-use model, cloud applications only pay for currently allocated resources and all usage costs are transparent.
- Enabled by self-service, on-demand, programmatic provisioning and releasing of resources, cloud scaling is automatable.
- Both enabled and constrained by multitenant services running on commodity hardware, cloud applications are optimized for cost rather than reliability; failure is routine, but downtime is rare.
- Enabled by a rich ecosystem of managed platform services such as for virtual machines, data storage, messaging, and networking, cloud application development is simplified.

While none of these are impossible outside the cloud, if they are all present at once, they are likely enabled by a cloud platform. In particular, Windows Azure and Amazon Web Services have all of these characteristics. Any significant cloud platform—public, private, or otherwise—will have most of these properties.

The patterns in this book apply to platforms with the above properties, though many will be useful on platforms with just some of these properties. For example, some private clouds may not have a metered pay-for-use mechanism, so pay-for-use may not literally apply. However, relevant patterns can still be used to drive down overall costs allowing the company to save money, even if the savings are not directly credited back to specific applications.

Where did these characteristics come from? There is published evidence that companies with a large web presence such as eBay, Facebook, and Yahoo! have internal clouds with some similar capabilities, though this evidence is not always as detailed as desired. The

best evidence comes from three of the largest players—Amazon, Google, and Microsoft—who have all used lessons learned from years of running their own internal high-capacity infrastructure to create public cloud platforms for other companies to use as a service.

These characteristics are leveraged repeatedly throughout the book.

Cloud-Native Application Defined

A *cloud-native application* is architected to take full advantage of cloud platforms. A cloud-native application is assumed to have the following properties, as applicable:

- Leverages cloud-platform services for reliable, scalable infrastructure. ("Let the platform do the hard stuff.")
- Uses non-blocking asynchronous communication in a loosely coupled architecture.
- Scales horizontally, adding resources as demand increases and releasing resources as demand decreases.
- Cost-optimizes to run efficiently, not wasting resources.
- Handles scaling events without downtime or user experience degradation.
- Handles transient failures without user experience degradation.
- Handles node failures without downtime.
- Uses geographical distribution to minimize network latency.
- Upgrades without downtime.
- Scales automatically using proactive and reactive actions.
- Monitors and manages application logs even as nodes come and go.

As these characteristics show, an application does not need to support millions of users to benefit from cloud-native patterns. Architecting an application using the patterns in this book will lead to a cloud-native application. Applications using these patterns should have advantages over applications that use cloud services without being cloud-native. For example, a cloud-native application should have higher availability, lower complexity, lower operational costs, better performance, and higher maximum scale.

Windows Azure and Amazon Web Services are full-featured public cloud platforms for running cloud-native applications. However, just because an application runs on Azure or Amazon does not make it cloud-native. Both platforms offer *Platform as a Service (PaaS)* features that definitely facilitate focusing on application logic for cloud-native applications, rather than plumbing. Both platforms also offer *Infrastructure as a Service*

(IaaS) features that allow a great deal of flexibility for running non-cloud-native applications. But using PaaS does not imply that the application is cloud-native, and using IaaS does not imply that it isn't. The architecture of your application and how it uses the platform is the decisive factor in whether or not it is cloud-native.

It is the application architecture that makes an application cloud-native, not the choice of platform.

A cloud-native application is not the best choice for every situation. It is usually most cost-effective to architect new applications to be cloud-native from the start. Significant (and costly) changes may be needed to convert a legacy application to being cloud-native, and the benefit may not be worth the cost. Not every application should be cloud-native, and many more cloud applications need not be 100% cloud-native. This is a business decision, guided by technical insight.

Patterns in this book can also benefit cloud applications that are not fully cloud-native.

Summary

Scalability impacts performance and efficiency impacts scalability. Two common scaling patterns are vertical and horizontal scaling. Vertical scaling is generally easier to implement, though it is more limiting than horizontal scaling. Cloud-native applications allocate resources horizontally, and scalability is only one benefit.

CHAPTER 2

Horizontally Scaling Compute Pattern

This fundamental pattern focuses on horizontally scaling compute nodes. Primary concerns are efficient utilization of cloud resources and operational efficiency.

The key to efficiently utilizing resources is stateless autonomous compute nodes. Stateless nodes do not imply a stateless application. Important state can be stored external to the nodes in a cloud cache or storage service, which for the web tier is usually done with the help of cookies. Services in the service tier typically do not use session state, so implementation is even easier: all required state is provided by the caller in each call.

The key to operations management is to lean on cloud services for automation to reduce complexity in deploying and managing homogeneous nodes.

Context

The Horizontal Scaling Compute Pattern effectively deals with the following challenges:

- Cost-efficient scaling of compute nodes is required, such as in the web tier or service tier.
- Application capacity requirements exceed (or may exceed after growth) the capacity of the largest available compute node.
- Application capacity requirements vary seasonally, monthly, weekly, or daily, or are subject to unpredictable spikes in usage.
- Application compute nodes require minimal downtime, including resilience in the event of hardware failure, system upgrades, and resource changes due to scaling.

This pattern is typically used in combination with the Node Termination Pattern (which covers concerns when releasing compute nodes) and the Auto-Scaling Pattern (which covers automation).

Cloud Significance

Public cloud platforms are optimized for horizontal scaling. Instantiating a single compute node (virtual machine) is as easy as instantiating 100. And with 100 nodes deployed, we can just as easily release 50 of them with a simple request to the cloud platform. The platform ensures that all nodes deploy with the same virtual machine image, offer services for node management, and provide load balancing as a service.

Impact

Availability, Cost Optimization, Scalability, User Experience

Mechanics

When a cloud-native application is ready to horizontally scale by adding or releasing compute nodes, this is achieved through the cloud platform management user interface, a scaling tool, or directly through the cloud platform management service. (The management user interface and any scaling tools ultimately also use cloud platform management service.)

The management service requires that a specific configuration is specified (one or more virtual machine images or an application image) and the number of desired nodes for each. If the number of desired compute nodes is larger than the current number, nodes are added. If the number of desired compute nodes is lower than the current number, nodes are released. The number of nodes in use (and commensurate costs) will vary over time according to needs, as shown in Figure 2-1.

The process is very simple. However, with nodes coming and going, care must be taken in managing user session state and maintaining operational efficiency.

It is also important to understand why we want an application with fluctuating resources rather than fixed resources. It is because reversible scaling saves us money.

Cloud Scaling is Reversible

Historically, scalability has been about *adding* capacity. While it has always been technically possible to reduce capacity, in practice it has been as uncommon as unicorn sightings. Rarely do we hear "hey everyone, the company time-reporting application is running great – let's come in this weekend and migrate it to less capable hardware and see what happens." This is the case for a couple of reasons.

It is difficult and time-consuming to ascertain the precise maximum resource requirements needed for an application. It is safer to overprovision. Further, once the hardware

is paid for, acquired, installed, and in use, there is little organizational pressure to fiddle with it. For example, if the company time-reporting application requires very little capacity during most of the week, but 20 times that capacity on Fridays, no one is trying to figure out a better use for the "extra" capacity that's available 6 days a week.

With cloud-native applications, it is far less risky and much simpler to exploit extra capacity; we just give it back to our cloud platform (and stop paying for it) until we need it again. And we can do this without touching a screwdriver.

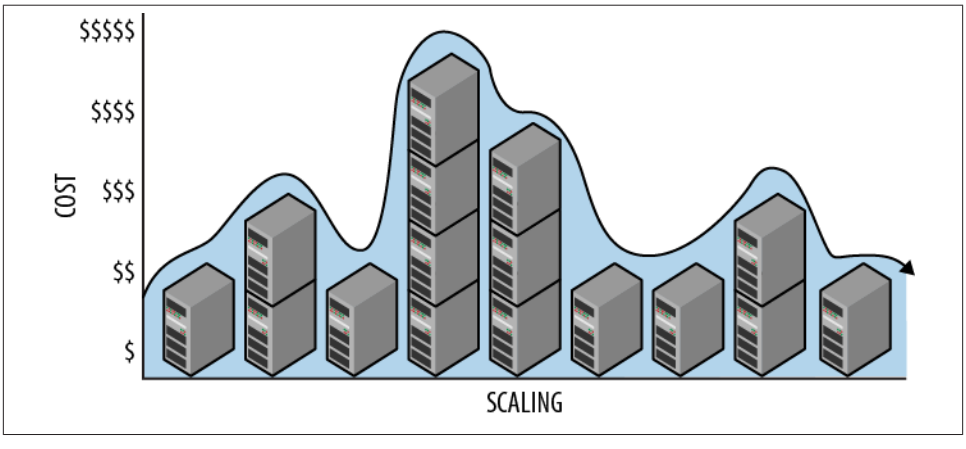


Figure 2-1. Cloud scaling is easily reversed. Costs vary in proportion to scale as scale varies over time.

Cloud resources are available on-demand for short-term rental as virtual machines and services. This model, which is as much a business innovation as a technical one, makes reversible scaling practical and important as a tool for cost minimization. We say reversible scaling is *elastic* because it can easily contract after being stretched.

Practical, reversible scaling helps optimize operational costs.

If our allocated resources exceed our needs, we can remove some of those resources. Similarly, if our allocated resources fall short of our needs, we can add resources to match our needs. We horizontally scale in either direction depending on the current resource needs. This minimizes costs because after releasing a resource, we do not pay for it beyond the current rental period.

Consider All Rental Options

The caveat "beyond the current rental period" is important. Rental periods in the cloud vary from instantaneous (delete a byte and you stop paying for its storage immediately) to increments of the wall clock (as with virtual machine rentals) to longer periods that may come with bulk (or long-term) purchasing. Bulk purchasing is an additional cost optimization not covered in this book. You, however, should not ignore it.

Consider a line-of-business application that is expected to be available only during normal business hours, in one time zone. Only 50 hours of availability are needed per week. Because there are 168 hours in a calendar week, we could save money by removing any excess compute nodes during the other 118 hours. For some applications, removing all compute nodes for certain time periods is acceptable and will maximize cost savings. Rarely used applications can be deployed on demand.

An application may be lightly used by relatively few people most of the time, but heavily used by tens of thousands of people during the last three business days of the month. We can adjust capacity accordingly, aligning cost to usage patterns: during most of the month two nodes are deployed, but for the last three business days of the month this is increased to ten.

The simplest mechanism for adjusting deployed capacity is through the cloud vendor's web-hosted management tool. For example, the number of deployed nodes is easily managed with a few clicks of the mouse in both the Windows Azure portal and the Amazon Web Services dashboard. In *Auto-Scaling Pattern* (Chapter 4) we examine additional approaches to making this more automated and dynamic.

Cloud scaling terminology

Previously in the book, we note that the terms *vertical scaling* and *scaling up* are synonyms, as are *horizontal scaling* and *scaling out*. Reversible scaling is so easy in the cloud that it is far more popular than in traditional environments. Among synonyms, it is valuable to prefer the more suitable terms. Because the terms scaling up and scaling out are biased towards *increasing* capacity, which does not reflect the flexibility that cloud-native applications exhibit, in this book the terms vertical and horizontal scaling are preferred.

The term *vertical scaling* is more neutral than *scaling up*, and *horizontal scaling* is more neutral than *scaling out*. The more neutral terms do not imply increase or decrease, just change. This is a more accurate depiction of cloud-native scaling.

For emphasis when describing specific scaling scenarios, the terms *vertically scaling up*, *vertically scaling down*, *horizontally scaling in*, and *horizontally scaling out* are sometimes used.

Managing Session State

Consider an application with two web server nodes supporting interactive users through a web browser. A first-time visitor adds an item to a shopping cart. Where is that shopping cart data stored? The answer to this simple question lies in how we manage session state.

When users interact with a web application, context is maintained as they navigate from page to page or interact with a single-page application. This context is known as *session state*. Examples of values stored in session state include security access tokens, the user's name, and shopping cart contents.

Depending on the application tier, the approach for session state will vary.

Session state varies by application tier

A web application is often divided into tiers, usually a web tier, a service tier, and a data tier. Each tier can consist of one or many nodes. The *web tier* runs web servers, is accessible to end users, and provides content to browsers and mobile devices. If we have more than one node in the web tier and a user visits our application from a web browser, which node will serve their request? We need a way to direct visiting users to one node or another. This is usually done using a *load balancer*. For the first page request of a new user session, the typical load balancer directs that user to a node using a round-robin algorithm to evenly balance the load. How to handle subsequent page requests in that same user session? This is tightly related to how we manage session state and is discussed in the following sections.

A *web service*, or simply *service*, provides functionality over the network using a standard network protocol such as HTTP. Common service styles include SOAP and REST, with SOAP being more popular within large enterprises and REST being more popular for services exposed publicly. Public cloud platforms favor the REST style.

The *service tier* in an application hosts services that implement business logic and provide business processing. This tier is accessible to the web tier and other service tier services, but not to users directly. The nodes in this tier are stateless.

The *data tier* holds business data in one or more types of persistent storage such as relational databases, NoSQL databases, and file storage (which we will learn later is called *blob storage*). Sometimes web browsers are given read-only access to certain types of storage in the data tier such as files (blobs), though this access typically does not extend to databases. Any updates to the data tier are either done within the service tier or managed through the service tier as illustrated in *Valet Key Pattern* (Chapter 13).

Sticky sessions in the web tier

Some web applications use *sticky sessions*, which assign each user to a specific web server node when they first visit. Once assigned, that node satisfies all of that user's page requests for the duration of the visit. This is supported in two places: the load balancer ensures that each user is directed to their assigned node, while the web server nodes store session state for users between page requests.

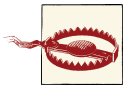
The benefits of sticky sessions are simplicity and convenience: it is easy to code and convenient to store users' session state in memory. However, when a user's session state is maintained on a specific node, that node is no longer stateless. That node is a *stateful* node.

The Amazon Web Services elastic load balancer supports sticky sessions, although the Windows Azure load balancer does not. It is possible to implement sticky sessions using Application Request Routing (ARR) on Internet Information Services (IIS) in Windows Azure.

Cloud-native applications do not need sticky session support.

Stateful node challenges

When stateful nodes hold the only copy of a user's session state, there are user experience challenges. If the node that is managing the sticky session state for a user goes away, that user's session state goes with it. This may force a user to log in again or cause the contents of a shopping cart to vanish.



A node holding the only copy of user session state is a single point of failure. If the node fails, that data is lost.

Sessions may also be unevenly distributed as node instances come and go. Suppose your web tier has two web server nodes, each with 1,000 active sessions. You add a third node to handle the expected spike in traffic during lunchtime. The typical load balancer randomly distributes new requests across all nodes. It will not have enough information to send new sessions to the newly added node until it also has 1,000 active sessions. It is effectively "catching up" to the other nodes in the rotation. Each of the 3 nodes will get

approximately one-third of the next 1,000 new sessions, resulting in an imbalance. This imbalance is resolved as older sessions complete, provided that the number of nodes remains stable. Overloaded nodes may result in a degraded user experience, while underutilized nodes are not operationally efficient. What to do?

Session state without stateful nodes

The cloud-native approach is to have session state without stateful nodes. A node can be kept stateless simply by avoiding storing user session state locally (on the node), but rather storing it externally. Even though session state will not be stored on individual nodes, session state does need to be stored somewhere.

Applications with a very small amount of session state may be able to store all of it in a web cookie. This avoids storing session state locally by eliminating all local session state; it is transmitted inside a cookie that is sent by the user's web browser along with page requests.

It gets interesting when a cookie is too small (or too inefficient) to store the session state. The cookie can still be used, but rather than storing all session state inside it, the cookie holds an application-generated session identifier that links to server-side session state; using the session identifier, session data can be retrieved and rehydrated at the beginning of each request and saved again at the end. Several ready-to-go data storage options are available in the cloud, such as NoSQL data stores, cloud storage, and distributed caches.

These approaches to managing session state allow the individual web nodes to remain autonomous and avoid the challenges of stateful nodes. Using a simple round-robin load balancing solution is sufficient (meaning even the load balancer doesn't need to know about session state). Of course, some of the responsibility for scalability is now shifted to the storage mechanism being used. These services are typically up for the task.

As an example, a distributed cache service can be used to externalize session state. The major public cloud platforms offer managed services for creating a distributed cache. In just a few minutes, you can provision a distributed cache and have it ready to use. You don't need to manage it, upgrade it, monitor it, or configure it; you simply turn it on and start using (and paying for) it.

Session state exists to provide continuity as users navigate from one web page to another. This need extends to public-facing web services that rely on session state for authentication and other context information. For example, a single-page web application may use AJAX to call REST services to grab some JSON data. Because they are user-accessible, these services are also in the web tier. All other services run in the service tier.

Stateless service nodes in the service tier

Web services in the service tier do not have public endpoints because they exist to support other internal parts of the application. Typically, they do not rely on any session information, but rather are completely stateless: all required state is provided by the caller in each call, including security information if needed. Sometimes internal web services do not authenticate callers because the cloud platform security prevents external callers from reaching them, so they can assume they are only being accessed by trusted subsystems within the application.

Other services in the service tier cannot be directly invoked. These are the processing services described in *Queue-Centric Workflow Pattern* (Chapter 3). These services pull their work directly from a queue.

No new state-related problems are introduced when stateless service nodes are used.

Managing Many Nodes

In any nontrivial cloud application, there will be multiple node types and multiple instances of each node type. The number of instances will fluctuate over time. Mixed deployments will be common if application upgrades are rolling upgrades, a few nodes at a time.

As compute nodes come and go, how do we keep track of them and manage them?

Efficient management enables horizontal scaling

Developing for the cloud means we need to establish a node image for each node type by defining what application code should be running. This is simply the code we think of as our application: PHP website code may be one node type for which we create an image, and a Java invoice processing service may be another.

To create an image with IaaS, we build a virtual machine image; with PaaS, we build a web application (or, more specifically, a Cloud Service on Windows Azure). Once a node image is established, the cloud platform will take care of deploying it to as many nodes as we specify, ensuring all of the nodes are essentially identical.

It is just as easy to deploy 2 identical nodes as is to deploy 200 identical nodes.

Your cloud platform of choice will also have a web-hosted management tool that allows you to view the current size and health of your deployed application.

Though you start with a pool of essentially identical nodes, you can change individual nodes afterwards. Avoid doing this as it will complicate operations at scale. For investigating issues, your cloud platform will have a way to take a node out of the load balancer rotation while you do some diagnostics; consider using that feature, then reimaging the node when you are done if you made changes. Homogeneity is your friend.

Capacity planning for large scale

Capacity planning is also different in the cloud. Non-cloud scenarios in big companies might have a hardware acquisition process that takes months, which makes ending up with too little capacity a big risk. In the cloud, where capacity is available on demand, capacity planning takes on a very different risk profile, and need not be so exacting. In fact, it often gives way to projections of operational expenses, rather than rigid capital investments and long planning cycles.

Cloud providers assume both the financial burden of over-provisioning and the reputation risk of under-provisioning that would destroy the illusion of infinite capacity. This amounts to an important simplification for customers; if you calculate wrong, and need more *or less* capacity than you planned, the cloud has you covered. It supports customer agility and capital preservation.

Are Cloud Resources Infinite?

We often hear that public cloud platforms offer the *illusion of infinite resources*. Obviously, resources are not *literally* infinite (infinite is rather a lot), but you can expect that any time you need more resources, they will be available (though not always instantly). This does not mean *each* resource has infinite capacity, just that you can request as many *instances* of the type of resource that you need.

This is why vertical scaling is limiting, and even more so in the cloud: an individual virtual machine or database instance has some maximum capacity, after which it cannot be increased. With horizontal scaling, if you need to go beyond that maximum capacity, you can do so by allocating an additional virtual machine or database instance. This, of course, introduces complexity of its own, but many of the patterns in this book help in taming that complexity.

Sizing virtual machines

A horizontal scaling approach supports increasing resources by adding as many node instances as we need. Cloud compute nodes are virtual machines. But not all virtual machines are the same. The cloud platforms offer many virtual machine configuration options with varying capabilities across the number of CPU cores, amount of memory, disk space, and available network bandwidth. The best virtual machine configuration depends on the application.

Determining the virtual machine configuration that is appropriate for your application is an important aspect of horizontal scaling. If the virtual machine is undersized, your application will not perform well and may experience failures. If the virtual machine is oversized, your application may not run cost-efficiently since larger virtual machines are more expensive.

Often, the optimal virtual machine size for a given application node type is the smallest virtual machine size that works well. Of course, there's no simple way to define "works well" across all applications. The optimal virtual machine size for nodes transcoding large videos may be larger than nodes sending invoices. How do you decide for your application? Testing. (And no excuses! The cloud makes testing with multiple virtual machine sizes more convenient than it has ever been.)

Sizing is done independently for each compute node type in your application because each type uses resources differently.

Failure is partial

A web tier with many nodes can temporarily lose a node to failure and still continue to function correctly. Unlike with single-node vertical scaling, the web server is not a single point of failure. (Of course, for this to be the case, you need at least two node instances running.) Relevant failure scenarios are discussed further in *Multitenancy and Commodity Hardware Primer* (Chapter 8) and *Node Failure Pattern* (Chapter 10).

Operational data collection

Operational data is generated on every running node in an application. Logging information directly to the local node is an efficient way to gather data, but is not sufficient. To make use of the logged data, it needs to be collected from individual nodes to be aggregated.

Collecting operational data can be challenging in a horizontally scaling environment since the number of nodes varies over time. Any system that automates gathering of log files from individual nodes needs to account for this, and care needs to be taken to ensure that logs are captured before nodes are released.

A third-party ecosystem of related products and open source projects exists to address these needs (and more), and your cloud platform may also provide services. Some Windows Azure platform services are described in the *Example* section.

Example: Building PoP on Windows Azure

The Page of Photos (PoP) application (which was described in the Preface and will be used as an example throughout the book) is designed to scale horizontally throughout. The web tier of this application is discussed here. Data storage and other facets will be discussed in other chapters.

Compute node and *virtual machine* are general industry terms. The equivalent Windows Azure-specific term for node is *role instance*, or *web role instance* or *worker role instance* if more precision is needed. Windows Azure role instances are running on virtual machines, so referring to role instances as virtual machines is redundant. Windows Azure terminology is used in the remainder of this section.

Web Tier

The web tier for PoP is implemented using ASP.NET MVC. Using a *web role* is the most natural way to support this. Web roles are a Windows Azure service for providing automated, managed virtual machines running Windows Server and Internet Information Services (IIS). Windows Azure automatically creates all the requested role instances and deploys your application to them; you only provide your application and some configuration settings. Windows Azure also manages your running role instances, monitors hardware and software health (and initiates recovery actions as warranted), patches the operating system on your role instances as needed, and other useful services.

Your application and configuration settings effectively form a template that can be applied to as many web role instances as required. Your effort is the same if you deploy 2 role instances or 20; Windows Azure does all the work.

It is instructive to consider the infrastructure management we no longer worry about with a web role: configuring routers and load balancers; installing and patching operating systems; upgrading to newer operating systems; monitoring hardware for failures (and recovering); and more.

Cloud Services Are Still Flexible

While Windows Azure Cloud Services are designed to shield you from infrastructure management so you can focus on simply building your application, you still have flexibility for advanced configuration if you need it. For example, using *Startup Tasks*, you can install additional Windows Services, configure IIS or Windows, and run custom installation scripts. For the most part, if an administrator can do it on Windows, you can do it on Windows Azure, though the more cloud-native your application is, the more likely things will "just work" without needing complex custom configuration. One advanced configuration possibility is to enable Application Request Routing (ARR) in IIS in order to support sticky sessions.

Stateless Role Instances (or Nodes)

As of this writing, the Windows Azure load balancer supports round robin delivery of web requests to the web role instances; there is no support for sticky sessions. Of course,

this is fine because we are demonstrating cloud-native patterns and we want our horizontally scalable web tier to consist of stateless, autonomous nodes for maximum flexibility. Because all web role instances for an application are interchangeable, the load balancer can also be stateless, as is the case in Windows Azure.

As described earlier, browser cookies can be used to store a session identifier linked to session data. In Windows Azure some of the storage options include SQL Azure (relational database), Windows Azure Table Storage (a wide-column NoSQL data store), Windows Azure Blob Storage (file/object store), and the Windows Azure distributed caching service. Because PoP is an ASP.NET application, we opt to use the Session State Provider for Windows Azure Caching, and the programming model that uses the familiar Session object abstraction while still being cloud-native with stateless, autonomous nodes. This allows PoP to benefit from a scalable and reliable caching solution provided by Windows Azure as a service.

Service Tier

PoP features a separate service tier so that the web tier can focus on page rendering and user interaction. The service tier in PoP includes services that process user input in the background.

Is a Separate Tier Necessary?

For PoP to be architected appropriately for its modest success, the service tier makes sense. Don't forget all the successful practices we've been using outside of the cloud, such as Service Oriented Architecture (SOA) techniques. This book is focused on architecture patterns that have a unique impact on cloud-native applications, but so many other practices of great value are not discussed directly, though SOA can be extremely valuable when developing applications for the cloud.

The PoP service tier will be hosted in *worker roles*, which are similar to web roles, though with a different emphasis. The worker role instances do not start the IIS service and

instances are not added to the load balancer by default. The worker role is ideal for application tiers that do not have interfaces to the outside world. Horizontal scaling works smoothly with the service tier; refer to *Queue-Centric Workflow Pattern* (Chapter 3) for details on its inner workings.

Loose Coupling Adds Implementation Flexibility

The web and service tiers can be implemented with different technologies. The web tier might use PHP, ASP.NET MVC, or other front-end solutions. The service tier might use Java, Python, Node.js, F#, C#, C++, and so forth. This adds flexibility when teams working on different application tiers have different skills.

Operational Logs and Metrics

Managing operational data is another challenge encountered when horizontally scaling out to many role instances. Operational data is generated during the process of operating your application, but is not usually considered part of the business data collected by the application itself.

Examples of operational data sources:

- Logs from IIS or other web servers
- Windows Event Log
- Performance Counters
- Debug messages output from your application
- Custom logs generated from your application

Collecting log data from so many instances can be daunting. The Windows Azure Diagnostics (WAD) Monitor is a platform service that can be used to gather data from all of your role instances and store it centrally in a single Windows Azure Storage Account. Once the data is gathered, analysis and reporting becomes possible.

Another source of operational data is the Windows Azure Storage Analytics feature that includes metrics and access logs from Windows Azure Storage Blobs, Tables, and Queues.

Examples of analytics data:

- Number of times a specific blob or blob container was accessed
- The most frequently accessed blob containers
- Number of anonymous requests originating from a given IP Address range
- Request durations

- Requests per hour that are hitting blob, table, or queue services
- Amount of space blobs are consuming

Analytics data is not collected by the WAD, so not automatically combined, but is available for analysis. For example, an application could combine blob storage access logs with IIS logs to create a more comprehensive picture of user activity.

Both Windows Azure Diagnostics and Windows Azure Storage Analytics support an application-defined data retention policy. This allows applications to easily limit the size and age of operational data because the cloud platform will handle purging details.

There are general purpose reporting tools in the Windows Azure Platform that might be useful for analyzing log and metric data. The Hadoop on Azure service is described in *MapReduce Pattern* (Chapter 6). The Windows Azure SQL Reporting service may also be useful.

Related Chapters

- *Queue-Centric Workflow Pattern* (Chapter 3)
- *Auto-Scaling Pattern* (Chapter 4)
- *MapReduce Pattern* (Chapter 6)
- *Database Sharding Pattern* (Chapter 7)
- *Multitenancy and Commodity Hardware Primer* (Chapter 8)
- *Node Failure Pattern* (Chapter 10)

Summary

The Horizontal Scaling Compute Pattern architecturally aligns applications with the most cloud-native approach for resource allocation. There are many potential benefits for applications, including high scalability, high availability, and cost optimization, all while maintaining a robust user experience. User state management should be handled without sticky sessions in the web tier. Keeping nodes stateless makes them interchangeable so that we can add nodes at any time without getting the workloads out of balance and can lose nodes without losing customer state.

CHAPTER 3
Queue-Centric Workflow Pattern

This essential pattern for loose coupling focuses on asynchronous delivery of command requests sent from the user interface to a back-end service for processing. This pattern is a subset of the CQRS pattern.

The pattern is used to allow interactive users to make updates through the web tier without slowing down the web server. It is especially useful for processing updates that are time consuming, resource intensive, or depend on remote services that may not always be available. For example, a social media site may benefit from this pattern when handling status updates, photo uploads, video uploads, or sending email.

The pattern is used in response to an update request from an interactive user. This is first handled by user interface code (in the web tier) that creates a message describing work needing to be done to satisfy the update request. This message is added to a queue. At some future time, a service on another node (running in the service tier) removes messages from the queue and does the needed work. Messages flow only in one direction, from the web tier, onto the queue, and into the service tier. This pattern does not specify how (or if) the user is informed of progress.

This is an asynchronous model, as the sender does not wait around for a response. In fact, no response is directly available. (In programming parlance, the return value is *void*.) It helps the user interface maintain consistently fast response times.

The web tier does not use this pattern for read-only page view requests; this pattern is for making updates.

Context

The Queue-Centric Workflow Pattern is effective in dealing with the following challenges:

- Application is decoupled across tiers, though the tiers still need to collaborate
- Application needs to guarantee *at-least-once processing* of messages across tiers
- A consistently responsive user experience is expected in the user interface tier, even though dependent processing happens in other tiers
- A consistently responsive user experience is expected in the user interface tier, even though third-party services are accessed during processing

This pattern is equally applicable to web applications and mobile applications that access the same functionality through web services. Any application serving interactive users is a candidate.

Cloud Significance

By using cloud services, the infrastructure aspects of this pattern are generally straightforward to implement. They can be far more complex outside the cloud. Reliable queues are available as a cloud service.

Storage of intermediate data is also simplified using cloud services. Cloud services are available for storage, NoSQL databases, and relational databases.

Impact

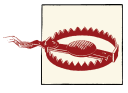
Availability, Reliability, Scalability, User Experience

Mechanics

The Queue-Centric Workflow Pattern is used in web applications to decouple communication between the web tier (which implements the user interface) and the service tier (where business processing happens).

Applications that do not use a pattern like this typically respond to a web page request by having user interface code call directly into the service tier. This approach is simple, but there are challenges in a distributed system. One challenge is that all service calls must complete before a web request is completed. This model also requires that the scalability and availability of the service tier meet or exceed that of the web tier, which can be tenuous with third-party services. A service tier that is unreliable or slow can ruin the user experience in the web tier and can negatively impact scalability.

The solution is to communicate asynchronously. The web tier sends *commands* to the service tier, where a *command* is a request to do something. Examples of commands include: create new user account, add photo, update status (such as on Twitter or Facebook), reserve hotel room, and cancel order.



The term asynchronous can apply to different aspects of application implementation. User interface code running in the web tier may invoke services asynchronously. This enables work to be done in parallel, potentially speeding up processing of that user request. Once all asynchronous services calls complete, the user request can be satisfied. This handy coding tactic should not be confused with this pattern.

Commands are sent in the form of messages over a queue. A queue is a simple data structure with two fundamental operations: *add* and *remove*. The behavior that makes it a queue is that the remove operation returns the message that has been in the queue the longest. Sometimes this is referred to as FIFO ordering: first in, first out. Invoking the add operation is commonly referred to as *enqueuing* and invoking the delete operation is *dequeuing*.

In the simplest (and most common) scenarios, the pattern is trivial: the sender adds command messages to the queue (*enqueues* messages), and a receiver removes those command messages from the queue (*dequeues* messages) and processes them. This is illustrated in Figure 3-1. (We'll see later that the programming model for removing messages from the queue is more involved than a simple dequeue.)

The sender and receiver are said to be *loosely coupled*. They communicate only through messages on a queue. This pattern allows the sender and receiver to operate at different paces or schedules; the receiver does not even need to be running when the sender adds a message to the queue. Neither one knows anything about the implementation of the other, though both sides do need to agree on which queue instance they will use, and on the structure of the command message that passes through the queue from sender to receiver.

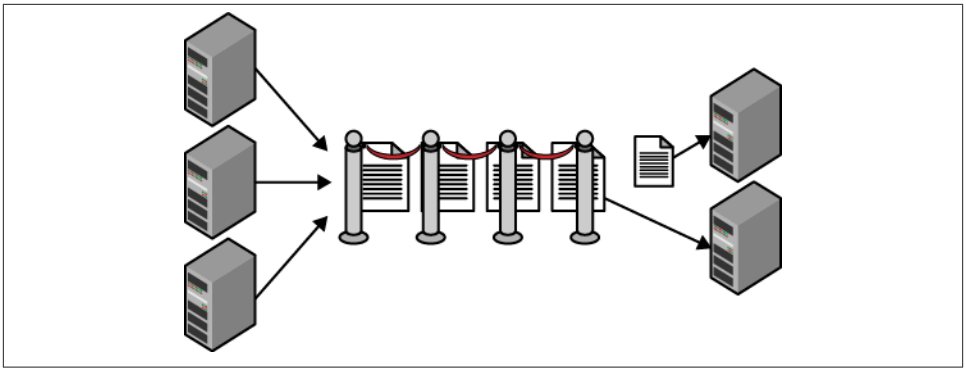


Figure 3-1. The web tier adds messages to the queue. The service tier removes and processes messages from the queue. The number of command messages in the queue fluctuates, providing a buffer so that the web tier can offload work quickly, while never overwhelming the service tier. The service tier can take its time, only processing new messages when it has available resources.

The sender need not be a web user interface; it could also be a native mobile application, for example, communicating through web services (as with a REST API). There could also be multiple senders and multiple receivers. The pattern still works.

The rest of this pattern description is concerned with guarding against failure scenarios and handling user experience concerns.

Queues are Reliable

The workflow involves the sender adding a message to a queue that is removed at some point by the receiver. Are we sure it will get there?

It is important to emphasize that the cloud queue service provides a *reliable queue*. The "reliable" claim stems primarily from two sources: durability of the data, and high throughput (at least hundreds of interactions per second).

The queue achieves data durability the same way that other cloud storage services do: by storing each byte entrusted to the service in triplicate (across three disk nodes) to overcome risks from hardware failure.

The queue itself is reliable and will not lose our data, but this pattern is not designed to shield our application from all failures. Rather, the pattern requires that our application implement specific behaviors to respond successfully to failure scenarios.

Programming Model for Receiver

When implementing the receiver, the programming model for using the reliable queue service sometimes surprises developers, as it is slightly more complicated than for a basic queue:

1. Get the next available message from the queue
2. Process the message
3. Delete the message from the queue

The implementation first *dequeues* the message, and then later *deletes* the message. Why the two-phase removal? This is to ensure *at-least-once processing*.

Invisibility window and at-least-once processing

Processing a command request involves getting a message from the queue, understanding the message contents, and carrying out the requested command accordingly. The details for this are specific to the application. If everything goes as planned, deleting the message from the queue is the last step. Only at that point is the command completely processed.

But everything does not always go as planned. For example, there might be a failure that is outside the control of your application code. These types of failures can happen for a number of reasons, but the easiest to understand is a hardware failure. If the hardware you are using fails out from under you, your process will be stopped, no matter where it is in its life cycle. Failure can occur if the cloud platform shuts down a running node because the auto-scaling logic decided it wasn't needed. Or, your node may be rebooted.

Refer to *Node Failure Pattern* (Chapter 10) for more scenarios that make use of this pattern to recover from interruptions.

Regardless of the reason for the failure, your processing has been interrupted and needs to recover. How does it do that?

When a message is dequeued, it is not removed entirely from the queue, but is instead hidden. The message is hidden for a specified amount of time (the duration is specified during the dequeue operation, and can be increased later). We call this period the *invisibility window*. When a message is within its invisibility window, it is not available for dequeuing.

Invisibility Window Nuances

During a message's invisibility window, there is usually exactly one copy of the message being processed. There are a couple of edge cases where this might not be true. One edge case is when the code processing the message has not finished, but the invisibility window lapses, and another copy of the same message gets dequeued. At this point, there are two active copies of that message being processed. If this happens, it may be due to a bug in your code. Instead of exceeding the invisibility window, you should inform the queue that you are still working on this message and increase its invisibility window to allow sufficient time with exclusive access. (See also the discussion of poison messages below.) However, as you will learn from the CAP Theorem discussion in *Eventual Consistency Primer* (Chapter 5), this may not always be possible in a distributed system due to partitioning. Though rare, the possibility should be accounted for.

An edge case can also occur with reliable queues that are *eventually consistent* (refer to Chapter 5 for more context). The bottom line here is that if two requests for the next queue item happen at nearly the same time, in rare cases, the queuing system may issue the same message in response to both requests. Amazon's Scalable Storage Service (S3) is eventually consistent and the documentation warns of this possibility. Both Windows Azure Storage Queues and Windows Azure ServiceBus Queues are *immediately consistent*, so this edge case does not apply.

The invisibility window comes into play only when processing takes longer than is allowed. The automatic reappearance of messages on the queue is one key to overcoming failures and is responsible for the *at-least-once* part of this *at-least-once processing* model. Any message not fully processed the first time it is dequeued will have another chance. The code keeps trying until processing completes (or we give up, as explained in the poison message handling section later).

Any message that is dequeued a second time may have been partially processed the first time. This can cause problems if not guarded against.

Idempotent processing for repeat messages

An *idempotent* operation is one that can be repeated such that any number of successful operations is indistinguishable from a single successful operation. For example, according to the HTTP specification, the HTTP verbs PUT, GET, and DELETE are all idempotent operations: we can DELETE a specific resource once or 100 times and the end result is equivalent; (assuming success) the resource is gone.

Some operations are considered *naturally idempotent*, such as HTTP DELETE, where idempotency essentially comes for free. A multistep financial transaction involving withdrawing money from one account and depositing it into another can be made to be idempotent, but it is definitely not naturally idempotent. Some cases are more difficult than others.

Idempotence Is Business Equivalent

No matter how many times it partially or fully completes, an idempotent process has an equivalent outcome, as long as the last instance completes successfully. Note that equivalent outcome means *business equivalence*, not *technical equivalence*. It is fine that application logs contain remnants of multiple attempts, for example, as long as the result is indistinguishable to a business user.

Cloud queue services keep track of how many times a message has been dequeued. Any time a message is dequeued, the queue service provides this value along with the message. We call this the *dequeue count*. The first time a message is dequeued, this value is one. By checking this value, application code can tell whether this is the first processing attempt or a repeat attempt.

Application logic can be streamlined for first-time processing, but for repeat attempts some additional logic may be needed to support idempotency.

Long Processing Tasks

Some cloud queue services support updating a message on the queue. For multi-step processes, as each step is completed, it can be handy to update the message on the queue with an indicator of the last completed step. As a simple example, you can design the message object so that it includes a `LastCompletedStep` field, which your application can use to track progress. If processing is interrupted, the updated `LastCompletedStep` field will be returned the next time the message is dequeued; message processing can resume with the `LastCompletedStep` step rather than starting from the beginning.

Consider a command to create a new user account based on a user-provided email address and the message dequeue count is two. Proper processing needs to consider the possibility that some (or all) of the processing work has been done previously and so needs to act smartly. Exactly how to "act smartly" will vary from application to application.

Simpler scenarios may not require any specific idempotency support. Consider the sending of a confirmation email. Because failure events are rare, and there is little harm in the occasional duplicate email, just sending the email every time may be sufficient.

Options for Idempotent Processing

Idempotent handling is easy to prescribe but not always easy to implement. More advanced approaches to idempotency are required for more complex idempotency scenarios, such as a multi-step financial transaction or an operation that spans multiple data stores.

A *database transaction* is sometimes very useful in the cloud: all operations succeed or they all fail. However, often a database transaction is not practical in a cloud application either because the supported transaction scope is too narrow (for example, a transaction cannot span partitions (or shards) in a NoSQL database or a sharded relational database) or the data is being written to multiple stores across which distributed transactions are simply not supported (for example, it is not possible to wrap a transaction around changes that span a relational database and blob storage).

A *compensating transaction*, where we reverse the net effect of a prior attempt, is one tool in our idempotency toolbox. Another is *event sourcing*, which is briefly mentioned in the context of CQRS in this chapter, and can sometimes provide a robust model for dealing with complex cases.

Idempotent handling is the correct first step in dealing with repeat messages. If the message repeats excessively, beyond an application-defined threshold, it should be treated as a poison message.

Poison messages handling for excessive repeats

Some messages cannot be processed successfully due to the contents of the message. These are known as poison messages.

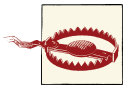
Consider a message containing a command to create a new user account based on a user-provided email address. If it turns out that the email address is already in use, your application should still process the message successfully, but not create a new user account. This is not a poison message.

But if the email address field contained a 10,000-character string and this is a scenario unanticipated in your application code, it may result in a crash. This is a poison message.

If our application crashes while processing a message, eventually its invisibility window will lapse, and the message will appear on the queue again for another attempt. The need for idempotent handling for that scenario is explained in the previous section. When dealing with a poison message, the idempotent handling will never terminate.

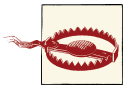
Two decisions need to be made around poison messages: how to detect one, and what to do with it.

As a message is dequeued, cloud queuing services offer a dequeue count that can be examined to determine if this is the first attempt at processing. This is the same value used for detecting repeats for purposes of idempotent handling. Your poison message detection logic must include a rule that considers any message that keeps reappearing to be treated as a poison message when it shows up the *Nth* time. Choosing a value for *N* is a business decision that balances the cost of wastefully processing a poison message with the risk of not processing a valid message. In practice, interruptions to execution tend to be infrequent, so take that into account when setting up your poison message strategy. If processing is resource intensive, perhaps taking 60 minutes, you may not want to retry any failed processes; so for $N > 1$, the message is treated as a poison message. It is common, however, to retry from once to a few times, depending on circumstances.



Correct poison message detection has some nuances. For example, having selected $N=3$ to trigger poison message handling, the application code needs to check for a dequeue count of at least 3, not exactly 3. A system interruption could have occurred during the time after detecting the dequeue count is 3, but before removing the message from the main queue.

Once a poison message has been identified, deciding how to deal with it is another business decision. If it is desirable to have a human review the poison messages to consider how to improve handling, then one approach is to use what is known as a *dead letter queue*, a place for storing messages that cannot be processed normally. Some queuing systems have built-in support for a dead letter queue, but it is not hard to roll your own. For low importance messages, you may even consider deleting them outright. The key point is to remove poison messages from the main processing queue as soon as the application detects them.



Unless we guard against the poison message scenario, a poison message will last a long time on our queue and waste processing resources. In extreme cases, with many active poison messages, all processing resources could end up dedicated to poison message processing!

A dequeue count greater than one does not necessarily mean a poison message is present. The value is a dequeue count, not a poison message count.

User Experience Implications

This pattern deals with asynchronous processing, repeated processing, and failed requests. All of these have user experience implications.

Handling asynchronous processing in a user interface can be tricky and application specific. We want the human-facing user interface to be responsive, so instead of performing lengthy work while the user waits, we queue up a command request for that work. This allows the user interface to return as soon as possible to the user (improving user experience) and allows the web server tier to remain focused on serving web pages (enhancing scalability).

The flip side here is that you now need your users to understand that even though the system has acknowledged their action (and a command posted), processing of that action was not immediately completed. There are a number of approaches to this.

In some cases, users cannot readily tell if the action completed, so special action is not required.

In cases where the user wants to be notified when their action will be completed, an email upon completion might do the trick. This is common in ecommerce scenarios where "your order has shipped" and other notifications of progress are common.

Sometimes users will prefer to wait around while a task completes. This requires either that the user interface layer polls the service tier until the task completes or the service tier proactively notifies the user interface layer. The proactive notification can be implemented using *long polling*. In long polling, the web client creates an HTTP connection to the server, but the server intentionally does not respond until it has an answer.

Ready-made implementations of the long polling (also known as *Comet*) technique are available. Examples include: SignalR for ASP.NET and Socket.IO for Node.js. These libraries will take advantage of HTML5 Web Sockets if available.

Using the long polling technique is different than having the original (time-consuming) action done "inline" from the web tier. Blocking at the web tier until the action is complete would hurt scalability. This approach still allows for the time-consuming work to be done in the service tier.

Is This the Same as CQRS?

Readers familiar with the Command Query Responsibility Segregation (CQRS) Pattern may wonder if the Queue-Centric Workflow (QCW) Pattern is really the same pattern. While similar, they are not the same. CQW is only a stepping-stone to CQRS. Understanding some of the differences and overlap will help you avoid confusing them.

The defining characteristic of CQRS is the use of two distinct models: one for writing (the *write model*), and one for reading (the *read model*). Two key terms are *command* and *query*. A command is a request to make an update via the write model. A query is a request for information from the read model. Serving commands and queries are distinct activities (the "responsibility segregation"); you would never issue a command and expect data results as a return value. Even though the read model and write model may be based on the same underlying data, they surface different data models.

QCW focuses on the flow of commands to the write model, while only alluding to the read model such as in support of long polling from the user interface. In this regard, it is consistent with CQRS, though not complete since QCW does not fully articulate the read model. A command in QCW is the same as a command in CQRS.

A full CQRS treatment would also consider *event sourcing* and Domain Driven Design (DDD). With event sourcing, as commands result in system state changes, resulting change *events* are captured and stored individually rather than simply reflecting the change in the master data. For example, an *address changed* event would hold the new address information rather than just overwrite a single address field in a database. The result is a chronological history that can be replayed to arrive at current state (or any state along the way). Using event sourcing may simplify handling idempotent operations. DDD is a technology-agnostic methodology to understand business context. Neither event sourcing nor DDD are required with CQRS, but they are frequently used together.

Scaling Tiers Independently

The queue length and the time messages spend in the queue are useful environmental signals for auto-scaling. The cloud queue services make these key metrics readily available. A growing queue may indicate the need to increase capacity in the service tier, for example. Note that the signals might indicate that only one tier or one specific processing service needs to be scaled. This concern-independent scaling helps to optimize for cost and efficiency.

At very high scale, the queue itself could become a bottleneck requiring multiple queue instances. This does not change the core pattern.

Example: Building PoP on Windows Azure

The Page of Photos (PoP) application (which was described in the Preface and will be used as an example throughout the book) uses Chapter 3 to handle ingestion of new photos into the system.

Two application tiers within PoP collaborate. The user interface on the web tier is responsible for facilitating photo uploads for logged-in users and enqueuing command messages. The service tier is responsible for dequeuing and processing command messages.

User Interface Tier

The PoP user interface running in the web tier consists of ASP.NET MVC code running on a variable number of web role instances. User authentication (logging in) is handled here, and authenticated users are allowed to upload photos.

The location of the photo being processed, a plain-text description of the photo, and the PoP user's account identifier are stored in a message object, which is then enqueued. With PoP, the photo being processed is assumed to already have been uploaded to blob storage (using *Valet Key Pattern* (Chapter 13), when possible) and stored in Windows Azure Storage as a blob; only the reference to this blob (a URL) is stored in the message object that is enqueued. And while this particular process operates on an external resource (the photo stored as a blob), that does not imply that external resources are needed in order for this pattern to be of value. It is also common for all of the required data to be entirely contained within the message object.

Regardless of how many web role instances are running, they all submit their messages to the same Windows Azure Queue.

PoP uses the Windows Azure Storage Queue service, but Windows Azure also offers a ServiceBus Queue service. The two services share many characteristics and either is an excellent choice for PoP.

The same message queue is used by both sender and receiver:

- Example message queue name: *http://popuploads.queue.core.windows.net*

Fields included in the message that goes in the queue are similar to the following:

- Location of uploaded photo (in blob storage): *http://popuploads.blob.core.windows.net/publicphotos/william.jpg*
- Authenticated account identifier from the *emailaddress claim*, as described in *Multisite Deployment Pattern* (Chapter 15): *kd1hn@example.com*
- `LastCompletedStep`: 0

Note that the image is not part of the message that goes on the queue, but rather a reference to the image. The practical reason for this is the queue does not allow messages to be larger than a certain size (64 KB as of this writing). The more philosophical reason is that blob storage is the "right" place to store uploaded images on Windows Azure.

Service Tier

PoP services are running on a variable number of worker role instances in the service tier. C# code in these services is written to constantly check the queue for new messages. Once a message becomes available on the queue, it is removed and processed.

There will be times when no messages are available on the queue. In such cases, any dequeue attempt returns immediately. It is important to avoid code hammering the queue service in a tight loop as every queue operation will cost a tiny amount of money. Be sure to add an appropriate delay.

Watch Out for Money Leaks!

As of this writing, ten million Windows Azure Storage operations will cost $1.00. Considering that this fee is charged for dequeue requests even if there is no message waiting, how expensive is that? Attempting to dequeue from an empty queue at the rate of 500 requests per second, every 200 seconds would cost one penny, and every day would add up to $4.32. Of course, you will never want to do this.

Be sure to code in a delay of at least a few seconds after each unsuccessful dequeue attempt to prevent a *money leak*, a cloud platform expense that adds no business value. Just like memory leaks that, unchecked, can bring down your application, money leaks can degrade the cost efficiency of your cloud applications. Also consider variable delay techniques, similar to those described in *Busy Signal Pattern* (Chapter 9), which offer an even more sophisticated approach to battling money leaks, while also further explaining that you risk being throttled by the queue service for being hyperactive.

Each message represents a new photo upload waiting to be integrated with the rest of the PoP site. A few steps are involved: a thumbnail is created, any geotagging data is extracted, and then user account data is updated to include the new photo on their public page. After each step, the message is updated back in blob storage with an updated value for `LastCompletedStep`.

PoP thumbnail creation is idempotent. This is important because if we are careless, we could end up with orphaned image files littering our blob storage account. To handle idempotent thumbnail creation, PoP chose to make the thumbnail filename deterministic by deriving it from the filename of the full-sized photo. All uploaded photos are issued a unique system-generated name such as *william.jpg*. The name of the thumbnail is derived from this by appending "_thumb" to the root filename resulting in *william_thumb.jpg*. This simple approach will ensure that we always end up with a single thumbnail in blob storage. Note that while the results are not *identical* to having successfully generated the thumbnail the first time through—the time stamp on the file will be different, for example—we can safely conclude that the results are *equivalent*.

PoP has a business rule that states any message coming from the *popuploads* queue with a dequeue count of 3 or more is considered poison. When a poison message is detected, PoP sends an email informing the user who submitted the photo that it has been rejected as invalid and has been deleted.

If an application does not deal proactively with poison messages, the Windows Azure Queue service will delete them from the queue after seven days.

Synopsis of Changes to Page of Photos System

In order to process new photo uploads:

- The photo is stored in a public blob container created for that purpose.
- A message containing relevant data about the newly created photo is enqueued. This is done from the web tier (from a web role).
- A worker role in the service tier monitors the queue for messages available for processing, processing them as available.

After the message processing has completed, the at-rest state of PoP includes:

- Original photo is stored as a blob.
- Generated thumbnail is stored as a blob.
- Metadata about the photo is stored in a Windows Azure SQL Database (discussed in *Database Sharding Pattern* (Chapter 7)).

Related Chapters

- *Scalability Primer* (Chapter 1)

- *Horizontally Scaling Compute Pattern* (Chapter 2)
- *Auto-Scaling Pattern* (Chapter 4)
- *Eventual Consistency Primer* (Chapter 5)
- *Node Failure Pattern* (Chapter 10)
- *Colocate Pattern* (Chapter 12)
- *Valet Key Pattern* (Chapter 13)

Summary

This pattern is for decoupling tiers of your application, especially between the web (user interface) tier and a service tier that does business processing. It is not useful for routine, read-only page requests. Communication is in one direction, from the web tier to the service tier, and is handled by adding messages onto a queue. Reliable cloud queue services simplify implementation. A decoupled web tier can be more responsive and reliable, providing a better user experience. Concern-independent scaling also allows each tier to be provisioned with the ideal level of resources for that tier.

CHAPTER 4

Auto-Scaling Pattern

This essential pattern for automating operations makes horizontal scaling more practical and cost-efficient.

Scaling manually through your cloud vendor's web-hosted management tool helps lower your monthly cloud bill, but automation can do a better job of cost optimization, with less manual effort spent on routine scaling tasks. It can also be more dynamic. Automation can respond to signals from the cloud service itself and scale reactively to actual need.

The two goals of auto-scaling are to optimize resources used by a cloud application (which saves money), and to minimize human intervention (which saves time and reduces errors).

Context

The Auto-Scaling Pattern is effective in dealing with the following challenges:

- Cost-efficient scaling of computational resources, such as web tier or service tier.
- Continuous monitoring of fluctuating resources is needed to maximize cost savings and avoid delays.
- Frequent scaling requirements involve cloud resources such as compute nodes, data storage, queues, or other elastic components.

This pattern assumes a horizontally scaling architecture and an environment that is friendly to reversible scaling. Vertically scaling (by changing virtual machine size) is also possible, although is not considered in this pattern.

Cloud Significance

Cloud platforms, supporting a full range of resource management, expose rich automation capabilities to customers. These capabilities allow reversible scaling and make it highly effective when used in combination with cloud-native applications that are built to gracefully adjust to dynamic resource management and scaling. This pattern embraces the inherent increase in overlap across the development and operations activities in the cloud. This overlap is known as *DevOps*, a portmanteau of "development" and "operations."

Impact

Cost Optimization, Scalability

This pattern does not impact scalability of the application, per se. It either scales or it does not, regardless of whether it is automated. The impact is felt on operational efficiency at scale. Operational efficiency helps to lower direct costs paid to the cloud provider, while reducing the time a staff devotes to routine manual operations.

Mechanics

Compute nodes are the most common resource to scale, to ensure the right number of web server or service nodes. Auto-scaling can maintain the *right resources for right now* and can do so across any resource that might benefit from auto-scaling, such as data storage and queues. This pattern focuses on auto-scaling compute nodes. Other scenarios follow the same basic ideas, but are generally more involved.

Auto-scaling with a minimum of human intervention requires that you schedule for known events (such as halftime during the Superbowl or business vs. non-business hours) and create rules that react to environmental signals (such as sudden surges or drops in usage). A well-tuned auto-scaling approach will resemble Figure 4-1, where resources vary according to need. Anticipated needs can be driven through a schedule, with less predictable scenarios handled in reaction to environmental signals.

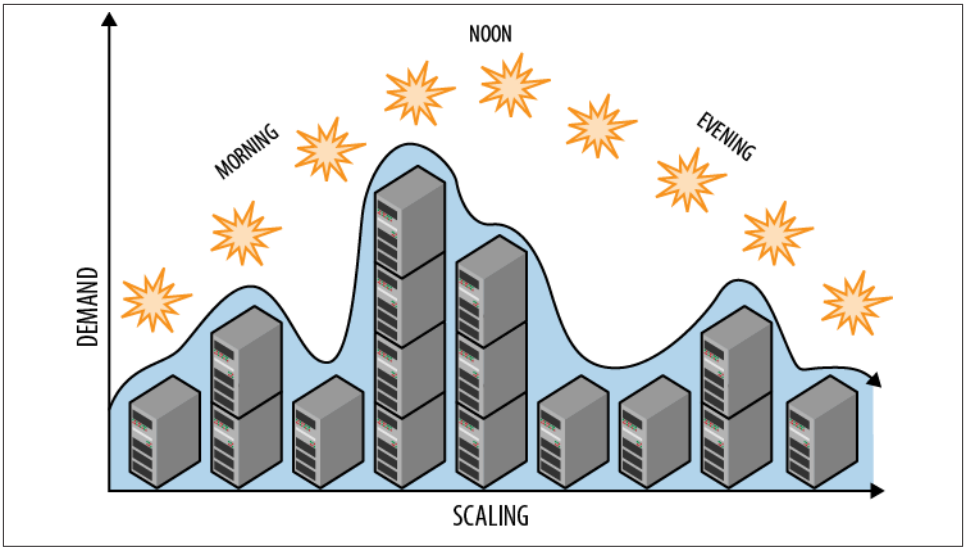
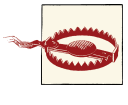


Figure 4-1. Proactive auto-scaling rules are planned, for example to add and release resources throughout the day on a schedule.

Automation Based on Rules and Signals

Cloud platforms can be automated, which includes programmatic interfaces for provisioning and releasing resources of all types. While it's possible to directly program an auto-scaling solution using cloud platform management services, using an off-the-shelf solution is more common. Several are available for Amazon Web Services and Windows Azure, some are available from the cloud vendors, and some from third parties.



Be aware that auto-scaling has its own costs. Using a Software as a Service (SaaS) offering may have direct costs, as can the act of probing your runtime environment for signals, using programmatic provisioning services (whether from your code or the SaaS solution), and self-hosting an auto-scaling tool. These are often relatively small costs.

Off-the-shelf solutions vary in complexity and completeness. Consult the documentation for any tools you consider. Functionality should allow expressing the following rules for a line-of-business application that is heavily used during normal business hours, but sporadically used outside of normal business hours:

- At 7:00 p.m. (local time), decrease the number of web server nodes to two.
- At 7:00 a.m., increase the number of web server nodes to ten.

- At 7:00 p.m. on Friday, decrease the number of web server nodes to one.

The individual rules listed are not very complicated, but still help drive down costs. Note that the last rule overlaps the first rule. Your auto-scaling tool of choice should allow you to express priorities, so that the Friday night rule takes precedent. Furthermore, rules can be scheduled; in the set listed above, the first two rules could be constrained to not run on weekends. You are limited only by your imagination and the flexibility of your auto-scaling solution.

Here are some more dynamic rules based on less predictable signals from the environment:

- If average queue length for past hour was less than 25, increase the number of invoice-processing nodes by 1.
- If average queue length for past hour was less than 5, decrease the number of invoice-processing nodes by 1.

These rules are reactive to environmental conditions. See *Queue-Centric Workflow Pattern* (Chapter 3) for an example where a rule based on queue length would be helpful.

Rules can also be written to respond to machine performance, such as memory used, CPU utilization, and so on. It is usually possible to express custom conditions that are meaningful to your application, for example:

- If average time to confirm a customer order for the past hour was more than ten minutes, increase the number of order processing nodes by one.

Some signals (such as response time from a compute node) may be triggered in the case of a node failure, as response time would drop to zero. Be aware that the cloud platform also intervenes to replace failed nodes or nodes that no longer respond.

Your auto-scaling solution may also support rules that understand overall cost and help you to stick to a budget.

Separate Concerns

The first set of example rules is applied to web server nodes and the second is applied to invoice-processing nodes. Both sets of rules could apply to the same application.

Scaling *part* of an application does not imply scaling *all* of an application.

In fact, the ability to independently scale the concerns within your architecture is an important property of cost-optimization.

When defined scale units require that resources be allocated in lockstep, they can be combined in the auto-scaling rules.

Be Responsive to Horizontally Scaling Out

Cloud provisioning is not instantaneous. Deploying and booting up a new compute node takes time (ten or more minutes, perhaps). If a smooth user experience is important, favor rules that respond to trends early enough that capacity is available in time for demand.

Some applications will opt to follow the *N+1 rule*, described in *Node Failure Pattern* (Chapter 10), where N+1 nodes are deployed even though only N nodes are really needed for current activity. This provides a buffer in case of a sudden spike in activity, while providing extra insurance in the event of an unexpected hardware failure on a node. Without this buffer, there is a risk that incoming requests can overburden the remaining nodes, reduce their performance, and even result in user requests that time out or fail. An overwhelmed node does not provide a favorable user experience.

Don't Be Too Responsive to Horizontally Scaling In

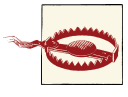
Your auto-scaling tool should have some built-in smarts relevant to your cloud platform that prevent it from being too active. For example, on Amazon Web Services and Windows Azure, compute node rentals happens in clock-hour increments: renting from 1:00-1:30 costs the same as renting from 1:00-2:00. More importantly, renting from 1:50-2:10 spans two clock hours, so you will be billed for both.

If you request that a node be released at 1:59 and it takes two minutes for it to drain in-process work before being released, will you be billed? Check the documentation for your cloud vendor on how strictly the clock-hour rules are enforced for these edge cases; there may be a small grace period.

Set lower limits thoughtfully. If you allow reduction to a single node, realize that the cloud is built on commodity hardware with occasional failures. This is usually not a best practice for nodes servicing interactive users, but may be appropriate in some cases, such as rare/sporadic over-the-weekend availability. Being down to a single node is usually fine for nodes that do not have interactive users (e.g., an invoice generation node).

Set Limits, Overriding as Needed

Implementing auto-scaling does not mean giving up full control. We can add upper and lower scaling boundaries to limit the range of permitted auto-scaling (for example, we may want to always have some redundancy on the low end, and we may need to stay within a financial budget on the high end). Auto-scaling rules can be modified as needs evolve and can be disabled whenever human intervention is needed. For tricky cases that cannot be fully automated, auto-scaling solutions can usually raise alerts, informing a human of any condition that needs special attention.



Your cloud platform most likely offers a Service Level Agreement (SLA) that asserts that your virtual machine instance will be running and accessible from the Internet some percent of time in any given month. Both Windows Azure and Amazon Web Services offer a 99.95% availability SLA. However, there is a caveat: they also require running at least two node instances of any given type in order for the SLA to be in effect.

This requirement makes perfect sense: when running with only a single node, any disruption to that node will result in downtime.

Take Note of Platform-Enforced Scaling Limits

Public cloud platforms usually have default scaling limits for new accounts. This serves to protect new account owners from themselves (by not accidentally scaling to 10,000 nodes), and limits exposure for the public cloud vendors. Because billing on pay-as-you-go accounts usually happens at the end of each month, consider the example of someone with nefarious intentions signing up with a stolen credit card; it would be hard to hold them accountable. Countermeasures are in place, including requiring a one-time support request in order to lift soft limits on your account, such as the number of cores your account can provision.

Example: Building PoP on Windows Azure

The Page of Photos (PoP) application (which was described in the Preface) will benefit from auto-scaling. There are a number of options for auto-scaling on Windows Azure. PoP uses a tool from the Microsoft Patterns & Practices team known officially as the Windows Azure Autoscaling Application Block, or (thankfully) WASABi for short.

Unlike some of the other options, WASABi is not Software as a Service but rather software we run ourselves. We choose to run it on a single worker role node using the smallest available instance (Extra Small, which costs $0.02/hour as of this writing).

It is common for cloud applications to make use of a single-role "admin" role instance for handling miscellaneous duties. This is a good home for WASABi.

WASABi can handle all the rules mentioned in this chapter. WASABi distinguishes two types of rules:

- Proactive rules, known as *constraint rules*, handle scheduled changes, minimum and maximum instance counts, and can be prioritized.
- *Reactive rules* respond to signals in the environment.

Rules for your application are chosen to align with budgets, historical activity, planned events, and current trends. PoP uses the following constraint rules:

- Minimum/Maximum number of web role instances = 2 / 6
- Minimum/Maximum number of image processing worker role instances (handling newly uploaded images) = 1 / 3

PoP has reactive rules and actions based on the following signals in the environment:

- If the ASP.NET Request Wait Time (performance counter) > 500ms, then add a web role instance.
- If the ASP.NET Request Wait Time (performance counter) < 50ms, then release a web role instance.
- If average Windows Azure Storage queue length for past 15 minutes > 20, then add an image processing worker role instance.
- If average Windows Azure Storage queue length for past 15 minutes < 5, then release an image processing worker role instance.

The third rule has the highest priority (so will take precedence over the others), while the other rule priorities are in the order listed.

WASABi rules are specified using an XML configuration file and can be managed (after deployment) using PowerShell cmdlets.

To implement the ASP.NET Request Wait Time, WASABi needs access to the performance counters from the running web role instances. This is handled through Windows Azure Diagnostics, operating at two levels.

First, each running instance is configured to gather data that you specify such as log files, application trace output ("debug statements"), and performance counters.

Second, there is a coordinated process that rounds up the data from each node and consolidates them in a central location in Azure Storage. It is from the central location that WASABi looks for the data it needs to drive reactive rules.

The Windows Azure Storage queue length data is gathered directly by WASABi. WASABi uses the Windows Azure Storage programmatic interface, which makes it a simple matter to access the current queue length. The queue in question here lies between the web tier and the service tier, where the image processing service runs; please see *Queue-Centric Workflow Pattern* (Chapter 3) for further context.

Although not described, PoP takes advantage of WASABi stabilizing rules that limit an overreaction to rules that might be costly. These help avoid thrashing (such as allocating a new virtual machine), releasing it before its rental interval has expired, and then allocating another right away; in some cases, the need could be met less expensively by just keeping the first virtual machine longer.

Throttling

WASABi also distinguishes between *instance scaling*, which is what we usually mean when scaling (adding or removing instances), and *throttling*. Throttling is selectively enabling or disabling features or functionality based on environmental signals. Throttling complements instance scaling. Suppose the PoP image processing implemented a really fancy photo enhancement technique that was also resource intensive. Using our throttling rules, we could disable this fancy feature when our image processing service was overloaded and wait for auto-scaling to bring up an additional node. The point is that the throttling can kick in very quickly, buying time for the additional node to spin up and start accepting work.

Beyond buying time while more nodes spin up, throttling is useful when constraint rules disallow adding any more nodes.

WASABi throttling should not be confused with throttling described in *Busy Signal Pattern* (Chapter 9).

Auto-Scaling Other Resource Types

Virtually all the discussion in this chapter has focused on auto-scaling for virtual machines, so what about other resource types? Let's consider a couple of more advanced scenarios.

The web tier funnels data to the service tier through a Windows Azure Queue. This is explained further in *Queue-Centric Workflow Pattern* (Chapter 3), but the key point is that both sides depend upon reliable and fast access to that queue. While most appli-

cations will not run into this problem, individual queues have scalability limits. What happens if PoP is so popular that it must process so many queue messages per second that one queue is no longer sufficient? When one queue isn't sufficient, the solution is to horizontally scale out to two queues, then three queues, and so on. Conversely, we can horizontally scale back (or *scale in*) as the demand recedes.

For databases, consider an online ticket sales scenario for a major event. Shortly before tickets go on sale, ticketing data is distributed (or *sharded*; see *Database Sharding Pattern* (Chapter 7)) over many SQL Database instances to handle the load needed to sell tens of thousands of tickets within minutes. Due to the many database instances, user traffic is easily handled. Once all the tickets are sold, data is consolidated and the excess database instances are released. This example is inspired by the TicketDirect case study mentioned in Appendix A.

Related Chapters

- *Horizontally Scaling Compute Pattern* (Chapter 2)
- *Queue-Centric Workflow Pattern* (Chapter 3)
- *Database Sharding Pattern* (Chapter 7)

Summary

The Auto-Scaling Pattern is an essential operations pattern for automating cloud administration. By automating routine scaling activities, cost optimization becomes more efficient with less effort. Cloud-native applications gracefully handle the dynamic increases or decreases in resource levels. The cloud makes it easy to plug into cloud monitoring and scaling services, with self-hosted options also available.

CHAPTER 5

Eventual Consistency Primer

The Eventual Consistency primer introduces eventual consistency and explains some ways to use it. This primer uses the CAP Theorem to highlight the challenges of maintaining data consistency across a distributed system and explains how eventual consistency can be a viable alternative.

In an eventually consistent database, simultaneous requests for the same data value can return different values. This condition is temporary, as the values become "eventually" consistent.

Eventual consistency stems from a choice in the way data is updated. It is an alternative to the use of distributed transactions. It can lead to better scalability, higher performance, and lower cost. Using it or not is a business decision.

At any moment, most of an eventually consistent database is consistent, with some small number of values still being updated. It is common for data values to be inconsistent for only seconds, but is not required. It depends on the application and can vary depending upon current circumstances.

CAP Theorem and Eventual Consistency

Brewer's CAP Theorem (or simply the *CAP Theorem*) considers three possible guarantees for data within a distributed application: *Consistency*, *Availability*, and *Partition Tolerance* (which spell CAPT, though the more pronounceable CAP is used). *Consistency* means everyone gets the same answer; *availability* means clients have ongoing access (even if there is partial system failure); and *partition tolerance* means correct operation, even if nodes within the application are cut off from the network and unable to communicate. The CAP Theorem posits that of these three possible guarantees, an application can only pick two.

Guaranteeing consistency is easy when data is on a single node, but once the data is distributed, partition tolerance needs to be considered. What happens if our nodes cannot communicate with each other due to failure, or simply cannot do so fast enough to meet our performance and scalability targets? Different tradeoffs are possible. One popular choice for cloud-native applications is to favor partition tolerance and availability and give up the guarantee of immediate consistency.

Applications that do not guarantee immediate consistency are said to be *eventually consistent*. The use of eventual consistency makes sense when the business value (risk, downside, or cost) is deemed superior to immediate consistency.

While this approach is not as familiar to those from the relational database world, eventual consistency can be a powerful feature that enables better scalability. Eventual consistency is not a deficiency or design flaw. When used appropriately, it is a feature.

Eventual Consistency Examples

The term eventual consistency is relatively new, but the idea is not. An old example can be found with the Domain Name System (DNS). DNS powers the Internet name resolution that is responsible for turning human-friendly web addresses (such as *http://www.pageofphotos.com*) into a computer-friendly IP address (such as 12.34.56.789). When the IP Address for a domain name is changed, it usually takes hours before the update propagates to all DNS servers (which may have the old address cached) across the Internet. This is considered a good tradeoff; IP addresses change infrequently enough that we tolerate the occasional inconsistency in exchange for superb scalability. After the IP address has been changed but before it is fully propagated, some users will be directed to the old site IP address, and some to the new site IP address.

Eventual Satisfaction

Eventually consistent does not mean that the system doesn't care about consistency. Most data is consistent virtually all of the time. There is just a business-tolerable delay before updates fully propagate. During this delay, different users may see different versions of the data.

The Page of Photos (PoP) application is eventually consistent because there is a delay after a photo is uploaded, but before it appears to visitors on the site. Furthermore, some viewers may see the photo before others do. Part of this is due to data replication across data centers, but some is just internal processing, such as the ingestion process for newly uploaded photos.

Data values that are no longer current are referred to as being *stale*. Sometimes stale data is visible to users. When a user sees data they know is stale, and there is a delay before they see the most current data, we call this resolution *eventual satisfaction*.

If you've ever experienced "buying" tickets online, only to find out that they have already been sold, you've seen eventual consistency in action.

Windows Azure, Amazon Web Services, Google App Engine, and other cloud platforms are themselves eventually consistent in a variety of circumstances. For example, it takes many minutes after activating global services such as CloudFront (a global CDN service from Amazon) and Traffic Manager (a global load-balancing service from Windows Azure) for them to propagate to nodes around the world.

The CAP Theorem has formalized these ideas and they have taken hold in distributed systems in the cloud and become popular with some databases.

Relational ACID and NoSQL BASE

The traditional relational database offers four so-called *ACID* guarantees:

Atomicity
: All of a transaction completes, or none of it does.

Consistency
: Data is always valid according to schema constraints.

Isolation
: Transactions competing to change the same data are applied sequentially.

Durability
: Committed changes are not lost.

These guarantees originated in a world where databases ran on a single node. They become more complex and expensive if the database is distributed.

For a single-node application, the CAP Theorem is not interesting, as partition tolerance need not be considered. As databases become more distributed (clustered, or with a geographically distributed failover node), the CAP Theorem consideration comes into play.

The CAP Theorem informs us that we must pick two of the three guarantees, which can be written in shorthand as CA, AP, and CP. All three combinations result in different behaviors. The one we will focus on here is AP (availability and partition tolerance), also known as eventually consistent.

By definition, eventually consistent databases do not support ACID guarantees, though they do support BASE. A *BASE* database is:

Basically Available
: The system will respond even with stale data.

Soft State
: State might not be consistent and might be corrected.

Eventually Consistent
: We allow for a delay before any individual data change completely propagates through the system.

BASE is commonly associated with NoSQL databases, and NoSQL database services are popular in the cloud. *NoSQL*, or *Not Only SQL*, is a database style that has emerged in recent years. NoSQL databases tend to be designed for very high scale at the expense of some advanced features of traditional relational databases. For example, they tend to have limited transactional capabilities and no ACID guarantees. Notably, they are usually designed to support sharding, which is further explored in *Database Sharding Pattern* (Chapter 7).

Unlike the acid and base you may have learned about in high school chemistry class, ACID and BASE can be used together safely, even in the same application.

Impact of Eventual Consistency on Application Logic

Previous examples have focused on eventual consistency scenarios that may be familiar or seem intuitive. Developers are often surprised when eventual consistency is used in a database. We have all come to expect that we can read a value back from a database after we've written it. This is not guaranteed if the database is eventually consistent.

Google's App Engine Datastore service and Amazon's S3 storage service are eventually consistent. Sometimes you get a choice: Amazon's SimpleDB database service has configurable consistency (with different performance characteristics). Many NoSQL databases are eventually consistent.

Windows Azure Storage is *immediately consistent*; you can immediately read back whatever you wrote. Sometimes this is also referred to as *strongly consistent* or *strictly consistent*.

It is important to note that eventually consistent databases always support some level of atomicity. Check the documentation for your eventually consistent database to understand what is considered atomic, but typically a database operation that writes a single record that changes ten properties will propagate as an atomic unit. Eventually consistent does not extend inside this atomic unit. None of the ten changes will show up individually; there is no partial update. None of the updates are visible until all of the updates are visible.

How should a developer deal with data storage that is eventually consistent?

User Experience Concerns

Sometimes, a reasonable approach is to act like it doesn't matter. Just go with the data you have at the moment. Surprisingly, this works very well in many scenarios where eventually consistent data makes sense. Often, users can't tell the difference.

However, sometimes that depends on who the user is. If the user is the one who just updated the data, it is more important to show the data the user expects, rather than wait for eventual satisfaction. In such cases, it may be sufficient for the user interface to update the user interface to reflect the most recent user-initiated change. In this case, the user interface intentionally does not refresh data from the database, knowing it may be stale.

Programmatic Differences

Data storage systems vary, but there are some common threads. Optimistic concurrency and "last write wins" models are common. These two features go hand-in-hand because they allow an application to retrieve a value, update it in memory, and then conditionally write it back. The condition is the timestamp from the original value; if the timestamp in storage is the same as the timestamp on the original value, there have been no changes in the meantime, so the update does not lose data.

Other systems are more sophisticated than "last write wins." The Amazon Dynamo Database was built for the shopping cart on Amazon.com. Dynamo is designed to merge multiple versions of the same shopping cart, such as might occur through temporary system partitions (the "P" in CAP), a sensible feature given the purpose.

If all writes go to a single location, dealing with eventual consistency is simplified. This is the case with the Couchbase and MongoDB NoSQL databases, which only accept writes to the master node for a particular data value. Once written, that updated value propagates to other nodes which are not allowed to modify it. In these scenarios, eventual consistency only matters during reads.

Summary

The CAP Theorem provides the theoretical basis that explains why we cannot guarantee both consistency and availability in a distributed database. A useful compromise is to allow for eventual consistency in favor of better scalability. Determining if your application data is a suitable candidate for eventual consistency is a business decision. The choice is between displaying stale data and scaling more efficiently.

CHAPTER 6
MapReduce Pattern

This pattern focuses on applying the MapReduce data processing pattern.

MapReduce in this chapter is explicitly tied to the use of Hadoop since that helps pin down its capabilities and avoid confusion with other variants. The term MapReduce is used except when directly referencing the Hadoop project (which is introduced below).

MapReduce is a data processing approach that presents a simple programming model for processing highly parallelizable data sets. It is implemented as a cluster, with many nodes working in parallel on different parts of the data. There is large overhead in starting a MapReduce job, but once begun, the job can be completed rapidly (relative to conventional approaches).

MapReduce requires writing two functions: a mapper and a reducer. These functions accept data as input and then return transformed data as output. The functions are called repeatedly, with subsets of the data, with the output of the mapper being aggregated and then sent to the reducer. These two phases sift through large volumes of data a little bit at a time.

MapReduce is designed for batch processing of data sets. The limiting factor is the size of the cluster. The same map and reduce functions can be written to work on very small data sets, and will not need to change as the data set grows from kilobytes to megabytes to gigabytes to petabytes.

Some examples of data that MapReduce can easily be programmed to process include text documents (such as all the documents in Wikipedia), web server logs, and users' social graphs (so that new connection recommendations can be discovered).

Data is divvied up across all nodes in the cluster for efficient processing.

Context

The MapReduce Pattern is effective in dealing with the following challenges:

- Application processes large volumes of relational data stored in the cloud
- Application processes large volumes of semi-structured or irregular data stored in the cloud
- Application data analysis requirements change frequently or are *ad hoc*
- Application requires reports that traditional database reporting tools cannot efficiently create because the input data is too large or is not in a compatible structure

Other cloud platforms may support Hadoop as an on-demand service. This pattern assumes a Hadoop-based service.

Hadoop implements MapReduce as a batch-processing system. It is optimized for the flexible and efficient processing of massive amounts of data, not for response time.

The output from MapReduce is flexible, but is commonly used for data mining, for reporting, or for shaping data to be used by more traditional reporting tools.

MapReduce as a Service

Amazon Web Services, Google, and Windows Azure all offer MapReduce as an on-demand service.

The Amazon Web Services and Windows Azure services are based on the open source Apache Hadoop project (*http://hadoop.apache.org*).

The Google service also uses the MapReduce pattern, but not with Hadoop. In fact, Google invented MapReduce to solve problems it faced in processing the vast data it was collecting, such as page-to-page hyperlinks that were gathered by crawling public websites. In particular, MapReduce was used to analyze these links to apply Google's famous PageRank algorithm to decide which websites are most worthy of showing up in searches. Afterwards, they published some academic papers explaining their approach, and the Hadoop project was modeled after it. (Relevant links are in Appendix A.)

Cloud Significance

Cloud platforms are good at managing large volumes of data. One of the tenets of big data analysis is *bring the compute to the data*, since moving large amounts of data is expensive and slow. A cloud service for the analysis of data already nearby in the cloud will usually be the most cost-effective and efficient.

Large volumes of data stored on-premises are more effectively analyzed by using a Hadoop cluster that is also on-premises.

Hadoop greatly simplifies building distributed data processing flows. Running *Hadoop as a Service* in the cloud takes this a step further by also simplifying Hadoop installation and administration. The Hadoop cloud service can access data directly from certain cloud storage services such as S3 on Amazon and Blob Storage on Windows Azure.

Using MapReduce through a Hadoop cloud platform service lets you rent instances for short amounts of time. Since a Hadoop cluster can involve many compute nodes, even hundreds, the cost savings can be substantial. This is especially convenient when data is stored in cloud storage services that integrate well with Hadoop.

Impact

Cost Optimization, Availability, Scalability

Mechanics

Map and Reduce from Computer Science

For context, it is helpful to be aware that the name of this pattern derives from the functional programming concepts of *map* and *reduce*. In computer science, map and reduce describe functions that can be applied to lists of elements. A map function is applied to each element in a list, resulting in a new list of elements (sometimes called a *projection*). A reduce function is applied to all elements in a list, resulting in a single scalar value.

Consider the list [“foo”, “bar”]. A map function that converts a single string to uppercase would result in a list where the letters have all been converted to uppercase ([“FOO”, “BAR”]); a corresponding reduce function that accepts a list of strings and concatenates them would produce a single result string (“FOOBAR”). A map function that returns the length of a string would result in a list of word lengths ([3, 3]); a corresponding reduce function that accepts a list of integers and adds them up would produce a sum (6).

Map produces a new list, while reduce produces a scalar result. While simple concepts, they are powerful.

The map and reduce functions implemented in this pattern are conceptually similar to the computer science versions, but not exactly the same. In the MapReduce Pattern, the lists consist of key/value pairs rather than just values. The values can also vary widely: a text block, a number, even a video file.

Hadoop is a sophisticated framework for applying map and reduce functions to arbitrarily large data sets. Data is divvied up into small units that are distributed across the cluster of data nodes. A typical Hadoop cluster contains from several to hundreds of data nodes. Each data node receives a subset of the data and applies the map and reduce functions to locally stored data as instructed by a *job tracker* that coordinates jobs across the cluster. In the cloud, "locally stored" may actually be durable cloud storage rather than the local disk drive of a compute node, but the principle is the same.

Data may be processed in a workflow where multiple sets of map and reduce functions are applied sequentially, with the output of one map/reduce pair becoming the input for the next. The resulting output typically ends up on the local disk of a compute node or in cloud storage. This output might be the final results needed or may be just a data shaping exercise to prepare the data for further analytical tools such as Excel, traditional reporting tools, or Business Intelligence (BI) tools.

MapReduce Use Cases

MapReduce excels at many data processing workloads, especially those known as *embarrassingly parallel problems*. Embarrassingly parallel problems can be effortlessly parallelized because data elements are independent and can be processed in any order. The possibilities are extensive, and while a full treatment is out of scope for this brief survey, they can range from web log file processing to seismic data analysis.

MapReduce and You

You may see the results of MapReduce without realizing it. LinkedIn uses it to suggest contacts you might want to add to your network. Facebook uses it to help you find friends you may know. Amazon uses it to recommend books. It is heavily used by travel and dating sites, for risk analysis, and in data security. The list of uses is extensive.

This pattern is not typically used on small data sets, but rather on what the industry refers to as *big data*. The criteria for what is or is not big data is not firmly established,

but usually starts in the hundreds of megabytes or gigabytes range and goes up to petabytes. Since MapReduce is a distributed computing framework that simplifies data processing, one might reasonably conclude that big data begins when the data is too big to handle with a single machine or with conventional tooling.

If the data being processed will grow to those levels, the pattern can be developed on smaller data and it will continue to scale. From a programming point of view, there is no difference between analyzing a couple of small files and analyzing petabytes of data spread out over millions of files. The map and reduce functions do not need to change.

Beyond Custom Map and Reduce Functions

Hadoop supports expressing map and reduce functions in Java. However, any programming language with support for standard input and standard output (such as C++, C#, Python) can be used to implement map and reduce functions using Hadoop streams. Also, depending on details of the cloud environment, other higher-level programming languages may be supported in some cases. For example, in Hadoop on Azure, JavaScript can be used to script *Pig* (Pig is introduced shortly).

Hadoop is more than a robust distributed map/reduce engine. In fact, there are so many other libraries in the Apache Hadoop Project, that it is more accurate to consider Hadoop to be an ecosystem. This ecosystem includes higher-level abstractions beyond map/reduce.

For example, the *Hive* project provides a query language abstraction that is similar to traditional SQL; when one issues a query, Hive generates map/reduce functions and runs them behind the scenes to carry out the requested query. Using Hive interactively as an *ad hoc* query tool is a similar experience to using a traditional relational database. However, since Hadoop is a batch-processing environment, it may not run as fast.

Pig is another query abstraction with a data flow language known as *Pig Latin*. Pig also generates map/reduce functions and runs them behind the scenes to implement the higher-level operations described in Pig Latin.

Mahout is a machine-learning abstraction responsible for some of the more sophisticated jobs, such as music classification and recommendations. Like Hive and Pig, Mahout generates map/reduce functions and runs them behind the scenes.

Hive, Pig, and Mahout are abstractions that, comparable to a compiler, turn a higher-level abstraction (such as Java code) into machine instructions.

The ecosystem includes many other tools, not all of which generate and execute map/reduce functions. For example, *Sqoop* is a relational database connector that gives you access to advanced traditional data analysis tools. This is often the most potent combination: use Hadoop to get the right data subset and shape it to the desired form, then use Business Intelligence tools to finish the processing.

More Than Map and Reduce

Hadoop is more than just capable of running MapReduce. It is a high-performance operating system for building distributed systems cost-efficiently.

Each byte of data is also stored in triplicate, for safety. This is similar to cloud storage services that typically store data in triplicate, but refers to Hadoop writing data to the local disk drives of its data nodes. Cloud storage can be used to substitute for this, but that is not required.

Automatic failure recovery is also supported. If a node in the cluster fails, it is replaced, any active jobs restarted, and no data will be lost. Tracking and monitoring administrative features are built in.

Example: Building PoP on Windows Azure

A new feature we want to add to the Page of Photos (PoP) application (which was described in the Preface) is to highlight the most popular page of all time. To do this, we first need data on page views. These are traditionally tracked in web server logs, and so can easily be parsed out. As described in *Horizontally Scaling Compute Pattern* (Chapter 2), the PoP IIS web logs are collected and conveniently available in blob storage.

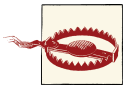
We can set up Hadoop on Azure to use our web log files as input directly out of blob storage. We need to provide map and reduce functions to process the web log files. These map and reduce functions would parse the web logs, one line at a time, extracting just the visited page from that line. Each line of the log file would contain a reference to a page; for example, a row in the web log indicating a visit to *http://www.pageofphotos.com/jebaka* would include the string "/jebaka." Our map function can remove the leading forward slash character. We could also have our map function ignore any rows that were not for visited pages, such as rows that were for image downloads. Because MapReduce expects a map function to return an attribute value pair, our simple map function would output a single string such as "jebaka, 1" where the "1" indicates a count of 1.

MapReduce will collect all instances of "jebaka, 1" and pass them on as a single list to our reduce function. The key here is "jebaka" and the list passed to the reduce function is a key followed by many values. The input to the reduce function would be "jebaka, 1 1 1 1 1 1 1 1 1 1" (and so on, depending upon how many views that page got). The reduce function needs to add up all the hits (10 in this example) and output it as "jebaka, 10" and that's all.

MapReduce will take care of the rest of the bookkeeping. In the end, there will be a bunch of output files with totals in them. While more map/reduce functions could be written to further simplify, we'll assume that a simple text scan (not using MapReduce) could find the page with the greatest number of views and cache that value for use by the PoP logic that displays the most popular site on the home page.

If we wanted to update this value once a week, we could schedule the launching of a MapReduce job. The individual nodes (which are worker role instances under the covers) in the Hadoop on Azure cluster will only be allocated for a short time each week.

Enabling clusters to be spun up only periodically without losing their data depends on their ability to persist data into cloud storage. In our scenario, Hadoop on Azure reads web logs from blob storage and writes its analysis back to blob storage. This is convenient and cuts down on the cost of compute nodes since they can be allocated on demand to run a Hadoop job, then released when done. If the data was instead maintained on local disks of compute nodes (which is how traditional, non-cloud Hadoop usually works), the compute nodes would need to be kept running continually.



As of this writing, the Hadoop on Azure service is in preview. It is not yet supported for production use.

Related Chapters

- *Horizontally Scaling Compute Pattern* (Chapter 2)
- *Eventual Consistency Primer* (Chapter 5)
- *Colocate Pattern* (Chapter 12)

Summary

The MapReduce Pattern provides simple tools to efficiently process arbitrary amounts of data. There are abundant examples of common use that are not economically viable using traditional means. The Hadoop ecosystem provides higher-level libraries that simplify creation and execution of sophisticated map and reduce functions. Hadoop also makes it easy to integrate MapReduce output with other tools, such as Excel and BI tools.

CHAPTER 7

Database Sharding Pattern

This advanced pattern focuses on horizontally scaling data through sharding.

To shard a database is to start with a single database and then divvy up its data across two or more databases (shards). Each shard has the same database schema as the original database. Most data is distributed such that each row appears in exactly one shard. The combined data from all shards is the same as the data from the original database.

The collection of shards is a single logical database, even though there are now multiple physical databases involved.

Context

The Database Sharding Pattern is effective in dealing with the following challenges:

- Application database query volume exceeds the query capability of a single database node resulting in unacceptable response times or timeouts
- Application database update volume exceeds the transactional capability of a single database node resulting in unacceptable response times or timeouts
- Application database network bandwidth needs exceed the bandwidth available to a single database node resulting in unacceptable response times or timeouts
- Application database storage requirements exceed the capacity of a single database node

This chapter assumes sharding is done with a database service that offers integrated sharding support. Without integrated sharding support, sharding happens entirely in the application layer, which is substantially more complex.

Cloud Significance

Historically, sharding has not been a popular pattern because sharding logic was usually custom-built as part of the application. The result was a significant increase in cost and complexity, both in database administration and in the application logic that interacts with the database. Cloud platforms significantly mask this complexity.

Integrated database sharding support is available with some cloud database services, in both relational and NoSQL varieties.

Integrated sharding support pushes complexity down the stack: out of the application code and into the database service.

Any database running on a single node is ultimately limited. Capacity limits are lower in the cloud than they are with customer-owned high-end hardware. Therefore, limits in the cloud will be reached sooner, requiring that database sharding occur sooner. This may lead to an increase in sharding popularity for cloud-native applications.

Impact

Scalability, User Experience

Mechanics

Traditional (non-sharded) databases are deployed to a single database server node. Any database running on a single node is limited by the capacity of that node. Contention for resources such as CPU, memory, disk speed, data size, and network bandwidth can impair the database's ability to handle database activity; excessive contention may overwhelm the database. This limited capacity is exacerbated with cloud database services because the database is running on commodity hardware and the database server is multitenant. (Multitenancy is described in *Multitenancy and Commodity Hardware Primer* (Chapter 8).)

There are many potential approaches for scaling an application database when a single node is no longer sufficient. Some examples include: distributing query load to *slave nodes*, splitting into multiple databases according to the type of data, and vertically scaling the database server. To handle query load (but not write/update), slave nodes are replicated from a master database; slave nodes are read-only and are typically *eventually consistent*. Another option is splitting into multiple databases according to the

type of data, such as inventory data in one database and employee data in another. In the cloud, vertically scaling the database is possible if you are willing to manage your own database server—a painful tradeoff—while still constrained by maximum available virtual machine size. The cloud-native option is database sharding.

The Database Sharding Pattern is a horizontal scaling approach that overcomes the capacity limits of a single database node by distributing the database across multiple database nodes. Each node contains a subset of the data and is known as a *shard*. Collectively, the data in all the shards represents a complete logical database. In a database service with integrated sharding, the collection of shards appears to applications as a single database with a single database connection string. This abstraction is a great boon to simplifying the programming model for applications. However, as we shall see, there are also limitations.

Shards Are Autonomous

Sharding is a horizontal scaling strategy in which resources from each shard (or node) contribute to the overall capacity of the sharded database. Database shards are said to implement a *shared nothing* architecture that simply means that nodes do not share with other nodes; they do not share disk, memory, or other resources.

For the approach to be efficient, common business operations must be satisfied by interacting with a single shard at a time. Cross-shard transactions are not supported.

Basically, shards do not reference other shards. Each shard is autonomous.

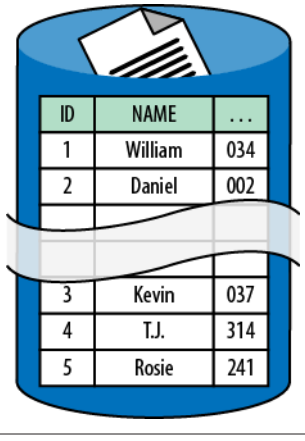


Figure 7-1. Data rows are distributed across shards, while maintaining the same structure. Two shards are depicted: one shard holds rows with ID=1-2, the other holds rows with ID=3-5.

In the most straightforward model as shown in Figure 7-1, all shards have the same database schema as the original database, so they are structurally identical, and the database rows are divvied up across the shards. The exact manner in which these rows are divvied up is critical if sharding is to provide the desired benefits.

Shard Identification

A specific database column designated as the *shard key* determines which shard node stores any particular database row. The shard key is needed to access data.

As a naïve but easily understood example, the shard key is the *username* column and the first letter is used to determine the shard. Any usernames starting with A-J are in the first shard, and K-Z in the second shard. When your customer logs in with their username, you can immediately access their data because you have a valid shard key.

A more complex example might shard on the *customerid* column that is a GUID. When your customer logs in with their username, you do not have a valid shard key. You can maintain a mapping table, perhaps in a distributed cache, to look up the shard key (*customerid*) from their username. Then you can access their data.

Shard Distribution

Why are you sharding? Answering this question is a good start for determining a reasonable sharding strategy. When you are sharding because data will not fit into a single node instance, divide the data into similarly sized shards to make it fit. When you are sharding for performance reasons, divide the data across shard nodes in such a way that all shard nodes experience a similar volume of database queries and updates.

When sharding for scalability or query performance, shards should be added before individual nodes can no longer keep pace with demand. Runtime logging and analytics are important in understanding the behavior of a sharded database. Some commercial databases do this analysis for you and shard automatically.

It is important that a single shard at a time can satisfy most of the common database operations. Otherwise, sharding will not be efficient.

Reporting functions such as aggregating data can be complicated because they span shards. Some database servers offer services to facilitate this. For example, Couchbase supports MapReduce jobs that span shards. If the database server does not have support for this, such scenarios must be handled in the application layer, which adds complexity.

More advanced scenarios are possible. Different sets of tables might use different shard keys, resulting in multiple sets of shards. For example, separate shards may be based on *customer* and *inventory* tables. It might also be possible to create a composite shard key by combining keys from different database entities.

When Not to Shard

Database schemas designed for cloud-native applications will support sharding. However, it should not be assumed that any database is easily updated to support sharding. This may be especially true of older databases that have evolved over many years. While a detailed analysis is beyond the scope of this chapter, a poorly modeled database is a bad choice.

It is worth emphasizing that even though they are not the subject of this book because they are not different in the cloud, database optimization techniques, proper indexing, query tuning, and caching are still completely useful and valid in the cloud. Just because the cloud allows you to more easily shard doesn't make it the solution to every database scaling or performance problem.

Is Cloud-Native Right for Your Application?

To maximize the value of the cloud, we build cloud-native applications. Building cloud-native applications requires that we think differently about some important aspects of architecture. However, building cloud-native applications does not mean that we forget all of our database tuning skills, as so many of them remain extremely useful.

Furthermore, while this is a book of patterns for building cloud-native applications, that's not the right approach for every application. Database architecture is central to many existing applications, and changing from a vertical scaling approach to a horizontal scaling approach is a big change. Sometimes the right business decision will be to host your own instance of a relational database so that you can still move to the cloud without changing the database architecture. Of course, you will forego many of the benefits, but at least it is possible through use of Linux or Windows virtual machines running on the Amazon or Windows Azure platforms. This can also serve as a temporary or transitional solution on the way to becoming fully cloud-native.

Not All Tables Are Sharded

Some tables are not sharded, but rather replicated into each shard. These tables contain *reference data*, which is not specific to any particular business entity and is *read-mostly*. For example, a list of zip codes is reference data; the application may use this data to determine the city from a zip code. We duplicate reference data in each shard to maintain shard autonomy: all of the data needed for queries must be in the shard.

The rest of the tables are typically sharded. Unlike reference data, any given row of a sharded table is stored on exactly one shard node; there is no duplication.

The sharded tables typically include those tables responsible for the bulk of the data size and database traffic. Reference data typically does not change very often and is much smaller than business data.

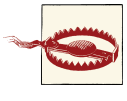
Cloud Database Instances

Using multiple database instances as a single logical database can bring some surprises. It is a distributed system, so the internal clocks on instances are never exactly the same. Be careful in making assumptions about timestamps across shards when using the database time to establish chronological order.

Further, cloud databases are set to Universal Coordinated Time (UTC), not local time. This means that application code is responsible for translating into local time as needed for reporting and user interface display.

Example: Building PoP on Windows Azure

The Page of Photos (PoP) application (which was described in the Preface) uses Windows Azure SQL Database to manage user-generated data about the photos. Basic account information includes name, login identifier (email address), and folder name; for example, the folder *kevin* corresponds to the photo page *http://pageofphotos.com/kevin*. Photo data includes a description, timestamp, and geographic location.



In June 2012, the SQL Azure service was renamed to Windows Azure SQL Database, or simply SQL Database. Because the SQL Azure name was around for a long time, many people still use the old name and most existing web information was written with the old name. The new name is used in this book.

There is also data that spans photos and spans accounts. Photo tags are shared across photos, geographic data is correlated ("other photos taken nearby"), and comments from one user to another create cross-references to other accounts. Users can follow other users' pages (so you can get an alert email when they post a new photo), and so forth. Data that is naturally relational suggests that using a relational database may be a good choice.

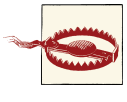
PoP grew in popularity over time and eventually reached the point where there were so many active users that the volume of database activity started to become a bottleneck. As explained in *Busy Signal Pattern* (Chapter 9), occasional throttling should be expected, but usage was too high for a single database instance to handle. To overcome this, the PoP database was sharded: we spread the demand across shards so that any individual shard can easily handle its share of the volume.

Windows Azure SQL Database offers integrated sharding support through a feature known as *Federations*. This feature helps applications flexibly manage a collection of shards, keeping the complexity out of the application layer. PoP uses the Federations feature to implement sharding.

The Federations feature uses different terminology than is used so far in this chapter. A *federation* is equivalent to a *shard*, a *federation key* is equivalent to a *shard key*, and a *federation member* is a database node hosting a federation. The remainder of this section will use the terminology specific to Windows Azure SQL Database.

PoP identifies users by their email address and uses that as its federation key. With Federations, the application is responsible for specifying the range of data in each federation based on the federation key. In the case of PoP, the first step is to simply spread the workload across two federations. This works well with data associated with email addresses beginning with characters "a" through "g" in the first federation, and the rest of the data in the second federation.

A one-time configuration is necessary to enable federations, with the most important operation being the establishment of the federation ranges. This is done using a new database update command, `ALTER FEDERATION`. Once federation ranges are defined by the application, the Federations feature gets busy moving data to the appropriate federation members. There is a small amount of boilerplate code needed in application code (issuing a `USE FEDERATION` command), but the application logic is otherwise essentially the same as when using SQL Database without Federations.



The `ALTER FEDERATION` command currently supports a `SPLIT AT` directive, which is used to specify how to divvy up the data across shards. The inverse, `MERGE AT`, has not yet been released. Until `MERGE AT` (or equivalent) is released, reducing the number of shards in your SQL Database is more cumbersome than it will be ultimately. See Appendix A for a reference on how to simulate `MERGE AT` today.

Rebalancing Federations

Through growth, PoP will eventually need to rebalance the federations. For example, the next step could be to spread the data equally across three federations. This is one of the most powerful aspects of Federations: it will handle all rebalancing details behind the scenes, without database downtime, and maintain all ACID guarantees, without the application code needing to change.

Federations will handle the distribution of data across federations, including redistributes, without database downtime. This is one of the two big differences between sharding at the application layer. The other big difference is that Federations appears as a single database to the application, regardless of how many federation members there are. This abstraction allows the application to deal with only a single database connection string, simplifying caching and database connection pooling.

Fan-Out Queries Across Federations

Recall that in database sharding, not all tables are federated. Reference data is replicated to all federation members in order to maintain autonomy. If reference tables are not replicated to individual federation members, they cannot be joined to during database queries without involving multiple federations.

PoP has some reference data that supports a feature that allows users to tag photos with a word such as ***`redsox`***, ***`theater`***, ***`heavy metal`***, or ***`manga`***. In order to promote consistency across users, these tags are stored in a shared table in a Windows Azure SQL Database that is frequently used in queries and updates across multiple federations. This is reference data, so we replicate it to each federation.

What happens when we need to add a new tag? We write application code to replicate our changes across all federations. We do this using a *fan-out query* that visits each federation member and applies the update. Because it happens from application code, the different federation members will have different sets of tags while the fan-out query is in progress. It is an acceptable business risk for PoP if the list of photos tags is eventually consistent.

Fanning Out

The PoP reference data example uses a fan-out query to replicate a reference data change across all federation members. This is known as a fan-out query since it "fans out" to every federation member, but it isn't actually a query. It is an update. Queries are also possible, though be aware that all joins happen in application code. Windows Azure SQL Data Sync could also be used to keep the tags in sync across federations.

A fanout query implementation resembles MapReduce: apply a query to individual federation members (map), then combine and refine that intermediate data into a final result (reduce).

Not surprisingly, Hadoop on Azure is another tool that can be useful in processing large amounts of data spanning federation members. Hadoop jobs can be used for data shaping where large data sets are preprocessed such that the resulting subset can be conveniently analyzed using Excel or traditional reporting tools. As mentioned previously, *Sqoop* can be used to allow Hadoop to connect to a Windows Azure SQL Database instance.

NoSQL Alternative

PoP data is relational, so a relational database is an appropriate tool for data management. However, other reasonable options are available, specifically a NoSQL database. There are tradeoffs.

Relational or NoSQL?

The PoP application is contrived to demonstrate use cases for cloud architecture patterns that, in turn, inspire the use of Windows Azure services and features. A reasonable review of the PoP data requirements may show that while the data contains some relationships, the relationships are not so complex that a full-featured relational database is needed. This is reasonable. No hard-and-fast rules exist regarding when to go with a full-featured relational database versus a NoSQL database, with pros and cons for either approach.

Windows Azure offers a NoSQL database as a service. This is the Table service that is part of Windows Azure Storage.

The Table service is an k*ey-value store*, meaning that application code specifies a *key* to set or get a *value*. It is a simple programming model. Each value can consist of many *properties* (up to 252 custom and 3 system properties). The programmer can choose among some standard data types for each custom property such as integer, string, binary data (up to 64 KB), double, GUID, and a few more. The three system properties are a timestamp (useful for optimistic locking), a partition key, and a row key.

The partition key is conceptually similar to a federation (or shard) key, and defines a grouping of data rows (each identified by row key) that are guaranteed to be stored on the same storage node. Federations are different in that each federation member hosts exactly one federation, whereas the Table service decides how many partition keys will be on each storage node, based on usage. The Table service automatically moves partitions between storage nodes to optimize for observed usage, which is more automated than Federations.

Hopefully the Federations feature will go beyond "integrated sharding" to a fully "auto-sharding" service in the future. In this (hypothetical) scenario, Federations monitors database analytics on your behalf and makes reasonable decisions to balance federations based on data size, query volume, and database transaction volume.

The Azure Table service is less capable than SQL Database in managing relationships. The Table service does not support a database schema, there is nothing akin to relational database referential integrity or a constraint between two tables, and there are no foreign keys. This means that the application layer is handling any relations between values. Transactions are supported, but only within a single partition. Two related values in separate partitions cannot be updated atomically, though they can be *eventually consistent*.

In summary, NoSQL can work just fine, and some aspects are easier than using SQL Database with Federations, but other aspects are more challenging.

Cost of using the services should also be a factor. Use the Windows Azure Pricing Calculator to model the costs.

Related Chapters

- *Horizontally Scaling Compute Pattern* (Chapter 2)
- *Auto-Scaling Pattern* (Chapter 4)
- *Eventual Consistency Primer* (Chapter 5)
- *MapReduce Pattern* (Chapter 6)

Summary

When using the Database Sharding Pattern, workloads can be distributed over many database nodes rather than concentrated in one. This helps overcome size, query performance, and transaction throughput limits of the traditional single-node database. The economics of sharding a database become favorable with managed sharding support, such as found in some cloud database services.

The data model must be able to support sharding, a possible barrier for some applications not designed with this in mind. Cross-shard operations can be more complex.

CHAPTER 8

Multitenancy and Commodity Hardware Primer

This primer introduces multitenancy and commodity hardware and explains why they are used by cloud platforms.

Cloud platforms are optimized for cost-efficiency. This optimization is partially driven by the high utilization of services running on cost-efficient hardware that manifests as multitenant services running on commodity hardware.

The decisions made in building the cloud platform also influence the applications that run on it. The impact to the application architecture of cloud-native applications manifests through horizontal scaling and handling failure.

Multitenancy

Multitenancy means there are multiple tenants sharing a system. Usually the system is a software application operated by one company, the host, for use by other companies, the tenants. Each tenant company has individual employees who access the software. All employees of a tenant company can be connected within the application while other tenants are invisible; this creates the illusion for each tenant that they are the only customers using the software.

In a single tenant model, if an application needs a database, it gets its own instance. This simplifies capacity management (for individual applications), but at the expense of overall efficiency, as many database servers (and other types of servers) will be running with low overall utilization much of the time.

In the cloud, multitenant services are standard: data services, DNS services, hardware for virtual machines, load balancers, identity management, and so forth. Cloud data centers are optimized for high hardware utilization and that drives down costs.

Multitenancy: Not Just for Cloud Platform Services

Cloud platforms have embraced multitenant services, so why not you? Software as a Service (SaaS) is a delivery model in which a software application is offered as a managed service; customers simply rent access. You may wish to build your SaaS application as multitenant on the cloud so that you can leverage the cost-efficiencies of shared instances. You can choose to be multitenant all the way through for maximum savings, or just in some areas but not others, such as with compute nodes but not database instances, for example.

Sometimes SaaS applications are also able to (perhaps anonymously) glean valuable business insights and analytics from the aggregate data they are managing across many customers.

There are also downsides to multitenant services. Your architecture will need to ensure *tenant isolation* so that one customer cannot access another customer's data, while still allowing individual customers access to their own data and reporting.

Two common areas of concern are security and performance management.

Security

Any individual tenant on a multitenant service is placed in a security sandbox that limits its ability to know anything about the other tenants, even the existence of other tenants. This is handled in different ways on different services. For example, hypervisors manage security on virtual machines, relational databases have robust user management features, and cryptographically secure keys are used as controls for cloud storage.

Unlike a tenant in an apartment building, you won't be running into neighbors, and won't need to remember their names. If tenant isolation is successful, you operate under the illusion that you are the only tenant.

Performance Management

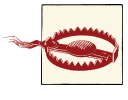
Applications in a multitenant environment compete for system resources. The cloud platform is responsible for fairly managing competing resource needs among tenants. The goal is to achieve high hardware utilization in all service instances without compromising the performance or behavior of the tenants. One strategy employed is to enforce quotas on individual tenants to prevent them from overwhelming specific shared resources. Another strategy is to deploy resource-hungry tenants alongside ten-

ants with low resource demands. Of course resource needs are dynamic and therefore unpredictable. The cloud platform is continuously monitoring, reorganizing (moving tenants around), and horizontally scaling service instances—but it's all done transparently. See *Auto-Scaling Pattern* (Chapter 4).

This type of automated performance management is less common in the non-cloud world, but the approach is important to understand as it will impact your cloud-native application.

Impact of Multitenancy on Application Logic

While the cloud platform can do a very good job of monitoring active tenants and continually rebalancing resources, there are scenarios where a burst of activity can temporarily overwhelm a service instance. This can happen when multiple applications get really busy all of a sudden. What happens? The cloud platform will proactively decide how to redistribute tenants as needed, but in the meantime (usually a few seconds to a few minutes), attempts to access these resources may experience transient failures that manifest as busy signals. For more information about responding to *transient failures* or busy signals, refer to *Busy Signal Pattern* (Chapter 9).



Multitenancy services get busy, occasionally responding to service calls with a busy signal. Plan on it happening to your application and plan on handling it.

Commodity Hardware

Cloud platforms are built using commodity hardware. In contrast to either low-end hardware or high-end hardware, commodity hardware is in the middle, chosen because it has the most attractive value-to-cost ratio; it's the-biggest-bang-for-the-buck hardware. High-end hardware is more expensive than commodity hardware. Twice as much memory and twice as many CPU cores typically will be more than twice the total cost. A dominant driver for using cloud data centers is cost-efficiency.

Data Center Is a Competitive Differentiator

It is not credible to claim that traditional data centers were developed without cost concerns. But with more heterogeneous and higher-end hardware populating those data centers, the emphasis was certainly different. These data centers were there to serve as home for the applications we already had on the hardware we were already using, optimized for individual vertically scaling applications rather than the far more ambitious goal of optimizing across all applications.

> The larger cloud platform vendors are tackling this ambitious goal of optimizing across the whole data center. While Windows Azure, Amazon Web Services, and other cloud platforms support virtual machine rentals that can run legacy software on Windows Server or Linux, the greatest runtime efficiency lies with cloud-native applications. This model should become attractive to more and more customers over time as it becomes increasingly cost-efficient as cloud platform vendors drive further efficiencies and pass along the cost savings.
>
> In particular, Microsoft enjoys economies of scale not available to most companies. Partly this is because it is a very large technology company in its own right, but also stems from its broad, mature product lines and platforms. By methodically updating its own internal applications and existing products to leverage Windows Azure, while also adding new cloud offerings, Microsoft benefits from a practice known as *eating your own dogfood*, or *dogfooding*. Through dogfooding, Microsoft's internal product teams use the Windows Azure platform as customers would, identify feature gaps or other concerns, and then work directly with the Windows Azure team so that more features can be developed using real world scenarios, resulting in a more mature platform sooner than might otherwise be possible.
>
> The largest cloud platform vendors are in a battle to produce and offer advanced features more efficiently than their competitors so that they can offer competitive pricing. Although I don't know which cloud platform vendors will win in the end (and I don't envision a world where Windows Azure and Amazon Web Services aren't both big players), the clear winners in this battle are the customers—that's us.

This is an economic decision that helps optimize for cost in the cloud. The main challenge to applications is that commodity hardware fails more frequently than high-end hardware.

Shift in Emphasis from MTBF to MTTR

The ethos in the traditional application development world emphasized minimizing the *mean time between failures (MTBF)*, meaning that we worked hard to ensure that hardware did not fail. This translated into high-end hardware, redundant components (such as RAID disk drives and multiple power supplies), and redundant servers (such as secondary servers that were not put into use unless the primary server failed for the most critical systems). On occasion when hardware did fail, the application was down until a human fixed the problem. It was expensive and complex to build software that effortlessly survived a hardware failure, so for the most part we attacked that problem with hardware.

The new ethos in the cloud-native world emphasizes minimizing the *mean time to recovery (MTTR)*, meaning that we work hard to ensure that when hardware fails, only

some application capacity is impacted, and the application keeps on working. In concert with patterns in this book and in alignment with the services offered by the major cloud platforms, this approach is not only viable, but also attractive due to the great reduction in complexity and new economic efficiencies.

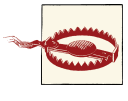
Hardware Failure Is Inevitable, but Not Frequent

Discussion of recovering from failures in commodity hardware can be misleading. Just because commodity hardware fails *more frequently* than high-end hardware does not mean it fails *frequently*. Hardware failures impact only a small percentage of the commodity servers in the data center every year. But be ready: eventually it will be your turn.

The cloud platform assumes that much of the MTTR duties are completed through automation, but also imposes requirements on your application, forming something of a partnership in handling failure.

Impact of Commodity Hardware on Application Logic

Cloud-native applications expect failure and are able to detect and automatically recover from common scenarios. Some of these failure scenarios are present because the application is relying on commodity hardware.



Commodity hardware fails occasionally. Plan on it happening to your compute nodes and plan on handling it.

Failure may simply be due to an issue with a specific physical server such as bad memory, a crashed disk drive, or coffee spilled on the motherboard. Other scenarios originate from software failures. For more information about responding to failures at the individual node level, refer to *Node Failure Pattern* (Chapter 10).

The failure scenario just described may be obvious: your application code is running on commodity hardware, and when that hardware fails your application is impacted. What is less obvious is that cloud services on which your application also depends (databases, persistent file storage, messaging, and so on) on are also running on commodity hardware. When these services experience a disruption due to a hardware failure, your application may also be impacted. In many scenarios, the cloud platform service recovers without any visible degradation, but sometimes capacity is temporarily reduced, forcing the calling application to handle transient failure. For more information about responding to failures encountered during calls to a cloud service, refer to Chapter 3.

Homogeneous Hardware

Cloud data centers also strive to use homogeneous hardware for easier management and maintenance of resources. Procurement of large scale homogeneous hardware is possible through inexpensive and readily available commodity hardware.

The level of homogeneity in the hardware is unlikely to directly impact applications as long as the allocated capacity in a virtual machine remains predictable.

Homogeneous Hardware Benefits in the Real World

Southwest Airlines is one of the most consistently profitable airlines in the world, in part fueled by their insistence on homogeneous commodity hardware: the Boeing 737. This is the only type of plane in the whole fleet, vastly reducing complexity in breadth of skills needed by mechanics and pilots, streamlining parts inventory, and probably even simplifying software that runs the airlines since there are fewer differences between flights.

Summary

Cloud platform vendors make choices around cost-efficiency that directly impact the architecture of applications. Architecting to deal with failure is part of what distinguishes a cloud-native application from a traditional application. Rather than attempting to shield the application from all failures, dealing with failure is a shared responsibility between the cloud platform and the application.

CHAPTER 9

Busy Signal Pattern

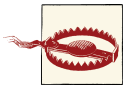
This pattern focuses on how an application should react when a cloud service responds to a programmatic request with a busy signal rather than success.

This pattern reflects the perspective of a client, not the service. The client is programmatically making a request of a service, but the service replies with a busy signal. The client is responsible for correct interpretation of the busy signal followed by an appropriate number of retry attempts. If the busy signals continue during retries, the client treats the service as unavailable.

Dialing a telephone occasionally results in a busy signal. The normal response is to retry, which usually results in a successful telephone call.

Similarly, invoking a service occasionally results in a failure code being returned, indicating the cloud service is not currently able to satisfy the request. The normal response is to retry which usually results in the service call succeeding.

The main reason a cloud service cannot satisfy a request is because it is too busy. Sometimes a service is "too busy" for just a few hundred milliseconds, or one or two seconds. Smart retry policies will help handle busy signals without compromising user experience or overwhelming busy services.



Applications that do not handle busy signals will be unreliable.

Context

The Busy Signal Pattern is effective in dealing with the following challenges:

- Your application uses cloud platform services that are not guaranteed to respond successfully every time

This pattern applies to accessing cloud platform services of all types, such as management services, data services, and more.

More generally, this pattern can be applied to applications accessing services or resources over a network, whether in the cloud or not. In all of these cases, periodic transient failures should be expected. A familiar non-cloud example is when a web browser fails to load a website fully, but a simple refresh or retry fixes the problem.

Cloud Significance

For reasons explained in *Multitenancy and Commodity Hardware Primer* (Chapter 8), applications using cloud services will experience periodic transient failures that result in a busy signal response. If applications do not respond appropriately to these busy signals, user experience will suffer and applications will experience errors that are difficult to diagnose or reproduce. Applications that expect and plan for busy signals can respond appropriately.

The pattern makes sense for robust applications even in on-premises environments, but historically has not been as important because such failures are far less frequent than in the cloud.

Impact

Availability, Scalability, User Experience

Mechanics

Use the Busy Signal Pattern to detect and handle normal transient failures that occur when your application (the *client* in this relationship) accesses a cloud service. A *transient failure* is a short-lived failure that is not the fault of the client. In fact, if the client reissues the identical request only milliseconds later, it will often succeed.

Transient failures are expected occurrences, not exceptional ones, similar to making a telephone call and getting a busy signal.

Busy Signals Are Normal

Consider making a phone call to a call center where your call will be answered by one of hundreds of agents standing by. Usually your call goes through without any problem, but not every time. Occasionally you get a busy signal. You don't suspect anything is wrong, you simply hit redial on your phone and usually you get through. This is a transient failure, with an appropriate response: retry.

However, many consecutive busy signals will be an indicator to stop calling for a while, perhaps until later in the day. Further, we only will retry if there is a true busy signal. If we've dialed the wrong number or a number that is no longer in service, we do not retry.

Although network connectivity issues might sometimes be the cause of transient failures, we will focus on transient failures at the service boundary, which is when a request reaches the cloud service, but is not immediately satisfied by the service. This pattern applies to any cloud service that can be accessed programmatically, such as relational databases, NoSQL databases, storage services, and management services.

Transient Failures Result in Busy Signals

There are several reasons for a cloud service request to fail: the requesting account is being too aggressive, an overall activity spike across all tenants, or it could be due to a hardware failure in the cloud service. In any case, the service is proactively managing access to its resources, trying to balance the experience across all tenants, and even reconfiguring itself on the fly in reaction to spikes, workload shifts, and internal hardware failures.

Cloud services have limits; check with your cloud vendor for documentation. Examples of limits are the maximum number of service operations that can be performed per second, how much data can be transferred per second, and how much data can be transferred in a single operation.

In the first two examples, operations per second and data transferred per second, even with no individual service operation at fault it is possible that multiple operations will cumulatively exceed the limits. In contrast, the third example, amount of data transfer-

red in a single operation, is different. If this limit is exceeded, it will not be due to a cumulative effect, but rather it is an invalid operation that should always be refused. Because an invalid operation should always fail, it is different from a transient failure and will not be considered further with this pattern.

Handling Busy Signals Does Not Replace Addressing Scalability Challenges

For cloud services, limits are not usually a problem except for very busy applications. For example, a Windows Azure Storage Queue is able to handle up to 500 operations per second for any individual queue. If your application needs to sustain more than 500 queue operations per second on an individual queue, this is no longer a transient failure, but rather a scalability challenge. Techniques for overcoming such a scalability challenge are covered under *Horizontally Scaling Compute Pattern* (Chapter 2) and *Auto-Scaling Pattern* (Chapter 4).

Limits in cloud services can be exceeded by an individual client or by multiple clients collectively. Whenever your use of a service exceeds the maximum allowed throughput, this will be detected by the service and your access would be subject to throttling. *Throttling* is a self-defense response by services to limit or slow down usage, sometimes delaying responses, other times rejecting all or some of an application's requests. It is up to the application to retry any requests rejected by the service.

Multiple clients that do not exceed the maximum allowed throughput *individually* can still exceed throttling limits *collectively*. Even though no individual client is at fault, aggregate demand cannot be satisfied. In this case the service will also throttle one or more of the connected clients. This second situation is known as the *noisy neighbor problem* where you just happen to be using the same service instance (or virtual machine) that some other tenant is using, and that other tenant just got real busy. You might get throttled even if, technically, you do nothing wrong. The service is so busy it needs to throttle *someone*, and sometimes that someone is you.

Cloud services are dynamic; a usage spike caused by a bunch of noisy neighbors might be resolved milliseconds later. Sustained congestion caused by multiple active clients who, as individuals, are compliant with rate limits, should be handled by the sophisticated resource monitoring and management capabilities in the cloud platform. Resource monitoring should detect the issue and resolve it, perhaps by spreading some of the load to other servers.

Cloud services also experience internal failures, such as with a failed disk drive. While the service automatically repairs itself by failing over to a healthy disk drive, redirecting

traffic to a healthy node, and initiating replication of the data that was on the failed disk (usually there are three copies for just this kind of situation), it may not be able to do so instantaneously. During the recovery process, the service will have diminished capacity and service calls are more likely to be rejected or time out.

Recognizing Busy Signals

For cloud services accessed over HTTP, transient failures are indicated by the service rejecting the request and usually responded to with an appropriate HTTP status code such as: *503 Service Unavailable*. For a relational database service accessed over TCP, the database connection might be closed. Other short-lived service outages may result in different error codes, but the handling will be similar. Refer to your cloud service documentation for guidance, but it should be clear when you have encountered a transient failure and documentation may also prescribe how best to respond. Handle (and log) unexpected status codes.

It is important that you clearly distinguish between busy signals and errors. For example, if code is attempting to access a resource and the response indicates it has failed because the resource does not exist or the caller does not have sufficient permissions, then retries will not help and should not be attempted.

Responding to Busy Signals

Once you have detected a busy signal, the basic reaction is to simply retry. For an HTTP service, this just means reissuing the request. For a database accessed over TCP, this may require reestablishing a database connection and then reissuing the query.

How should your application respond if the service fails again? This depends on circumstances. Some responses to consider include:

- Retry immediately (no delay).
- Retry after delay (fixed or random delay).
- Retry with increasing delays (linear or exponential backoff) with a maximum delay.
- Throw an exception in your application.

Access to a cloud service involves traversing a network that already introduces a short delay (longer when accessing over the public Internet, shorter when accessing within a data center). A *retry immediately* approach is appropriate if failures are rare and the documentation for the service you are accessing does not recommend a different approach.

When a service throttles requests, multiple client requests may be rejected in a short time. If all those clients retry quickly at the same time, the service may need to reject many of them again. A *retry after delay* approach can give the service a little time to clear its queue or rebalance. If the duration of the delay is random (e.g., 50 to 250ms), retries to the busy service across clients will be more distributed, improving the likelihood of success for all.

The least aggressive retry approach is *retry with increasing delays*. If the service is experiencing a temporary problem, don't make it worse by hammering the service with retry requests, but instead get less aggressive over time. A retry happens after some delay; if further retries are needed, the delay is increased before each successive retry. The delay time can increase by a fixed amount (*linear backoff*), or the delay time can, for example, double each time (*exponential backoff*).

Cloud platform vendors routinely provide client code libraries to make it as easy as possible to use your favorite programming language to access their platform services. Avoid duplication of effort: some client libraries may already have retry logic built in.

Regardless of the particular retry approach, it should limit the number of retry attempts and should cap the backoff. An aggressive retry may degrade performance and overly tax a system that may already be near its capacity limits. Logging retries is useful for analysis to identify areas where excessive retrying is happening.

After some reasonable number of delays, backoffs, and retries, if the service still does not respond, it is time to give up. This is both so the service can recover and so the application isn't locked up. The usual way for application code to indicate that it cannot do its job (such as store some data) is to throw an exception. Other code in the application will handle that exception in an application-appropriate manner. This type of handling needs to be programmed into every cloud-native application.

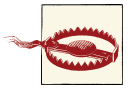
User Experience Impact

Handling transient failures sometimes impacts the user experience. The details of handling this well are specific to every application, but there are a couple of general guidelines.

The choice of a retry approach and the maximum number of retry attempts should be influenced by whether there is an interactive user waiting for some result or if this is a batch operation. For a batch operation, exponential backoff with a high retry limit may make sense, giving the service time to recover from a spike in activity, while also taking advantage of the lack of interactive users.

With an interactive user waiting, consider several retries within a small interval before informing the user that "the system is too busy right now – please try again later". The social networking service Twitter is well-known for this behavior. Consider the *Queue-Centric Workflow Pattern* (Chapter 3) for ways to decouple time-consuming work from the user interface.

When a service does not succeed within a reasonable time or number of retries, your application should take action. Though unsatisfying, sometimes passing the information back to the user is a reasonable approach, such as "the server is busy and your update will be retried in ten seconds". (This is similar to how Google Mail and Quora handle temporary network connectivity issues in their web user interfaces.)



Be careful with server-side code that ties up resources while retrying some operation, even when that code is retrying in an attempt to improve the user experience. If a busy web application has lots of user requests, each holding resources during retries, this could bump up against other resource constraints, reducing scalability.

Logging and Reducing Busy Signals

Logging busy signals can be helpful in understanding failure patterns. Robustly tracking and handling transient failures is extremely important in the cloud due to the innate challenges in debugging and managing distributed cloud applications.

Analysis of busy signal logs can lead to changes that will reduce future busy signals. For example, analysis may reveal that busy signals trend higher when accessing a cloud database service. Remember, the cloud provides the illusion of infinite resources, but this does not mean that each resource has infinite capacity. To access more capacity, you must provision more instances. When one database instance is not enough, it may be time to apply *Database Sharding Pattern* (Chapter 7).

Testing

It is common to test cloud applications in non-cloud environments. Code that runs in a development or test environment, especially at lower-than-production volumes or with dedicated hardware, may not experience the transient failures seen in the cloud. Be sure to test and load test in an environment as close to production as possible.

It is becoming more common for companies to test against the production environment because it is the most realistic. For example, load testing against production, though perhaps at non-peak times. Netflix goes even further by continually stressing their production (cloud) environment with errors using a home-grown tool they call Chaos Monkey. To ensure they can handle any kind of disruption, Chaos Monkey continually and randomly turns off services and reboots servers in the production environment.

Example: Building PoP on Windows Azure

The Page of Photos (PoP) application (which was described in the Preface) uses a number of Windows Azure cloud services. Two of these are Windows Azure SQL Databases for storing account profile information and Windows Azure Storage blobs for storing photos. Because these are cloud services, PoP code is written to handle transient failures.

The Windows Azure Platform includes software libraries for use with a variety of programming environments such as C#, Java, Node.js, Python, and mobile devices. These libraries all simplify accessing blobs by automatically detecting transient failures and retrying up to three times. There are a number of other predefined retry behaviors available, and it is also possible to create a custom retry policy. Most applications won't need anything beyond the default behavior.

Busy Signals in the Real World

How frequently do retries happen in practice? Here are some real numbers gathered in running an upload utility that saturated a network connection for an extended period of time (measured in days), pushing data over the public Internet into Windows Azure Blob storage. A custom retry policy was used to implement exponential backoff while also logging each retry attempt. During the uploading of nearly a million files averaging four megabytes in size, around 1.8% of the *files* required at least one retry. Note that because the files were so large, each file upload involved many storage operations, so retries on storage operations were needed much less frequently than 0.1% of the time. However, because logging was at the granularity of a file upload level, exact statistics are not available. Further, many factors can impact retry behavior, such as competition for the network resources locally and network connectivity across the public Internet. Your results will vary.

Applications accessing Windows Azure SQL Databases use the same TCP protocol used for accessing SQL Server. However, a more robust approach to transient failure detection and retry logic is needed. This comes in the form of a library known as the Transient Fault Handling Application Block, also known as Topaz. Like the retry support in the

libraries mentioned previously for blob access, Topaz has some predefined retry behaviors, but also a rich model for customization. The easiest way to take advantage of Topaz features when writing database access code is to replace standard database objects with the transient-failure-aware equivalents that ship with Topaz. For example, use a `ReliableSqlConnection` object in place of the standard .NET `SqlConnection` object.

Some SQL Database transient failures are due to throttling and inform you of this by returning a well-known error code. Sometimes SQL Database will drop your database connection, requiring that you reconnect.

Expect Your Database Connection to Drop

Why would SQL Database drop your connection? When you connect, behind the scenes it is not a single instance but rather a cluster of three collaborating servers. This provides multiple benefits. One benefit is resilience to disk drive failure as each byte written to SQL Database is written to all three servers within the cluster. Another benefit is that, as a multitenant service, SQL Database can distribute (and redistribute) load across servers to keep it balanced. These benefits come at a cost. If SQL Database chooses your connection to move to another of the three servers in the cluster, it needs to first disconnect you; when you reconnect, you will connect to your newly assigned cluster node, though you won't be able to tell the difference.

While the focus here has been on using Topaz with SQL Database, it also supports Windows Azure Storage, Windows Azure Caching, and Windows Azure Service Bus, including advanced retry possibilities if you need them.

Related Chapters

- *Queue-Centric Workflow Pattern* (Chapter 3)
- *Multitenancy and Commodity Hardware Primer* (Chapter 8)
- *Node Failure Pattern* (Chapter 10)

Summary

Handling transient failures is essential for building reliable cloud-native applications. Using the Busy Signal Pattern, your application can detect and handle transient failures appropriately. Further, your approach can be tuned for batch or interactive user scenarios.

It may be difficult to test your application's response to transient failure conditions if running on non-cloud hardware or with an unrealistically light load.

CHAPTER 10

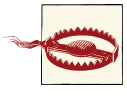
Node Failure Pattern

This pattern focuses on how an application should respond when the compute node on which it is running shuts down or fails.

This pattern reflects the perspective of application code running on a node (virtual machine) that is shut down or suddenly fails due to a software or hardware issue. The application has three responsibilities: prepare the application to minimize issues when nodes fail, handle node shutdowns gracefully, and recover once a node has failed.

Some common reasons for shutdown are unresponsive application due to application failure, routine maintenance activities managed by the cloud vendor, and auto-scaling activities initiated by the application. Failures might be caused by hardware failure or an unhandled exception in your application code.

While there are many reasons for a node shutdown or failure, we can still treat them uniformly. Handling the various forms of failure is sufficient; all shutdown scenarios will also be handled. The pattern name derives from the more encompassing node failures.



Applications that do not handle node shutdowns and failures will be unreliable.

Context

The Node Interruption Pattern is effective in dealing with the following challenges:

- Your application is using the Queue-Centric Workflow Pattern and requires *at-least-once processing* for messages sent across tiers

- Your application is using the Auto-Scaling Pattern and requires graceful shutdown of compute nodes that are being released
- Your application requires graceful handling of sudden hardware failures
- Your application requires graceful handling of unexpected application software failures
- Your application requires graceful handling of reboots initiated by the cloud platform
- Your application requires sufficient resiliency to handle the unplanned loss of a compute node without downtime

All of these challenges result in interruptions that need to be addressed and many of them have overlapping solutions.

Cloud Significance

Cloud applications will experience failures that result in node shutdowns. Cloud-native applications expect these and respond appropriately when notified of shutdowns by the cloud platform.

Impact

Availability, User Experience

Mechanics

The goal of this pattern is to allow individual nodes to fail while the application as a whole remains available. There are several categories of failure scenarios to consider, but all of them share characteristics for preparing for failure, handling node shutdown (when possible), and then recovering.

Failure Scenarios

When your application code runs in a virtual machine (especially on commodity hardware and on a public cloud platform), there are a number of scenarios in which a shutdown or failure could occur. There are other potential scenarios, but from the point of view of the application, the scenarios always look like one of those listed in Table 10-1.

Table 10-1. Node Failure Scenarios

Scenario	Initiated By	Advanced Warning	Impact
Sudden failure + restart	Sudden hardware failure	No	Local data is lost

Scenario	Initiated By	Advanced Warning	Impact
Node shutdown + restart	Cloud platform (vacate failing hardware, software update)	Yes	Local data may be available
Node shutdown + restart	Application (application update, application bug, managed reboot)	Yes	Local data is available
Node shutdown + termination	Application (node released such as via auto-scale)	Yes	Local data is lost

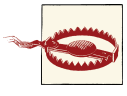


Table 10-1 is correct for the Windows Azure and Amazon Web Services platforms. It is not correct for every platform, though the pattern is still generally applicable.

The scenarios have a number of triggers listed; some provide advanced warning, some don't. Advanced warning can come in multiple forms, but amounts to a proactive signal, sent by the cloud platform that allows the node to gracefully shut down. In some cases, local data written to the local virtual machine disk is available after the interruption, sometimes it is lost. How do we deal with this?

Scenarios can be initiated by hardware or software failures, by the cloud platform, or by your application.

Treat All Interruptions as Node Failures

None of the scenarios are predictable from the point of view of the individual node. Further, the application code does not know which scenario it is in, but it can handle all of the scenarios gracefully if it treats them all as failures.

All application compute nodes should be ready to fail at any time.

Given that failure can happen at any time, your application should not use the local disk drive of the compute node as the system of record for any business data. As mentioned in Table 10-1, it can be lost. Remember that an application using stateless nodes is not the same as a stateless application. Application state is persistent in reliable storage, not on the local disk of an individual node.

By treating all shutdown and failure scenarios the same, there is a clear path for handling them: maintain capacity, handle node shutdowns, shield users when possible, and resume work-in-progress after the fact.

Maintain Sufficient Capacity for Failure with N+1 Rule

An application prepares first by assuming node failures will happen, then taking proactive measures to ensure failures don't result in application downtime. The primary proactive measure is to ensure sufficient capacity.

How many node instances are needed for high availability? Apply the *N+1 rule*: If N nodes are needed to support user demand, deploy N+1 nodes. One node can fail or be interrupted without impact to the application. It is especially important to avoid any single points of failure for nodes directly supporting users. If a single node is required, deploy two.

The *N+1 rule* should be considered and applied independently for each type of node. For example, some types of node can experience downtime without anyone noticing or caring. These are good candidates for not applying the *N+1 rule*.

A buffer more extensive than a single node may be needed, though rarely, due to an unusual failure. A severe weather-related incident could render an entire data center unavailable, for example. A top-of-rack switch failure is discussed in the *Example* and cross-data center failover is touched on in *Multisite Deployment Pattern* (Chapter 15).

There is a time lag between the failure occurrence and the recognition of that failure by the cloud platform monitoring system. Your service will run at diminished capacity until a new node has been fully deployed. For nodes configured to accept traffic through the cloud platform load balancer, traffic will continue to be routed there; once the failed node is recognized it will be promptly removed from the load balancer, but there will still be more time until the replacement node is booted up and added to the load balancer. See *Auto-Scaling Pattern* (Chapter 4) for a discussion of borrowing already-deployed resources from less critical functions to service more important ones during periods of diminished capacity.

Handling Node Shutdown

We can responsibly handle node shutdown, but there is no equivalent handling for sudden node failure because the node just stops in an instant. Not all hardware failures result in node failure: the cloud platforms are adept at detecting signals that hardware failure is imminent and can preemptively initiate a controlled shutdown to move tenants to different hardware.

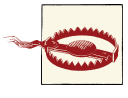
Some node failures are also preventable: is your application code handling exceptions? At a minimum, do this for logging purposes.

Regardless of the reason the node is shutting down, the goals are to allow in-progress work to complete, gather operational data from that node before shutdown completes, and do so without negatively impacting the user experience.

Cloud platforms provide signals to let running applications know that a node is either about to be, or is in the process of being, shut down. For example, one of the signals provided by Amazon Web Services is an advanced alert, and one of the signals provided by Windows Azure is to raise an event within the node indicating that shutdown has begun.

Node shutdown with minimal impact to user experience

Cloud platforms managing both load balancing and service shutdown stop sending web traffic to nodes they are in the process of shutting down. This will prevent many types of user experience issues, as new web requests will be routed to other instances. Only those pending web requests that will not be complete by the time the service shuts down are problematic. Some applications will not have any problem as all requests are responded to nearly instantly.



If your application is using sticky sessions, new requests will not always be routed to other instances. This topic was covered in *Horizontally Scaling Compute Pattern* (Chapter 2). However, cloud-native applications avoid sticky sessions.

It is possible that a node will begin shutting down while some work visible to a user is in progress. The longer it takes to satisfy a web request, the more of a problem this can be. The goal is to avoid having users see server errors just because they were accessing a web page when a web node shuts down while processing their request. The application should wait for these requests to be satisfied before shutting down. How to best handle this is application-, operating system-, and cloud platform-specific; an example for a Windows Cloud Service running on Windows Azure is provided in the *Example*.

Failures that result in sudden termination can cause user experience problems if they happen in nodes serving user requests. One tactic for minimizing this is to design the user interface code to retry when there are failures, as described in *Busy Signal Pattern* (Chapter 9). Note that the busy signal here is not coming from a cloud platform service, but the application's own internal service.

Node shutdown without losing partially completed work

The node should immediately stop pulling new items from the queue once shutdown is underway and let current work finish if possible.

Queue-Centric Workflow Pattern (Chapter 3) describes processing services that can sometimes be lengthy. If processing is too lengthy to be completed before the node is terminated, that node has the option of saving any in-progress work outside of the local node (such as to cloud storage) in a form that can be resumed.

Node shutdown without losing operational data

Web server logs and custom application logs are typically written to local disk on individual nodes. This is a reasonable approach, as it makes efficient use of available local resources to capture this log data.

As described in *Horizontally Scaling Compute Pattern* (Chapter 2), part of managing many nodes is the challenge of consolidating operational data. Usually this data is collected periodically. If the collection process is not aware that a node is in the process of shutdown, that data may not be collected in time, and may be lost. This is your opportunity to trigger collection of that operational data.

Recovering From Node Failure

There are aspects to recovering from node failure: maintaining a positive user experience during the failure process, and resuming any work-in-progress that was interrupted.

Shielding interactive users from failures

Some node instances support interactive users directly. These nodes might be serving up web pages directly or providing web services that power mobile apps or web applications. A positive user experience would shield users from the occasional failure. This happens in a few ways.

If a node fails on the server, the cloud platform will detect this and stop sending traffic to that node. Once detected, service calls and web page requests will be routed to healthy node instances. You don't need to do anything to make this happen (other than follow the *N+1 rule*), and this is sometimes good enough.

However, to be completely robust, you should consider the small window of exposure to failure that can occur while in the middle of processing a request for a web page or while executing a service call. For example, the disk drive on the node may suddenly fail. The best response to this edge case is to place retry logic on the client, basically applying *Busy Signal Pattern* (Chapter 9). This is straightforward for a native mobile app or a single-page web application managing updates through web service calls (Google Mail is an example of a single-page web application that handles this scenario nicely). For a traditional web application that redisplays the entire page each time, it is possible that users could see an error surface through their web browser. This is out of your control. It will be up to the users to retry the operation.

Don't Be Overprotective

The strategy of shielding users from errors only makes sense up to a point. Eventually you will need to communicate errors to the user, though "service is not available, please retry

in a couple of minutes" is sometimes the best we can do. Further, make sure any retry policies are appropriate for interactive users; in other words, a policy that retries 10 times with a 30-second sleep in between may be fine for a nightly batch process, but totally inappropriate when supporting an interactive user.

Resuming work-in-progress on backend systems

In addition to user experience impact, sudden node failure can interrupt processing in the service tier. This can be robustly handled by building processes to be idempotent so they can safely execute multiple times on the same inputs. Successful recovery depends on nodes being stateless and important data being stored in reliable storage rather than on a local disk drive of the node (assuming that disk is not backed by reliable storage). A common technique for cloud-native applications is detailed in *Queue-Centric Workflow Pattern* (Chapter 3), which covers restarting interrupted processes as well as saving in-progress work to make recovery faster.

Example: Building PoP on Windows Azure

Page of Photos (PoP) application (which was described in the Preface) is architected to provide a consistently reliable user experience and to not lose data. In order to maintain this through occasional failures and interruptions, PoP prepares for failure, handles role instance (node) shutdowns gracefully, and recovers from failures as appropriate.

Preparing PoP for Failure

PoP is constantly increasing and decreasing capacity to take advantage of the cost savings; we only want to pay for the capacity needed to run well, and no more. PoP also does not wish to sustain downtime or sacrifice the user experience due to diminished capacity in the case of an occasional failure or interruption.

N+1 rule

The *N+1 rule* is honored for roles that directly serve users, because user experience is very important. However, because no users are directly impacted by occasional interruptions, PoP management made the business decision to not apply the *N+1 rule* for worker roles which make up the service tier.

These decisions are accounted for in PoP auto-scaling rules.

Windows Azure fault domains

Windows Azure (through a service known as the Windows Azure Fabric Controller) determines where in the data center to deploy each role instance of your application

within specific constraints. The most important of these constraints for failure scenarios is a fault domain. A *fault domain* is a potential single point of failure (SPoF) within a data center, and any roles with a minimum of two instances are guaranteed to be distributed across at least two fault domains.

The discerning reader will realize that this means up to half of your application's web and worker role instances can go down at once (assuming two fault domains). While it is possible, it is unlikely. The good news is that if it does happen, the Fabric Controller begins immediate remediation, although your application will run with diminished capacity during recovery.

Auto-Scaling Pattern (Chapter 4) introduced the idea of internally throttling some application features during periods of diminished or insufficient resources. This is another scenario where that tactic may be useful. Note that you may not want to auto-scale in response to such a failure, but rather allow the Fabric Controller to recover for you.

For the most part, a hardware failure results in the outage of a single role instance. The *N+1* rule accounts for this possibility.

PoP makes the business decision that N+1 is sufficient, as described previously.

When N + 1 Is Not Enough

If a failure impacts more than a single hardware node (for example, a failure at the rack level), N+1 may not be enough, depending on your business requirements. If this level of potential (albeit rare) downtime is too much, consider even more capacity, depending on how many fault domains are in effect. Currently, all Windows Azure cloud services run with two fault domains and (as of this writing) the number cannot be changed. Extra capacity for critical times can be scheduled as a proactive auto-scaling task. Also consider *Multisite Deployment Pattern* (Chapter 15).

Note that fault domains only apply with unexpected hardware failures. For operating system updates, hypervisor updates, or application-initiated upgrades, downtime is distributed across upgrade domains.

Upgrade domains

Conceptually similar to fault domains, Windows Azure also has *upgrade domains* that provide logical partitioning of role instances into groups (by default, five) which are considered to be independently updatable. This goes hand-in-hand with the *in-place upgrade* feature, which upgrades your application in increments, one upgrade domain at a time.

If your application has N upgrade domains, the Fabric Controller will manage the updates such that 1/N role instances will be impacted at a time. After these updates complete, updating the next 1/N role instances can begin. This continues until the application is completely updated. Progressing from one upgrade domain to the next can be automated or manual.

Upgrade domains are useful to the Fabric Controller since it uses them during operating system upgrades (if the customer has opted for automatic updates), hypervisor updates, emergency moves of role instances from one machine or rack to another, and so forth. Your application uses upgrade domains if it updates itself using the *in-place upgrade* process where the application is upgraded one upgrade domain at a time. You may choose to manually approve the progression from one upgrade domain to the next, or have the Fabric Controller proceed automatically.

Multiple Concurrent Versions

Cloud-native applications commonly have a single deployment with mixed releases (and features) across the role instances. This staged upgrade can be managed using upgrade domains with an in-place upgrade. Another way to look at this: an application upgrade does not happen at a particular time, rather it spans time. During that time span the application is running two versions. Two customers visiting the application at the same time may be routed to different versions. If you wish to avoid deploying multiple concurrent versions, look into Windows Azure's VIP Swap feature.

What is the right number of upgrade domains for my application? There is a tradeoff: more upgrade domains take longer to roll out, while fewer upgrade domains increase the level of diminished capacity. For PoP, the default of five upgrade domains works well.

Handling PoP Role Instance Shutdown

Graceful role shutdown avoids loss of operational data and degradation of the user experience.

When we instruct Windows Azure to release a role instance, there is a defined process for shutdown; it does not shut off immediately. In the first step, Windows Azure chooses the role instance that will be terminated

You don't get to decide which role instances will be terminated.

First, let's consider terminating a web role instance.

Web role instance shutdown

Once a specific web role instance is selected for termination, Windows Azure (specifically the Fabric Controller) removes that instance from the load balancer so that no new requests will be routed to it. The Fabric Controller then provides a series of signals to the running role that a shutdown is coming. The last of these signals is to invoke the role's `OnStop` method, which is allowed five minutes for a graceful shutdown. If the instance finishes the cleanup within five minutes, it can return from `OnStop` at that point; if your instance does not return from `OnStop` within five minutes, it will be forcibly terminated. Only after `OnStop` has completed does Azure consider your instances to be released for billing purposes, providing an incentive for you to make `OnStop` exit as efficiently as possible. However, there are a couple of steps you could take to make the shutdown more graceful.

Although the first step in the shutdown process is to remove the web role from the load balancer, it may have existing web requests being handled. Depending on how long it takes to satisfy individual web requests, you may wish to take proactive steps to allow these existing web requests to be satisfied before allowing the shutdown to complete. One example of a slower-than-usual operation is a user uploading a file, though even routine operations may need to be considered if they don't complete quickly enough. A simple technique to ensure your Web Role is gracefully shutting down is to monitor an IIS performance counter such as *Web Service\Current Connections* in a loop inside your `OnStop` method, continuing to termination only when it has drained to zero.

Application diagnostic information from role instances is logged to the current virtual machine and collected by the Diagnostics Manager on a schedule of your choosing, such as every ten minutes. Regardless of your schedule, triggering an unscheduled collection within your `OnStop` method will allow you to wrap up safely and exit as soon as you are ready. An on-demand collection can be initiated using Service Management features.

Worker role instance shutdown

Gracefully terminating a worker role instance is similar to terminating a web role instance, although there are some differences. Because Azure does not load balance PoP worker role instances, there are no active web requests to drain. However, triggering an on-demand collection of application diagnostic information does make sense, and you would usually want to do this.

Although PoP worker role instances do not have public-facing endpoints, Windows Azure does support it, so applications may have active connections when instance shutdown begins. A worker role can define public-facing endpoints that will be load balanced. This feature is useful in running your own web server such as Apache, Mongoose, or Nginx. It may also be useful for exposing self-hosted WCF endpoints. Further, public endpoints are not limited to HTTP, as TCP and UDP are also supported.

A worker role instance will want to stop pulling new items from the queue for processing as soon as possible once shutdown is underway. As with the web role, instance shutdown signals such as `OnStop` can be useful; a worker role could set a global flag to indicate that the role instance is shutting down. The code that pulls new items from the queue could check that flag and not pull in new items if the instance is shutting down. Worker roles can use `OnStop` as a signal, or hook an earlier signal such as the `OnStopping` event.

If the worker role instance is processing a message pulled from a Windows Azure Storage queue and it does not complete before the instance is terminated, that message will time out and be returned to the queue so it can be pulled again. (Refer to *Queue-Centric Workflow Pattern* (Chapter 3) for details.) If there are distinct steps in processing, the underlying message can be updated on the queue to indicate progress. For example, PoP processes newly uploaded photos in three steps: extract geolocation information from the photo, create two sizes of thumbnail, and update all SQL Database references. If two of these steps were completed, but not the third, it is indicated by a `LastCompletedStep` field in the message object that was pulled from the queue, and that message object can be used to update the copy on the queue. When this message is pulled again from the queue (by a subsequent role instance), processing logic will know to skip the first two steps because `LastCompletedStep` field is set to two. The first time a message is pulled, the value is set to zero.

Use controlled reboots

Sometimes you may wish to reboot a role instance. The controlled shutdown that allows draining active work is supported if reboots are triggered using the Reboot Role Instance operation in the Windows Azure Service Management service.

If other means of initiating a reboot are used, such as directly on the node using Windows commands, Windows Azure is not able to manage the process to allow for a graceful shutdown.

Recovering PoP From Failure

Queue-Centric Workflow Pattern (Chapter 3) details recovering from a long-running process.

Otherwise, there is usually little to do in recovering a stateless role instance, whether a web role or worker role.

Some user experience glitches are addressed in PoP because client-side JavaScript code that fetches data (though AJAX calls to REST services) implements *Busy Signal Pattern* (Chapter 9). This code will retry if a service does not respond, and the retry will eventually be load balanced to a different role instance that is able to satisfy the request.

Related Chapters

- *Queue-Centric Workflow Pattern* (Chapter 3)
- *MapReduce Pattern* (Chapter 6)
- *Multitenancy and Commodity Hardware Primer* (Chapter 8)
- *Busy Signal Pattern* (Chapter 9)
- *Multisite Deployment Pattern* (Chapter 15)

Summary

Failure in the cloud is commonplace, though downtime is rare for cloud-native applications. Using the Node Failure Pattern helps your application prepare for, gracefully handle, and recover from occasional interruptions and failures of compute nodes on which it is running. Many scenarios are handled with the same implementation.

Cloud applications that do not account for node failure scenarios will be unreliable: user experience will suffer and data can be lost.

CHAPTER 11

Network Latency Primer

This basic primer explains network latency and why delays due to network latency matter.

The time it takes to transmit data across a network is known as *network latency*. Network latency slows down our application.

While individual networking devices like routers, switches, wireless access points, and network cards all introduce latencies of their own, this primer blends them all together into a bigger picture view, the total delay experienced by data having to travel over the network.

As cloud application developers, we can decrease the impact of network latency through caching, compression, moving nodes closer together, and shortening the distance between users and our application.

Network Latency Challenges

Highly scalable and high performing (even infinitely fast!) servers do not guarantee that our application will perform well. This is due to the main performance challenge that lies outside of raw computational power: movement of data. Transmitting data across a network does not happen instantly and the resultant delay is known as *network latency*.

Network latency is a function of distance and bandwidth: how far the data needs to travel and how fast it moves. The challenge is that any time compute nodes, data sources, and end users are not all using a single computer, network latency comes into play; the more distribution, the greater the impact. Network quality plays an important role, although it is one you might not be able to control; quality may vary as users connect through networks from disparate locations.

Application performance and scalability will suffer if it takes too long for data to get to the client computer. The effects of network latency can also vary based on the user's geographical location: to some users of the system, it may seem blazingly fast, while to other users it may seem as slow as cold molasses (which means *really slow*).

Admittedly, understanding the actual distance traveled by the data and the effective bandwidth can be challenging. The actual path traveled is not the same as the ideal great circle path you might imagine from a map. The path from router to router is jagged, and indeed may even be impacted by the capability of individual routers, and individual legs of the route may have different bandwidths. And it can vary over time.

A simple way to estimate effective bandwidth (at some point in time) is to use ping. *Ping* is a simple but useful program that calculates the time it takes to travel from one point on the Internet to another point, by actually sending network packets, counting the nodes along the way, and showing how long it takes the packets to get to the destination and back. Some measured ping times are shown in Table 11-1. These pings originated from Boston which is in the eastern USA. The fastest networks today use fiber optic cable, which supports data transmission at around 66% of the speed of light (186,000 miles/sec × 66% = 122,760 miles/sec). As you can see from Table 11-1, shorter distances are dominated by factors other than pure distance traveled, and none are close to the theoretical maximum speed. HTTP and TCP traffic will be slower than pings. Network latency impact can add up quickly, especially as web pages get heavier with more objects to download. In *An Analysis of Application Performance Data and Trends* (Jan 2012), Compuware reports that the average (yes, *average*) page download size for a non-mobile news site is more than a megabyte and 790 KB for a non-mobile travel site.

Table 11-1. Ping Times from Boston – distance matters

Domain	Location	Elapsed Time each way (ms)	Approximate Distance (miles)	Speed (miles/sec)
www.bu.edu	Boston, USA	17	10	1,000
www.gsu.edu	Atlanta, USA	75	900	12,000
www.usc.edu	Los Angeles, USA	51	2600	52,000
www.cam.ac.uk	Cambridge, UK	50	3300	67,000
www.msu.ru	Moscow, RU	81	4500	56,000
www.u-tokyo.ac.jp	Tokyo, JP	107	6700	63,000
www.auckland.ac.nz	Auckland, NZ	115	9000	79,000

Doing the Math

How much data transfer can I get for my bandwidth? Let's do some math. Consider a T1 connection that provides 1.544 Mb/sec of bandwidth. Translating from kilo**bits** (Kb) to kilo**bytes** (KB) we end up with about 193 KB/sec (1544/8=193 kilo**bytes** per second, ig-

noring errors, retries, speed degradation, and assuming no other uses of the connection). Uploading a full CD-ROM (~660 MB) would take nearly an hour. Uploading a full DVD (~4.7 GB) would take around 7 hours. These calculations assume ideal network conditions, which of course you will never have.

Reducing Perceived Network Latency

The *perceived network latency*, the network latency as experienced by a user, can be reduced through techniques such as:

- Data compression
- Background processing, where screen updates don't happen until the data arrives (though not actually faster, it may improve a user's subjective experience, as with single page web applications)
- Predictive fetching, where data is loaded in anticipation of need (as with map tiles in a mapping app, although it is possible that not all the map tiles will be referenced)

Eventual consistency is another tool we can use, if we can serve users with slightly stale data. These are reasonable approaches for reducing the *impact* of network latency, but do not *change* the network latency. They are essentially the same for cloud-native applications as they are for non-cloud applications.

Reducing Network Latency

We can reduce network latency by:

- Moving application closer to users
- Moving application data closer to users
- Ensuring nodes within our application are close together

These reduction techniques are the topics of the Colocate, Valet Key, CDN, and Multisite Deployment patterns.

Summary

A comprehensive strategy for dealing with network latency will use multiple strategies. One set of strategies focuses on reducing the perceived network latency. Another set of strategies focuses on actually reducing network latency by shortening the distance between users and the instances of our application.

CHAPTER 12
Colocate Pattern

This basic pattern focuses on avoiding unnecessary network latency.

Communication between nodes is faster when the nodes are close together. Distance adds network latency. In the cloud, "close together" means in the same data center (sometimes even closer, such as on the same rack).

There are good reasons for nodes to be in different data centers, but this chapter focuses on ensuring that nodes that should be in the same data center actually are. Accidentally deploying across multiple data centers can result in terrible application performance and unnecessarily inflated costs due to data transfer charges.

This applies to nodes running application code, such as compute nodes, and nodes implementing cloud storage and database services. It also encompasses related decisions, such as where log files should be stored.

Context

The Colocation Pattern effectively deals with the following challenges:

- One node makes frequent use of another node, such as a compute node accessing a database
- Application deployment is basic, with no need for more than a single data center
- Application deployment is complex, involving multiple data centers, but nodes within each data center make frequent use of other nodes, which can be colocated in the same data center

In general, resources that are heavily reliant on each other should be colocated.

A multitier application generally has a web or application server tier that accesses a database tier. It is often desirable to minimize network latency across these tiers by colocating them in the same data center. This helps maximize performance between these tiers and can avoid the costs of cloud provider data transmission.

This pattern is typically used in combination with the Valet Key and CDN Patterns. Reasons to deviate from this pattern, such as proximity to consumers and overall reliability, are discussed in *Multisite Deployment Pattern* (Chapter 15).

Cloud Significance

Public cloud platforms typically offer many worldwide data center locations to which applications can be easily deployed. These data centers span continents, and sometimes multiple data centers are available within the same continent. An application can be easily deployed to and use the cloud platform services in any of these data centers. This multi-data center flexibility brings great advantages, but also introduces the risk that application code and services will be unnecessarily distributed across multiple data centers, resulting in extra costs and a degraded user experience.

Impact

Cost Optimization, Scalability, User Experience

Mechanics

When you think about it, this may seem an obvious pattern, and in many respects it is. Depending on the structure of your company's hardware infrastructure (whether a private data center or rented space), it may have been very difficult to do anything *other than* colocate databases and the servers that accessed them.

With public cloud providers, multiple data centers are typically offered across multiple continents, sometimes with more than one data center per continent or region. If you plan to deploy to a single data center, there may be more than one reasonable choice. This is good news and bad news, since it is possible (and easy) to choose any data center as a deployment target. This makes it possible to deploy a database to one data center, while the servers that access the database are deployed to a different data center.

The performance penalty of a split deployment can be severe for a data-intensive application.

Hadoop-based "big data" applications tend to be data-intensive in the extreme and require that data and compute be colocated. See *MapReduce Pattern* (Chapter 6).

Automation Helps

Enforcing colocation is really not a technological problem, but rather a process issue. However, it can be mitigated with automation.

You will want to take proactive steps to avoid accidentally splitting a deployment across data centers. Automating deployments into the cloud is a good practice, as it limits human error from repetitive tasks. If your application spans multiple data centers but each site operates essentially independently, add checks to ensure that data access is not accidentally spanning data.

Outside of automation, your cloud platform may have specific features that make co-location mistakes less likely.

Cost Considerations

Operations will generally be less expensive if your databases and the compute resources that access them are in the same data center. There are cost implications when splitting them.

As of this writing, the Amazon Web Services and Windows Azure platforms do not charge for network traffic to enter a data center, but do charge for network traffic leaving a data center (even if it is to another of their data centers). There are no traffic charges when data stays within a single data center.

Non-Technical Considerations

There may be non-technical influences on application architecture that result in databases being stored at a different location than the compute resources that access them. This topic is considered further in *Multisite Deployment Pattern* (Chapter 15).

Example: Building PoP on Windows Azure

When the Page of Photos (PoP) application (which was described in the Preface) was first developed, it made sense to deploy it and use cloud services inside of a single data center.

Windows Azure allows you to specify the target data center for any resource that is deployable to a specific data center. As examples, the target data center can be specified for code deployment (such as Web Roles, Worker Roles, or Virtual Machines) and cloud services (such as Windows Azure Storage, SQL Database, and more). When such resources will be used together, they should be colocated in the same data center.

Affinity Groups

Windows Azure goes a step further for some resources, supporting affinity groups. An *affinity group* is a logical grouping of resources tied to a data center. You can provide a custom name for an affinity group, using a name that makes sense for your business and is not simply a generic data center name. This is an example of a cloud platform feature that can help avoid colocation mistakes.

In the case of PoP, if we deploy to a single North American data center, we can have an affinity group called *PoP-North America-Production* for our production data center. Upon creation, the decision is made as to which North American data center will be chosen, which removes this as a decision point for any downstream deployments that use the *PoP-North America-Production* affinity group.

As of this writing, not all Windows Azure resources support affinity groups; currently only Windows Azure Compute and Windows Azure Storage are supported. Most notably, SQL Database is not supported, although it can still be placed in the same data center as the other resources.

While affinity groups are tied to a specific data center, they also provide a hint to Windows Azure that allows for further local optimization for supported resource types. Not only are they in the same data center, but your Windows Azure Storage, the code accessing it from web, and the worker roles are even closer together, with fewer router hops and less distance for data to traverse. This further reduces network latency.

Using the same affinity group across storage accounts and cloud services will ensure that they are all colocated in the same data center.

Operational Logs and Metrics

You will also likely be gathering operational log files with Windows Azure Diagnostics (WAD) and Windows Azure Storage Analytics, available with Windows Azure Storage accounts. Apply the same affinity group to storage as you apply to compute instances.

It is a best practice to persist WAD into a special operational storage account (different from other storage accounts within your production system) both to minimize potential for contention and to make it easier to manage access to logs and metrics. Depending on the specific type of data items, individual diagnostic values are stored in either Windows Azure Blob or Windows Azure Table storage. Use the same affinity group for WAD storage to ensure it is stored in the same data center as the rest of the application.

Windows Azure Storage Analytics data are stored alongside the regular data, and are not stored in a separate storage account, so colocation is automatic.

More details about the capabilities of Windows Azure Diagnostics and Windows Azure Storage Analytics can be found in *Horizontally Scaling Compute Pattern* (Chapter 2) in the *Example* section under *Operational Logs and Metrics*. Both features support allow applications to set a basic data retention policy that Windows Azure Storage uses to automatically purge data.

When colocation is not possible due to technical or business reasons, Windows Azure has some services that can help. These services are mentioned in the *Example* section of *Multisite Deployment Pattern* (Chapter 15).

Related Chapters

- *MapReduce Pattern* (Chapter 6)
- *Network Latency Primer* (Chapter 11)
- *Valet Key Pattern* (Chapter 13)
- *CDN Pattern* (Chapter 14)
- *Multisite Deployment Pattern* (Chapter 15)

Summary

The simplest way to get started in the cloud is to colocate nodes, usually all in a single data center. This is appropriate for many applications, and should be the usual configuration. Only deviate for good reason, and avoid the mistake of accidental deployment across more than one data center, including for storage of operational data.

CHAPTER 13

Valet Key Pattern

This pattern focuses on efficiently using cloud storage services with untrusted clients.

This pattern loosely models the use of valet keys from the real world. Valet keys are useful when you are willing to trust a valet parking attendant to park your car, but don't want to also give them access to areas in the car not needed for this purpose, such as the glove compartment. This pattern enables specifying that a user of your application is allowed to access very specific areas within your cloud storage account, with specific permissions, and for a limited amount of time. You can issue as many cloud storage valet keys as you like and they can all be different.

Cloud storage services simplify securely transferring data directly between untrusted clients and to secure data storage, without the data needing to pass through a trusted intermediate layer (the web tier in this case) to implement security. Both uploads and downloads are supported.

Going directly to the final storage location eliminates the need for data to unnecessarily pass through an intermediary, so these operations will be faster with lower latency, while reducing the load on the web tier. Because this pattern avoids load on the web tier, it will result in needing less capacity, which may further result in needing fewer web server nodes (thus saving money). Realize, however, that this pattern also requires the ability to securely allow access to cloud storage, typically by issuing temporary access URLs. Depending on how this is done, it may necessitate the creation of a web service, which may require some infrastructure of its own. Generally, however, the performance, scalability, and cost savings favor the use of this pattern.

Context

The Valet Key Pattern is effective in dealing with the following challenges:

- Application data files are stored in cloud storage, which clients need to access
- Application data files are stored in cloud storage, which clients need to access on a case-by-case basis
- Application supports clients uploading data that will end up in cloud storage

This pattern is useful when reading and writing data from mobile apps, desktop apps, and during server-to-server communication. Reading data from a web application is also a very common use, though due to security limitations imposed by web browsers it is much trickier to write data to blob storage from a web application; however, see the appendix as it is possible in some cases using emerging HTML5 standards. Still, many web browsers do not support this scenario today.

Example of practical uses are many. A for-pay video or training site can allow access to a video for 24 hours. A photo sharing site can allow a user to upload a new photo or video directly into cloud storage. An enterprise can securely and efficiently share assets with employees or partners who are outside the firewall. A backup service running on a consumer desktop can efficiently interact directly with cloud storage, while only ever having access to files for that single user. A translation service can allow customers to upload text, audio, and video files; translated can be delivered back through the same secure means.

Cloud Significance

This pattern is possible because cloud services mask significant technical complexity by offering an easy-to-use service to applications. Additionally, software libraries supporting a broad range of clients further simplifies client coding.

The cloud platform data storage services are optimized for reading and writing files of all sizes at very high scale, shielding the application's web tier from needing to provide the bandwidth and computational power to handle this same data.

Some applications use the Gatekeeper Pattern, which does exactly what this pattern avoids: move all data through an intermediary in order to control access. Both patterns can be used securely, though this pattern offers efficiency and scalability benefits.

Impact

Scalability, User Experience

Mechanics

Access to cloud storage services is a privileged operation. Typically, you will allow only *trusted subsystems* within your application to have unfettered access to your cloud storage accounts, where trusted subsystems are running entirely under your control on the server.

Securing and managing access to data is always a high priority, but there is also a tension between security and efficiency; we want to remain secure, but would like to do so with the most efficient use of resources that allows us to deliver an optimized user experience. This pattern focuses on securely using blob storage while also reading and writing data with maximum efficiency. This is illustrated in Figure 13-1, which shows the flow of files directly between the client and the cloud storage service. The data flow is as efficient as possible, and in particular does not need to flow through the web tier of the application.

Using vendor-neutral terms can be challenging. A *blob* is a *binary large object* in the sense that's been used in the industry for many years and is commonly used with relational databases. Basically, a blob is a file. Windows Azure Storage happens to also call this concept a *blob*, while Amazon's Simple Storage Service (S3) calls it an *object*, Rackspace has *cloud files*, and the list goes on. In this chapter, a blob is just a file, and blob storage is a cloud platform service that allows you to store files in the cloud.

A *storage access key* is cryptographically generated by the cloud platform for each storage account, and applications are expected to use this key to access blob storage. It is not easy to safely store this key on client devices, whether for a desktop, mobile, or web application. A compromised key exposes the entire storage account. This pattern will not cover any techniques for protecting keys on client devices, but will instead focus on two other approaches: *public access* and *temporary access*.

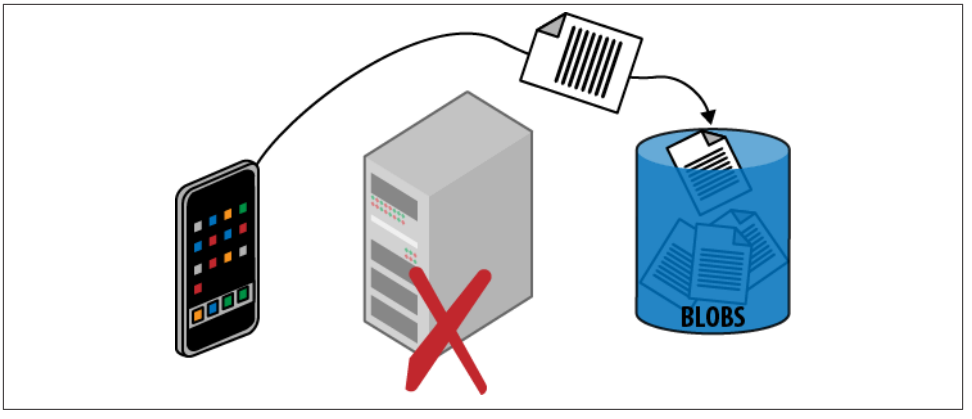


Figure 13-1. Data is uploaded efficiently, going directly from the client to cloud storage without passing through the web tier. The issuing of a temporary access URL—the valet key—is not shown, but is needed before the data can be uploaded.

Public Access

There are two sides to this pattern from the point of view of the client: reading blob data and writing blob data. For blob data that is intended to be publicly visible anyway, blob storage allows us to configure blobs for public read access. Because each blob resource is addressable as a URL, this opens up a considerable number of possibilities. Sharing is easy: just distribute the individual links. Blob URL references can be referenced just like other files from HTML pages so that web applications can benefit from the scalability features of blob storage, reducing the burden on the web tier that would otherwise be responsible for delivering files such as images, videos, JavaScript, and documents.

Anonymous public read access is very powerful. However, anonymous public write access is not supported. Temporary access for both read and write are possible if managed by the application.

Granting Temporary Access

Public read access allows any client knowing the URL to access the blob resource at any time without limit. This may not be what you want. If more control is desired, it is possible to grant temporary access to specific resources by constructing a *temporary access URL* using the storage key. The temporary access URL can be shared with clients who can then access the specific resource, but not all resources, just like with a valet key.

The Valet Key

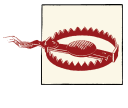
Some cars come with two types of keys: regular keys that work on every lock in the car, and valet keys that provide only limited access. The valet key is useful when you wish to temporarily allow some access, but do not want to give full access. The classic case is valet parking. With valet parking, you turn over your car to a stranger who will park it for you. In exchange for this convenience, you need to give this stranger a key to your car. The valet key is ideal for this, as it will only allow the driver to unlock the doors and start the car; it will not allow access to storage areas. In some newer cars, it will even limit the speed at which the car can be driven.

In the cloud storage world, the temporary access URL serves as a flexible and powerful valet key, allowing efficient and convenient access to storage while still being able to limit that access to only specific resources and during specific times.

Temporary access URLs are not limited to reading, but can be extended to allow other permissions, including writing. Thus is it possible to create one that would enable a client to upload a file directly into a predetermined location in blob storage.

The temporary access URLs are time-limited by your application-supplied expiration. The expiration time might be selected to be just long enough to safely finish an upload, watch a movie, or set to match the expected duration of the login session; this is a policy decision for your application.

Temporary access URLs need to be generated on the server and made available to the client. Reasonable approaches will depend on your application, but might include providing them during login, or separately issuing them on demand to authenticated callers via a web service.



When uploading blobs, transient failures may require retries. See *Busy Signal Pattern* (Chapter 9).

Security Considerations

Any code (in the cloud or elsewhere) with access to the storage access key will be able to create temporary access URLs.

Temporary access URLs are secured through *hashing*, a proven cryptographic technique that requires accesses to the storage key and uses it to create a unique signature for any string, in this case the temporary access URL. The associated hash is checked every time a client attempts to use the temporary access URL; without a correct one, access is denied. Adversaries without access to the storage key cannot tamper with an existing temporary access URL, create a new one, or guess a valid one.

However, as with a real-life valet key, any client in possession of a temporary access URL can use it. Any permissions granted by the temporary access URL are available. Like any other security access token, follow the principle of least privilege and provide only those rights necessary, and only for as long as necessary. Further, when transporting a temporary access URL, do so over a secure channel. Since the cloud platforms support secure HTTPS access to blob storage, clients outside the cloud provider's data center can safely use temporary access URLs. Temporary access URLs can even be used safely to serve pages to a regular web browser; as far as the browser is concerned, it is just requesting resources via HTTPS since the security key is part of the URL (the query string), HTTPS protects the query string during transport, and the cloud storage service handles authorization.

In the physical world, a lost valet key cannot be revoked; the only way to render it useless is by changing the locks. In the cloud, we have other tools such as expiration dates. The Windows Azure cloud platform goes further, supporting temporary access URLs that do not directly contain permissions or an expiration date, but rather reference a policy, known as a stored access policy, which dictates permissions and expiration. This policy can be changed independently of the URLs that reference it. This allows temporary access URLs to be issued that can later be revoked, extended, or have their permissions tweaked. Many URLs can reference the same policy.

There is no limit on the number of temporary access URLs that can be issued and all are independent of one another.

Example: Building PoP on Windows Azure

The Page of Photos (PoP) application (which was described in the Preface) supports publishing photos directly from a smart phone. Since smart phones can now routinely take high-resolution photos and capture HD video, the file may be large (multiple megabytes for a photo to tens of megabytes for a video), making this an especially good use case for this pattern.

From the point of view of the mobile application, it does not care whether it is uploading directly to cloud storage or uploading through a wrapper web service in the web tier; it is the same amount of work. To write a blob into Windows Azure, a mobile application only requires a few lines of code if it is using one of the mobile libraries provided as part of the platform. At the time of this writing, mobile client libraries were available for Android, iOS (iPhone, iPad), and Windows Phone.

Windows Azure Blobs offer a couple of handy features that allow us to streamline this process: public read access and Shared Access Signatures.

Public Read Access

Enabling public read access to blobs is trivial.

Blob storage containers (which are like directories or folders) can be marked as available for either public or private access. Photos stored in PoP are intended to be publicly viewable by anyone, so we can go ahead and mark them all as publicly visible. Once marked, anyone who knows the URL can read it. This is ideal for PoP for uploaded photos and videos, generated thumbnails, and any other size or format variants.

It is not possible to configure blobs in Windows Azure Storage to allow anonymous public updates of any kind; updates always require security credentials (which are described next).

Public read access to photos in PoP will likely result in users sharing URLs to specific photos. This exposes the underlying URL to users. By default, the URL points to a Windows Azure domain, but in our case we would like it to point to a *http://pageofphotos.com* domain; we do this with a vanity domain.

Using a Vanity Domain

Example of a regular photo URL from blob storage:

http://pageofphotos.blob.core.windows.net/photos/daniel.png

Example of a photo URL form blob storage after configuring our vanity domain:

http://www.pageofphotos.com/photos/daniel.png

Shared Access Signatures

A *Shared Access Signature (SAS)* is the Windows Azure Storage feature used to construct temporary access URLs for blobs that have temporary permission for reading or writing blobs.

PoP takes advantage of SAS for blob storage. Windows Azure Storage also features SAS support for its NoSQL database, Windows Azure Tables, as well as Windows Azure Queues.

For PoP, the goal is to permit users to upload photos from a mobile application directly into blob storage without allowing them to upload photos to other user accounts. Permissions are part of the special URL and will not allow one user to interfere with blobs belonging to another user. This level of security is sufficient for PoP. Once the temporary access URL is constructed and delivered to the mobile application, it is a simple matter for the mobile application to upload directly from the mobile device to blob storage, perhaps using one of the mobile client libraries mentioned previously.

SAS for Selective Read Access

The SAS technique can also be used to provide temporary access URLs for reading non-public blob resources. PoP does not require this since all images are public anyway.

If PoP did require that some photos be publicly accessible while others were not, it could use a private container to hold photos and use the SAS technique to provide temporary access to all URLs as needed. Alternatively, private photos could be moved into a separate blob container, allowing public photos in the original blob container to maintain open permissions.

Once a temporary access URL is generated, it is up to your application to safely deliver it to the appropriate client. In the case of the PoP mobile app, the SAS will be returned to the mobile app in response to an authenticated web service call over a secure (HTTPS) connection.

Windows Azure Storage also supports the notion of an *access policy* which is a named collection of permissions. An access policy can be used when creating a SAS. If a SAS

is constructed using an access policy, it is possible to later change that access policy. Any valid SAS will be immediately adjusted to use the new permissions. One common use of access policies is to maintain the ability to revoke permissions if it later becomes necessary.

Related Chapters

- *Eventual Consistency Primer* (Chapter 5)
- *Busy Signal Pattern* (Chapter 9)
- *Network Latency Primer* (Chapter 11)
- *Colocate Pattern* (Chapter 12)
- *CDN Pattern* (Chapter 14)
- *Multisite Deployment Pattern* (Chapter 15)

Summary

Use of this pattern should be considered anywhere it can be safely applied. The ability to manage temporary access for reading and writing makes this a broadly usable pattern. The most common troublesome use case will be an upload directly from a web browser, but reading is well supported, and writing from more flexible clients such as mobile apps is also well supported.

When used with storage containers that support this pattern, applications can avoid having a web page or web service act as a security proxy to read or write data stored in a secure container. This reduces load on the web tier because it is not acting as a middleman for data transfer. Also, the code for implementing all the variants of data passing through is replaced by the far simpler generation and issuing of temporary access URLs, while the upload code is offloaded to the client. The client code should utilize existing helper libraries where available to minimize complexity.

The end result is that applications scale better and the user experience is improved.

CHAPTER 14
CDN Pattern

This pattern focuses on reducing network latency for commonly accessed files through globally distributed edge caching.

The goal is to speed up delivery of application content to users. Content is anything that can be stored in a file such as images, videos, and documents. The Content Delivery Network (CDN) is a service that functions as a globally distributed cache. The CDN keeps copies of application files in many places around the world. When these places are close to users, content does not need to travel as far to be delivered, so it will arrive faster, improving the user experience.

CDN nodes are strategically located around the globe, hopefully close to application users. The CDN has its own URLs that are resolved by a geographic load balancer that directs users to the nearest node, regardless of where the user is.

The flow of files is shown in Figure 14-1. When a CDN URL is requested for the first time, the CDN will retrieve the file from the main source (usually known as the origin server), and then return that to the requesting user while also caching a local copy to satisfy subsequent requests for that file. When a CDN URL is requested and the file has already been cached, the CDN returns a locally cached copy of that file to the requesting user. This is faster than if the user retrieved it from the origin server, which is (presumably) further away. When many users access the same content from a CDN node, the initial request is slower (so the first user to request a file has to wait longer), but subsequent requests are much faster.

A CDN is only helpful for files that will be accessed multiple times. Files that are intended to be rarely accessed or that are for only a single user are usually not good candidates for a CDN.

Each CDN node operates independently of the others.

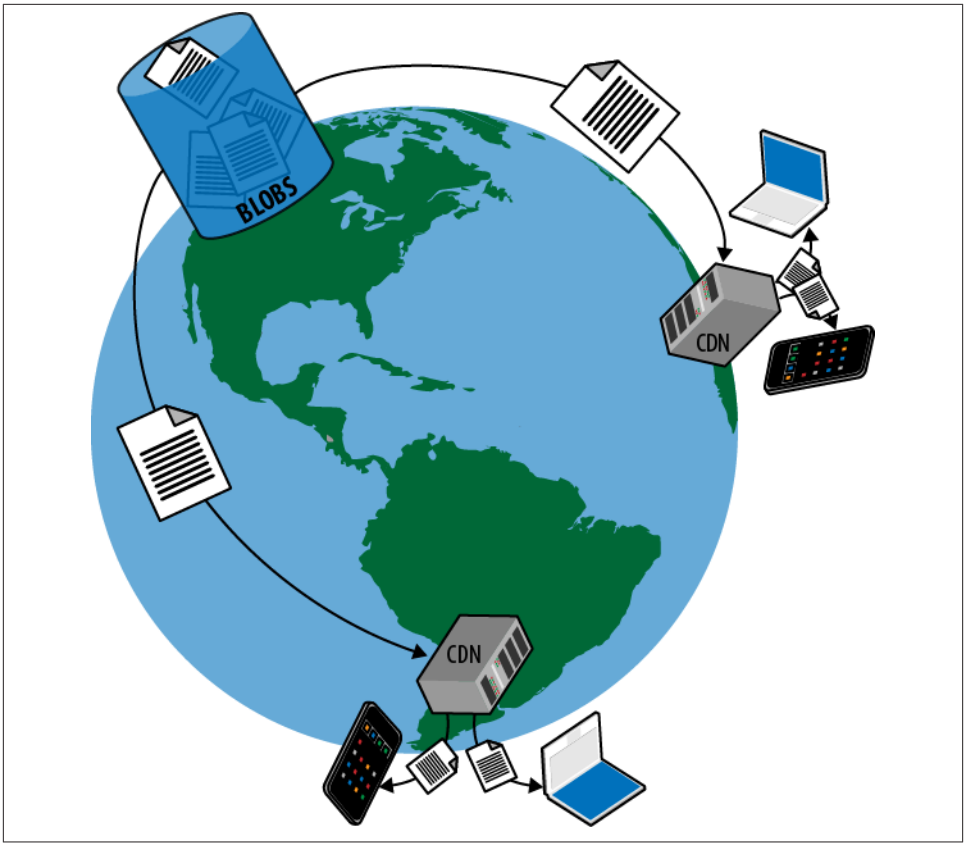


Figure 14-1. Users get content directly from the nearest CDN node. The CDN node fetches content from the source as needed.

Context

The CDN Pattern is effective in dealing with the following challenges:

- Application data is accessed from geographic locations that may not be close to the data center from which it originates
- Multiple clients access the same application data objects (such as HTML, JavaScript, image, video, or other files)
- Application includes large downloads, streaming video, or other heavyweight content delivery

A CDN can effectively reduce the load on other types of nodes that might be serving up content.

Cloud Significance

Cloud platform CDN services are easy to enable and are integrated into the cloud vendor's web-hosted management tool, appear on the same bill, and are supported by the same organization as other services offered within the cloud platform. Some other services, such as blob storage, can be easily CDN-enabled with a few mouse clicks. For these reasons, cloud CDN services can be more convenient than working directly with an independent CDN service provider.

Impact

Scalability, User Experience

Mechanics

Generally speaking, transmitting data over the Internet is faster when the source data and recipient are closer together. One way to bring source data and recipients closer together is by caching copies of the source data in locations that are closer to the recipients. When a user needs the data, retrieving it from the closest location will be faster than retrieving it from the origin, which might potentially be much further away. This practice is commonly known as *edge caching* and is the role of the CDN. Specifically, we will focus on accessing files over HTTP.

To enable the CDN to cache our files, we need to make one change: instead of using our normal domain name in our file URLs, we use one designated for us by the CDN provider. All file resources that we want to be cached by the CDN are changed to the new scheme.

Whenever a user attempts to access a file being managed by the CDN, the request goes to the CDN service to be fulfilled. However, because one of the benefits of the CDN service is that it has cache nodes in many geographical locations, somehow the request needs to be directed to the location nearest to the requesting user. This is usually handled through the *anycast* routing protocol that helps identify the closest CDN node, and the request is smartly routed accordingly.

Even though all users see the same URL for the file, different users will be routed to different CDN nodes. This is the whole point of the CDN: route requests to the nearest CDN node to improve responsiveness.

A CDN is a specialized cache, and as such, is not necessarily already populated with every desired file. This would be the case if the file had never been requested before through this CDN node, or the file had been requested but subsequently expired from the CDN cache. Any resource stored in a CDN has a defined expiration time that your

application sets, so this is something you will want to think about. Some resources, such as a company logo, may be stable for months or years, while other resources such as a user profile photo may have a shorter shelf life of perhaps one hour (users like to update them frequently). The expiration is expressed through the HTTP *Cache-Control* header. Because the Cache-Control caching directive is a long-established Internet standard, web browsers understand and handle this directive with ease. Beyond individual browsers, sometimes proxy servers or other intermediaries will cache, extending the caching benefits beyond a single user and beyond the CDN. Once the resource has expired, the CDN will remove it from its cache. (There are other possible behaviors with some CDN services, such as having the CDN check with the origin server to ascertain whether expired content has indeed changed, and still using the cached object if it has not.)

If a requested file is not available at the CDN node, the CDN service effectively acts as a middleman, and behind the scenes retrieves the file from the origin server, caches (stores) it in the CDN node, and then responds to the original request (which it can now do successfully).

Limitations of CDN

A CDN is not effective for use on resources that are infrequently accessed. A CDN is only efficient for use on resources that are usually accessed at least twice before expiring from the CDN cache.

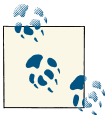
A CDN is not effective for use on resources that change constantly, such as those changing with each request.

A CDN may be a poor choice for content that is not intended for public viewing.

A CDN really shines when used on frequently accessed files that seldom change.

Caches Can Be Inconsistent

The CDN stores copies of resources with a specified expiration date; thus a cached image file might be declared to be valid for one hour, one month, or whatever is appropriate for that resource. Any resource cached in a CDN is potentially stale because there is always a delay between updates to the source copy and propagation to the copy that is cached at the CDN. This is not necessarily a problem, but is a factor to consider carefully when deciding what to cache and for how long.



Caching resources in a CDN is an application of *eventual consistency*: immediate consistency is traded off for performance and scalability benefits.

Example: Building PoP on Windows Azure

The Page of Photos (PoP) application (which was described in the Preface) stores all photos in Windows Azure Blob Storage. To improve the download experience for PoP users, all photo downloads are CDN-enabled.

In Windows Azure Blob Storage, individual blobs are stored in a blob *container* which, among other benefits, establishes a default security context for the blobs within it. Only blobs stored in public containers—where *public* means anyone knowing the URL to a file within the container can view that file—are eligible for caching in the CDN.

Configuring the CDN

Enabling the CDN for blobs is a trivial configuration change at the storage account level. In order for the CDN to sit between users and blob storage, a new domain name is created. This new domain name is system generated. That is, rather than using our own domain name (such as example.com), we use one assigned to us by the CDN service (such as jaromijo1213.vo.msecnd.net). A URL that used to be *http://example.com/maura.png* before the CDN becomes *http://jaromijo.vo.msecnd.net/maura.png* with the CDN. You can optionally configure a vanity domain name of your own to make the CDN addresses appear less chaotic; this vanity domain name is also known as *canonical name* (or CNAME) in DNS terms. PoP takes advantage of this vanity feature, creating *http://cdn.pageofphotos.com* as the CDN domain so that photo URLs will be more consistent with the URL for the main application.

As of this writing, there are 8 Windows Azure data centers worldwide, but 24 Windows Azure CDN node locations. Since there are so many additional geographic locations brought into the mix, use of the CDN network greatly increases global coverage for lower latency content distribution. A resource needs to have been requested at least once in order to be loaded into the CDN cache. The first request for that resource will therefore have a poorer user experience than subsequent requestors. This scenario plays out at each CDN node, as each node needs to fill its own local cache. The scenario also repeats any time a resource is requested, but has expired from the CDN cache.

Each of the 24 CDN nodes operates independently, so each populates independently.

PoP photos do not change frequently, so when the photos are saved in blob storage, a Cache-Control header is set as a property on the individual blob, with the duration set to one month. If we ever needed to update a photo sooner, we would need to change the filename.

The Windows Azure CDN does not currently have built-in support for pre-fetching an object to warm up a particular CDN node. Further, once an object is cached by the CDN, you can't easily change your mind about it since there is no way to evict a particular object from the CDN before its specified cache expiration. Thus, tricks like file renaming or appending a "cache-buster" like a query string are used. For example, we could append a query string so that the cached image *kd1hn.png* becomes *kd1hn.png?v=1* to cause it to reload without finding the one in the current CDN. This requires a change to the reference used to access the image (such as the HTML page referencing it). These are cache tricks that happen to work with the CDN (since it is also a cache honoring HTTP headers); nothing is specific to the Windows Azure Cache.

Cost Considerations

The Windows Azure CDN access charges are the same as directly from blob storage, with no additional at-rest storage charge. Other than the cost of the one additional request needed to populate the CDN node, the cost will be identical to direct-from-blob access. Note that this "one additional request" happens for (a) each resource, (b) each time it is loaded from blob storage (the first time as well as the next time after cache expiration), and (c) in each CDN node from which the resource is accessed.

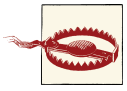
Security Considerations

The Windows Azure CDN only supports caching of files that are already freely visible to the public, so they do not decrease security or increase the attack surface.

The Windows Azure CDN supports access using HTTP or HTTPS. This is primarily a user experience benefit (because the files are already freely visible to the public) and comes into play when a user loads a page using HTTPS. Modern browsers typically display a warning if an HTTPS page references an image as HTTP, complaining about mixed security models. Such images can instead be accessed as HTTPS to improve the user experience.

Additional Capabilities

Though not explored in this chapter, there are other Windows Azure services related to CDN. For example, the CDN can serve files directly from a Web Role. Also, Windows Azure Media Services provide video streaming to the Windows Azure CDN. There are also other integrated services that help applications prepare media for download by many types of devices, such as transcoding services that make content equally accessible across mobile phone brands and desktop operating systems.



As of this writing, the Windows Azure Media Services are currently in preview and not yet supported for production use.

Related Chapters

- *Eventual Consistency Primer* (Chapter 5)
- *Network Latency Primer* (Chapter 11)
- *Colocate Pattern* (Chapter 12)
- *Valet Key Pattern* (Chapter 13)
- *Multisite Deployment Pattern* (Chapter 15)

Summary

Adding CDN support to a cloud application is a great example of a low-friction adoption of a cloud service. Enabling a CDN can be accomplished either programmatically or through a one-time manual configuration via the cloud vendor's web-hosted management tool. This is substantially easier to get started with than traditional CDNs due to the degree of convenience and integration.

Once enabled, this is a great technique for reducing the load on web servers, distributing load across many servers (there are many more CDN locations than data centers), while decreasing network latency. All this helps to both improve scalability and improve user experience.

CHAPTER 15

Multisite Deployment Pattern

This advanced pattern focuses on deploying a single application to more than one data center.

Deploying to multiple data centers helps reduce network latency by routing a client to the nearest data center, which improves the user experience. This also provides the seeds of a solution that can handle failover across data centers and improve availability.

If multisite deployment does not improve user experience and your application does not need cross-data center failover, use of this pattern may be overkill.

Context

The Multisite Deployment Pattern is effective in dealing with the following challenges:

- Users are not clustered near any single data center, but form clusters around multiple data centers or are widely distributed geographically
- Regulations limit options for storing data in specific data centers
- Circumstances require that the public cloud be used in concert with on-premises resources
- Application must be resilient to the loss of a single data center

This pattern helps deal with a user base that is not conveniently clustered in a single geographic area. Some of the reasons for this include use of the application from unpredictable locations during travel, globally distributed mobile applications, and companies with offices distributed across many geographical locations.

This pattern is similar in some ways to *CDN Pattern* (Chapter 14), in that we strive to bring our application closer to our users. The CDN focus is on bringing files closer to our users, and it only helps when sending data to our users, not users sending data to

the application. This pattern is more powerful than a CDN because it brings more application facilities closer to users with lower latency, even when users send data to the application. This pattern is more limited in one way: there are many more CDN nodes than there are full-blown data centers. For these reasons, these two patterns are often used together.

Cloud Significance

Networking cloud services are available for geographic load balancing and cross-data center failover. Data-oriented cloud services are also available for database synchronization and geo-replication of cloud storage. These services, plus access to multiple geographically distributed data centers, simplify some of the most complex aspects of multisite deployment.

Impact

Availability, Reliability, Scalability, User Experience

Mechanics

If all global traffic for an application is served from a single data center, then responsiveness is better near that data center than from more distant regions of the world. Generally speaking, the farther away, the poorer the responsiveness. Which data center to choose then? This pattern is about choosing more than one data center to offer the best available user experience throughout the world.

The major public cloud platforms have multiple data centers on multiple continents. With Windows Azure and Amazon Web Services, you select the specific data center to which you will deploy. Let's assume you choose one data center location in Asia, one in Europe, and one in the United States, and deploy an instance of your application to each data center.

Once your application is deployed to multiple data centers, you need to decide how users will be directed to the appropriate one. In this pattern, we will strive to make this process transparent to users in order to provide the best user experience. In the most basic scenarios, this requires smart routing and data replication services.

To route users to the closest data center, use your cloud platform's service for geographic load balancing and configure it to direct users to the closest data center. These services include the Windows Azure Traffic Manager and Elastic Load Balancing from Amazon Web Services.

You also need to replicate data across all data centers if you want users to see the same data, regardless of the data center to which they route. This is further explored in the example given later in this chapter. The end result can be seen in Figure 15-1, where

data centers replicate data as needed. Since each data center has a local copy of any needed data, users can be conveniently routed to the one nearest to them for the best performance. As we shall see later, this topology also can be leveraged to handle rare scenarios when one of the data centers is not available.

Another possibility is to assign each user to a single data center, though that scenario is not explored.

With geographic load balancing and data replication in place, you have the basics of a multisite deployment. Users will experience better performance than if they always had to directly access data and services from a data center located on another continent.

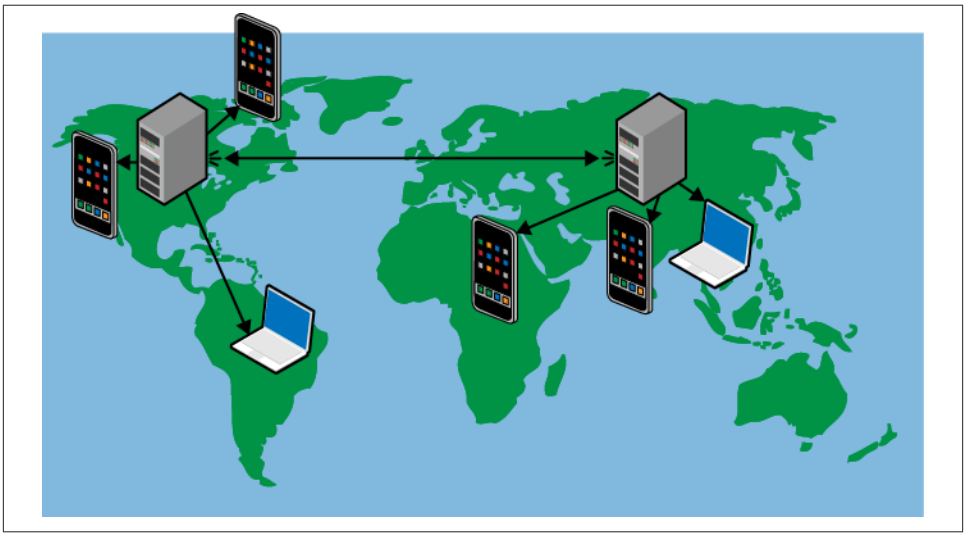


Figure 15-1. Data centers stay in sync so that users can access the closest one, or an alternate in case of failure. The connection spanning the continents represents data center-to-data center synchronization. External user devices are shown accessing the closest data center.

Non-Technical Considerations in Data Center Selection

Sometimes there are non-technical considerations that compel you to separate application tiers across data centers or to choose one data center over another. Sometimes a data center location is chosen in order to comply with government regulations, industry requirements, and other so-called *data sovereignty* issues. For example, some countries

in the European Union (EU) require that applications operating in the EU store personally identifying information within the EU. As another example, the credit card industry has specific expectations of applications handling credit card data. These expectations may further limit or influence options in data center selection.

Finally, consider a business constraint in which a customer is not ready to move a database from its on-premises location into the cloud but wants to access it from cloud servers. Cloud platform services can extend a company's internal network into the cloud with a Virtual Private Network (VPN). It securely handles this scenario so that applications running on cloud resources can easily query databases located elsewhere. This is yet another factor influencing your data center footprint.

Thus, business considerations may override purely performance-based data center selection criteria. While these considerations can impact cloud architecture, a treatment of this complex topic is not the focus of this pattern.

Cost Implications

As of this writing, both Windows Azure and Amazon platforms charge a fee for data leaving the data center, but not for data coming into the data center. Thus, traffic across data centers would incur these charges. Of course, this applies even within a single data center location because remotely accessing a cloud application from outside the data center results in data leaving the data center. In the context of this pattern, this also applies to keeping databases in sync across data centers and accessing on-premises databases.

Deploying your application across multiple data centers has cost implications. Such costs should be weighed against the user experience and the robustness limitations of a single data center. And of course, if multisite deployment is needed, consider the cost and complexity of doing this without the convenience of cloud services.

Failover Across Data Centers

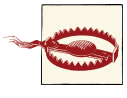
Failover is a feature of an application that enables the application to continue to function using secondary resources when there is a failure with the primary resources. In the context of this pattern, we are concerned with the loss of a data center and the ability of our application to continue to function using the other available data centers. We may want to failover from one data center to another due to a natural disaster, such as a hurricane, or due to a failure by the cloud provider, such as a software bug or configuration error that impacts services.

The tools and techniques used to create a multisite deployment can also help to make your application more failure-resistant. The same services used for geographic load

balancing that route users to the closest data center can be used to account for failover scenarios. In particular, they can be configured to monitor the health of the services to which they are directing users, and if any service is unavailable, traffic will be routed to a healthy instance.

It takes time for the geographic load balancer to figure out that a service is not responding before it fails over to a healthy copy. During that time, application access may result in errors. Client-side retries, such as *Busy Signal Pattern* (Chapter 9), can help shield users from these errors. Once failover is complete, users will be routed to data centers; if these data centers are on other continents, their experience will be degraded, but it will work. This is usually better than not working at all, although that depends on the specific application.

As unhealthy services become healthy again, traffic can be delivered, returning system responsiveness to maximum levels.



This scheme does not guarantee instant or seamless failover. There will be downtime.

One approach to failover is to have the secondary resources already deployed, just in case they are needed. This arrangement is known as either *active/passive* or *active/active*, depending on whether the secondary resources are just sitting around in case they are needed or are in active use (the primary data center accounts for the first "active" resource). Another option is to not have any spare resources that are already running, but to deploy those resources as needed for a failover. This obviously would take longer, but is also less costly. This topic is explored further in the *Example*.

This is part of a larger *disaster recovery (DR)* plan. Many of these concerns parallel those for non-cloud applications, but are beyond the scope of this chapter.

Example: Building PoP on Windows Azure

The Page of Photos (PoP) application (which was described in the Preface) faces a few challenges in providing a seamless user experience across the globe:

- Routing users to the nearest data center so that they have the best user experience
- Replicating account data so that it is available at all data centers, both for account owners and visitors
- Replicating identity information so that we can correctly authenticate a user in any data center and know they are the same user
- Serving photos globally and efficiently

Choosing a Data Center

As of this writing, Windows Azure offers eight geographically distributed data centers:

- Two in Asia: East (Hong Kong) and Southeast (Singapore)
- Two in Europe: North (Netherlands) and West (Ireland)
- Four in the United States: North Central (Illinois), East (Virginia), South Central (Texas), and West (California)

Since PoP users are all over the world, we will deploy to one data center in Asia (Singapore), one in Europe (Ireland), and one in North America (Virginia). However, PoP users should not need to know about the three data centers; using PoP should just work, regardless of where the user is. In other words, *http://www.pageofphotos.com* should be the only address a user ever needs to remember.

Routing to the Closest Data Center

Windows Azure Traffic Manager is a scalable, highly available, easily configured, geographic load balancing service that continually monitors the responsiveness of all application deployments behind the scenes. It ensures that a visitor to *http://www.pageofphotos.com* is transparently directed to the data center that will give the visitor the best response.

Configuring Traffic Manager

Traffic Manager configuration includes specifying a public domain name (*http://www.pageofphotos.com*) that users see, listing the individual instances (*http://pop-asia.cloudapp.net*, *http://pop-eur.cloudapp.net*, *http://pop-usa.cloudapp.net*), and configuring the routing rules for the closest instance. The endpoint names listed are just examples, but show use of a consistent naming pattern that can help avoid confusion and errors.

In our case, as an example, visiting *http://www.pageofphotos.com/timothy* from Asia would be satisfied by the *http://pop-asia.cloudapp.net* instance.

Replicating User Data for Performance

Consider a PoP user in Ireland who posts some new photos. They can be neatly and efficiently stored in the Ireland data center with a minimum of network latency. The collection can be shared with friends in Ireland who also access them from the Ireland data center. So far, so good.

Now consider that a link to this page of photos is emailed to some friends in Boston. Traffic from Boston will be routed to the Virginia data center. What will happen? To make it work, the Virginia data center needs access to the same data that was saved to the Ireland data center. Our approach with PoP is split into two complementary schemes: one for uploaded photos, another for the rest of the account data.

The photos are uploaded to blobs in Window Azure Storage in the nearest data center. The application makes no effort to explicitly replicate these photos to other data centers. Thumbnails for these photos are generated in the same data center [using *Queue-Centric Workflow Pattern* (Chapter 3)] and also stored as blobs. Photos are not replicated to all data centers as we plan to rely on *CDN Pattern* (Chapter 14) to deliver them most efficiently to users.

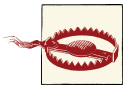
In addition to the photos, there is metadata around the photos such as:

- The account responsible for uploading the photo
- The URL needed for accessing the photo
- The text description of the photo
- And so on.

This data is stored in a Windows Azure SQL Database. The new photo is uploaded to the nearest data center. The metadata is saved to SQL Database in the nearest data center. But, unlike the photo blobs, the SQL Database data is replicated across all three data centers using SQL Database Data Sync service.

Configuring Data Sync

SQL Data Sync service configuration requests the provisioning of a Data Sync Server, creates a Sync Group that will specify the database instances to be kept in sync, then adds the database names of the three instances that will be kept in sync to the Sync Group. It selects the desired sync rules, and lets it run. Syncs happen on a schedule; the most aggressive supported frequency is five minutes between synchronization jobs.



As of this writing, the SQL Database Data Sync service is currently in preview. It is not yet supported for production use.

A user from anywhere in the world will be directed to the nearest data center when accessing a page of photos. It returns the HTML page of photos; the individual photos will be served up from one of the 24 global CDN nodes.

Note the eventual consistency of SQL Database instances. Consider our earlier example of a PoP user from Ireland sharing a page of photos with a friend in Boston. There will be a short window of time during which PoP users in Ireland see new photos, but PoP users in the United States (and Asia) do not; this is eventually resolved as the SQL Database Data Sync service catches up.

Replicating Identity Information for Account Owners

While any anonymous user can view any page of photos, all uploaded photos belong to a PoP account holder. An account on PoP is easily created because it does not manage user credentials, but rather relies on *federated authentication*. PoP is configured to allow users to log in using an *identity provider (IdP)* that they are likely already using. Supported IdPs leveraged by PoP include Google, Yahoo!, and Facebook. (Many other identity providers can be supported, but PoP chooses not to make use of them.) The first time a user signs into their IdP, they give their IdP permission to share login information with PoP. (Actually, the IdP shares with the Access Control Service (ACS), not directly with PoP. PoP gets the information from ACS.) For any IdP supported by PoP, the important item we want is a validated email address. Because each IdP trusted by PoP is reputable, we trust it. We then use the email address as a unique account key in our SQL Database instances. This way, we can tie uploaded photos to a specific user account.

Note further that PoP does not know or store the user's password. Only the underlying IdP does. Using cryptographic techniques, the IdP can securely communicate to PoP that users are who they say they are and pass along a cryptographically secure *claim* that provides their email address. This is all PoP needs. The PoP development effort can be focused on features, not infrastructure.

Federated identity, claims, and identity providers are important concepts for building applications that deal with identity in a cloud-native manner. A whole book could be written to explain these important topics. (In fact, Appendix A references such a book.) The key takeaway for this chapter is that some external entity (an IdP) is telling us the current user owns a specific email address (indicated by a claim) and that we can trust that this is accurate.

The Windows Azure Access Control Service (ACS) acts as an intermediary between PoP and each IdP. It turns out that this is a huge simplification for PoP because there is so much variation in how an IdP can behave. It can be fairly complicated if your application needs to interact with the IdP directly. ACS helps applications manage the relationship with one or more identity providers, abstracting away the implementation details and normalizing them so that claims are consistent when they reach your application.

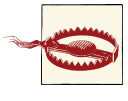
Configuring ACS

Access Control Service configuration consists of establishing a set of rules for each supported provider. The basic process is mostly handled through a set of well-defined steps in a wizard-like user interface; this is appropriate as such configurations are routine. Google and Yahoo! configurations can be completed in the ACS interface, though Facebook requires additional configuration at the Facebook site. ACS also can be configured programmatically. The configuration for PoP is simple: pass through all claims from the IdP (which will include an email address claim), and inject an "Admin" role claim for a few special email accounts (for PoP site administrators).

This description of how ACS helps authentication work for PoP is necessary to provide context for the multisite deployment. The upshot is that this configuration needs to be repeated for each identity provider in each supported data center.

Data Center Failover

Given the measures we have taken to make PoP available in three data centers, what more do we need to do to ensure we can failover in the event one of the data centers fails?



Supporting data center failover is a big decision with many risks and tradeoffs. The approach outlined here is appropriate for the PoP application and business. These risks and tradeoffs may or may not be acceptable for your application and business.

PoP replicates Windows Azure SQL Database instances. As mentioned previously, there is still a window within which data loss could occur. If this is not good enough, it may be challenging to overcome.

PoP has Access Control Service identity providers configured in each data center so that PoP will recognize the data center, regardless of which one is used to authenticate.

PoP uses Windows Azure Storage to save photos and thumbnails as blobs. We have taken no measures to replicate them because the Windows Azure Platform geo-replicates blobs on our behalf. (Windows Azure Storage Tables are also geo-replicated.) Each data center is paired with another on the same continent. In the list of data centers above, the two Asia data centers replicate, the two Europe data centers replicate, and in the United States, North Central replicates with South Central, and East replicates with West. In the extremely rare event of a disaster so great that a data center is lost or unavailable for

an extended period of time, storage will failover to its replication pair. You do not need to do anything. There will be a period of time when blob requests fail, but once the failover is complete, they will resume working correctly. Note, however, that there is still a window within which data loss could occur.

PoP uses the CDN to deliver photos and thumbnails close to the users. Because all of the eight Windows Azure data centers also host CDN nodes, a data center failure could likewise impact the CDN. Furthermore, the other CDN nodes could be impacted independently. The built-in CDN routing logic will notice a failure of a CDN node and change where traffic is routed. There is still a window within which files may not be delivered. Given the nature of the CDN, it is likely that the many distributed CDN nodes will somewhat shield users during the interval where blobs are unavailable and in the process of being failed over because many photos will already be cached around the world.

The final step is to update the configuration in Traffic Manager. Earlier we described how Traffic Manager routes to the closest (most responsive) data center. Traffic Manager also handles failover scenarios. For example, if a hurricane rendered the Virginia data center unavailable, Traffic Manager can route users to Singapore or Ireland (whichever it determines to be faster for any individual user). While the user experience is degraded compared to direct access to the North American data center, it is a better user experience than no availability. There is still a window within which Traffic Manager may route to a failing data center.

Because the data center can failover to resources that are already provisioned and in active use, this is an *active/active* failover configuration.

Considering Disaster Recovery and User Experience

How should use of multisite deployment balance disaster recovery (DR) and user experience (primarily performance) benefits? There is no single right answer, but it is generally helpful to look at DR and user experience as independent concerns that happen to have overlapping solutions. Either could drive the decision to go multisite, or the combined value may be needed to justify a multisite deployment.

Colocation Alternatives

As mentioned earlier, there are a number of non-technical business factors that may limit location options for data storage. In Windows Azure, there are a number of features that might be of use in making the most of such constraints. Briefly, these include:

- Windows Azure Virtual Networking and Windows Azure Connect support a secure Virtual Private Network (VPN) connection between your Windows Azure service and another network. This enables many integration scenarios, such as allowing compute resources running in Windows Azure to access a database running in a private data center.
- Windows Azure Service Bus supports several secure scenarios for communication across firewalls, publish/subscribe to connected devices, and publish/subscribe to disconnected devices. These capabilities enable secure and efficient communication across applications regardless of firewall or data center configurations.
- SQL Data Sync Service use for cloud-to-cloud synchronization was already described. This service can also be used to synchronize between the cloud and on-premises databases.

These services can help integrate with a database or services that are on premises or in another data center.

Related Chapters

- *Horizontally Scaling Compute Pattern* (Chapter 2)
- *Queue-Centric Workflow Pattern* (Chapter 3)
- *Eventual Consistency Primer* (Chapter 5)
- *Node Failure Pattern* (Chapter 10)
- *Network Latency Primer* (Chapter 11)
- *Colocate Pattern* (Chapter 12)
- *Valet Key Pattern* (Chapter 13)
- *CDN Pattern* (Chapter 14)

Summary

Using the Multisite Deployment Pattern primarily helps improve the user experience for a geographically distributed user base. Users need not be all over the world, but at least distributed such that more than one data center provides sufficient value if the goal is to improve performance.

This pattern is also useful for applications requiring a failover strategy in case one data center becomes unavailable. This is a complex subject, but many of the components in this chapter will get you started.

Because the use of this pattern will result in a more complex and more expensive application than a single data center solution, the business value needs to be assessed and compared with the cost.

APPENDIX A

Further Reading

Page of Photos (PoP) Sample

The Page of Photos sample application is used throughout the book. Parts of it have already been implemented using Windows Azure to demonstrate ideas in the book. The code for Page of Photos (minimalist at first, then built out over time by the author and some accomplices) will be shared in a public repo on GitHub.

- Run the Page of Photos (PoP) sample application: *http://www.pageofphotos.com*
- View the Page of Photos (PoP) sample application source code: *http://www.github.com/codingoutloud/pageofphotos*
- Find information about the book and possibly related content in the future: *http://www.cloudarchitecturebook.com*

Resources From Preface and Chapters

- Windows Azure Platform: *http://www.windowsazure.com*
- Amazon Web Services: *http://aws.amazon.com/*
- Google App Engine: *http://developers.google.com/appengine/*
- A NIST Definition of Cloud Computing (SP 800-145 Sept. 2011): *http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf*
- (NIST) Cloud Computing Synopsis and Recommendations (SP 800-146 May 2012): *http://csrc.nist.gov/publications/nistpubs/800-146/sp800-146.pdf*

Chapter 1

- "A Compuware analysis of 33 major retailers across 10 million home page views showed that a 1-second delay in page load time reduced conversions by 7%." Source: Compuware, April 2011.
- "Google observed that adding a 500-millisecond delay to page response time caused a 20% decrease in traffic" Source: Marissa Mayer, "What Google Knows" talk at Web 2.0 Conf 2006 (11/09/2006): *http://conferences.oreillynet.com/presentations/web2con06/mayer.ppt*
- "Yahoo! observed a 400-millisecond delay caused a 5-9% decrease [in traffic]." And "Amazon.com reported that a 100-millisecond delay caused a 1% decrease in retail revenue." Source: *http://blog.yottaa.com/2010/11/secret-sauce-for-successful-web-site-web-performance-optimization-wpo*
- "Google has started using web-site performance as a signal in its search engine rankings." Source: Using site speed in web search ranking: *http://googlewebmastercentral.blogspot.com/2010/04/using-site-speed-in-web-search-ranking.html*
- Example of a self-inflicted scaling failure: *http://glinden.blogspot.com/2006/11/amazon-crashes-itself-with-promotion.html*
- ITIL: *http://en.wikipedia.org/wiki/Information_Technology_Infrastructure_Library*

Chapter 2

- Windows Azure Storage Analytics: Logs and Metrics: *http://blogs.msdn.com/b/windowsazurestorage/archive/2011/08/03/windows-azure-storage-analytics.aspx*
- Windows Azure Diagnostics: *http://msdn.microsoft.com/en-us/library/windowsazure/gg433048.aspx*
- About Load Balancers: *http://1wt.eu/articles/2006_lb/*

Chapter 3

- Comparing Windows Azure Storage Queues with Amazon Simple Queue Service: *http://gauravmantri.com/2012/04/15/comparing-windows-azure-queue-service-and-amazon-simple-queue-servicesummary/*
- Comparing the two queue services offered by Windows Azure: *http://msdn.microsoft.com/en-us/library/windowsazure/hh767287.aspx*
- Update Message on a Windows Azure Queue: *http://msdn.microsoft.com/en-us/library/windowsazure/hh452234*
- Loose coupling: *http://en.wikipedia.org/wiki/Loose_coupling*

- Long Polling with SignalR for ASP.NET: *http://signalr.net*
- Long Polling with Socket.IO for Node.js: *http://socket.io*
- CQRS Pattern: *http://martinfowler.com/bliki/CQRS.html*
- CQRS: *http://www.cqrsinfo.com/*
- Event Sourcing: *http://martinfowler.com/eaaDev/EventSourcing.html*

Chapter 4

- Enterprise Library 5.0 Integration Pack for Windows Azure (contains WASABi): *http://msdn.microsoft.com/en-us/library/hh680918*
- Hosted services that can be used to monitor and autoscale your Windows Azure applications include AzureWatch from Paraleap Technologies and AzureOps from Opstera.
- Auto-Scaling on Amazon Web Services: *http://aws.amazon.com/autoscaling*
- Ticket Direct case study that sharded databases then consolidated depending on ticket sales: New Zealand-based TicketDirect International: *http://www.microsoft.com/casestudies/Case_Study_Detail.aspx?CaseStudyID=4000005890*

Chapter 5

- Life beyond Distributed Transactions: an Apostate's Opinion (by Pat Helland): *http://www-db.cs.wisc.edu/cidr/cidr2007/papers/cidr07p15.pdf*
- CAP Twelve Years Later: How the "Rules" Have Changed (by Eric Brewer): *http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed*
- Introducing BASE: *http://queue.acm.org/detail.cfm?id=1394128*
- Introducing Eventual Consistency: *http://queue.acm.org/detail.cfm?id=1466448*
- Eventual consistency in CloudFront: *http://docs.amazonwebservices.com/AmazonCloudFront/latest/DeveloperGuide/Concepts.html*
- Google BigTable: *https://developers.google.com/appengine/docs/python/datastore/overview*
- Amazon Dynamo SOSP paper: *http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf*
- Azure Storage SOSP paper: *http://sigops.org/sosp/sosp11/current/2011-Cascais/printable/11-calder.pdf*

Chapter 6

- Hadoop: *http://hadoop.apache.org*
- Hadoop on Windows Azure: *http://www.hadooponazure.com*

Chapter 7

- An Unorthodox Approach to Database Design: The Coming of the Shard from High Scalability blog: *http://highscalability.com/unorthodox-approach-database-design-coming-shard*.
- The official source for learning about Federations in Windows Azure SQL Database.
- For anyone interested in Federations for Windows Azure SQL Database, Cihan Biyikoglu's blog is a must-read. Some particularly useful posts are listed below.
  - Implementing MERGE command using SQL Azure Migration Wizard by *@gihuey*: *http://blogs.msdn.com/b/cbiyikoglu/archive/2012/02/20/implementing-alter-federation-merge-at-command-using-sql-azure-migration-wizard-by-gihuey.aspx*.
  - Introduction to Fan-out Queries for Federations in SQL Azure (Part 1): Scalable Queries over Multiple Federation Members, MapReduce Style!: *http://blogs.msdn.com/b/cbiyikoglu/archive/2011/12/29/introduction-to-fan-out-queries-querying-multiple-federation-members-with-federations-in-sql-azure.aspx*.
- Integrated sharding support with Windows Azure SQL Database Federations: *http://blogs.msdn.com/b/cbiyikoglu/archive/2012/02/08/connection-pool-fragmentation-scale-to-100s-of-nodes-with-federations-and-you-won-t-need-to-ever-learn-what-these-nasty-problems-are.aspx*
- Federations: *http://msdn.microsoft.com/en-us/magazine/hh848258.aspx*
- Choosing a shard key in MongoDB: *http://www.mongodb.org/display/DOCS/Choosing+a+Shard+Key*
- SQL Azure Data Sync: *http://msdn.microsoft.com/en-us/library/windowsazure/hh667301.aspx*
- Windows Azure Table Storage service: *http://www.windowsazure.com/en-us/develop/net/how-to-guides/table-services/*
- Generating a GUID as a cluster key with NEWID for Federations on SQL Database: *http://msdn.microsoft.com/en-us/library/ms190348.aspx*

Chapter 8

- Definition of multitenancy: *http://en.wikipedia.org/wiki/Multitenancy*
- Definition of commodity hardware: *http://en.wikipedia.org/wiki/Commodity_hardware*

Chapter 9

- Definition of multitenancy: *http://en.wikipedia.org/wiki/Multitenancy*
- Transient Fault Handling Application Block (Topaz): *http://msdn.microsoft.com/en-us/library/hh680934(v=PandP.50).aspx*
- Scalability Targets for Windows Azure Storage: *http://blogs.msdn.com/b/windowsazurestorage/archive/2010/05/10/windows-azure-storage-abstractions-and-their-scalability-targets.aspx*
- Implementing Retry Logic on Windows Azure: *http://www.davidaiken.com/2011/10/10/implementing-windows-azure-retry-logic/*
- Fault isolation and recovery: *http://www.faqs.org/rfcs/rfc816.html*
- Chaos Monkey from Netflix: *http://techblog.netflix.com/2011/07/netflix-simian-army.html*

Chapter 10

- Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications: *http://research.microsoft.com/en-us/um/people/navendu/papers/greenberg09vl2.pdf*
- How Windows Azure knows a Role Instance (node) is faulty: *http://blogs.msdn.com/b/mcsuksoldev/archive/2010/05/10/how-does-azure-identify-a-faulty-role-instance.aspx*
- Windows Azure Troubleshooting Best Practices: *http://msdn.microsoft.com/en-us/library/windowsazure/hh771389.aspx*
- Updating a Windows Azure deployment, including Fault Domains and Update Domains: *http://msdn.microsoft.com/en-us/library/ff966479.aspx*

Chapter 11

- Ping utility: *http://en.wikipedia.org/wiki/Ping*

- It's the Latency Stupid essay: *http://rescomp.stanford.edu/~cheshire/rants/Latency.html*

Chapter 12

- On the importance of affinity groups: *https://msmvps.com/blogs/nunogodinho/archive/2012/03/04/importance-of-affinity-groups-in-windows-azure.aspx*

Chapter 13

- Windows Azure Toolkits for Mobile Devices (Android, iOS, Windows Phone, and more): *https://github.com/WindowsAzure-Toolkits*
- Restricting Access to Containers and Blobs Windows Azure: *http://msdn.microsoft.com/en-us/library/windowsazure/dd179354*
- Web Browser Same Origin Policy: *http://en.wikipedia.org/wiki/Same_origin_policy*
- Using a Shared Access Signature (REST API): *http://msdn.microsoft.com/en-us/library/windowsazure/ee395415.aspx*
- Rahul Rai's sample code showing access to Windows Azure Blob Storage from HTML 5 Web Browser: *http://code.msdn.microsoft.com/windowsazure/Silverlight-Azure-Blob-3b773e26*
- Trusted Subsystem Design: *http://msdn.microsoft.com/en-us/library/aa905320.aspx*

Chapter 14

- Anycast protocol enables geographic load balancing for CDN: *http://en.wikipedia.org/wiki/Anycast*
- Windows Azure Media Service: *https://www.windowsazure.com/en-us/home/features/media-services/*
- Recorded talk on Windows Azure CDN: *http://channel9.msdn.com/Events/TechEd/NorthAmerica/2011/COS401*

Chapter 15

- Windows Azure SQL Data Sync service: *http://msdn.microsoft.com/en-us/library/windowsazure/hh456371.aspx*

- Windows Azure Traffic Manager: *http://msdn.microsoft.com/en-us/wazplatformtrainingcourse_windowsazuretrafficmanager.aspx*
- A Guide to Claims-Based Identity and Access Control: *http://msdn.microsoft.com/en-us/library/ff423674.aspx*
- Windows Azure Access Control Service: *http://msdn.microsoft.com/en-us/library/windowsazure/gg429786.aspx*
- Automating the Windows Azure Access Control Service (ACS): *http://msdn.microsoft.com/en-us/library/gg185927.aspx*
- ACS automation sample code: *http://acs.codeplex.com/releases/view/57595*
- SQL Azure Point In Time Restore now available in preview: *http://www.microsoft.com/en-us/download/details.aspx?id=28364*
- Business Continuity in Windows Azure SQL Database: *http://msdn.microsoft.com/en-us/library/windowsazure/hh852669.aspx*

Index

Symbols

A

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

B

C

D

E

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

About the Author

Bill Wilder is a hands-on developer, architect, consultant, trainer, speaker, writer, and community leader focused on helping companies and individuals succeed with the cloud using the Windows Azure Platform. Bill began working with Windows Azure when it was unveiled at the Microsoft PDC in 2008 and subsequently founded Boston Azure, the first/oldest Windows Azure user group in the world, in October 2009. Bill is recognized by Microsoft as a Windows Azure MVP and is the author of *Cloud Architecture Patterns*. Bill can be found blogging at *blog.codingoutloud.com* and on Twitter at @codingoutloud.