O'REILLY®

Foundations of Scalable Systems

Designing Distributed Architectures

Early Release

RAW & UNEDITED

Ian Gorton

Foundations of Scalable Systems

Designing Distributed Architectures

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Ian Gorton

Foundations of Scalable Systems

by Ian Gorton

Copyright © Ian Gorton. All rights reserved.

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://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Melissa Duffield

Development Editor: Virginia Wilson

Production Editor: Daniel Elfanbaum

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

August 2022: First Edition

Revision History for the Early Release

- 2021-05-10: First Release
- 2021-06-22: Second Release
- 2021-09-08: Third Release
- 2021-10-04: Fourth Release

- 2021-11-19: Fifth Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098106065 for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Foundations of Scalable Systems*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher’s views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-10599-0

[LSI]

Preface

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

This book is built around the thesis that the ability of software systems to operate at scale is increasingly a driving system quality. As our world becomes more interconnected, this characteristic will only accelerate. Hence the goal of the book is to provide the reader with the core knowledge of distributed and concurrent systems. It will also introduce a collection of software architecture approaches and distributed technologies that can be used to build scalable systems.

Why Scalability?

The pace of change in our world is daunting. Innovations appear daily, creating new capabilities for us all to interact with, conduct business, be entertained, end pandemics. The fuel for much of this innovation is software, written by veritable armies of developers in major internet companies, crack small teams in startups, and all shapes and sizes of teams in between.

Delivering software systems that are responsive to user needs is difficult enough, but it becomes an order of magnitude more difficult to do for

systems at scale. We all know of systems that fail suddenly when exposed to unexpected high loads - such situations are minimally bad publicity for organizations, and at worst can lose jobs and destroy companies.

Software is unlike physical systems in that it's amorphous—its physical form (1's and 0's) bears no resemblance to its actual capabilities. We'd never expect to transform a small village of 500 people into a city of 10 million overnight. But we sometimes expect our software systems to suddenly handle 1000x the number of requests they were designed for. Not surprisingly, the outcomes are rarely pretty.

Who This Book Is For

The major target audience for this book is software engineers and architects who have no or limited experience with distributed, concurrent systems. They need to deepen both their theoretical and practical design knowledge in order to meet the challenges of building larger scale, typically Internet-facing applications.

Much of the content of this book has been developed in the context of an advanced undergraduate/graduate course at Northeastern University. It has proven a very popular and effective approach for equipping students with the knowledge and skills needed to launch their careers with major Internet companies. Additional materials on the book web site are available to support educators who wish to use the book for their course.

What You Will Learn

This book covers the landscape of concurrent and distributed systems through the lens of scalability. While it's impossible to totally divorce scalability from other architectural qualities, scalability is the main focus of discussion. Of course, other qualities necessarily come in to play, with performance, availability and consistency regularly raising their heads.

Building distributed systems requires some fundamental understanding of distribution and concurrency - this knowledge is a recurrent theme throughout this book. It's needed because at their core, there are two problems in distributed systems that make them complex, as I describe below.

First, although systems operate perfectly correctly nearly all the time, an individual part of the system may fail at any time. When a component fails (hardware crash, network down, bug in server), we have to employ techniques that enable the system as a whole to continue operations and recover from failures. And any distributed system will experience component failure, often in weird and mysterious and unanticipated ways.

Second, creating a scalable distributed system requires the coordination of multiple moving parts. Each component of the system needs to keep its part of the bargain and process requests as quickly as possible. If just one component causes requests to be delayed, the whole system may perform poorly and even eventually crash.

To deal with these problems there is a rich deep body of literature available to draw on. And luckily for us engineers, there's a rich, extensive collection of technologies that are designed to help us build distributed systems that are tolerant to fail and scalable. These technologies embody theoretical approaches and complex algorithms that are incredibly hard to build correctly. Using these platform level, widely applicable technologies, our applications can stand on the shoulders of giants, enabling us to build sophisticated business solutions.

Specifically, readers of this book will learn:

- The fundamental characteristics of distributed systems, including state management, time coordination, concurrency, communications and coordination
- Architectural approaches and supporting technologies for building scalable, robust services
- How distributed databases operate and can be used to build scalable distributed systems
- Architectures and technologies such as Apache Kafka and Flink for building streaming, event-based systems

Part I. Scalability in Modern Software Systems

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

The first four chapters in Part 1 of this book motivate the need for scalability as a key architectural attribute in modern software systems. The chapters provide broad coverage of the basic mechanisms for achieving scalability, the fundamental characteristics of distributed systems, and an introduction to concurrent programming. This knowledge lays the foundation for what follows, and if you are new to the areas of distributed, concurrent systems, you'll need to spend some time on these four chapters. They will make the rest of the book much easier to digest.

Chapter 1. Introduction to Scalable Systems

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 1st chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

The last 20 years have seen unprecedented growth in the size, complexity and capacity of software systems. This rate of growth is hardly likely to slow in the next 20 years – what these future systems will look like is close to unimaginable right now. The one thing we can guarantee is that more and more software systems will need to be built with constant growth - more requests, more data, more analysis - as a primary design driver.

Scalable is the term used in software engineering to describe software systems that can accommodate growth. In this chapter I'll explore what precisely is meant by the ability to scale – known, not surprisingly, as scalability. I'll also describe a few examples that put hard numbers on the capabilities and characteristics of contemporary applications and give a brief history of the origins of the massive systems we routinely build today. Finally, I'll describe two general principles for achieving scalability, namely replication and optimization, that will recur in various forms throughout the

rest of this book and examine the indelible link between scalability and other software architecture quality attributes.

What is Scalability?

Intuitively, scalability is a pretty straightforward concept. If we ask Wikipedia for a definition, it tells us "scalability is the property of a system to handle a growing amount of work by adding resources to the system." We all know how we scale a highway system – we add more traffic lanes so it can handle a greater number of vehicles. Some of my favorite people know how to scale beer production – they add more capacity in terms of the number and size of brewing vessels, the number of staff to perform and manage the brewing process, and the number of kegs they can fill with tasty fresh brews. Think of any physical system – a transit system, an airport, elevators in a building – and how we increase capacity is pretty obvious.

Unlike physical systems, software systems are somewhat amorphous. They are not something you can point at, see, touch, feel, and get a sense of how it behaves internally from external observation. It's a digital artifact. At its core, the stream of 1's and 0's that make up executable code and data are hard for anyone to tell apart. So, what does scalability mean in terms of a software system?

Put very simply, and without getting into definition wars, scalability defines a software system's capability to handle growth in some dimension of its operations. Examples of operational dimensions are:

- The number of simultaneous user or external (e.g. sensor) requests a system can process
- The amount of data a system can effectively process and manage
- The value that can be derived from the data a system stores
- The ability to maintain a stable. consistent response time as the request load grows

For example, imagine a major supermarket chain is rapidly opening new stores and increasing the number of self-checkout kiosks in every store. This requires the core supermarket software systems to:

- Handle increased volume from item scanning without decreased response time. Instantaneous responses to item scans are necessary to keep customers happy.
- Process and store the greater data volumes generated from increased sales. This data is needed for inventory management, accounting, planning and likely many other functions.
- Derive ‘real-time’ (e.g. hourly) sales data summaries from each store, region and country and compare to historical trends. This trend data can help highlight unusual events in regions (e.g. unexpected weather conditions, large crowds at events, etc.) and help the stores affected quickly respond.
- Evolve the stock ordering prediction subsystem to be able to correctly anticipate sales (and hence the need for stock reordering) as the number of stores and customers grow

These dimensions are effectively the scalability requirements of the system. If, over a year, the supermarket chain opens 100 new stores and grows sales by 400 times (some of the new stores are big!), then the software system needs to scale to provide the necessary processing capacity to enable the supermarket to operate efficiently. If the systems don’t scale, we could lose sales as customers are unhappy. We might hold stock that will not be sold quickly, increasing costs. We might miss opportunities to increase sales by responding to local circumstances with special offerings. All these reduce customer satisfaction and profits. None are good for business.

Successfully scaling is therefore crucial for our imaginary supermarket’s business growth, and likewise is in fact the lifeblood of many modern internet applications. But for most business and Government systems, scalability is not a primary quality requirement in the early stages of development and deployment. New features to enhance usability and utility

become the drivers of our development cycles. As long as performance is adequate under normal loads, we keep adding user-facing features to enhance the system's business value. In fact, introducing some of the sophisticated distributed technologies I'll describe in this book before there is a clear requirement can actually handicap a project, with the additional complexity causing development inertia.

Still, it's not uncommon for systems to evolve into a state where enhanced performance and scalability become a matter of urgency, or even survival. Attractive features and high utility breed success, which brings more requests to handle and more data to manage. This often heralds a tipping point, where design decisions that made sense under light loads are now suddenly technical debt. External trigger events often cause these tipping points – look in the March/April 2020 media for the many reports of Government Unemployment and supermarket online ordering sites crashing under demand caused by the coronavirus pandemic.

Increasing a systems' capacity in some dimension by increasing resources is called *scaling up* or *scaling out* – I'll explore the difference between these later. In addition, unlike physical systems, it is often equally important to be able to *scale down* the capacity of a system to reduce costs.

The canonical example of this is Netflix, which has a predictable regional diurnal load that it needs to process. Simply, a lot more people are watching Netflix in any geographical region at 9pm than are at 5am. This enables Netflix to reduce its processing resources during times of lower load. This saves the cost of running the processing nodes that are used in the Amazon cloud, as well as societally worthy things such as reducing data center power consumption. Compare this to a highway. At night when few cars are on the road, we don't retract lanes (except for repairs). The full road capacity is available for the few drivers to go as fast as they like. In software systems, we can expand and contract our processing capacity in a matter of seconds to meet instantaneous load. Compared to physical systems, the strategies we deploy are very, very different.

There's a lot more to consider about scalability in software systems, but let's come back to these issues after examining the scale of some contemporary software systems circa 2021.

System scale in early 2020's: Examples

Looking ahead in this technology game is always fraught with danger. In 2008 I wrote [1]:

> *While petabyte datasets and gigabit data streams are today's frontiers for data-intensive applications, no doubt 10 years from now we'll fondly reminisce about problems of this scale and be worrying about the difficulties that looming exascale applications are posing.*

Reasonable sentiments, it is true, but exascale? That's almost commonplace in today's world. Google reported multiple exabytes of Gmail in 2014, and by now, do all Google services manage a yottabyte or more? I don't know. I'm not even sure I know what a yottabyte is! Google won't tell us about their storage, but I wouldn't bet against it. Similarly, how much data does Amazon store in the various AWS data stores for their clients. And how many requests does, say, DynamoDB process per second collectively, for all client applications supported? Think about these things for too long and your head will explode.

A great source of information that sometimes gives insights into contemporary operational scales are the major Internet company's technical blogs. There are also Web sites analyzing Internet traffic that are highly illustrative of traffic volumes. Let's take a couple of 'point in time' examples to illustrate a few things we do know today. Bear in mind these will look almost quaint in a year or four.

- Facebook's engineering blog describes Scribe, their solution for collecting, aggregating, and delivering petabytes of log data per hour, with low latency and high throughput. Facebook's computing infrastructure comprises millions of machines, each of which generates log files that capture important events relating to system

and application health. Processing these log files, for example from a Web server, can give development teams insights into their application's behavior and performance, and support fault finding. Scribe is a custom buffered queuing solution that can transport logs from servers at a rate of several terabytes per second and deliver them to downstream analysis and data warehousing systems. That, my friends, is a lot of data!

- You can see live Internet traffic for numerous services at www.internetlivestats.com. Dig around and you'll find statistics like Google handles around 3.5 billion search requests a day, Instagram uploads about 65 million photos per day, and there is something like 1.7 billion web sites. It is a fun site with lots of information to amaze you. Note the data is not really 'live', just estimates based on statistical analyses of multiple data sources.
- In 2016 Google published a paper describing the characteristics of their code base. Amongst the many startling facts reported is: "The repository contains 86TBs of data, including approximately two billion lines of code in nine million unique source files." Remember, this was 2016.

Still, real, concrete data on the scale of the services provided by major Internet sites remain shrouded in commercial-in-confidence secrecy. Luckily, we can get some deep insights into the request and data volumes handled at Internet scale through the annual usage report from one tech company. You can browse their incredibly detailed usage statistics from 2019. It's a fascinating glimpse into the capabilities of massive scale systems. Beware though, this is Pornhub.com.[1]

How Did We Get Here? A Brief History of System Growth

I am sure many readers will have trouble believing there was civilized life without Internet search, YouTube and social media. In fact, the first video

upload to YouTube occurred in 2005. Yep, it is hard for even me to believe. So, let's take a brief look back in time at how we arrived at the scale of today's systems. Below are some historical milestones of note:

1980s

An age dominated by timeshared mainframe and minicomputers. PCs emerged in the early 1980s but were rarely networked. By the end of the 1980s, development labs, universities and increasingly businesses had email and access to primitive Internet resources.

1990-95

Networks became more pervasive, creating an environment ripe for the creation of the World Wide Web (WWW) with HTTP/HTML technology that had been pioneered at CERN by Tim Berners-Lee during the 1980s. By 1995, the number of web sites was tiny, but the seeds of the future were planted with companies like Yahoo! in 1994 and Amazon and eBay in 1995

1996-2000

The number of web sites grew from around 10,000 to 10 million, a truly explosive growth period. Networking bandwidth and access also grew rapidly. Companies like Amazon, eBay, Google, Yahoo! and the like were pioneering many of the design principles and early versions of advanced technologies for highly scalable systems that we know and use today. Everyday businesses rushed to exploit the new opportunities that *e-business* offered, and this brought system scalability to prominence, as explained in the sidebar.

2000-2006

The number of web sites grew from around 10 to 80 million during this period, and new service and business models emerged. In 2005, YouTube was launched. 2006 saw Facebook become available to the public. In the same year, Amazon Web Services, which had low key beginnings in 2004, relaunched with its S3 and EC2 services.

2007-today

> We now live in a world with around 2 billion web sites, of which about 20% are active. There are something like 4 billion Internet users. Huge data centers operated by public cloud operators like AWS, GCP and Azure, along with a myriad of private data centers, for example Twitter's operational infrastructure, are scattered around the planet. Clouds host millions of applications, with engineers provisioning and operating their computational and data storage systems using sophisticated cloud management portals. Powerful cloud services make it possible for us to build, deploy and scale our systems literally with a few clicks of a mouse. All you do is pay your cloud provider bill at the end of the month.

This is the world that this book targets. A world where our applications need to exploit the key principles for building scalable systems and leverage highly scalable infrastructure platforms. Bear in mind, in modern applications, most of the code executed is not written by your organization. It is part of the containers, databases, messaging systems and other components that you compose into your application through API calls and build directives. This makes the selection and use of these components at least as important as the design and development of your own business logic. They are architectural decisions that are not easy to change.

HOW SCALE IMPACTED BUSINESS SYSTEMS

The surge of users with Internet access in the 1990s brought new online money making opportunities for businesses. There was a huge rush to expose business functions - sales, services - to users through a Web browser. This heralded a profound change in how we had to think about building systems.

Take for example a retail bank. Before providing online services, it was possible to accurately predict the loads the bank's business systems would experience. You knew how many people worked in the bank and used the internal systems, how many terminals/PCs were connected to the bank's networks, how many ATMs you had to support, and the number and nature of connections to other financial institutions. Armed with this knowledge, we could build systems that support, say, a maximum of say 3000 concurrent users, safe in the knowledge that this number could not be exceeded. Growth would also be relatively slow, and probably most of the time (eg outside business hours) the load would be a lot less than the peak. This made our software design decisions and hardware provisioning a lot easier.

Now imagine our retail bank decides to let all customers have Internet banking access. And the bank has 5 million customers. What is our maximum load now? How will load be dispersed during a business day? When are the peak periods? What happens if we run a limited time promotion to try and sign up new customers? Suddenly our relatively simple and constrained business systems environment is disrupted by the higher average and peak loads and unpredictability you see from Internet-based user populations.

Scalability Basic Design Principles

The basic aim of scaling a system is to increase its capacity in some application-specific dimension. A common dimension is increasing the

number of requests that a system can process in a given time period. This is known as the system's throughput. Let's use an analogy to explore two basic principles we have available to us for scaling our systems and increasing throughput: replication and optimization.

In 1932, one of the world's great icons, the Sydney Harbor Bridge, was opened. Now it is a fairly safe assumption that traffic volumes in 2021 are somewhat higher than in 1932. If by any chance you have driven over the bridge at peak hour in the last 30 years, then you know that its capacity is exceeded considerably every day. So how do we increase throughput on physical infrastructures such as bridges?

This issue became very prominent in Sydney in the 1980s, when it was realized that the capacity of the harbor crossing had to be increased. The solution was the rather less iconic Sydney Harbor tunnel, which essentially follows the same route underneath the harbor. This provides 4 more lanes of traffic, and hence added roughly 1/3rd more capacity to harbor crossings. In not too far away Auckland, their harbor bridge also had a capacity problem as it was built in 1959 with only 4 lanes. In essence, they adopted the same solution as Sydney, namely, to increase capacity. But rather than build a tunnel, they ingeniously doubled the number of lanes by expanding the bridge with the hilariously named 'Nippon Clipons', which widened the bridge on each side.

These examples illustrate the first strategy we have in software systems to increase capacity. We basically replicate the software processing resources to provide more capacity to handle requests and thus increase throughput, as shown in Figure 1-1. These replicated processing resources are analogous to the traffic lanes on bridges, providing a mostly independent processing pathway for a stream of arriving requests. Luckily, in cloud-based software systems, replication can be achieved at the click of a mouse, and we can effectively replicate our processing resources thousands of times. We have it a lot easier than bridge builders in that respect. Still. we need to take care to replicate resources in order to alleviate bottlenecks, otherwise our resources will simply cause needless costs and give no scalability benefit.

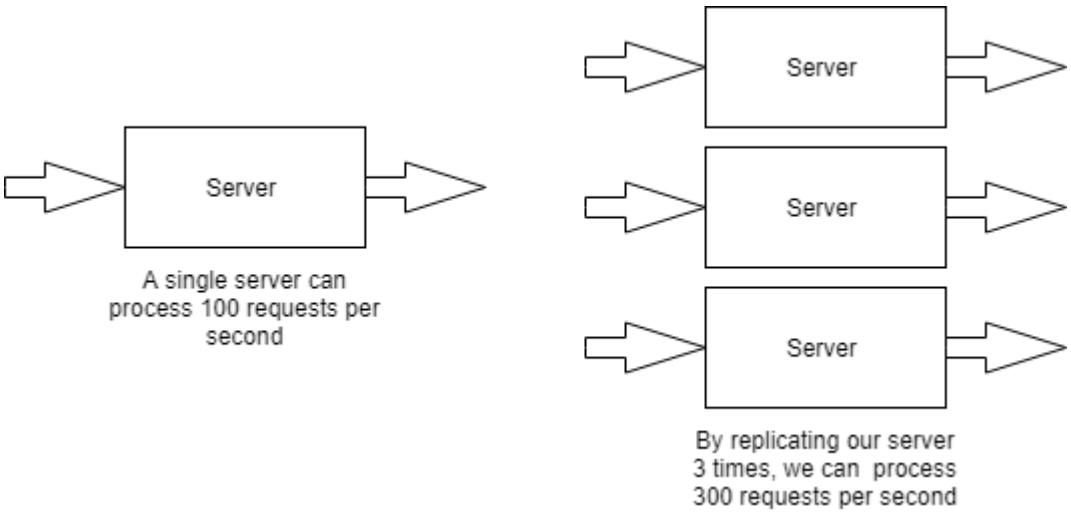

Figure 1-1. Increasing Capacity through Replication

The second strategy for scalability can also be illustrated with our bridge example. In Sydney, some observant person realized that in the mornings a lot more vehicles cross the bridge from north to south, and in the afternoon we see the reverse pattern. A smart solution was therefore devised – allocate more of the lanes to the high demand direction in the morning, and sometime in the afternoon, switch this around. This effectively increased the capacity of the bridge without allocating any new resources – we *optimized* the resources we already had available.

We can follow this same approach in software to scale our systems. If we can somehow optimize our processing, by maybe using more efficient algorithms, adding extra indexes in our databases to speed up queries, or even rewriting our server in a faster programming language, we can increase our capacity without increasing our resources. The canonical example of this is Facebook's creation of (the now discontinued) HipHop for PHP, which increased the speed of Facebook's web page generation by up to 6 times by compiling PHP code to C++.

I'll revisit these two design principles – namely replication and optimization - many times in the remainder of this book. You will see that there are many complex implications of adopting these principles that arise from the fact that we are building distributed systems. Distributed systems have properties that make building scalable systems 'interesting', where interesting in this context has both positive and negative connotations.

Scalability and Costs

Let's take a trivial hypothetical example to examine the relationship between scalability and costs. Assume we have a Web-based (e.g. web server and database) system that can service a load of 100 concurrent requests with a mean response time of 1 second. We get a business requirement to scale up this system to handle 1000 concurrent requests with the same response time. Without making any changes, a simple load test of this system reveals the performance shown in Figure 1-2 (left). As the request load increases, we see the mean response time steadily grow to 10 seconds with the projected load. Clearly this does not satisfy our requirements in its current deployment configuration. The system doesn't scale.

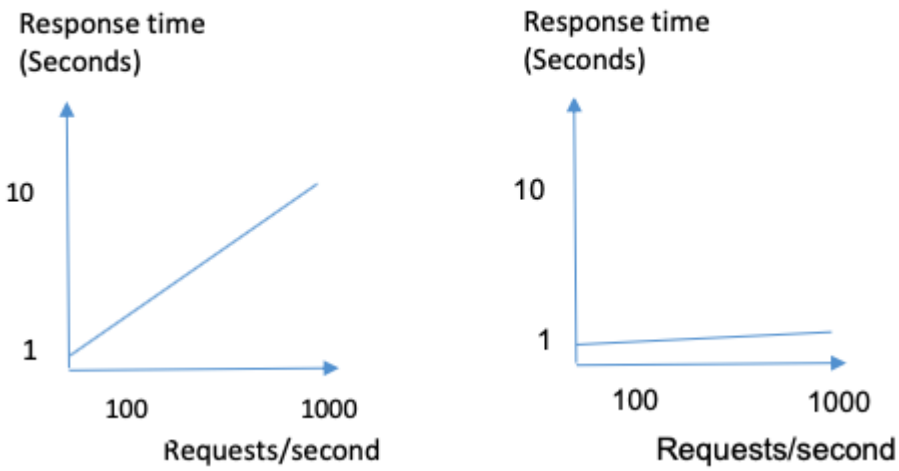

Figure 1-2. Scaling an application. (Left) – non-scalable performance. (Right) – scalable performance

Clearly some engineering effort is needed in order to achieve the required performance. Figure 1-2 (right) shows the system's performance after this effort has been modified. It now provides the specified response time with 1000 concurrent requests. Hence, we have successfully scaled the system. Party time!

A major question looms however. Namely, how much effort and resources were required to achieve this performance? Perhaps it was simply a case of

running the Web server on a more powerful (virtual) machine. Performing such reprovisioning on a cloud might take 30 minutes at most. Slightly more complex would be reconfiguring the system to run multiple instances of the Web server to increase capacity. Again, this should be a simple, low-cost configuration change for the application, with no code changes needed. These would be excellent outcomes.

However, scaling a system isn't always so easy. The reasons for this are many and varied, but here's some possibilities:

1. The database becomes less responsive with 1000 requests per second, requiring an upgrade to a new machine
2. The Web server generates a lot of content dynamically and this reduces response time under load. A possible solution is to alter the code to more efficiently generate the content, thus reducing processing time per request.
3. The request load creates hot spots in the database when many requests try to access and update the same records simultaneously. This requires a schema redesign and subsequent reloading of the database, as well as code changes to the data access layer.
4. The Web server framework that was selected emphasized ease of development over scalability. The model it enforces means that the code simply cannot be scaled to meet the request load requirements, and a complete rewrite is required. Use another framework? Use another programming language even?

There's a myriad of other potential causes, but hopefully these illustrate the increasing effort that might be required as we move from possibility (1) to possibility (4).

Now let's assume option (1), upgrading the database server, requires 15 hours of effort and a thousand dollars extra cloud costs per month for a more powerful server. This is not prohibitively expensive. And let's assume option (4), a rewrite of the Web application layer, requires 10,000 hours of development due to implementing in a new language (e.g. Java instead of

Ruby). Options (2) and (3) fall somewhere in between options (1) and (4). The cost of 10,000 hours of development is seriously significant. Even worse, while the development is underway, the application may be losing market share and hence money due to its inability to satisfy client requests loads. These kinds of situations can cause systems and businesses to fail.

This simple scenario illustrates how the dimensions of resource and effort costs are inextricably tied to scalability. If a system is not designed intrinsically to scale, then the downstream costs and resources of increasing its capacity to meet requirements may be massive. For some applications, such as Healthcare.gov, these (more than $2 billion) costs are borne and the system is modified to eventually meet business needs. For others, such as Oregon's health care exchange, an inability to scale rapidly at low cost can be an expensive ($303 million) death knell.

We would never expect someone would attempt to scale up the capacity of a suburban home to become a 50 floor office building. The home doesn't have the architecture, materials and foundations for this to be even a remote possibility without being completely demolished and rebuilt. Similarly, we shouldn't expect software systems that do not employ scalable architectures, mechanisms and technologies to be quickly evolved to meet greater capacity needs. The foundations of scale need to be built in from the beginning, with the recognition that the components will evolve over time. By employing design and development principles that promote scalability, we can more rapidly and cheaply scale up systems to meet rapidly growing demands. I'll explain these principles in Part 2 of this book.

Software systems that can be scaled exponentially while costs grow linearly are known as hyperscale systems, defined as:

> *Hyper scalable systems exhibit exponential growth in computational and storage capabilities while exhibiting linear growth rates in the costs of resources required to build, operate, support and evolve the required software and hardware resources.*

You can read more about hyperscale systems in this article [3].

Scalability and Architecture Trade-offs

Scalability is just one of the many quality attributes, or non-functional requirements, that are the *lingua franca* of the discipline of software architecture. One of the enduring complexities of software architecture is the necessity of quality attribute trade-offs. Basically a design that favors one quality attribute may negatively or positively affect others. For example we may want to write log messages when certain events occur in our services so we can do forensics and support debugging of our code. We need to be careful however how many events we capture because logging introduces overheads and negatively affects performance.

Experienced software architects constantly tread a fine line, crafting their designs to satisfy high priority quality attributes, while minimizing the negative effects on other quality attributes.

Scalability is no different. When we point the spotlight at the ability of a system to scale, we have to carefully consider how our design influences other highly desirable properties such as performance, availability, security and the oft overlooked manageability. I'll briefly discuss some of these inherent trade-offs below.

Performance

There's a simple way to think about the difference between performance and scalability. When we target performance, we attempt to satisfy some desired metrics for individual requests. This might be a mean response time of less than 2 seconds, or a worst-case performance target such as the 99th percentile response time less than 3 seconds.

Improving performance is in general a good thing for scalability. If we improve the performance of individual requests, we create more capacity in our system, which helps us with scalability as we can use the unused capacity to process more requests.

However, it's not always that simple. We may reduce response times in a number of ways. We might carefully optimize our code, by for example

removing unnecessary object copying, using a faster JSON serialization library, or even completely rewriting your code in a faster programming language. These approaches optimize performance without increasing resource usage.

An alternative approach might be to optimize individual requests by keeping commonly accessed state in memory rather than writing to the database on each request. Eliminating a database access nearly always speeds things up. However, if our system maintains large amounts of state in memory for prolonged periods, we may (and in a heavily loaded system, will) have to carefully manage the number of requests our system can handle. This will likely reduce scalability as our optimization approach for individual requests uses more resources (in this case, memory) than the original solution, and hence reduces system capacity.

We'll see this tension between performance and scalability reappear throughout this book. In fact, it's sometimes judicious to make individual requests slightly slower so we can utilize additional system capacity. A great example of this is described when I discuss load balancing in the next chapter.

Availability

Availability and scalability are in general highly compatible partners. As we scale our systems through replicating resources, we create multiple instances of services that can be used to handle requests from any users. If one of our instances fails, the others remain available. The system just suffers from reduced capacity due to a failed, unavailable resource. Similar thinking holds for replicating network links, network routers, disks, and pretty much any resource in a computing system.

Things start to get complicated with scalability and availability when state is involved. Think of a database. If our single database server becomes overloaded, we can replicate it and send requests to either instance. This also increases availability as we can tolerate the failure of one instance. This scheme works great if our databases are read only. But as soon as we

update one instance, we somehow have to figure out how and when to update the other instance. This is where the issue of replica consistency raises its ugly head.

In fact, whenever state is replicated for scalability and availability, we have to deal with consistency. This will be a major topic when I discuss distributed databases in Part 3 of this book.

Security

Security is a complex, highly technical topic worthy of its own book. No one wants to use an insecure system, and systems that are hacked and compromise user data cause CTOs to resign, and in extreme cases companies to fail.

The basic elements of a secure system are authentication, authorization and integrity. We need to ensure data cannot be intercepted in transit over networks, and data at rest (persistent store) cannot be accessed by anyone who does not have permissions to access that data. Basically I don't want anyone seeing my credit card number as it is communicated between systems or stored in a company's database.

Hence security is a necessary quality attribute for any Internet facing systems. The costs of building secure systems cannot be avoided, so let's briefly examine how these affect performance and scalability.

At the network level, systems routinely exploit the Transport Layer Security[2] (TLS) protocol, which runs on top of TCP/IP (see Chapter 3). TLS provides encryption, authentication and integrity using asymmetric cryptography.[3] This has a performance cost for establishing a secure connection as both parties need to generate and exchange keys. TLS connection establishment also includes an exchange of certificates to verify the identity of the server (and optionally client), and the selection of an algorithm to check the data is not tampered with in transit. Once a connection is established, in-flight data is encrypted using symmetric cryptography, which has a negligible performance penalty as modern CPUs have dedicated encryption hardware.

Connection establishment requires usually two message exchanges between client and server, and is hence comparatively slow. Reusing connections as much as possible minimizes these performance overheads.

There are multiple options for protecting data at rest. Popular database engines such as SQL Server and Oracle have features such as Transparent Data Encryption (TDE) that provides efficient file level encryption. Finer grain encryption mechanisms, down to field level, are increasingly required in regulated industries such as finance. Cloud providers offer various features too, ensuring data stored in cloud based data stores is secure. The overheads of secure data at rest are simply costs that must be borne to achieve security - studies suggest the overheads are in the 5-10% range.[4]

Another perspective on security is the CIA Triad,[5] which stands for *Confidentiality*, *Integrity* and *Availability*. The first two are pretty much what I have described above. Availability refers to a system's ability to operate reliably under attack from adversaries. Such attacks might be attempts to exploit a system design weakness to bring the system down. Another attack is a classic Distributed Denial of Service (DDOS) where an adversary gains control over multitudes of systems and devices and coordinates a flood of requests that effectively make a system unavailable.

In general, security and scalability are opposing forces. Security necessarily introduces performance degradation. The more layers of security a system emcompasses, then a greater burden is placed on performance, and hence scalability. This eventually affects the bottom line—more powerful and expensive resources are required to achieve a system's performance and scalability requirements.

Manageability

As the systems we build become more distributed and complex in their interactions, their management and operations comes to the fore. We need to pay attention to ensuring every component is operating as expected, and the performance is continuing to meet expectations.

The platforms and technologies we use to build our systems provide a multitude of standards-based and proprietary monitoring tools that can be used for these purposes. Monitoring dashboards can be used to check the ongoing health and behavior of each system component. These dashboards, built using highly customizable and open tools such as Grafana, can display system metrics, and send alerts when various thresholds or events occur that need operator attention. The term used for this sophisticated monitoring capability is *observability*.

There are various APIs such as Java's MBeans, AWS CloudWatch and Python's AppMetrics that engineers can utilize to capture custom metrics for their systems - a typical example is request response times. Using these APIs, monitoring dashboards can be tailored to provide live charts and graphs that give deep insights into a system's behaviour. Such insights are invaluable to ensure ongoing operations and highlight parts of the system that may need optimization or replication.

Scaling a system invariably means adding new system components - hardware and software. As the number of components grows, we have more moving parts to monitor and manage. This is never effort-free. It adds complexity to the operations of the system as and costs in terms of monitoring code that needs developing and observability platform evolution.

The only way to control the costs and complexity of manageability as we scale is through automation. This is where the world of devops enters the scene. *Devops* is a set of practices and tooling that combine software development and system operations. Devops reduces the development life cycle for new features, and automates ongoing test, deployment, management, upgrade and monitoring of the system. It's an integral part of any successful scalable system.

Summary and Further Reading

The ability to scale an application quickly and cost-effectively should be a defining quality of the software architecture of contemporary Internet-

facing applications. We have two basic ways to achieve scalability, namely increasing system capacity, typically through replication, and performance optimization of system components.

Like any software architecture quality attribute, scalability cannot be achieved in isolation. It inevitably involves complex trade-offs that need to be tuned to an application's requirements. I'll be discussing these fundamental trade-offs throughout the remainder of this book, starting in fact in the next chapter when I describe concrete architecture approaches to achieve scalability.

References

[1] Ian Gorton, Paul Greenfield, Alex Szalay, and Roy Williams. 2008. Data-Intensive Computing in the 21st Century. Computer 41, 4 (April 2008), 30–32.

[2] Rachel Potvin and Josh Levenberg. 2016. Why Google stores billions of lines of code in a single repository. Commun. ACM 59, 7 (July 2016), 78–87.

[3] Ian Gorton (2017). Chapter 2. Hyperscalability – The Changing Face of Software Architecture. 10.1016/B978-0-12-805467-3.00002-8.

---

1 The report is not for the squeamish. Here's one PG-13 illustrative data point – they had 42 billion visits in 2019! Some of the statistics will definitely make your eyes bulge!

2 Transport Layer Security - Wikipedia

3 https://en.wikipedia.org/wiki/Public-key_cryptography

4 https://www.hashicorp.com/blog/understanding-the-performance-overhead-of-encryption

5 https://www.oreilly.com/library/view/building-secure-and/9781492083115/ch01.html#confidentialitycomma_integritycomma_ava

Chapter 2. Distributed Systems Architectures: An Introduction

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 2nd chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

In this chapter I'll broadly cover some of the fundamental approaches to scaling a software system. You can regard this as a 30,000 feet view of the content that is covered in Parts 2, 3 and 4 of this book. I'll take you on a tour of the main architectural approaches used for scaling a system, and give pointers to later chapters where these issues are dealt with in depth. You can regard this as a "Why" we need these architectural tactics, and the remainder of the book explaining the "How".

The type of systems this book is oriented towards are the internet-facing systems we all utilize every day. I'll let you name your favorite. These systems accept requests from users through Web and mobile interfaces, store and retrieve data based on user requests or events (e.g. a GPS-based system), and have some *intelligent* features such as providing recommendations or notifications based on previous user interactions.

I'll start with a simple system design and show how it can be scaled. In the process, I'll introduce several concepts that will be covered in much more

detail later in this book. Hence this chapter just gives a broad overview of these concepts and how they aid in scalability – truly a whirlwind tour!

Basic System Architecture

Virtually all massive scale systems start off small and grow due to their success. It's common, and sensible, to start with a development framework such as Ruby on Rails or Django or equivalent, which promotes rapid development to get a system quickly up and running. A typical, very simple software architecture for 'starter' systems which closely resembles what you get with rapid development frameworks is shown in Figure 2-1. This comprises a client tier, application service tier, and a database tier. If you use Rails or equivalent, you also get a framework which hardwires a Model-View-Controller (MVC) pattern for Web application processing and an Object-Relational Mapper (ORM) that generates SQL queries.

With this architecture, users submit requests to the application from their mobile app or Web browser. The magic of Internet networking (see Chapter 3) delivers these requests to the application service which is running on a machine hosted in some corporate or commercial cloud data center. Communications uses a standard application-level network protocol, typically HTTP.

The application service runs code that supports an application programming interface (API) that clients use to format data and send HTTP requests to. Upon receipt of a request, the service executes the code associated with the requested API. In the process, it may read from or write to a database or some other external system, depending on the semantics of the API. When the request is complete, the service sends the results to the client to display in their app or browser.

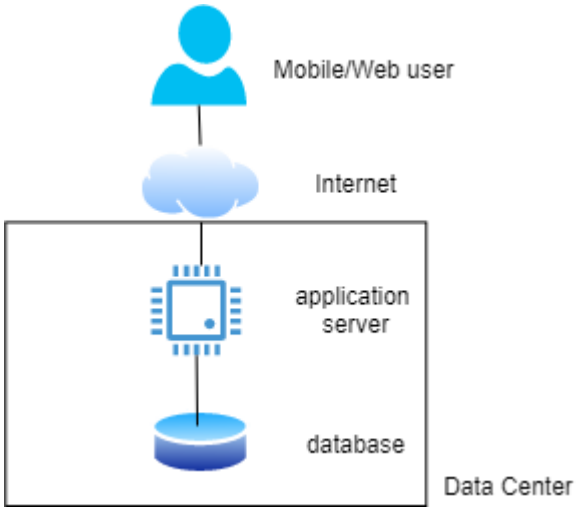

Figure 2-1. Basic Multi-Tier Distributed Systems Architecture

Many, if not most systems conceptually look exactly like this. The application service code exploits a server execution environment that enables multiple requests from multiple users to be processed simultaneously. There's a myriad of these application server technologies – JEE and Spring for Java, Flask for Python[1] – that are widely used in this scenario.

This approach leads to what is generally known as a monolithic architecture.[2] Monoliths tend to grow in complexity as the application becomes more feature rich. All API handlers are built into the same server code body. This eventually makes it hard to modify and test rapidly, and the execution footprint can become extremely heavyweight as all the API implementations run in the same application service.

Still, if request loads stay relatively low, this application architecture can suffice. The service has the capacity to process requests with consistently low latency. But if request loads keep growing, this means latencies will increase as the service has insufficient CPU/memory capacity for the concurrent request volume and hence requests will take longer to process. In these circumstances, our single server is overloaded and has become a bottleneck.

In this case, the first strategy for scaling is usually to 'scale up' the application service hardware. For example, if your application is running on

AWS, you might upgrade your server from a modest t3.xlarge instance with 4 (virtual) CPUs and 16GBs of memory to a t3.2xlarge instance which doubles the number of CPUs and memory available for the application.[3]

Scale up is simple. It gets many real-world applications a long way to supporting larger workloads. It obviously just costs more money for hardware, but that's scaling for you.

It's inevitable however that for many applications the load will grow to a level which will swamp a single server node, no matter how many CPUs and how much memory you have. That's when you need a new strategy – namely scaling out, or horizontal scaling, that I touched on in Chapter 1.

Scale Out

Scaling out relies on the ability to replicate a service in the architecture and run multiple copies on multiple server nodes. Requests from clients are distributed across the replicas so that in theory, if we have N replicas, each server node processes {#requests/N}. This simple strategy increases an application's capacity and hence scalability.

To successfully scale out an application, you need two fundamental elements in our design. As illustrated in Figure 2-2, these are:

Load balancer

> All user requests are sent to a load balancer, which chooses a service replica target to process the request. Various strategies exist for choosing a target service, all with the core aim of keeping each resource equally busy. The load balancer also relays the responses from the service back to the client. Most load balancers belong to a class of Internet components known as reverse proxies,[4] which control access to server resources for client requests. As an intermediary, reverse proxies add an extra network hop for a request, and hence need to be extremely low latency to minimize the overheads they introduce. There are many off-the-shelf load balancing solutions as well as cloud-provider specific

ones, and I'll cover the general characteristics of these in much more detail in Chapter 5.

Stateless services

For load balancing to be effective and share requests evenly, the load balancer must be free to send consecutive requests from the same client to different service instances for processing. This means the API implementations in the services must retain no knowledge, or state, associated with an individual client's session. When a user accesses an application, a user session is created by the service and a unique session is managed internally to identify the sequence of user interactions and track session state. A classic example of session state is a shopping cart. To use a load balancer effectively, the data representing the current contents of a user's cart must be stored somewhere – typically a data store – such that any service replica can access this state when it receives a request as part of a user session. In Figure 2-2 this is labeled as a Session Store.

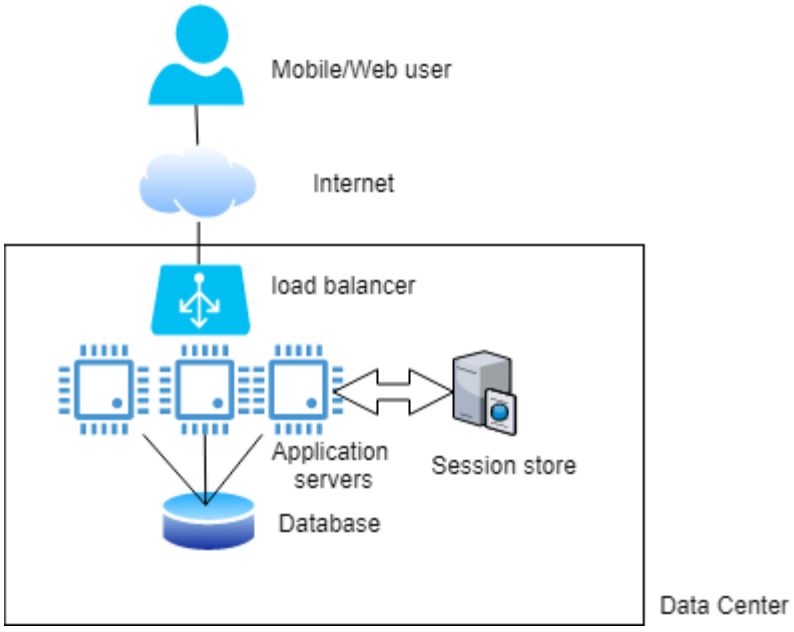

Figure 2-2. Scale out Architecture

Scale out is attractive as, in theory, you can keep adding new (virtual) hardware and services to handle increased request loads and keep request

latencies consistent and low. As soon as you see latencies rising, you deploy another server instance. This requires no code changes with stateless services and hence is relatively cheap – you just pay for the hardware you deploy.

Scale out has another highly attractive feature. If one of the services fails, the requests it is processing will be lost. But as the failed service manages no session state, these requests can be simply reissued by the client and sent to another service instance for processing. This means the application is resilient to failures in the service software and hardware, thus enhancing the application's availability.

Unfortunately, as with any engineering solution, simple scaling out has limits. As you add new service instances, the request processing capacity grows, potentially infinitely. At some stage however, reality will bite and the capability of your single database to provide low latency query responses will diminish. Slow queries will mean longer response times for clients. If requests keep arriving faster than they are being processed, some system component will become overloaded and fail due to resource exhaustion, and clients will see exceptions and request timeouts. Essentially your database has become a bottleneck that you must engineer away in order to scale your application further.

Scaling the Database with Caching

Scaling up by increasing the number of CPUs, memory and disks in a database server can go a long way to scaling a system. For example, at the time of writing Google Cloud Platform can provision a SQL database on a *db-n1-highmem-96* node, which has 96 vCPUs, 624GB of memory, 30TBs of disk and can support 4000 connections. This will cost somewhere between $6K and $16K per year, which sounds a good deal to me! Scaling up is a very common database scalability strategy.

Large databases need constant care and attention from highly skilled database administrators to keep them tuned and running fast. There's a lot of wizardry in this job – e.g. query tuning, disk partitioning, indexing, on-

node caching – and hence database administrators are valuable people that you want to be very nice to. They can make your application services highly responsive indeed.

In conjunction with scale up, a highly effective approach is querying the database as infrequently as possible from your services. This can be achieved by employing *distributed caching* in the scaled out service tier. Caching stores recently retrieved and commonly accessed database results in memory so they can be quickly retrieved without placing a burden on the database. For example, the weather forecast for the next hour won't change, but may be queried by 100s or thousands of clients. You can use a cache to store the forecast once it is issued. All client requests will read from the cache until the forecast expires.

For data that is frequently read and changes rarely, your processing logic can be modified to first check a distributed cache, such as a Redis[5] or memcached[6] store. These cache technologies are essentially distributed Key-Value stores with very simple APIs. This scheme is illustrated in Figure 2-3. Note that the *Session Store* from Figure 2-2 has disappeared. This is because you can use a general-purpose distributed cache to store session identifiers along with application data.

Accessing the cache requires a remote call from your service. If the data you need is in the cache, on a fast network you can expect sub-millisecond cache reads. This is far less expensive than querying the shared database instance, and also doesn't require a query to contend for typically scarce database connections.

Introducing a caching layer also requires your processing logic to be modified to check for cached data. If what you want is not in the cache, your code must still query the database and load the results into the cache as well as return it to the caller. You also need to decide when to remove or invalidate cached results – this depends on the nature of your data (e.g. weather forecasts expire naturally) and your application's tolerance to serving out of date, known as stale, results to clients.

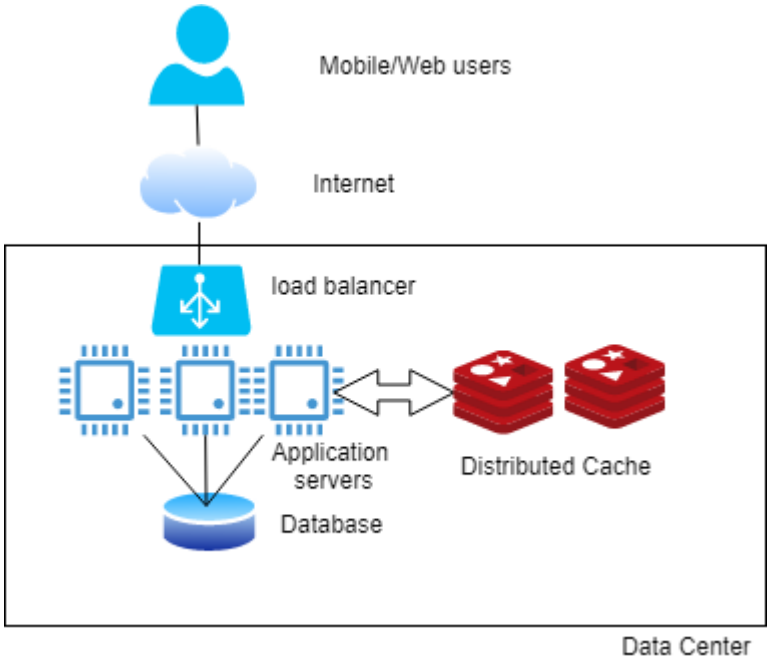

Figure 2-3. Introducing Distributed Caching

A well-designed caching scheme can be absolutely invaluable in scaling a system. Caching works great for data that rarely changes and is accessed frequently, such as inventory catalogs, event information and contact data. If you can handle a large percentage, like 80% or more, of read requests from your cache, then you effectively buy extra capacity at your databases as they never see a large proportion of requests.

Still, many systems need to rapidly access terabyte and larger data stores that make a single database effectively prohibitive. In these systems, a distributed database is needed.

Distributing the Database

There are more distributed database technologies around in 2020 than you probably want to imagine. It's a complex area, and one I'll cover extensively later in the chapters in Part 3 of this book. In very general terms, there are two major categories:

- Distributed SQL stores from major vendors such as Oracle and IBM. These enable organizations to scale out their SQL database

relatively seamlessly by storing the data across multiple disks that are queried by multiple database engine replicas. These multiple engines logically appear to the application as a single database, hence minimizing code changes. There is also a class of 'born distributed' SQL databases that are commonly known as NewSQL stores that fit in this category.

- Distributed so-called NoSQL stores from a whole array of vendors. These products use a variety of data models and query languages to distribute data across multiple nodes running the database engine, each with their own locally attached storage. Again, the location of the data is transparent to the application, and typically controlled by the design of the data model using hashing functions on database keys. Leading products in this category are Cassandra, MongoDB and Neo4j.

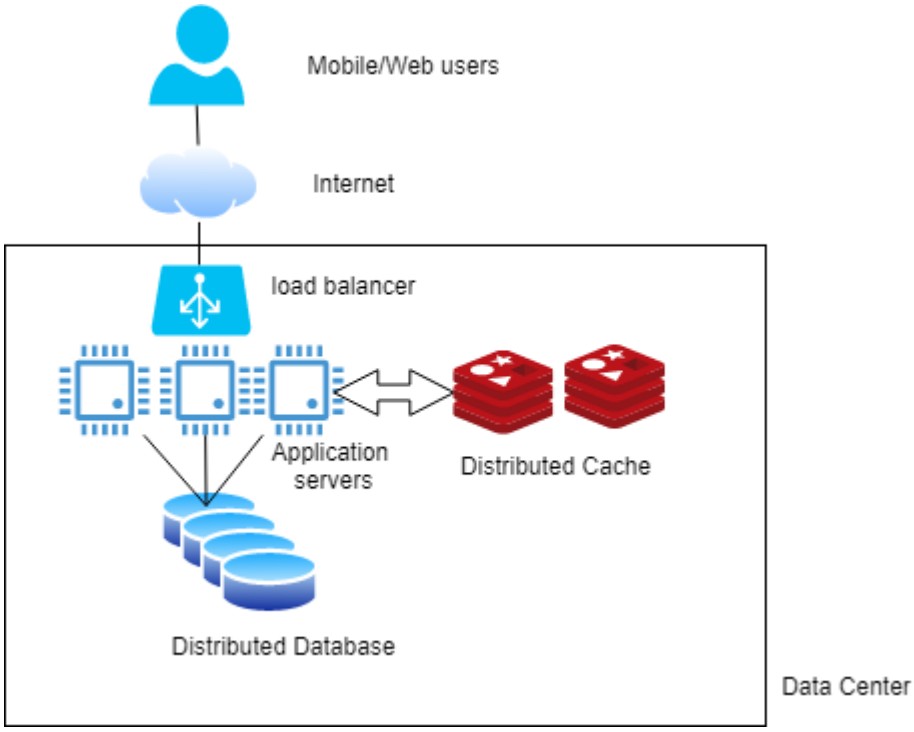

Figure 2-4. Scaling the Data Tier using a Distributed Database

Figure 2-4 shows how our architecture incorporates a distributed database. As the data volumes grow, a distributed database has features to enable the number of storage nodes to be increased. As nodes are added (or removed),

the data managed across all nodes is *rebalanced* to attempt to ensure the processing and storage capacity of each node is equally utilized.

Distributed databases also promote availability. They support replicating each data storage node so if one fails or cannot be accessed due to network problems, another copy of the data is available. The models utilized for replication and the trade-offs these require (spoiler – consistency) are covered in later chapters.

If you are utilizing a major cloud provider, there are also two deployment choices for your data tier. You can deploy your own virtual resources and build, configure, and administer your own distributed database servers. Alternatively, you can utilize cloud-hosted databases. The latter simplifies the administrative effort associated with managing, monitoring and scaling the database, as many of these tasks essentially become the responsibility of the cloud provider you choose. As usual, the no free lunch principle applies.

Multiple Processing Tiers

Any realistic system that you need to scale will have many different services that interact to process a request. For example, accessing a Web page on the Amazon.com web site can require in excess of 100 different services being called before a response is returned to the user.[7]

The beauty of the stateless, load balanced, cached architecture I am elaborating in this chapter is that it's possible to extend the core design principles and build a multi-tiered application. In fulfilling a request, a service can call one or more dependent services, which in turn are replicated and load-balanced. A simple example is shown in Figure 2-5. There are many nuances in how the services interact, and how applications ensure rapid responses from dependent services. Again, I'll cover these in detail in later chapters.

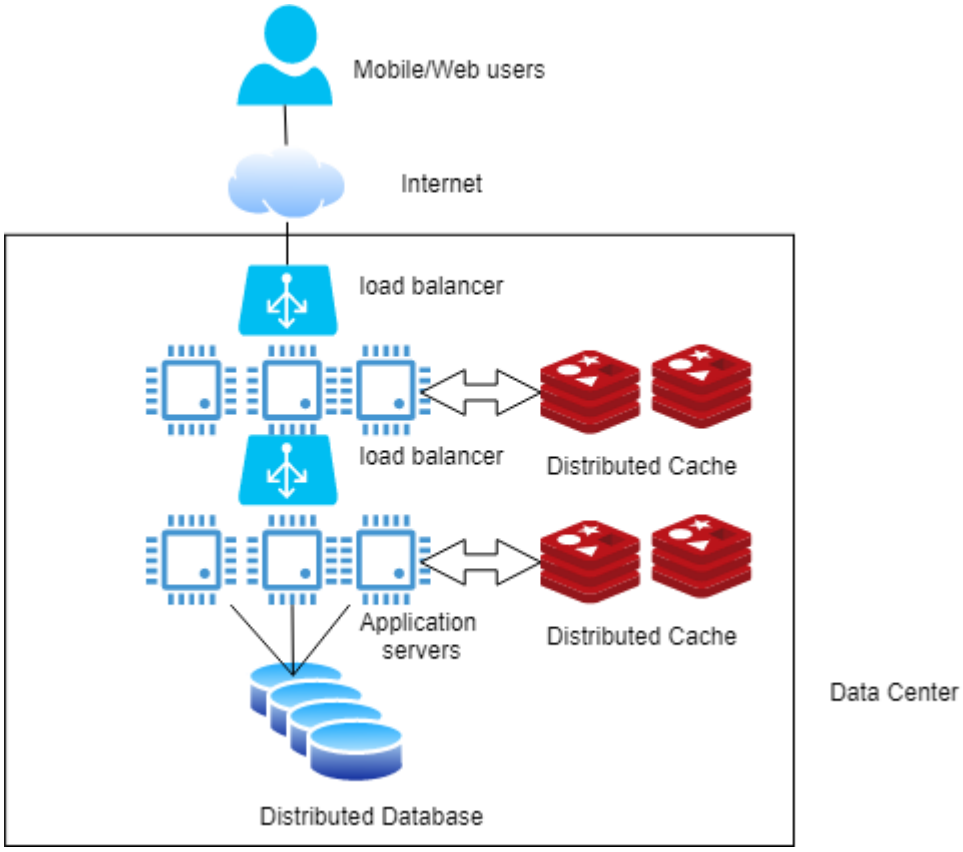

Figure 2-5. Scaling Processing Capacity with Multiple Tiers

This design also promotes having different, load balanced services at each tier in the architecture. For example, Figure 2-6 illustrates two replicated Internet-facing services that both utilized a core service that provides database access. Each service is load balanced and employs caching to provide high performance and availability. This design is often used to provide a service for Web clients and a service for mobile clients, each of which can be scaled independently based on the load they experience. Its commonly called the Backend For Frontend (BFF) pattern.[8]

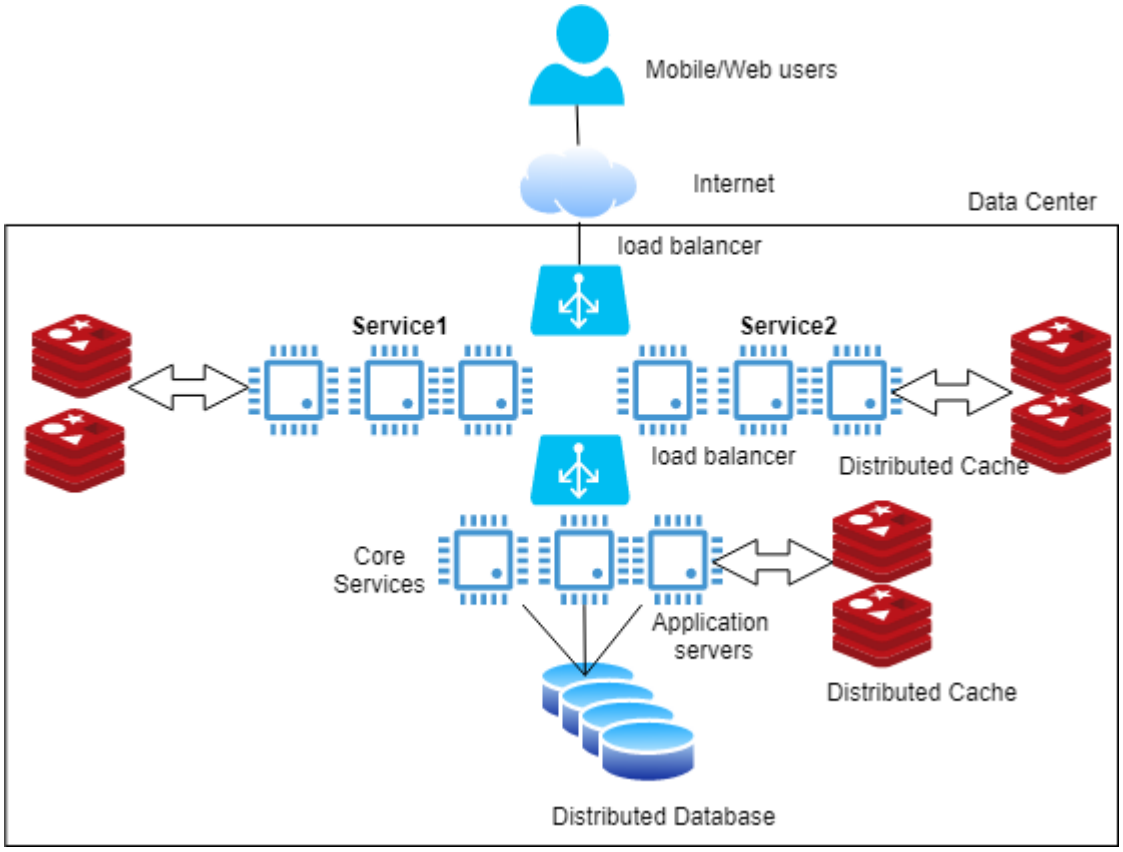

Figure 2-6. Scalable Architecture with Multiple Services

In addition, by breaking the application into multiple independent services, you can scale each based on the service demand. If for example you see an increasing volume of requests from mobile users and decreasing volumes from Web users, it's possible to provision different numbers of instances for each service to satisfy demand. This is a major advantage of refactoring monolithic applications into multiple independent services, which can be separately built, tested, deployed and scaled. I'll explore some of the major issues in designing systems based on such services, known as microservices, in Chapter 9.

Increasing Responsiveness

Most client application requests expect a response. A user might want to see all auction items for a given product category or see the real estate that is available for sale in a given location. In these examples, the client sends a

request and waits until a response is received. This time interval between sending the request and receiving the result is the response time of the request. You can decrease response times by using caching and precalculated responses, but many requests will still result in a database access.

A similar scenario exists for requests that update data in an application. If a user updates their delivery address immediately prior to placing an order, the new delivery address must be persisted so that the user can confirm the address before they hit the ‘purchase’ button. The response time in this case includes the time for the database write, which is confirmed by the response the user receives.

Some update requests however can be successfully responded to without fully persisting the data in a database. For example, the skiers and snowboarders amongst you will be familiar with lift ticket scanning systems that check you have a valid pass to ride the lifts that day. They also record which lifts you take, the time you get on, and so on. Nerdy skiers/snowboarders can then use the resort’s mobile app to see how many lifts they ride in a day.

As a person waits to get on a lift, a scanner device validates the pass using an RFID chip reader. The information about the rider, lift, and time are then sent over the Internet to a data capture service operated by the ski resort. The lift rider doesn’t have to wait for this to occur, as the response time could slow down the lift loading process. There’s also no expectation from the lift rider that they can instantly use their app to ensure this data has been captured. They just get on the lift, talk smack with their friends, and plan their next run.

Service implementations can exploit this type of scenario to improve responsiveness. The data about the event is sent to the service, which acknowledges receipt and concurrently stores the data in a remote queue for subsequent writing to the database. Distributed queueing platforms can be used to reliably sent data from one service to another, typically but not always in a First-In First-Out (FIFO) mode.

Writing a message to a queue is typically much faster than writing to a database, and this enables the request to be successfully acknowledged much more quickly. Another service is deployed to read messages from the queue and write the data to the database. When the user checks their lift rides – maybe 3 hours or 3 days later – the data has been persisted successfully in the database.

The basic architecture to implement this approach is illustrated in Figure 2-7.

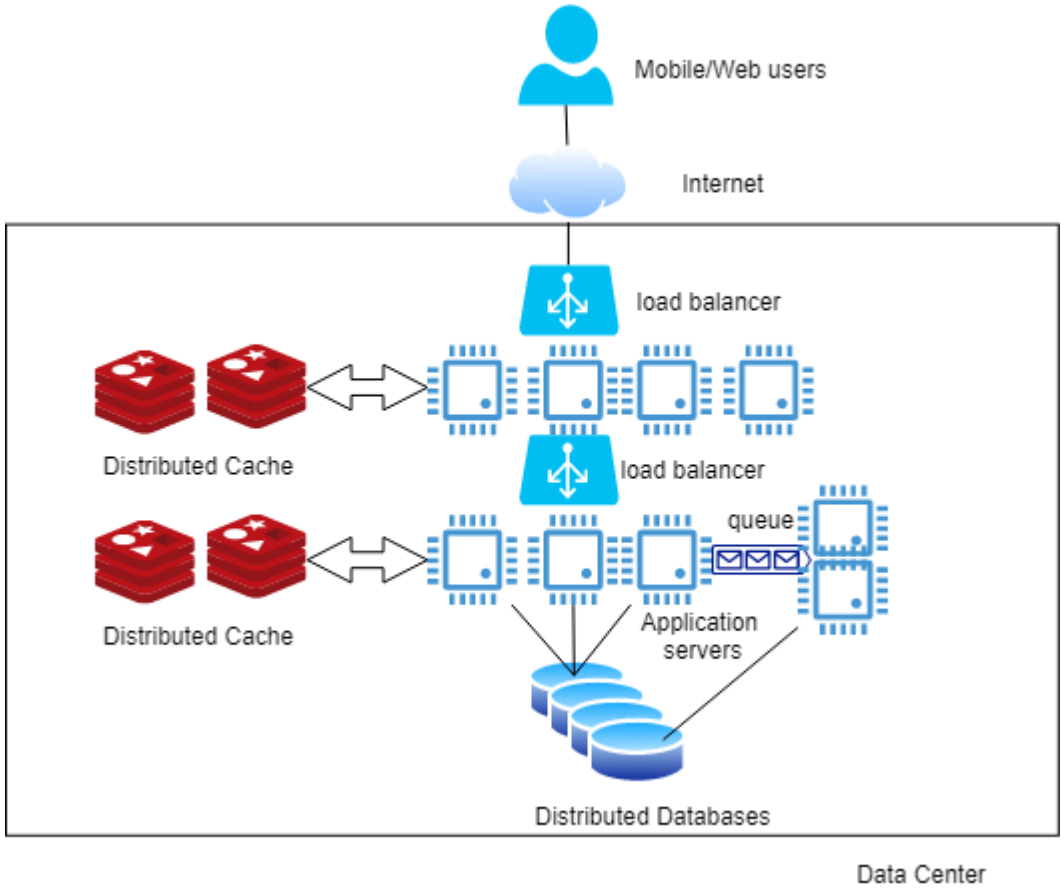

Figure 2-7. Increasing Responsiveness with Queueing

Whenever the results of a write operation are not immediately needed, an application can use this approach to improve responsiveness and hence scalability. Many queueing technologies exist that applications can utilize, and I'll discuss how these operate in Chapter 7. These queueing platforms all provide asynchronous communications. A *producer* service writes to the queue, which acts as temporary storage, while another *consumer* service

removes messages from the queue and makes the necessary updates to, in our example, a database that stores skier lift ride details.

The key is that the data *eventually* gets persisted. Eventually typically means a few seconds at most but use cases that employ this design should be resilient to longer delays without impacting the user experience.

Systems and Hardware Scalability

Even the most carefully crafted software architecture and code will be limited in terms of scalability if the services and datastores run on inadequate hardware. The open source and commercial platforms that are commonly deployed in scalable systems are designed to utilize additional hardware resources in terms of CPU cores, memory and disks. It's a balancing act between achieving the performance and scalability you require, and keeping your costs as low as possible.

That said, there are some cases where upgrading the number of CPU cores and available memory is not going to buy you more scalability. For example:

A code is single threaded. Running this on a node with more cores is not going to improve performance. It'll just use one care at any time. The rest are simply not utilized.

A multithreaded code contains many serialized sections, meaning that only one threaded can proceed at a time to ensure the results are correct. This phenomena is described by Amdahl's Law. This gives us a way to calculate the theoretical speedup of a code when adding more CPU cores based on the amount of code that executes serially.

Two data points from Amdahl's Law are:

- If only 5% of a code executes serially, the rest in parallel, adding more than 2048 cores has essentially no effect
- If 50% of a code executes serially, the rest in parallel, adding more than 8 cores has essentially no effect

This demonstrates why efficient multithreaded code is essential to achieving scalability. If your code is not running as highly independent tasks implemented as threads, then not even money will buy you scalability. That's why I devote Chapter 4 to the topic of multithreading - it's a core knowledge component for building scalable distributed systems.

To illustrate the effect of upgrading hardware, Figure 2-8 shows how the throughput of a benchmark system improves as the database is deployed on more powerful (and expensive) hardware.[9] The benchmark employs a Java service which accepts requests from a load generating client, queries a database and returns the results to the client. The client, service and database run on different hardware resources deployed in the same regions in the AWS cloud.

In the tests, the number of concurrent requests grows from 32 to 256 (x-axis) and each line represents the system throughput (y-axis) for a different hardware configuration on the AWS EC2's Relational Database Service (RDS). The different configurations are listed at the top of the chart, with the least powerful on the left and most powerful on the right. Each client sends a fixed number of requests synchronously over HTTP, with no pause between receiving results from one request and sending the nest. This consequently exerts a high request load on the server.

From this chart, it's possible to make some straightforward observations:

1. In general, the more powerful hardware selected for the database, the higher the throughput. That is good
2. The difference between the db.t2.xlarge and db.t2.2xlarge instances in terms of throughput is minimal. This could be because the service tier is becoming a bottleneck, or our data base model and queries are not exploiting the additional resources of the db.t2.2xlarge RDS instance. Regardless - more bucks, no bang.
3. The two least powerful instances perform pretty well until the request load isincreased to 256 concurrent requests. The dip in

throughput for these two instances indicates they are overloaded and things will only get worse if the request load increases.

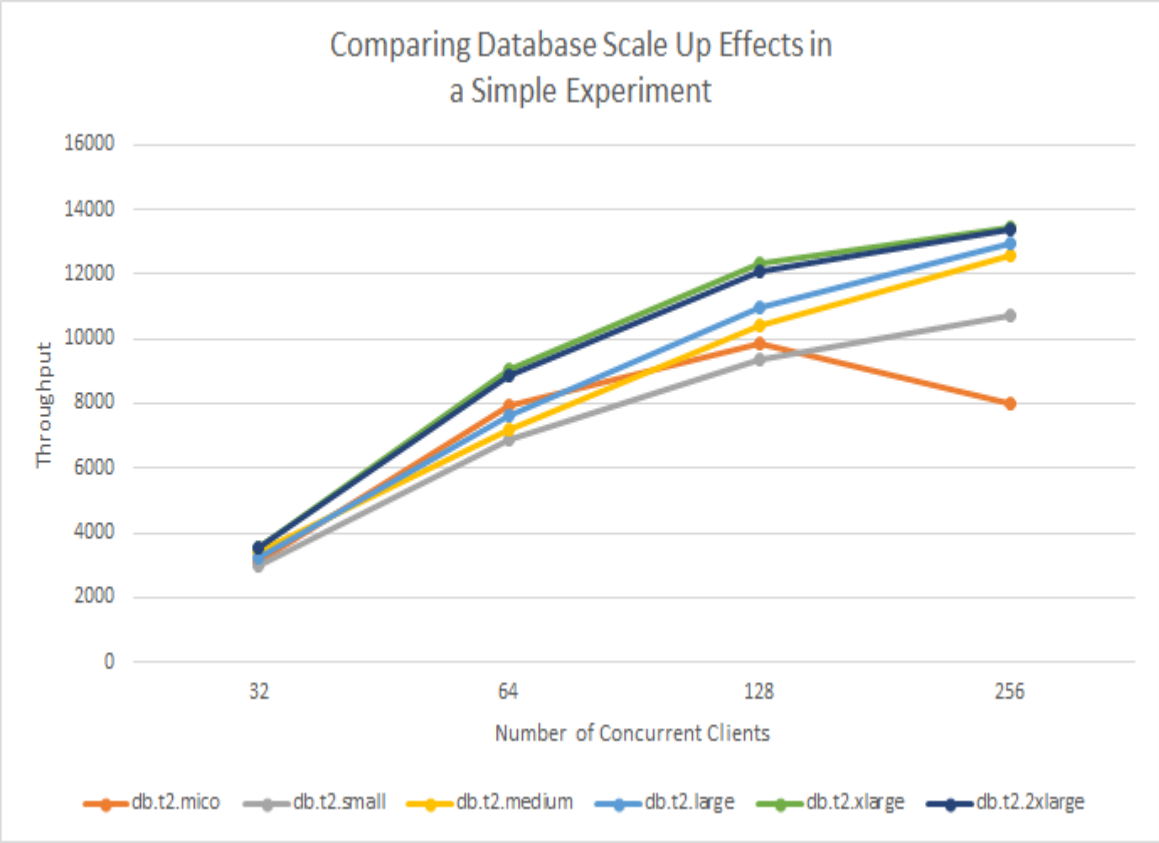

Figure 2-8. An Example of Scaling Up a Database Server

Hopefully this simple example illustrates why scaling through simple upgrading of hardware needs to be approached carefully. Adding more hardware always increases costs, but may not always give the performance improvement you expect. Running simple experiments and taking measurements is essential for assessing the effects of hardware upgrades. It gives you solid data to guide your design, and justify costs to stakeholders.

Summary and Further Reading

In this chapter I've provided a whirlwind tour of the major approaches you can utilize to scale out a system as a collection of communicating services and distributed databases. Much detail has been brushed over, and as you have no doubt realized - in software systems the devil is in the detail.

Subsequent chapters will therefore progressively start to explore these details, starting with some fundamental characteristics of distributed systems in Chapter 3 that everyone should be aware of.

Another area this chapter has skirted around is the subject of software architecture. I've used the term *services* for distributed components in an architecture that implement application business logic and database access. These services are independently deployed processes that communicate using remote communications mechanisms such as HTTP. In architectural terms, these services are most closely mirrored by those in the Service Oriented Architecture (SOA) pattern, an established architectural approach for building distributed systems. A more modern evolution of this approach revolves around microservices. These tend to be more cohesive, encapsulated services that promote continuous development and deployment.

If you'd like a much more in-depth discussion of these issues, and software architecture concepts in general, then Mark Richards' and Neal Ford's book[10] is an excellent place to start.

Finally, there's a class of *big data* software architectures that address some of the issues that come to the fore with very large data collections. One of the most prominent is data reprocessing. This occurs when data that has already been stored and analyzed needs to be re-analyzed due to code or business rule changes. This reprocessing may occur due to software fixes, or the introduction of new algorithms that can derive more insights from the original raw data. There's a good discussion of the Lambda and Kappa architectures, both of which are prominent in this space, in Jay Krepps's article, Questioning the Lambda Architecture.

---

1 https://en.wikipedia.org/wiki/Flask_(web_framework)

2 Mark Richards and Neal Ford, *Fundamentals of Software Architecture: An Engineering Approach* 1st edition, O'Reilly Media, 2020.

3 https://aws.amazon.com/ec2/instance-types/

4 https://en.wikipedia.org/wiki/Reverse_proxy

5 https://redis.io/

6 https://memcached.org/

7 https://www.allthingsdistributed.com/2019/08/modern-applications-at-aws.html

8 https://samnewman.io/patterns/architectural/bff/

9 Results are courtesy of Ruijia Xiao from Northeastern University, Seattle

10 Mark Richards and Neal Ford, *Fundamentals of Software Architecture: An Engineering Approach*, 1st Edition, O'Reilly Media, 2020

Chapter 3. Distributed Systems Essentials

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 3rd chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

As I described in Chapter 2, scaling a system naturally involves adding multiple independently moving parts. We run our software components on multiple machines and our databases across multiple storage nodes, all in the quest of adding more processing capacity. Consequently, our solutions are distributed across multiple machines in multiple locations, with each machine processing events concurrently, and exchanging messages over a network.

This fundamental nature of distributed systems has some profound implications on the way we design, build and operate our solutions. This chapter provides the basic 'nature of the beast' information you need to know to appreciate the issues and complexities of distributed software systems. We briefly cover communications networks hardware and software, remote method invocation, how to deal with the implications of communications failures, distributed coordination, and the thorny issue of time in distributed systems.

Communications Basics

Every distributed system has software components that communicate over a network. If a mobile banking app requests the user's current bank account balance, a (very simplified) sequence of communications occurs along the lines of:

1. The mobile banking app sends a request over the cellular network addressed to the bank to retrieve the user's bank balance.
2. The request is routed across the Internet to where the bank's web servers are located.
3. The bank's web server authenticates the request (checks if it originated from the supposed user) and sends a request to a database server for the account balance.
4. The database server reads the account balance from disk and returns it to the web server
5. The web server sends the balance in a reply message addressed to the app, which is routed over the Internet and the cellular network until the balance magically appears on the screen of the mobile device

It almost sounds simple when you read the above, but in reality, there's a huge amount of complexity hidden beneath this sequence of communications. Let's examine some of these complexities in the following sections.

Communications Hardware

The bank balance request example above will inevitably traverse multiple different networking technologies and devices. The global Internet is a heterogeneous machine, comprising different types of network communications channels and devices that shuttle many millions of messages a second across networks to their intended destinations.

Different types of communications channels exist. The most obvious categorization is wired versus wireless. For each category there are multiple network transmission hardware technologies that can ship bits from one machine to another. Each technology has different characteristics, and the ones we typically care about are speed and range.

For physically wired networks, the two most common types are local area networks (LANs) and wide area networks (WANs). LANs are networks that can connect devices at 'building scale', being able to transmit data over a small number (e.g. 1-2) of kilometers. Contemporary LANs can transport between 100 megabits per second (Mbps) to 1 gigabits per second (Gbps). This is known as the network's bandwidth, or capacity. The time taken to transmit a message across a LAN – the network's latency – is sub-millisecond with modern LAN technologies.

WANs are networks that traverse the globe and make up what we collectively call the Internet. These long-distance connections are the high speed data 'pipelines' connecting cities and countries and continents with fiber optic cables. These cables support a networking technology known as wavelength division multiplexing[1] which makes it possible to transmit up 171 Gbps over 400 different channels, giving more than 70 Terabits per second (Tbps) of total bandwidth for a single fiber link. The fiber cables that span the world normally comprise four or more strands of fiber, giving bandwidth capacity of hundreds of Tbps for each cable.

Latency is more complicated with WANs however. WANs transmit data over 100s to 1000s of kilometers, and the maximum speed that the data can travel in fiber optic cables is the theoretical speed of light. In reality, these cables can't reach the speed of light, but do get pretty close to it as we can see in Table 3-1.

Table 3-1. WAN Speeds

Path	Distance	Time - Speed of Light	Time - Fiber Optic Cable
New York to San Francisco	4,148 km	14 ms	21 ms
New York to London	5,585 km	19 ms	28 ms
New York to Sydney	15,993 km	53 ms	80 ms

Actual times will be slower than this as the data needs to pass through networking equipment known as routers[2]. The global Internet has a complex hub-and-spoke topology, with many potential paths between nodes in the network. Routers are therefore responsible for transmitting data on the physical network connections to ensure data is transmitted across the Internet from source to destination.

Routers are specialized, high speed devices that can handle several hundred Gbps of network traffic, pulling data off incoming connections and sending the data out to different outgoing network connections based on their destination. Routers at the core of the Internet comprise racks of these devices and hence can process 10s to hundreds of Tbps. This is how you

and 1000's of your friends get to watch a steady video stream on Netflix at the same time.

Wireless technologies have different range and bandwidth characteristics. Wi-Fi routers that we are all familiar with in our homes and offices are wireless ethernet networks and use 802.11 protocols to send and receive data. The most widely used Wi-Fi protocol, 802.11ac, allows for maximum (theoretical) data rates of up to 5,400Mbps. The most recent 802.11ax protocol, also known as Wi-Fi 6, is an evolution of 802.11ac technology that promises increased throughput speeds of up to 9.6Gbps. The range of Wi-Fi routers is of the order of 10's of meters, and of course is affected by physical impediments like walls and floors.

Cellular wireless technology uses radio waves to send data from our phones to routers mounted on cell towers, which are generally connected by wires to the core Internet for message routing. Each cellular technology introduces improved bandwidth and other dimensions of performance. The most common technology at the time of writing is 4G LTE wireless broadband. 4G LTE is around 10 times faster than the older 3G, able to handle sustained download speeds around 10 Mbps (peak download speeds are nearer 50 Mbps) and upload speeds between 2 and 5 Mbps.

Emerging 5G cellular networks promise 10x bandwidth improvements over existing 4G, with 1-2 millisecond latencies between devices and cell towers. This is a great improvement over 4G latencies which are in the 20-40 millisecond range. The trade-off is range. 5G base station range operates at about 500m maximum, whereas 4G provides reliable reception at distances of 10-15kms.

This whole collection of different hardware types for networking comes together in the global Internet. The Internet is a heterogeneous network, with many different operators around the world and every type of hardware imaginable. Figure 3-1 shows a simplified view of the major components that comprise the Internet. Tier 1 networks are the global high-speed Internet backbone. There are around 20 Tier 1 Internet Service Providers (ISPs) who manage and control global traffic. Tier 2 ISPs are typically

regional (e.g. one country), have lower bandwidth than Tier 1 ISPs, and deliver content to customers through Tier 3 ISPs. Tier 3 ISPs are the ones that charge your exorbitant fees for your home Internet every month.

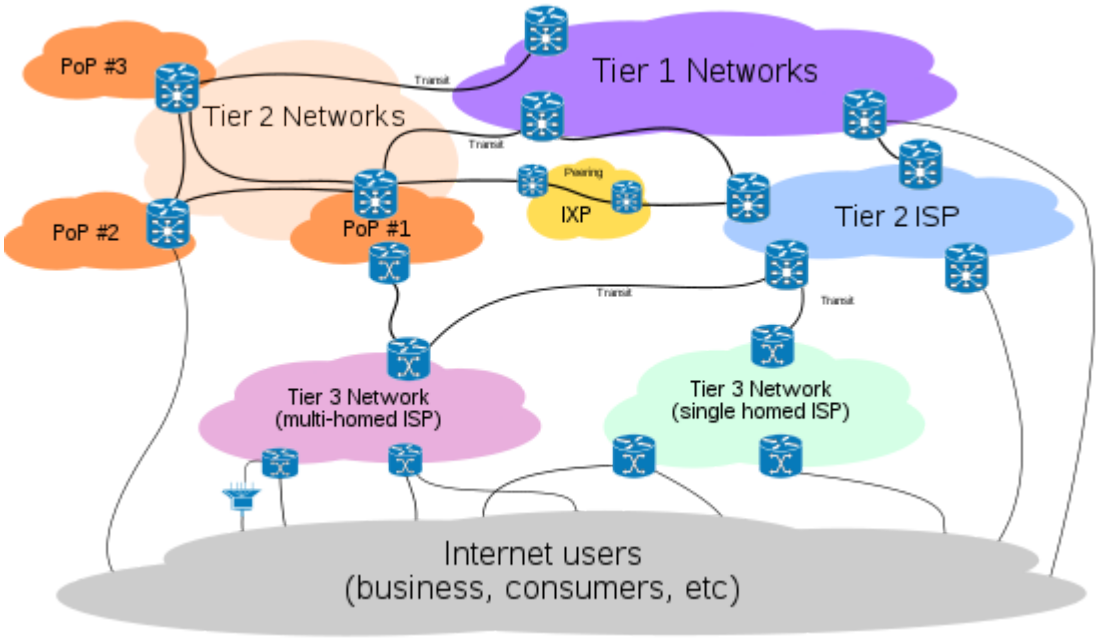

Figure 3-1. Simplified view of the Internet

There's a lot more complexity to how the Internet works than described here. That level of networking and protocol complexity is beyond the scope of this chapter. From a distributed systems software perspective, we need to understand more about the 'magic' that enables all this hardware to route messages from say my cell phone, to my bank and back. This is where the IP protocol comes in.

Communications Software

Software systems on the Internet communicate using the Internet Protocol suite.[3] The Internet Protocol suite specifies host addressing, data transmission formats, message routing and delivery characteristics. There are four abstract layers, which contain related protocols that support the functionality required at that layer. These are, from lowest to highest:

1. The data link layer, specifying communication methods for data across a single network segment. This is implemented by the

device drivers and network cards that live inside your devices.

2. The Internet layer specifies addressing and routing protocols that make it possible for traffic to traverse the independently managed and controlled networks that comprise the Internet. This is the IP protocol in the Internet protocol suite.
3. The transport layer, specifying protocols for reliable and best-effort host-to-host communications. This is where the well-known TCP and UDP protocols live.
4. The application layer, which comprises several application level protocols such as HTTP and SCP.

Each of the higher layer protocols builds on the features of the lower layers. In the following, I'll briefly cover the IP protocol for host discovery and message routing, and the TCP and UDP transport protocols that can be utilized by distributed applications.

Internet Protocol (IP)

IP defines how hosts are assigned addresses on the Internet and how messages are transmitted between two hosts who know each other's addresses.

Every device on the Internet has its own address. These are known as Internet Protocol (IP) addresses. The location of an IP address can be found using an Internet wide directory service known as Domain Naming Service (DNS). DNS is a widely distributed, hierarchical database that acts as the address book of the Internet.

The technology currently used to assign IP addresses, known as Internet Protocol version 4 (IPv4), will eventually be replaced by its successor, IPv6. IPv4 is a 32-bit addressing scheme that before long will run out of addresses due to the number of devices connecting to the Internet. IPv6 is a 128-bit scheme that will offer an (almost) infinite number of IP addresses. As an indicator, in July 2020 about 33% of the traffic processed by Google.com[4] is IPv6.

DNS servers are organized hierarchically. A small number of root DNS servers, which are highly replicated, are the starting point for resolving an IP address. When an Internet browser tries to find a web site, a network host known as the local DNS server that is managed by your employer or ISP, will contact a root DNS server with the requested host name. The root server replies with a referral to a so-called *authoritative* DNS server that manages name resolution for, in our banking example, *.com* addresses. There is an authoritative name server for each top-level Internet domain (e.g. .com, .org, .net, etc).

Next the local DNS server will query the *.com* DNS server, which will reply with the address of the DNS server which knows about all the IP addresses managed by mybank.com. This DNS is queried, and it returns the actual IP address we need to communicate with the application. The overall scheme is illustrated in Figure 3-2.

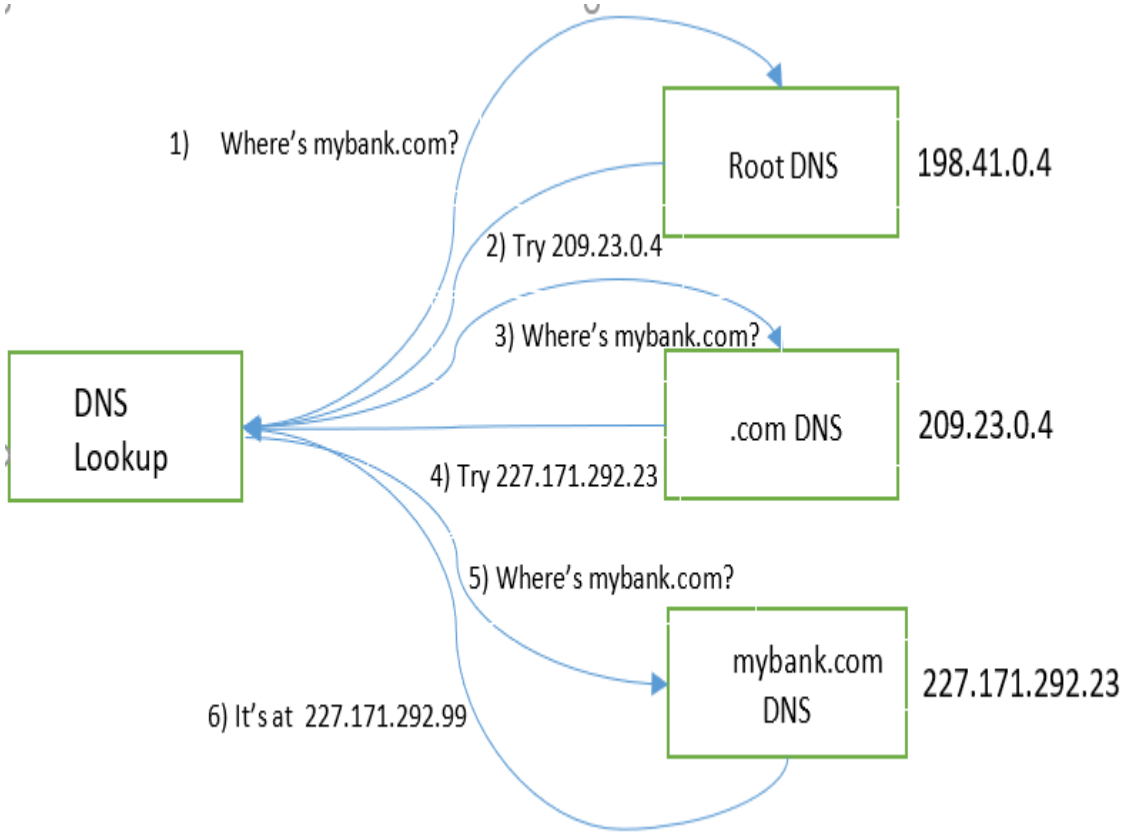

Figure 3-2. Example DNS Lookup for mybank.com

The whole DNS database is highly geographically replicated so there are no single points of failure, and requests are spread across multiple physical servers. Local DNS servers also remember the IP addresses of recently contacted hosts, which is possible as IP addresses don't change very often. This means the complete name resolution process doesn't occur for every site we contact.

Armed with a destination IP address, a host can start sending data across the network as a series of IP data packets. IP has the task of delivering data from the source to the destination host based on the IP addresses in the packet headers. IP defines a packet structure that contains the data to be delivered, along with header data including source and destination IP addresses. Data sent by an application is broken up into a series of packets which are independently transmitted across the Internet.

IP is known as a best-effort delivery protocol. This means it does not attempt to compensate for the various error conditions that can occur during packet transmission. Possible transmission errors include data corruption, packet loss and duplication. In addition, every packet is routed across the Internet from source to destination independently. Treating every packet independently is known as packet-switching. This allows the network to dynamically respond to conditions such as network link failure and congestion, and hence is a defining characteristic of the Internet. This does mean however that different packets may be delivered to the same destination via different network paths, resulting in out-of-order delivery to the receiver.

Because of this design, the IP is unreliable. If two hosts require reliable data transmission, they need to add additional features to make this occur. This is where the next layer in the IP protocol suite, the transport layer, enters the scene.

Transmission Control Protocol (TCP)

Once an application or browser has discovered the IP address of the server it wishes to communicate with, it can send messages using a transport protocol API. This is achieved using Transmission Control Protocol (TCP)

or User Datagram Protocol (UDP), which are the established standard transport protocols for the IP network stack.

Distributed applications can choose which of these protocols to use. Implementations are widely available in mainstream programming languages such as Java, Python and C++. In reality, use of these APIs is not common as higher-level programming abstractions hide the details from most applications. In fact, the IP protocol suite application layer contains several of these application-level APIs, including HTTP, which is very widely used in mainstream distributed systems.

Still, it's important to understand TCP, UDP and their differences. Most requests on the Internet are sent using TCP. TCP is:

- Connection-oriented
- Stream-oriented
- Reliable

I'll explain each of these below.

TCP is known as a connection-oriented protocol. Before any messages are exchanged between applications, TCP uses a 3-step handshake to establish a two-way connection between the client and server applications. The connection stays open until the TCP client calls close to terminate the connection with the TCP server. The server responds by acknowledging the close request before the connection is dropped.

Once a connection is established, a client sends a sequence of requests to the server as a data stream. When a data stream is sent over TCP, it is broken up into individual network packets, with a maximum packet size of 65535 bytes. Each packet contains a source and destination address, which is used by the underlying IP protocol to route the messages across the network.

The Internet is a packet-switched network, which means every packet is individually routed across the network. The route each packet traverses can vary dynamically based on the conditions in the network, such as link

congestion or failure. This means the packets may not arrive at the server in the same order they are sent from the client. To solve this problem, a TCP sender includes a sequence number in each packet so the receiver can reassemble packets into a stream that is identical to the order they were sent.

Reliability is needed as network packets can be lost or delayed during transmission between sender and receiver. To achieve reliable packet delivery, TCP uses a cumulative acknowledgement mechanism. This means a receiver will periodically send an acknowledgement packet that contains the highest sequence number of the packets received without gaps in the packet stream. This implicitly acknowledges all packets sent with a lower sequence number, meaning all have been successfully received. If a sender doesn't receive an acknowledgement within a timeout period, the packet is resent.

TCP has many other features, such as checksums to check packet integrity, and dynamic flow control to ensure a sender doesn't overwhelm a slow receiver by sending data too quickly. Along with connection establishment and acknowledgments, this makes TCP a relatively heavyweight protocol, which trades off reliability over efficiency.

This is where UDP comes into the picture. UDP is a simple connectionless protocol, which exposes the user's program to any unreliability of the underlying network. There is no guarantee of in order delivery, or even delivery for that matter. It can be thought of as a thin veneer (layer) on top of the underlying IP protocol, and deliberately trades off raw performance over reliability.

This however is highly appropriate for many modern applications where the odd lost packet has very little effect. Think streaming movies, video conferencing and gaming, where one lost packet is unlikely to be perceptible by a user.

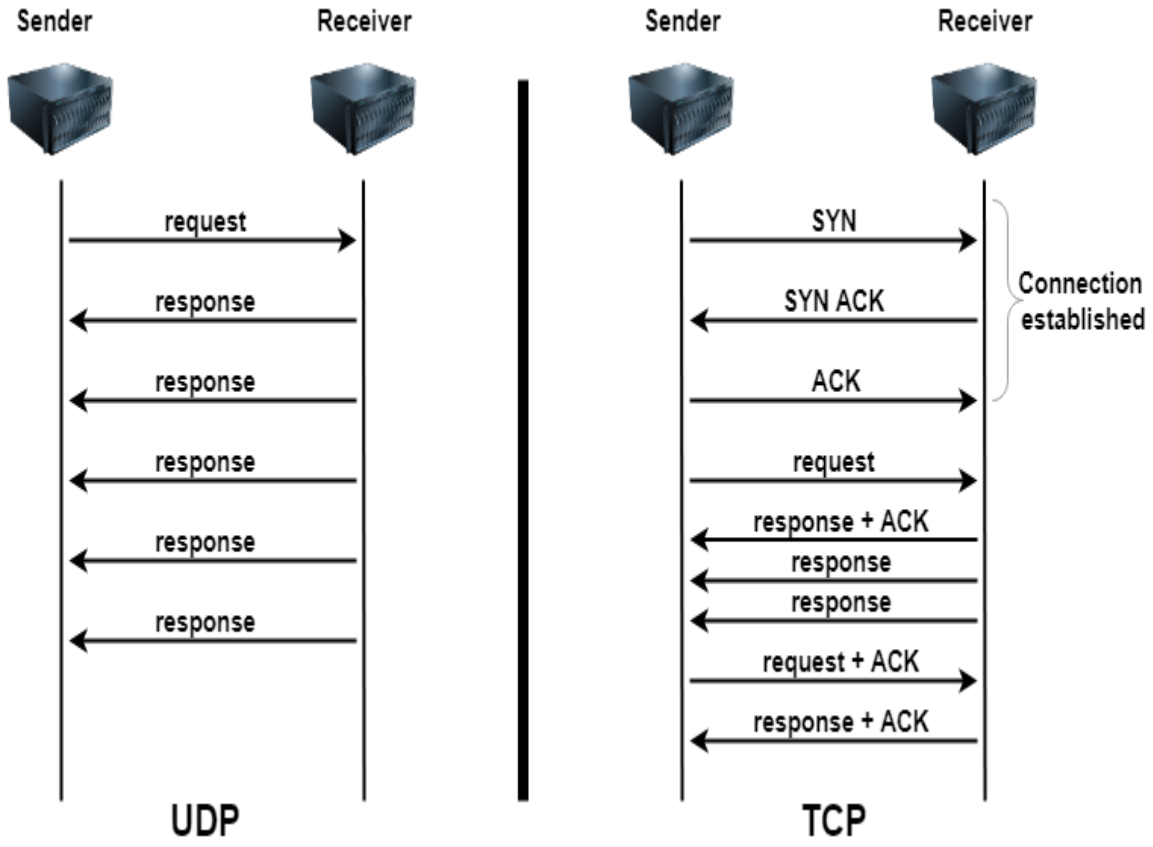

Figure 3-3. Comparing TCP and UDP

Figure 3-3 depicts some of the major differences between TCP and UDP. TCP incorporates a connection establishment 3-packet handshake (SYN, SYN ACK), and piggybacks acknowledgements (ACK) of packets so that any packet loss can be handled by the protocol. There's also a TCP connection close phase involving a 4-way handshake that is not shown in the diagram. UDP dispenses with connection establishment, tear down, acknowledgements and retries. Hence applications using UDP need to be tolerant of packet loss and client or server failures and behave accordingly.

Remote Method Invocation

It's perfectly feasible to write our distributed applications using low-level APIs that interact directly with the transport layer protocols TCP and UDP. The most common approach is the standardized sockets library - see the brief overview in the sidebar. This is something you'll hopefully never need

to do, as sockets are complex and error prone. Essentially sockets create a bi-directional pipe between two nodes that you can use to send streams of data. There are luckily much better ways to build distributed communications as I'll describe in this section. These approaches abstract away much of the complexity of using sockets. However sockets lurk underneath, so some knowledge is necessary.

AN OVERVIEW OF SOCKETS

A socket is one endpoint of a two-way network connection between a client and a server. Sockets are identified by a combination of the nodes' IP address and an abstraction known as a *port*. A port is a unique numeric identifier, which allows a node to support communications for multiple applications running on the node. Each {IP Address, port} combination can be associated with an application. This combination forms a unique address that is used by the transport layer to deliver data to the correct application.

Each node can support 65,535 TCP ports and another 65,535 UDP ports. A connection is identified by a combination of source socket and destination socket addresses. Once the connection is created, the client sends data to the server in a stream, and the server responds with results. The sockets library supports both protocols, with the SOCK_STREAM option for TCP, and the SOCK_DGRAM for UDP.

You can write your distributed applications directly to the sockets API, which is an operating system core component. Socket APIs are available in all mainstream programming languages. The sockets library is however a low level, hard to use API, and should be avoided unless you have a real need to write system level code.

Using sockets, in our mobile banking example, the client might request a balance for the user's checking account. Ignoring specific language issues (and security!!), the client could send a message payload as follows over a connection to the server:

```
{“balance”, “000169990”}
```

In this message, “balance” represents the operation we want the server to execute, and “000169990” is the bank account number.

In the server, we need to know that the first string in the message is the operation identifier, and based on this value being “balance”, the second is the bank account number. The server then uses these values to presumably query a database, retrieve the balance and send back the results, perhaps as a message formatted with the account number and current balance, as below:

```
{“000169990”, “220.77”}
```

In any complex system, the server will support many operations. In mybank.com, there might be for example “login”, “transfer”, “address”, “statement”, “transactions”, and so on. Each will be followed by different message payloads that the server needs to interpret correctly to fulfill the client’s request.

What we are defining here is an *application specific protocol*. As long as we send the necessary values in the correct order for each operation, the server will be able to respond correctly. If we have an erroneous client that doesn’t adhere to our application protocol, well, our server needs to do thorough error checking.

The socket library provides a primitive, low level method for client-server communications. It provides highly efficient communications but is difficult to correctly implement and evolve the application protocol to handle all possibilities. There are better mechanisms.

Stepping back, if we were defining the mybank.com server interface in an object-oriented language such as Java, we would have each operation it can process as a method. Each method is passed an appropriate parameter list for that operation, as shown in the example code below.

```
// Simple mybank.com server interface
public interface MyBank {
    public float balance  (String accNo);
    public boolean  statement(String month) ;
```

```
    // other operations
}
```

There are several advantages of having such an interface, namely:

- Calls from the client to the server can be statically checked by the compiler to ensure they are of the correct format and argument types
- Changes in the server interface (e.g. add a new parameter) force changes in the client code to adhere to the new method signature
- The interface is clearly defined by the class definition, and hence straightforward for a client programmer to understand and utilize

These benefits of an explicit interface are of course well known in software engineering. The whole discipline of object-oriented design is pretty much based upon these foundations, where an interface defines a contract between the caller and callee. Compared to the implicit, application protocol we need to program to with sockets, the advantages are significant.

This fact was recognized reasonably early in the creation of distributed systems. Since the early 1990's, we have seen an evolution of technologies that enable us to define explicit server interfaces and call these across the network using essentially the same syntax as we would in a sequential program. A summary of the major approaches is given in Table 3-2. Collectively they are known as Remote Procedure Call (RPC), or Remote Method Invocation (RMI) technologies.

Table 3-2. Summary of major RPC/RMI Technologies

Technology	Dates	Main features
Distributed Computing Environment (DCE) [a]	Early 1990s	DCE RPC provides a standardized approach for client-server systems. Primary languages were C/C++.
Common Object Request Broker Architecture (CORBA) [b]	Early 1990s	Facilitates language-neutral client-server communications based on an object-oriented Interface Definition Language (IDL). Primary language support in C/C++, Java, Python, Ada.
Java Remote Method Invocation (RMI) [c]	Late 1990s	A pure Java-based remote method invocation that facilitates distributed client-server systems with the same semantics as Java objects.
XML Web Services	2000	Supports client-server communications based on HTTP and XML. Servers define their remote interface in the Web Services Description Language (WSDL)

a http://www.opengroup.org/dce/

b http://www.corba.org

c https://docs.oracle.com/javase/9/docs/specs/rmi/index.html

While the syntax and semantics of these RPC/RMI technologies vary, the essence of how each operates is the same. Let's continue with our Java example of mybank.com to use this as an example of the whole class of approaches. Java offers a Remote Method Invocation (RMI) API for building client-server applications.

Using Java RMI, we can trivially make our MyBank interface example from above into a remote interface, as illustrated below.

```
import java.rmi.*;
// Simple mybank.com server interface
public interface MyBank extends Remote{
    public float balance  (String accNo)
         throws RemoteException;
    public boolean  statement(String month)
         throws RemoteException ;
    // other operations
 }
```

The `java.rmi.Remote` interface serves as a marker to inform the Java compiler we are creating an RMI server. In addition, each method must throw `java.rmi.RemoteException`. These exceptions represent errors that can occur when a distributed call between two objects is invoked over a network. The most common reasons for such an exception would be a communications failure or the server object having crashed.

We then must provide a class that implements this remote interface. The sample code below shows an extract of the server implementation - the complete code for this example is in this book's code github repository.

```
public class MyBankServer extends UnicastRemoteObject
                          implements MyBank  {
   // constructor/method implementations omitted
   public static void main(String args[]){
        try{
          MyBankServer server=new MyBankServer();
          // create a registry in local JVM on default port
          Registry registry = LocateRegistry.createRegistry(1099);
          registry.bind("MyBankServer", server);
          System.out.println("server ready");
```

```
        }catch(Exception e){
                // code omitted for brevity}
        }
    }
```

Points to note are:

- The server extends the `UnicastRemoteObject` class. This essentially provides the functionality to instantiate a remotely callable object.
- Once the server object is constructed, its availability must be advertised to remote clients. This is achieved by storing a reference to the object in a system service known as the *RMI Registry,* and associating a logical name with it – in this example, *MyBankServer*. The registry is a simple directory service that enables clients to look up the location (network address and object reference) of and obtain a reference to an RMI server by simply supplying the logical name it is associated with in the registry.

An extract from the client code to connect to the server is shown below. It obtains a reference to the remote object by performing a `lookup` operation (line 3) in the RMI Registry and specifying the logical name that identifies the server. The reference returned by the lookup operation can then be used to call the server object in the same manner a local object. However there is a difference – the client must be ready to catch a `RemoteException` that will be thrown by the Java runtime when the server object cannot be reached.

```
// obtain a remote reference to the server
MyBank bankServer=
       (MyBank)Naming.lookup("rmi://localhost:1099/MyBankServer");
//now we can call the server
System.out.println(bankServer.balance("00169990"));
```

Figure 3-4 depicts the call sequence amongst the components that comprise a RMI system. The *Stub* and *Skeleton* are objects generated by the compiler from the RMI interface definition, and these facilitate the actual remote

communications. The skeleton is in fact a TCP network endpoint (host, port) that listens for calls to the associated server.

The sequence of operations is as follows:

1. When the server starts, its logical reference is stored in the RMI Registry. This entry contains the Java client stub that can be used to make remote calls to the server.
2. The client queries the registry, and the stub for the server is returned.
3. The client stub accepts a method call to the server interface from the Java client implementation.
4. The stub transforms the request into one or more network packets that are sent to the server host. This transformation process is known as marshalling.
5. The skeleton accepts network requests from the client, and unmarshalls the network packet data into a valid call to the RMI server object implementation. Unmarshalling is the opposite of marshalling - it takes a sequence of network packets and transforms them into a call to an object.
6. The skeleton waits for the method to return a response.
7. The skeleton marshalls the method results into a network reply packet that is sent the client
8. The stub unmarshalls the data passes the result to the Java client call site

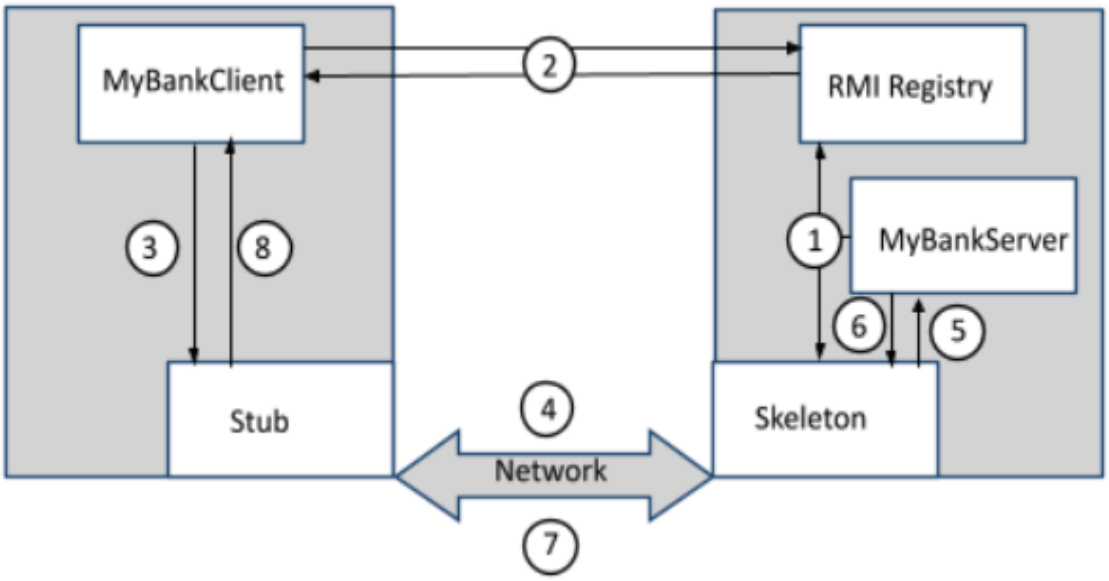

Figure 3-4. Schematic depicting the call sequence for establishing a connection and making a call to a RMI server object

This Java RMI example illustrates the basics that are used for implementing any RPC/RMI mechanism, even in modern languages like Erlang[5] and Go[6]. You are most likely to encounter Java RMI when using the Java Enterprise Edition's (JEE) Enterprise Java Bean (EJB) technology. EJB's are a server side component model built on RMI, which have seen wide usage in the last 20 or so years in enterprise systems.

Regardless of the precise implementation, the basic attraction of RPC/RMI approaches is to provide an abstract calling mechanism that supports *location transparency* for clients making remote server calls. Location transparency is provided by the Registry, or in general any mechanism that enables a client to locate a server through a directory service. This means it is possible for the server to update its network location in the directory without affecting the client implementation.

RPC/RMI is not without its flaws. Marshalling and unmarshalling can become inefficient for complex object parameters. Cross language marshalling – client in one language, server in another – can cause problems due to types being represented differently in different languages, causing subtle incompatibilities. And if a remote method signature changes,

all clients need to obtain a new compatible stub which can be cumbersome in large deployments.

For these reasons, most modern systems are built around simpler protocols based on HTTP and using JSON for parameter representation. Instead of operation names, HTTP verbs (PUT, GET, POST, etc) have associated semantics that are mapped to a specific URL. This approach originated in the work by Roy Fielding on the REST approach[7]. REST has a set of semantics that comprise a RESTful architecture style, and in reality, most systems do not adhere to these. We'll discuss REST and HTTP API mechanisms in the Chapter 5.

Partial Failures

The components of distributed systems communicate over a network. In communications technology terminology, the shared local and wide area networks that our systems communicate over are known as *asynchronous* networks.

With asynchronous networks:

- Nodes can choose to send data to other nodes at any time
- The network is *half-duplex*, meaning that one node sends a request and must wait for a response from the other. These are two separate communications.
- The time for data to be communicated between nodes is variable, due to reasons like network congestion, dynamic packet routing and transient network connection failures.
- The receiving node may not be available due to a software or machine crash.
- Data can be lost. In wireless networks, packets can be corrupted and hence dropped due to weak signals or interference. Internet routers can drop packets during congestion.

- Nodes do not have identical internal clocks, hence they are not synchronized

(This is in contrast with synchronous networks, which essentially are full duplex, transmitting data in both directions at the same time with each node having an identical clock for synchronization[8].)

What does this mean for our applications? Well, put simply, when a client sends a request to a server, how long does it wait until it receives a reply? Is the server node just being slow? Is the network congested and the packet has been dropped by a router? If the client doesn't get a reply, what should it do?

Let's explore these scenarios in detail. The core problem here, namely whether and when a response is received, is known as handling partial failures, and the general situation is depicted in Figure 3-5.

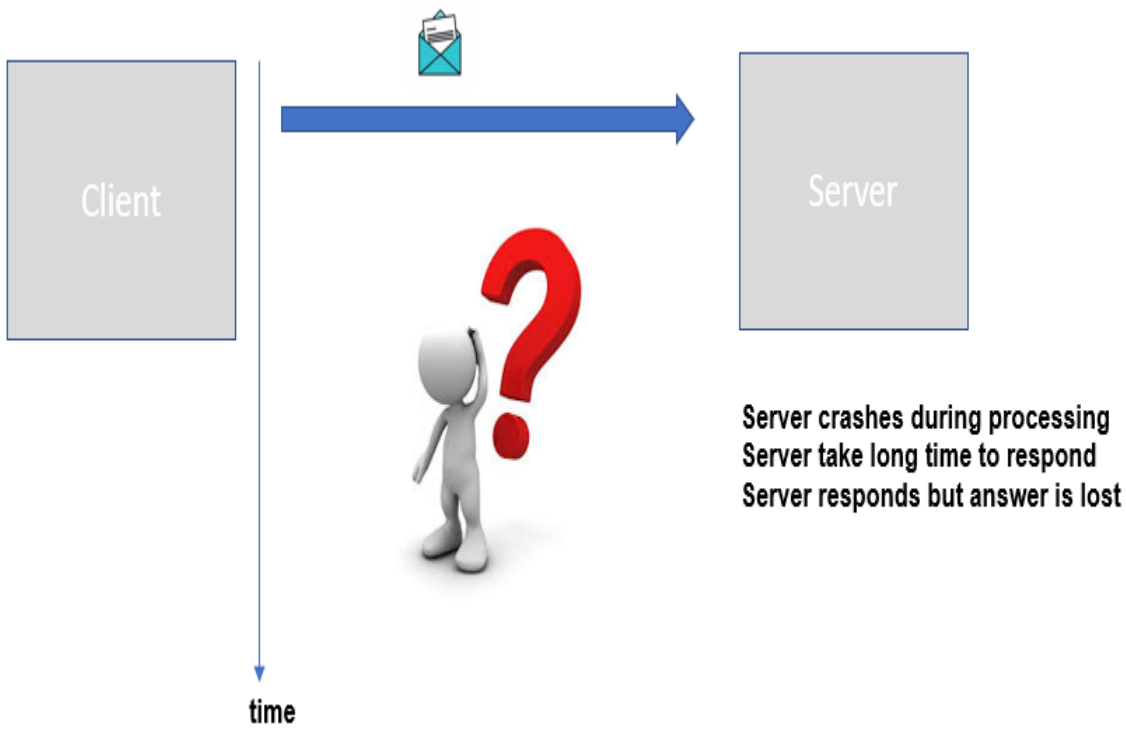

Figure 3-5. Handling Partial Failures

Once a client node has established a TCP connection, it can send a request to a server node and wait for a response. In this situation, the following

outcomes may occur:

1. The request succeeds and a rapid response is received. All is good. (In reality, this outcome occurs for almost every request. Almost is the operative word here though.)
2. The destination IP address lookup may fail. In this case the client rapidly receives an error message and can act accordingly.
3. The IP address is valid but the destination node or target server process has failed. Again the sender will receive an error message and can inform the user.
4. The request is received by the target server, which fails while processing the request and no response is ever sent.
5. The request is received by the target server, which is heavily loaded. It processes the request but takes a long time (e.g 34 seconds) to respond.
6. The request is received by the target server and a response is sent. However, the response is not received by the client due to a network failure.

Numbers (1) to (3) are easy for the client to handle, as a response is received rapidly. A result from the server or an error message – either allows the client to proceed. Failures that can be detected quickly are easy to deal with.

Numbers (4) to (6) pose a problem for the client. They do not provide any insight into the reason why a response has not been received. From the client's perspective, these three outcomes look exactly the same. The client cannot know, without waiting potentially forever, whether the response will arrive eventually, or never arrive. And waiting forever doesn't get much work done.

More insidiously, the client cannot know if the operation succeeded and a server or network failure caused the result to never arrive, or if the request

is on its way - delayed simply due to congestion in the network/server. These faults are collectively known as *crash faults*[9].

The typical solution that clients adopt to handle crash faults is to resend the request after a configured timeout period. This however is fraught with danger, as Figure 3-6 illustrates. The client sends a request to the server to deposit money in a bank account. When it receives no response after a timeout period, it resends the request. What is the resulting balance? The server may have applied the deposit, or it may not, depending on the partial failure scenario.

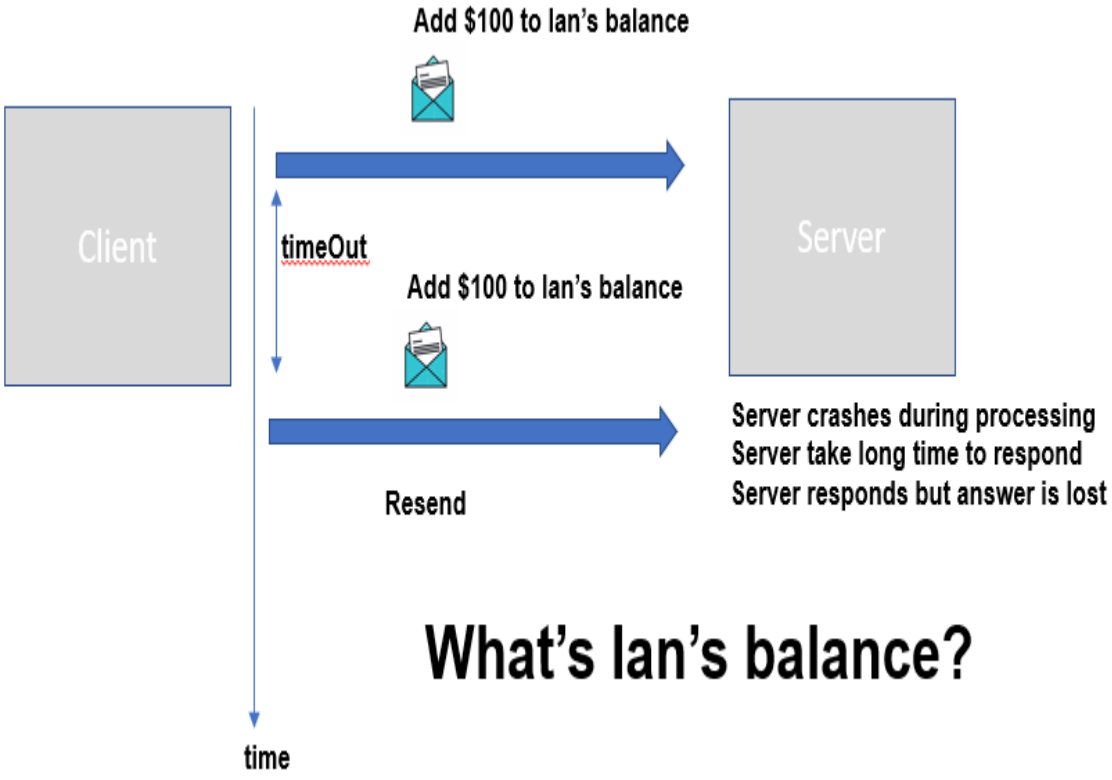

Figure 3-6. Client retries a request after timeout

The chance that the deposit may occur twice is a fine outcome for the customer. The bank though is unlikely to be amused by this possibility. Hence, we need a way to ensure in our server operations implementation that retried, duplicate requests from clients only result in the request being applied once. This is necessary to maintain correct application semantics.

This property is known as idempotence. Idempotent operations can be applied multiple times without changing the result beyond the initial application. This means that for the example in Figure 10, the client can retry the request as many times as it likes, and the account will only be increased by $100.

Requests that make no persistent state changes are naturally idempotent. This means all read requests are inherently safe and no extra work is needed in the server. Updates are a different matter. The system needs to devise a mechanism such that duplicate client requests can be detected by the server, and they do not cause any state changes. In API terms, these endpoints cause mutation of the server state and must therefore be idempotent.

The general approach to building idempotent operations is as follows:

- Clients include a unique idempotence-key in all requests that mutate state. The key identifies a single operation from the specific client or event source. It is usually a composite of a user identifier, such as the session key, and a unique value such as a local timestamp, UUID or a sequence number.
- When the server receives a request, it checks to see if it has previously seen the idempotence key value by reading from a database that is uniquely designed for implementing idempotence. If the key is not in the database, this is a new request. The server therefore performs the business logic to update the application state. It also stores the idempotence key in a database to indicate that the operation has been successfully applied.
- If the idempotence key is in the database, this indicates that this request is a retry from the client and hence should not be processed. In this case the server returns a valid response for the operation so that (hopefully) the client won't retry again.

The database used to store idempotence keys can be implemented in, for example:

- A separate database table or collection in the transactional database used for the application data.
- A dedicated database that provides very low latency lookups, such as a simple key-value store.

Unlike application data, idempotence keys don't have to be retained forever. Once a client receives an acknowledgement of a success for an individual operation, the idempotence key can be discarded. The simplest way to achieve this is to automatically remove idempotence keys from the store after a specific time period, such as 60 minutes or 24 hours, depending on application needs and request volumes.

In addition, an idempotent API implementation must ensure that the application state is modified, **and** the idempotence key is stored. Both must occur for success. If the application state is modified and, due to some failure, the idempotent key is not stored, then a retry will cause the operation to be applied twice. If the idempotence key is stored but for some reason the application state is not modified, then the operation has not been applied. If a retry arrives, it will be filtered out as duplicate as the idempotence key already exists, and the update will be lost.

The implication here is that the updates to the application state and idempotence key store must **both** occur, or **neither** must occur. If you know your databases, you'll recognize this as a requirement for transactional semantics. We'll discuss how distributed transactions are achieved in Chapter 12. Essentially transactions ensure *exactly-once semantics for operations*, which guarantees that all messages will always be processed exactly once – precisely what we need for idempotence.

Exactly once does not mean that there are no message transmission failures, retries and no application crashes. These are all inevitable. The important thing is that the retries eventually succeed and the result is always the same.

We'll return to the issue of communications delivery guarantees in later chapters. As Figure 3-7 illustrates, there's a spectrum of semantics, each with different guarantees and performance characteristics. *At most once*

delivery is fast and unreliable – this is what the UDP protocol provides. *At least once* delivery is the guarantee provided by TCP/IP, meaning duplicates are inevitable. *Exactly-once* delivery, as we've discussed here, requires guarding against duplicates and hence trades off reliability against slower performance.

As we'll see, some advanced communications mechanisms can provide our applications with exactly once semantics. However, these don't operate at Internet scale because of the performance implications. That is why, as our applications are built on the at least once semantics of TCP/IP, we must implement exactly once semantics in our APIs that cause state mutation.

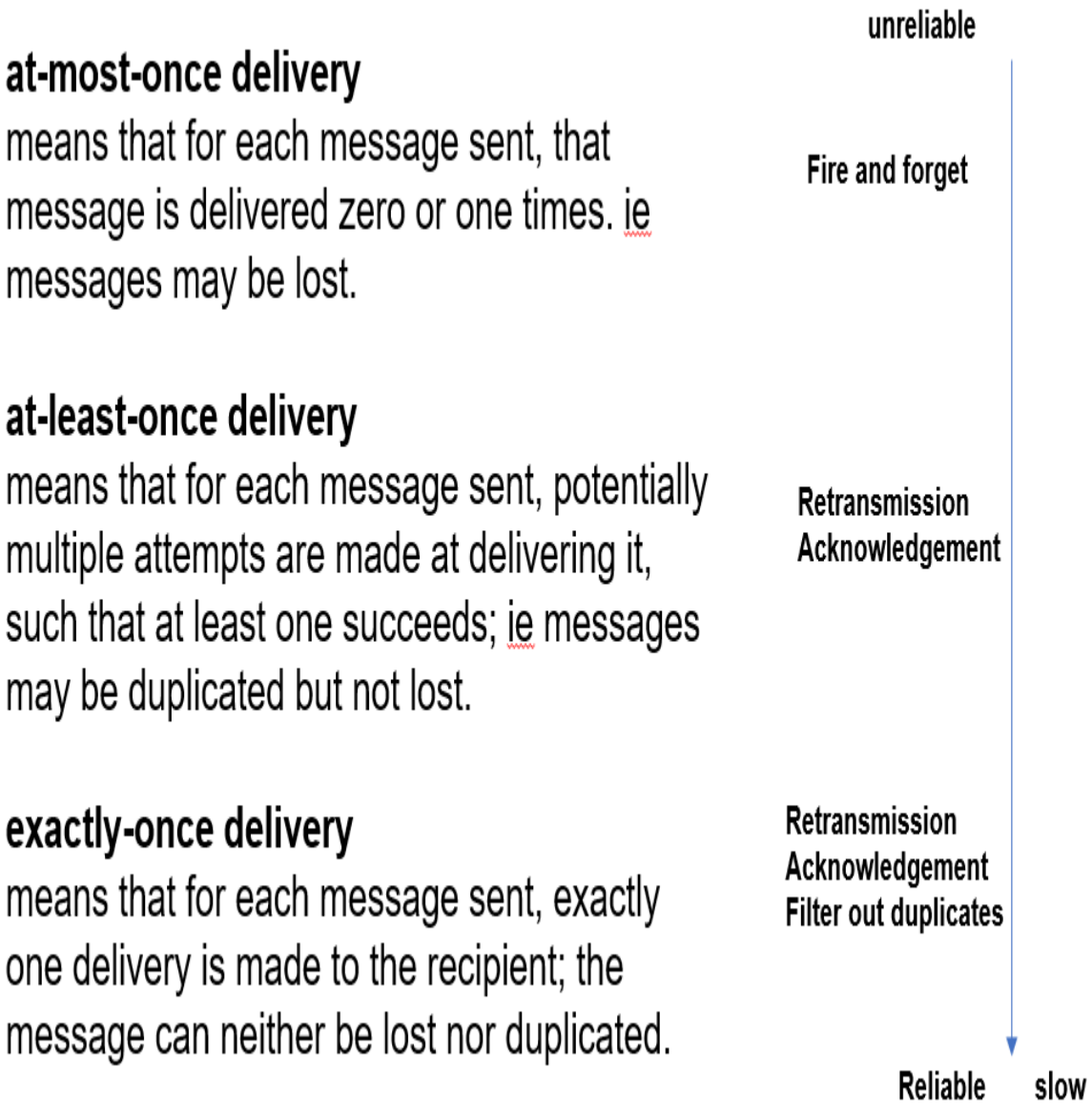

Figure 3-7. Communications Delivery Guarantees

Consensus in Distributed Systems

Crash faults have another implication for the way we build distributed systems. This is best illustrated by the Two Generals Problem[10], which is illustrated in Figure 3-8.

Imagine a city under siege by two armies. The armies lie on opposite sides of the city, and the terrain surrounding the city is difficult to travel through

and visible to snipers in the city. In order to overwhelm the city, it's crucial that both armies attack at the same time. This will stretch the city's defenses and make victory more likely for the attackers. If only one army attacks, then they will likely be repelled.

Given these constraints, how can the two generals reach agreement on the exact time to attack, such that both generals know for certain that agreement has been reached? They both need certainty that the other army will attack at the agreed time, or disaster will ensue.

To coordinate an attack, the first general sends a messenger to the other, with instructions to attack at a specific time. As the messenger may be captured or killed by snipers, the sending general cannot be certain the message has arrived unless they get an acknowledgement messenger from the second general. Of course, the acknowledgement messenger may be captured or killed, so even if the original messenger does get through, the first general may never know. And even if the acknowledgement message arrives, how does the second general know this, unless they get an acknowledgement from the first general?

Hopefully the problem is apparent. With messengers being randomly captured or extinguished, there is no guarantee the two generals will ever reach consensus on the attack time. In fact, it can be proven that it is not possible to *guarantee* agreement will be reached. There are solutions that increase the likelihood of reaching consensus. For example, *Game of Thrones* style, each general may send 100 different messengers every time, and even if most are killed, this increases the probability that at least one will make the perilous journey to the other friendly army and successfully deliver the message.

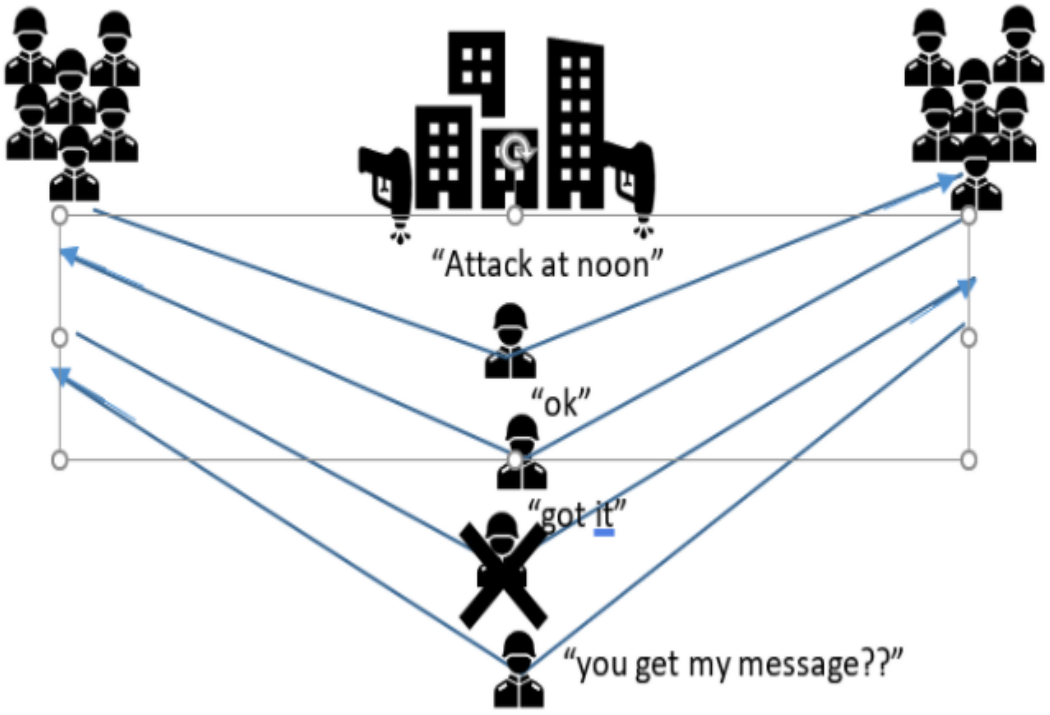

Figure 3-8. The Two Generals Problem

The Two Generals problem is analogous to two nodes in a distributed system wishing to reach agreement on some state, such as the value of a data item that can be updated at either. Partial failures are analogous to losing messages and acknowledgements. Messages may be lost or delayed for an indeterminate period of time – the characteristics of asynchronous networks, as I described earlier in this chapter.

In fact it can be demonstrated that consensus on an asynchronous network in the presence of crash faults, where messages can be delayed but not lost, is impossible to achieve within bounded time. This is known as the FLP Impossibility Theorem[11].

Luckily, this is only a theoretical limitation, demonstrating it's not possible to *guarantee* consensus will be reached with unbounded message delays on an asynchronous network. In reality, distributed systems reach consensus all the time. This is possible because while our networks are asynchronous, we

can establish sensible practical bounds on message delays and retry after a timeout period. FLP is therefore a worst-case scenario, and as I'll discuss algorithms for establishing consensus in distributed databases in Chapter 12.

Finally, we should note the issue of Byzantine failures. Imagine extending the Two Generals problem to N Generals, who need to agree on a time to attack. However, in this scenario, traitorous messengers may change the value of the time of the attack, or a traitorous general may send false information to other generals.

This class of *malicious* failures are known as Byzantine faults and are particularly sinister in distributed systems. Luckily, the systems we discuss in this book typically live behind well-protected, secure enterprise networks and administrative environments. This means we can in practice exclude handling Byzantine faults. Algorithms that do address such malicious behaviors exist, and if you are interested in a practical example, take a look at Blockchain technologies[12] and BitCoin[13].

Time in Distributed Systems

Every node in a distributed system has its own internal clock. If all the clocks on every machine were perfectly synchronized, we could always simply compare the timestamps on events across nodes to determine the precise order they occurred in. If this were reality, many of the problems I'll discuss with distributed systems would pretty much go away.

Unfortunately, this is not the case. Clocks on individual nodes *drift* due to environmental conditions like changes in temperature or voltage. The amount of drift varies on every machine, but values like 10-20 seconds a day are not uncommon. Or with my current coffee machine at home, about 5 minutes a day!

If left unchecked, clock drift would render the time on a node meaningless – like my coffee machine if I don't correct it every few days. To address this problem, a number of *time services* exist. A time service represents an

accurate time source, such as a GPS or atomic clock, which can be used to periodically reset the clock on a node to correct for drift on packet-switched, variable-latency data networks.

The most widely used time service is NTP[14], which provides a hierarchically organized collection of time servers spanning the globe. The root servers, of which there are around 300 worldwide, are the most accurate. Time servers in the next level of the hierarchy (approximately 20,000) synchronize to within a few milliseconds of the root server periodically, and so on throughout the hierarchy, with a maximum of 15 levels. Globally there are more than 175,000 NTP servers.

Using the NTP protocol, a node in an application running an NTP client can synchronize to an NTP server. The time on a node is set by a UDP message exchange with one or more NTP servers. Messages are time stamped and through the message exchange the time taken for message transit is estimated. This becomes a factor in the algorithm used by NTP to establish what the time on the client should be reset to. A simple NTP configuration is shown in Figure 3-9. On a LAN, machines can synchronize to an NTP server within a small number of milliseconds accuracy.

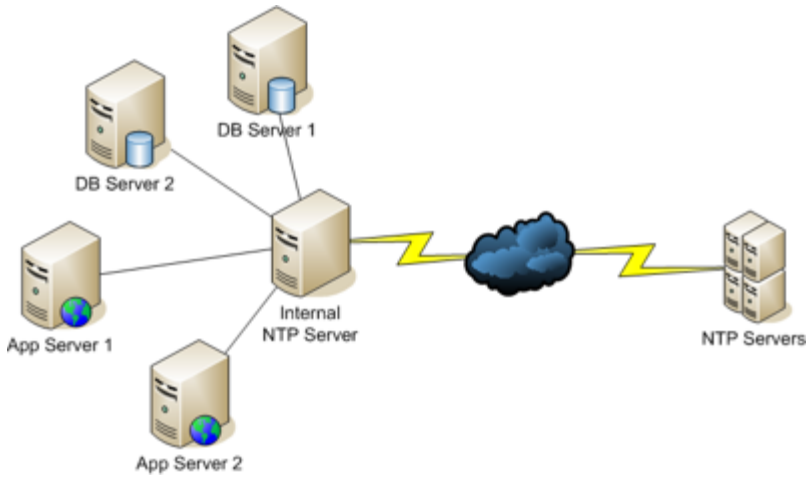

Figure 3-9. Illustrating using the NTP Service

One interesting effect of NTP synchronization for our applications is that the resetting of the clock can move the local node time forwards or backwards. This means that if our application is measuring the time taken for events to occur (e.g. to calculate event response times), it is possible that

the end time of the event may be earlier than the start time if the `NTP` protocol has set the local time backwards.

In fact, a compute node has two clocks. These are:

Time of Day Clock

> This represents the number of milliseconds since midnight January 1970. In Java, you can get the current time using `System.currentTimeMillis()`. This is the clock that can be reset by NTP, and hence may jump forwards or backwards if it is a long way behind or ahead of NTP time.

Monotonic Clock

> This represents the amount of time (in seconds and nanoseconds) since an unspecified point in the past, such as the last time the system was restarted. It will only ever move forward, however it again may not be a totally accurate measure of elapsed time because it stalls during an event such as virtual machine suspension. In Java, you can get the current monotonic clock time using `System.nanoTime()`.

Applications can use an NTP service to ensure the clocks on every node in the system are closely synchronized. It's typical for an application to resynchronize clocks on anything from a one hour to one day time interval. This ensures the clocks remain close in value. Still, if an application really needs to precisely know the order of events that occur on different nodes, clock drift is going to make this fraught with danger.

There are other time services that provide higher accuracy than NTP. Chrony[15] supports the NTP protocol but provides much higher accuracy and greater scalability than NTP – the reason it has recently been adopted by Facebook[16]. Amazon has built the Amazon Time Sync Service by installing GPS and atomic clocks in its data centers. This service is available for free to all Amazon cloud customers.

The takeaway from this discussion is that our applications cannot rely on timestamps of events on different nodes to represent the actual order of

these events. Clock drift even by a second or two makes cross-node timestamps meaningless to compare. The implications of this will become clear when we start to discuss distributed databases in detail.

Summary and Further Reading

This chapter has covered a lot of ground to explain some of the essential characteristics of communications and time in distributed systems. These characteristics are important for application designers and developers to understand.

The key issues that should resonate from this chapter are as follows:

- Communications in distributed systems can transparently traverse many different types of underlying physical networks – e.g. Wi-Fi, wireless, WANs and LANs. Communication latencies are hence highly variable, and influenced by the physical distance between nodes, physical network properties, and transient network congestion. At large scale, latencies between application components are something that should be minimized as much as possible, within the laws of physics of course
- The IP protocol stack ensures reliable communications across heterogeneous networks through a combination of the IP and TCP protocols. Communications can fail due to network communications fabric and router failures that make nodes unavailable, as well as individual node failure. Your code will experience various TCP/IP overheads, for example for connection establishment, and errors when network failures occur. Hence understanding the basics of the IP suite is important for design and debugging.
- RMI/RPC technologies build the TCP/IP layer to provide abstractions for client-server communications that mirror making local method/procedure calls. However, these more abstract programming approaches still need to be resilient to network issues

such as failures and retransmissions. This is most apparent in application APIs that mutate state on the server, and must be designed to be idempotent.

- Achieving agreement, or consensus on state across multiple nodes in the presence of crash faults is not possible in bounded time on asynchronous networks. Luckily, real networks, especially LANs, are fast and mostly reliable, meaning we can devise algorithms that achieve consensus in practice. I'll cover these in Part 3 of the book when we discuss distributed databases.
- There is no reliable global time source that nodes in an application can rely upon to synchronize their behavior. Clocks on individual nodes vary and cannot be used for meaningful comparisons. This means applications cannot meaningfully compare clocks on different nodes to determine the order of events.

These issues will pervade the discussions in the rest of this book. Many of the unique problems and solutions that are adopted in distributed systems stem from these fundamentals. There's no escaping them!

An excellent source for more detailed, more theoretical coverage of all aspects of distributed systems is *George Colouris et al., Distributed Systems: Concepts and Design, 5th Edition, Pearson, 2011*.

Likewise for computer networking, you'll find out all you wanted to know and no doubt more in *James Kurose, Keith Ross, Computer Networking: A Top-Down Approach, 7th Edition, Pearson 2017*.

---

1 https://www.smartoptics.com/this-is-wdm/the-basics-of-wavelength-division-multiplexing-wdm/

2 https://www.easytechjunkie.com/what-is-a-core-router.htm

3 https://docs.oracle.com/cd/E19455-01/806-0916/6ja85398m/index.html

4 https://www.google.com/intl/en/ipv6/statistics.html

5 http://erlang.org/doc/man/rpc.html

6 https://golang.org/pkg/net/rpc/

7 Fielding, Roy Thomas (2000). “Architectural Styles and the Design of Network-based Software Architectures”. Dissertation. University of California, Irvine.

8 https://study.com/academy/lesson/synchronous-asynchronous-networks-in-wan.html#:~:text=Synchronous%20Networks,accomplished%20with%20a%20signal%20clock.

9 https://medium.com/baseds/modes-of-failure-part-1-6687504bfed6#

10 https://en.wikipedia.org/wiki/Two_Generals%27_Problem

11 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. J. ACM 32, 2 (April 1985), 374–382.

12 https://medium.com/@chrshmmmr/consensus-in-blockchain-systems-in-short-691fc7d1fefe

13 https://ieeexplore.ieee.org/abstract/document/8123011

14 www.ntp.org

15 https://chrony.tuxfamily.org/

16 https://engineering.fb.com/production-engineering/ntp-service/

Chapter 4. An Overview of Concurrent Systems

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 4th chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Distributed systems comprise multiple independent pieces of code executing in parallel, or concurrently, on many processing nodes across multiple locations. Any distributed system is hence by definition a concurrent system, even if each node is processing events one at a time. The behavior of the various nodes must of course be coordinated in order to make the application behave as desired.

As I described in Chapter 3, coordinating nodes in a distributed system is fraught with dangers. Luckily, our industry has matured sufficiently to provide complex, powerful software frameworks that hide many of these distributed system perils from our applications – most of the time anyway. The majority of this book focuses on describing how we can utilize these frameworks to build scalable distributed systems.

This chapter however is concerned with concurrent behavior in our systems on a single node. By explicitly writing our software to perform multiple actions concurrently, we can optimize the processing and resource

utilization on a single node, and hence increase our processing capacity both locally and system wide.

I'll use the Java 7.0 concurrency capabilities for examples, as these are at a lower level of abstraction than those introduced in Java 8.0. Knowing how concurrent systems operate 'closer to the machine' is essential foundational knowledge when building concurrent and distributed systems. Once you understand the lower mechanisms for building concurrent systems, the more abstract approaches are easier to optimally exploit. And while this chapter is Java specific, the fundamental problems of concurrent systems don't change when you write systems in other languages. Mechanisms for handling concurrency exist in all mainstream programming languages. The sidebar *Concurrency Models* gives some more details on alternative approaches and how they are implemented in modern languages.

One final point. This chapter is a concurrency primer. It won't teach you everything you need to know to build complex, high performance concurrent systems. It will also be useful if your experience writing concurrent programs is rusty, or you have some exposure to concurrent code in another programming language. The further reading section at the end of the chapter points you to more comprehensive coverage of this topic for those who wish to delve deeper.

Why Concurrency?

Think of a busy coffee shop. If everyone orders a simple coffee, then the barista can quickly and consistently deliver each drink. Suddenly, the person in front of you orders a soy, vanilla, no sugar, quadruple shot iced brew. Everyone in line sighs and starts reading their social media. In two minutes the line is out of the door.

Processing requests in Web applications is analogous to our coffee example. In a coffee shop, we enlist the help of a new barista to simultaneously make coffees on a different machine to keep the line length in control and serve customers quickly. In software, to make applications responsive, we need to

somehow process requests in our server in an overlapping manner, handling requests concurrently.

In the good old days of computing, each CPU was only able to execute a single machine instruction at any instant. If our server application runs on such a CPU, why do we need to structure our software systems to potentially execute multiple instructions concurrently? It all seems slightly pointless.

There is actually a very good reason. Virtually every program does more than just execute machine instructions. For example, when a program attempts to read from a file or send a message on the network, it must interact with the hardware subsystem (disk, network card) that is peripheral to the CPU. Reading data from a modern hard disk takes around 10 milliseconds (ms). During this time, the program must wait for the data to be available for processing.

Now, even an ancient CPU such as a circa 1988 Intel 80386[1] can execute more than 10 million instructions per second (mips). 10ms is 1/100th of a second. How many instructions could our 80386 execute in 1/100th second. Do the math. It's a lot! A lot of wasted processing capacity, in fact.

This is how operating systems such as Linux can run multiple programs on a single CPU. While one program is waiting for an input-output (I-O) event, the operating system schedules another program to execute. By explicitly structuring our software to have multiple activities that can be executed in parallel, the operating system can schedule tasks that have work to do while others wait for I-O. We'll see in more detail how this works with Java later in this chapter.

In 2001, IBM introduced the world's first multicore processor, a chip with two CPUs – see Figure 4-1 for a simplified illustration. Today, even my laptop has 16 CPUs, or cores as they are commonly known. With a multicore chip, a software system that is structured to have multiple parallel activities can be executed concurrently on each core, up the number of available cores. In this way, we can fully utilize the processing resources on a multicore chip, and hence increase our application's processing capacity.

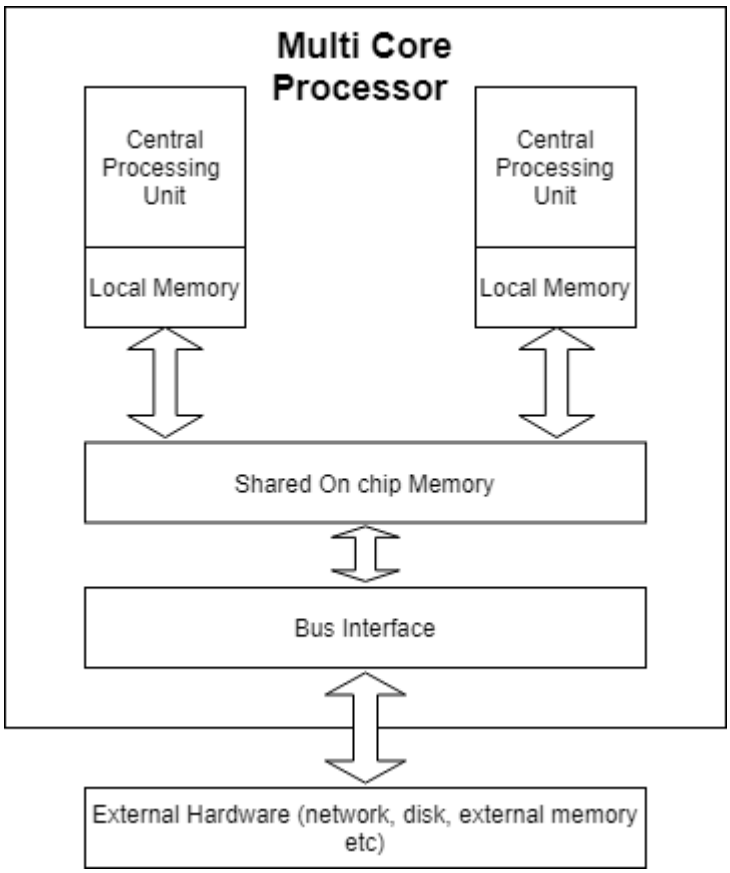

Figure 4-1. Simplified view of a multicore processor

The primary way to structure a software system as concurrent activities is to use *threads*. Virtually every programming language has its own threading mechanism. The underlying semantics of all these mechanisms are similar – there are only a few primary threading models in mainstream use – but obviously the syntax varies by language. In the following sections, I'll explain how threads are supported in Java, and how we need to design our programs to be safe (i.e correct) and efficient when executing in parallel. Armed with this knowledge, leaping into the concurrency features supported in other languages shouldn't be too arduous.

CONCURRENCY MODELS

This chapter describes one model for concurrent systems, based on independently executing threads using locks to operate on shared mutable resources. Concurrency models have been a much studied and explored topic in computer science for roughly the last 50 years. Many theoretical proposals have been put forward, and some of these are implemented in modern programming languages. These models provide alternative approaches for structuring and coordinating concurrent activities in programs. Here's a sampler that you might well encounter in your work.

- **Go:** The Communicating Sequential Processes (CSP) model forms the basis of Go's concurrency features.[2] In CSP, processes synchronize by sending messages using communication abstractions known as channels. In Go, the unit of concurrency is a goroutine, and goroutines communicate by sending messages using unbuffered or buffered channels. Unbuffered channels are used to synchronize senders and receivers, as communications only occur when both goroutines are ready to exchange data.
- **Erlang:** Erlang implements the Actor model of concurrency[3]. Actors are lightweight processes that have no shared state, and communicate by asynchronously sending messages to other actors. Actors use a mailbox, or queue, to buffer messages and can use pattern matching to choose which messages to process.
- **Node.js:** Node.js eschews anything resembling multiple threads, and instead utilizes a single threaded, non-blocking model managed by an event loop.[4] This means when an input-output (IO) operation is required, such as accessing a database, Node.js instigates the operation but does not wait until it completes. Operations are delegated to the operating system to execute asynchronously, and upon completion the results are

placed on the main thread's stack as callbacks. These callbacks are subsequently executed in the event loop. This model works well for codes performing frequent IO requests, as it avoids the overheads associated with thread creation and management. However if your code needs to perform a CPU intensive operation, such as sorting a large list, you only have one thread. This will therefore block all other requests until the sort is complete. Rarely an ideal situation.

Hopefully this gives you a feel for the diversity of concurrency models and primitives in modern programming languages. Luckily, when you know the fundamentals and one model, the rest are straightforward to learn.

Threads

Every software process has a single thread of execution by default. This is the thread that the operating system manages when it schedules the process for execution. In Java, for example, the `main()` function you specify as the entry point to your code defines the behavior of this thread. This single thread has access to the program's environment and resources such as open file handles and network connections. As the program calls methods in objects instantiated in the code, the program's runtime stack is used to pass parameters and manage variable scopes. Standard programming language run time stuff, that we all know and love. This is a sequential process.

In your systems, you can use programming language features to create and execute additional threads. Each thread is an independent sequence of execution and has its own runtime stack to manage local object creation and method calls. Each thread also has access to the process' global data and environment. A simple depiction of this scheme is shown in Figure 4-2.

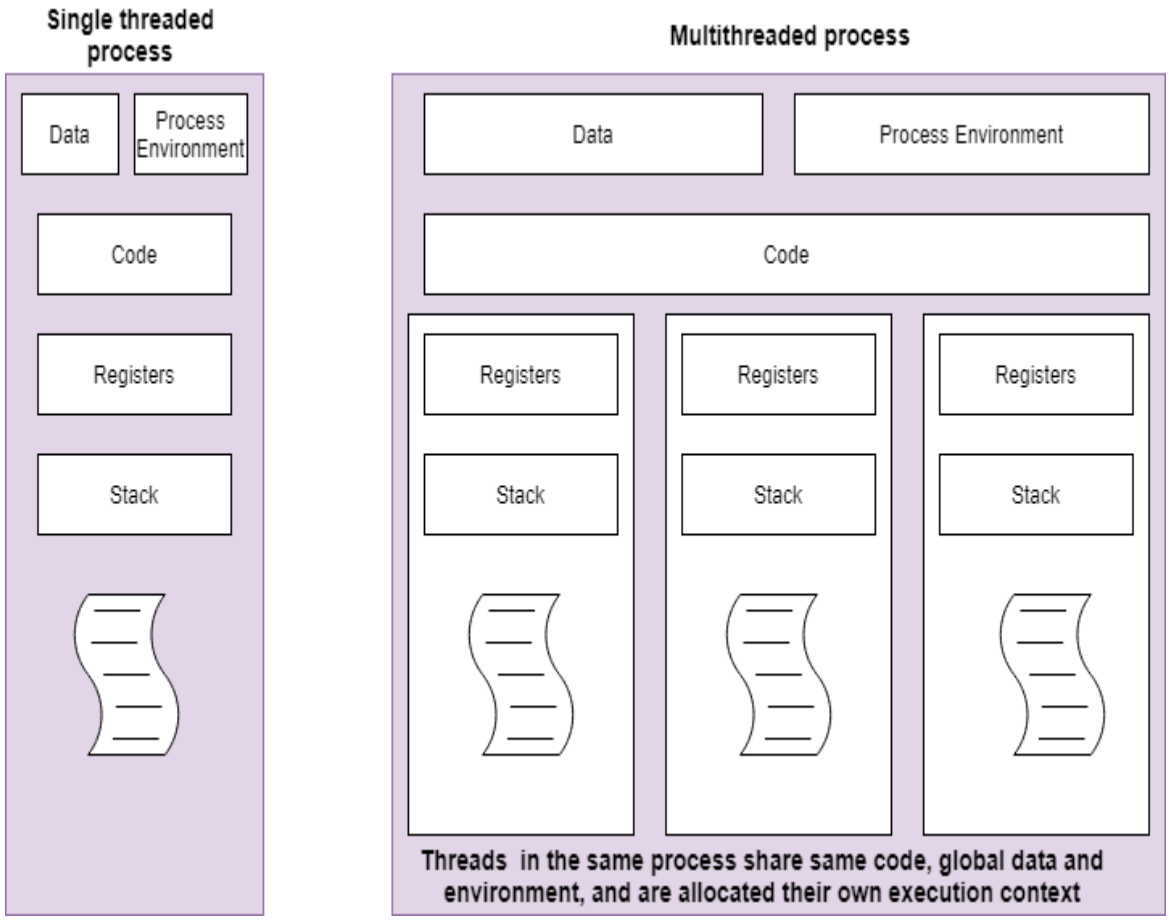

Figure 4-2. Comparing a single and multithreaded process

In Java, we can define a thread using a class that implements the `Runnable` interface and defines the `run()` method. A simple example is depicted in the example below:

```
class NamingThread implements Runnable {

  private String name;

        public NamingThread(String threadName) {
          name = threadName ;
    System.out.println("Constructor called: " + threadName) ;
  }

  public void run() {
          //Display info about this  thread
    System.out.println("Run called : " + name);
    System.out.println(name + " : " + Thread.currentThread());
    // and now terminate  ....
```

```
        }
  }
```

To execute the thread, we need to construct a `Thread` object using an instance of our `Runnable` and call the `start()` method to invoke the code in its own execution context. This is shown in the code example below, along with the output of running the code in bold text. Note this example has two threads – the `main()` thread and the `NamingThread`. The main thread starts the `NamingThread`, which executes asynchronously, and then waits for 1 second to give our `run()` method in `NamingThread` ample time to complete.

```
public static void main(String[] args) {

  NamingThread name0 = new NamingThread("My first thread");

  //Create the thread
  Thread t0 = new Thread (name0);

  // start the threads
  t0.start();

  //delay the main thread for a second (1000 milliseconds)
  try {
    Thread.currentThread().sleep(1000);
  } catch (InterruptedException e) {}

      //Display info about the main thread and terminate
      System.out.println(Thread.currentThread());

}
```

```
    ===EXECUTION OUTPUT===
    Constructor called: My first thread
    Run called : My first thread
    My first thread : Thread[Thread-0,5,main]
    Thread[main,5,main]
```

For illustration, we also call the static `currentThread()` method, which returns a string containing:

- The system generated thread identifier

- The thread priority, which by default is 5 for all threads. We'll cover thread priorities later.
- The identifier of the parent thread – in this example both parent threads are the main thread

Note to instantiate a thread, we call the `start()` method, not the `run()` method we define in the `Runnable`. The `start()` method contains the internal system magic to create the execution context for a separate thread to execute. If we call `run()` directly, the code will execute, but no new thread will be created. The `run()` method will execute as part of the `main` thread, just like any other Java method invocation that you know and love. You will still have a single threaded code.

In the example, we use `sleep()` to pause the execution of the `main` thread and make sure it does not terminate before the `NamimgThread`. This approach, namely coordinating two threads by delaying for an absolute time period (e.g. 1 second in the example) is not a very robust mechanism. What if for some reason - a slower CPU, a long delay reading disk, additional complex logic in the method – our thread doesn't terminate in the expected time frame? In this case, `main` will terminate first – this is not what we intend. In general, if you are using absolute times for thread coordination, you are doing it wrong. Almost always. Like 99.99999% of the time.

A simple and robust mechanism for one thread to wait until another has completed its work is to use the `join()` method. We could replace the `try-catch` block in the above example with:

```
t0.join();
```

This method causes the calling thread (in this case, `main`) to block until the thread referenced by `t0` terminates. If the referenced thread has terminated before the call to `join()`, then the method call returns immediately. In this way we can coordinate, or synchronize, the behavior of multiple threads. Synchronization of multiple threads is in fact the major focus of the rest of this chapter.

Order of Thread Execution

The system scheduler (in Java, this lives in the JVM) controls the order of thread execution. From the programmer's perspective, the order of execution is *non-deterministic*. Get used to that term, I'll use it a lot. The concept of non-determinism is fundamental to understanding multithreaded code.

I'll illustrate this by building on the earlier `NamingThread` example. Instead of creating a single `NamingThread`, I'll create and start up a few. Three in fact, as shown in the following code example. Again, sample output from running the code is in bold text beneath.

```
NamingThread name0 = new NamingThread("thread0");
NamingThread name1 = new NamingThread("thread1");
NamingThread name2 = new NamingThread("thread2");

//Create the threads
Thread t0 = new Thread (name0);
Thread t1 = new Thread (name1);
Thread t2 = new Thread (name2);

// start the threads
t0.start();
t1.start();
t2.start();
```

```
===EXECUTION OUTPUT===
Run called : thread0
thread0 : Thread[Thread-0,5,main]
Run called : thread2
Run called : thread1
thread1 : Thread[Thread-1,5,main]
thread2 : Thread[Thread-2,5,main]
Thread[main,5,main]
```

The output shown is a sample from just one execution. You can see the code starts three threads sequentially, namely *t0*, *t1* and *t2* (lines 11-13). Looking at the output, we see thread *t0* completes (line 17) before the others start. Next *t2's* run() method is called (line 18) followed by *t1's* run() method, even though *t1* was started before *t2*. Thread *t1* then runs to completion

(line 20) before *t2*, and eventually the *main* thread and the program terminate.

This is just one possible order of execution. If we run this program again, we will almost certainly see a different execution trace. This is because the JVM scheduler is deciding which thread to execute, and for how long. Put very simply, once the scheduler has given a thread an execution time slot on a CPU, it can interrupt the thread after a specified time period and schedule another one to run. This interruption is known as preemption. Preemption ensures each thread is given an opportunity to make progress. Hence the threads run independently and asynchronously until completion, and the scheduler decides which thread runs when based on a scheduling algorithm.

There's more to thread scheduling than this, and I'll explain the basic scheduling algorithm used later in this chapter. But for now, there is a major implication for programmers, namely, regardless of the order of thread execution, which you don't control, your code should produce correct results. Sounds easy?

Read on.

Problems with Threads

The basic problem in concurrent programming is coordinating the execution of multiple threads so that whatever order they are executed in, they produce the correct answer. Given that threads can be started and preempted non-deterministically, any moderately complex program will have essentially an infinite number of possible orders of executions. These systems aren't easy to test.

There are two fundamental problems that all concurrent programs need to avoid. These are race conditions and deadlocks, and these topics are covered in the next two subsections.

Race Conditions

Non-deterministic execution of threads implies that the code statements that comprise the threads:

- Will execute sequentially as defined within each thread
- Can be overlapped in any order across threads. This is because the number of statements that are executed for each thread execution slot is determined by the scheduler.

Hence, when many threads are executed on a single processor, their execution is *interleaved*. The CPU executes some steps from one thread, then performs some steps from another, and so on. If we are executing on a multicore CPU, then we can execute one thread per core. The statements of each thread execution are still however interleaved in a non-deterministic manner.

Now, if every thread simply does its own thing, and is completely independent, this is not a problem. Each thread executes until it terminates, as in our trivial `NamingThread` example. This stuff is a piece of cake! Why are these thread things meant to be complex?

Unfortunately, totally independent threads are not how most multithreaded systems behave. If you refer back to Figure 4-2, you will see that multiple threads share the global data within a process. In Java this is both global and static data.

Threads can use shared data structures to coordinate their work and communicate status across threads. For example, we may have threads handling requests from Web clients, one thread per request. We also want to keep a running total of how many requests we process each day. When a thread completes a request, it increments a global *RequestCounter* object that all threads share and update after each request. At the end of the day, we know how many requests were processed. A simple and elegant solution indeed. Well, maybe?

The code below shows a very simple implementation that mimics the request counter example scenario. It creates 50k threads to update a shared counter. Note we use a lambda function for brevity to create the threads,

and a ‘really bad idea’[5] 5 second delay in `main` to allow the threads to finish.

```
public class RequestCounter {
  final static private int NUMTHREADS = 50000;
  private int count = 0;

  public  void inc() {
    count++;
  }

  public int getVal() {
    return this.count;
  }

  public static void main(String[] args) throws
InterruptedException {
    final RequestCounter counter = new RequestCounter();

    for (int i = 0; i < NUMTHREADS; i++) {
      // lambda runnable creation
      Runnable thread = () -> {counter.inc(); };
            new Thread(thread).start();
    }

    Thread.sleep(5000);
    System.out.println("Value should be " + NUMTHREADS + "It is: "
+     counter.getVal());
  }
}
```

What you can do at home is clone this code from the book github repo, run this code a few times and see what results you get. In 10 executions my mean was 49995. I didn’t once get the correct answer of 50000. Weird.

Why?

The answer lies in how abstract, high-level programming language statements, in Java in this case, are executed on a machine. In this example, to perform an increment of a counter, the CPU must (1) load the current value into a register, (2) increment the register value, and (3) write the results back to the original memory location. This simple increment is actually a sequence of three machine-level operations.

As Figure 4-3 shows, at the machine level these three operations are independent and not treated as a single *atomic* operation. By atomic, we mean an operation that cannot be interrupted and hence once started will run to completion.

As the increment operation is not atomic at the machine level, one thread can load the counter value into a CPU register from memory, but before it writes the incremented value back, the scheduler premempts the thread and allows another thread to start. This thread loads the old value of the counter from memory and writes back the incremented value. Eventually the original thread executes again, and writes back its incremented value, which just happens to be the same as what is already in memory.

This means we've lost an update. From our 10 tests of the counter code above, we see this is happening on average 5 times in 50000 increments. Hence such events are rare, but even if it happens 1 time in 10 million, you still have an incorrect result.

Thread 1	Thread 2
Reads (x) into register	
Register value + 6	
Writes register value to (x)	
	Reads (x) into register
	Register value + 1
	Writes register value to (x)

Thread 1	Thread 2
Reads (x) into register	
Register value + 6	
	Reads (x) into register
	Register value + 1
	Writes register value to (x)
Writes register value to (x)	

Figure 4-3. Increments are not atomic at the machine level

When we lose updates in this manner, it is called a race condition. Race conditions can occur whenever multiple threads make changes to some shared state, in this case a simple counter. Essentially, different interleavings of the threads can produce different results.

Race conditions are insidious, evil errors, because their occurrence is typically rare, and they can be hard to detect as most of the time the answer will be correct. Try running the multithreaded counter code example with 1000 threads instead of 50000, and you will see this in action. I got the correct answer nine times out of ten.

So, this situation can be summarized as ‘same code, occasionally different results’. Like I said – race conditions are evil! Luckily, eradicating them is straightforward if you take a few precautions.

The key is to identify and protect *critical sections.* A critical section is a section of code that updates shared data structures, and hence must be executed atomically if accessed by multiple threads. The example of incrementing a shared counter is an example of a critical section. Another is removing an item from a list. We need to delete the head node of the list, and move the reference to the head of the list from the removed node to the next node in the list. Both operations must be performed atomically to maintain the integrity of the list. This is a critical section.

In Java, the `synchronized` keyword defines a critical section. If used to decorate a method, then when multiple threads attempt to call that method on the same shared object, only one is permitted to enter the critical section. All others block until the thread exits the synchronized method, at which point the scheduler chooses the next thread to execute the critical section. We say the execution of the critical section is serialized, as only one thread at a time can be executing the code inside it.

To fix the counter example, you therefore just need to identify the `inc()` method as a critical section and make it a synchronized method, ie:

```
synchronized public void inc() {
    count++;
  }
```

Test it out as many times as you like. You’ll always get the correct answer. Slightly more formally, this means any interleaving of the threads that the scheduler throws at us will always produce the correct results.

The synchronized keyword can also be applied to blocks of statements within a method. For example, we could rewrite the above example as:

```
public void inc() {
synchronized(this){
            count++;
        }
}
```

Underneath the covers, every Java object has a *monitor lock*, sometimes known as an intrinsic lock, as part of its runtime representation. The monitor is like the bathroom on a long distance bus – only one person is allowed to (and should!) enter at once, and the door lock stops others from entering when in use.

In our totally sanitary Java runtime environment, a thread must acquire the monitor lock to enter a synchronized method or synchronized block of statements.. Only one thread can own the lock at any time, and hence execution is serialized. This, very basically, is how Java and similar languages implement critical sections.

As a rule of thumb, you should keep critical sections as small as possible so that the serialized code is minimized. This can have positive impacts on performance and hence scalability. I'll return to this topic later, but I'm really talking about Amdahl's Law again, as introduced in Chapter 2. Synchronized blocks are the serialized parts of a system as described by Amdahl, and the longer they execute for, then the less potential we have for system scalability.

Deadlocks

To ensure correct results in multithreaded code, I explained that we have to restrict the inherent non-determinism to serialize access to critical sections. This avoids race conditions. However, if we are not careful, we can write code that restricts non-determinism so much that our program stops. And never continues. This is formally known as a deadlock.

A deadlock occurs when two or more threads are blocked forever, and none can proceed. This happens when threads need exclusive access to a shared set of resources, and the threads acquire locks in different orders. This is illustrated in the example below in which two threads need exclusive access to critical sections A and B. Thread 1 acquires the lock for critical section A, and thread 2 acquires the lock for critical section B. Both then block forever as they cannot acquire the locks they need to continue.

Two threads sharing access to two shared variables via synchronized blocks

1. *thread 1: enters critical section A*
2. *thread 2: enters critical section B*
3. *thread 1: blocks on entry to critical section B*
4. *thread 2: blocks on entry to critical section A*
5. *Both threads wait forever*

A deadlock, also known as a deadly embrace, causes a program to stop. It doesn't take a vivid imagination to realize that this can cause all sorts of undesirable outcomes. I'm happily texting away while my autonomous vehicle drives me to the bar. Suddenly, the vehicle code deadlocks. It won't end well.

Deadlocks occur in more subtle circumstances than the simple example above. The classic example is the Dining Philosophers problem. The story goes like this.

Five philosophers sit around a shared table. Being philosophers, they spend a lot of time thinking deeply. In between bouts of deep thinking, they replenish their brain function by eating from a plate of food that sits in front of them. Hence a philosopher is either eating or thinking, or transitioning between these two states.

In addition, the philosophers must all be physically very close, highly dexterous and Covid19 vaccinated friends, as they share chopsticks to eat with. Only five chopsticks are on the table, placed between each philosopher. When one philosopher wishes to eat, they follow a protocol of picking up their left chopstick first, then their right chopstick. Once they are ready to think again, they first return the right chopstick, then the left.

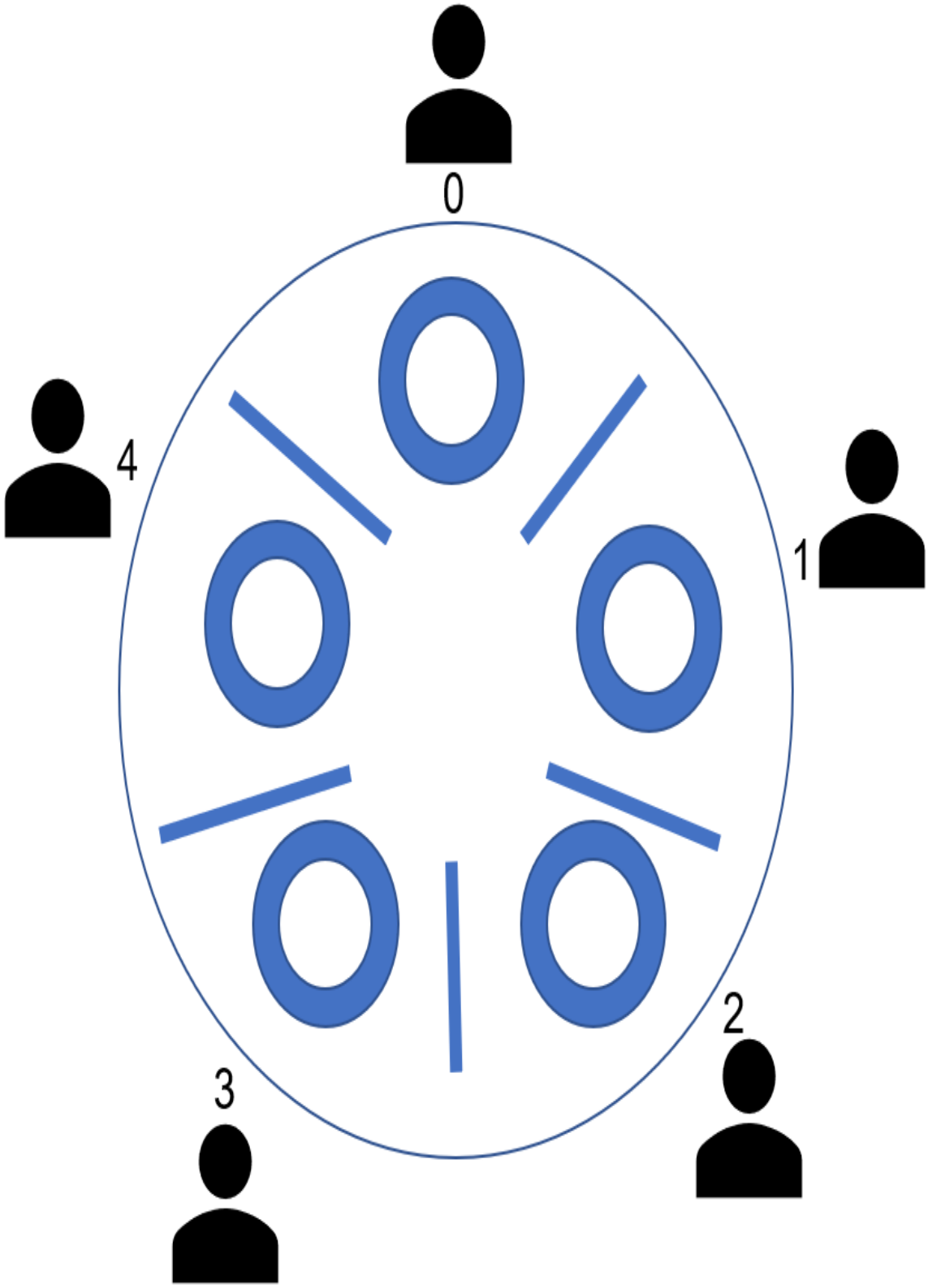
0
1
2
3
4

Figure 4-4. The Dining Philosophers Problem

Figure 4-4 depicts our philosophers, each identified by a unique number. As each is either concurrently eating or thinking, we can model each philosopher as a thread. The code is shown in Example 4-7. The shared chopsticks are represented by instances of the Java `Object` class. As only one object can hold the monitor lock on an object at any time, they are used as entry conditions to the critical sections in which the philosophers acquire the chopsticks they need to eat. After eating, the chopsticks are returned to the table and the lock is released on each so that neighboring philosophers can eat whenever they are ready.

Example 4-7. The Philosopher Thread

```
1   public class Philosopher implements Runnable {
2
3     private final Object leftChopStick;
4     private final Object rightChopStick;
5
6     Philosopher(Object leftChopStick, Object rightChopStick) {
7       this.leftChopStick = leftChopStick;
8       this.rightChopStick = rightChopStick;
9     }
10     private void LogEvent(String event) throws
InterruptedException {
11      System.out.println(Thread.currentThread()
12                                      .getName() + " " + event);
13      Thread.sleep(1000);
14    }
15
16    public void run() {
17      try {
18        while (true) {
19          LogEvent(": Thinking deeply");
20          synchronized (leftChopStick) {
21            LogEvent( ": Picked up left chop stick");
22            synchronized (rightChopStick) {
23              LogEvent(": Picked up right chopstick - eating");
24              LogEvent(": Put down right chopstick");
25            }
26            LogEvent(": Put down left chopstick. Ate too much");
27          }
28        } // end while
```

```
29      } catch (InterruptedException e) {
30         Thread.currentThread().interrupt();
31    }
32   }
33  }
```

To bring the philosophers described in Example 4-7 to life, we must instantiate a thread for each and give each philosopher access to its neighboring chopsticks. This is done through the thread constructor call on line 16 in Example 4-8. In the `for` loop we create five philosophers and start these as independent threads, where each chopstick is accessible to two threads, one as a left chopstick, and one as a right.

Example 4-8. Dining Philosophers - deadlocked version

```
private final static int NUMCHOPSTICKS = 5 ;
private final static int NUMPHILOSOPHERS = 5;
public static void main(String[] args) throws Exception {

  final Philosopher[] ph = new Philosopher[NUMPHILOSOPHERS];
  Object[] chopSticks = new Object[NUMCHOPSTICKS];

  for (int i = 0; i < NUMCHOPSTICKS; i++) {
    chopSticks[i] = new Object();
  }

  for (int i = 0; i < NUMPHILOSOPHERS; i++) {
    Object leftChopStick = chopSticks[i];
    Object rightChopStick = chopSticks[(i + 1) %
chopSticks.length];

    ph[i] = new Philosopher(leftChopStick, rightChopStick);
            }

    Thread th = new Thread(ph[i], "Philosopher " + (i + 1));
    th.start();
  }
}
```

Running this code produces the following output on my first attempt. If you run the code you will almost certainly see different outputs, but the final outcome will be the same.

```
Philosopher 4 : Thinking deeply
Philosopher 5 : Thinking deeply
Philosopher 1 : Thinking deeply
Philosopher 2 : Thinking deeply
Philosopher 3 : Thinking deeply
Philosopher 4 : Picked up left chop stick
Philosopher 1 : Picked up left chop stick
Philosopher 3 : Picked up left chop stick
Philosopher 5 : Picked up left chop stick
Philosopher 2 : Picked up left chop stick
```

10 lines of output, then … nothing! We have a deadlock. This is a classic circular waiting deadlock. Imagine the following scenario:

1. Each philosopher indulges in a long thinking session
2. Simultaneously, they all decide they are hungry and reach for their left chop stick.
3. No philosopher can eat (proceed) as none can pick up their right chop stick

Real philosophers in this situation would figure out some way to proceed by putting down a chopstick or two until one or more of their colleagues can eat. We can sometimes do this in our software by using timeouts on blocking operations. When the timeout expires a thread releases the critical section and retries, allowing other blocked threads a chance to proceed. This is not optimal though, as blocked threads hurt performance, and setting timeout values in an inexact science.

It is much better therefore to design a solution to be deadlock free. This means that one or more threads will always be able to make progress. With circular wait deadlocks, this can be achieved by imposing a resource allocation protocol on the shared resources, so that threads will not always request resources in the same order.

In the Dining Philosophers problem, we can do this by making sure one of our philosophers picks up their right chopstick first. Let’s assume we instruct Philosopher 4 to do this. This leads to a possible sequence of operations such as below:

```
Philosopher 0 picks up left chopstick (chopStick[0]) then right
(chopStick[1])
Philosopher 1 picks up left chopstick (chopStick[1]) then right
(chopStick[2])
Philosopher 2 picks up left chopstick (chopStick[2]) then right
(chopStick[3])
Philosopher 3 picks up left chopstick (chopStick[3]) then right
(chopStick[4])
Philosopher 4 picks up right chopstick (chopStick[0]) then left
(chopStick[4])
```

In this example, Philosopher 4 must block, as Philosopher 0 already has acquired access to chopstick[0]. With Philosopher 4 blocked, Philosopher 3 is assured access to chopstick[4] and can then proceed to satisfy their appetite.

The fix for the Dining Philosophers solution is shown in Example 4-10.

Example 4-10. Solving the Dining Philosophers deadlock

```
if (i == NUMPHILOSOPHERS - 1) {
  // The last philosopher picks up the right fork first
  ph[i] = new Philosopher(rightChopStick, leftChopStick);
} else {
  // all others pick up the left chop stick first
  ph[i] = new Philosopher(leftChopStick, rightChopStick);
}
            }
```

More formally we are imposing an ordering on the acquisition of shared resources, such that:

```
chopStick[0]< chopStick[1]< chopStick[2]< chopStick[3]<
chopStick[4]
```

This means each thread will always attempt to acquire chopstick[0] before chopstick[1], and chopstick[1] before chopstick[2], and so on. For philosopher 4, this means it will attempt to acquire chopstick[0] before chopstick[4], thus breaking the potential for a circular wait deadlock.

Deadlocks are a complicated topic and this section has just scratched the surface. You'll see deadlocks in many distributed systems. For example, a

user request acquires a lock on some data in a *Students* database table, and must then update rows in the *Classes* table to reflect student attendance. Simultaneously another user request acquires locks on the *Classes* table, and next must update some information in the *Students* table. If these requests interleave such that each request acquires locks in an overlapping fashion, we have a deadlock.

I'll revisit deadlocks later when discussing concurrent database access and locking in Part 3 of this book.

Thread States

Multithreaded systems have a system scheduler that decides which threads to run when. In Java, the scheduler is known as a preemptive, priority-based scheduler. In short this means it chooses to execute the highest priority thread which wishes to run.

Every thread has a priority (by default 5, range 0 to 10). A thread inherits its priority from its parent thread. Higher priority threads get scheduled more frequently than lower priority threads, but in most applications having all threads as the default priority suffices.

The scheduler cycles threads through four distinct states, based on their behavior. These are:

Created

> A thread object has been created but its `start()` method has not been invoked. Once `start()` is invoked, the thread enters the runnable state.

Runnable

> A thread is able to run. The scheduler will choose which thread(s) to execute in a first-in first-out (FIFO) manner – one thread can be allocated at any time to each core in the node. Threads then execute until they block (e.g. on a `synchronized` statement), execute a

`yield()`, `suspend()` or `sleep()` statement, the `run()` method terminates, or are preempted by the scheduler. Preemption occurs when a higher priority thread becomes runnable, or when a system-specific time period, known as a time slice, expires. Preemption based on time slicing allows the scheduler to ensure that all threads eventually get a chance to execute – no execution hungry threads can hog the CPU.

Blocked

A thread is blocked if it is waiting for a lock, a notification event to occur (e.g. sleep timer to expire, resume() method executed), or is waiting for a network or disk request to complete. When the specific event a blocked thread is waiting for occurs, it moves back to the runnable state.

Terminated

A thread's run() method has completed or it has called the stop() method. The thread will no longer be scheduled.

An illustration of this scheme is in Figure 4-5. The scheduler effectively maintains FIFO queue in the *Runnable* state for each thread priority. High priority threads are used typically to respond to events (eg an emergency timer), and execute for a short period of time. Low priority threads are used for background, ongoing tasks like checking for corruption of files on disk through recalculating checksums. Background threads basically use up idle CPU cycles.

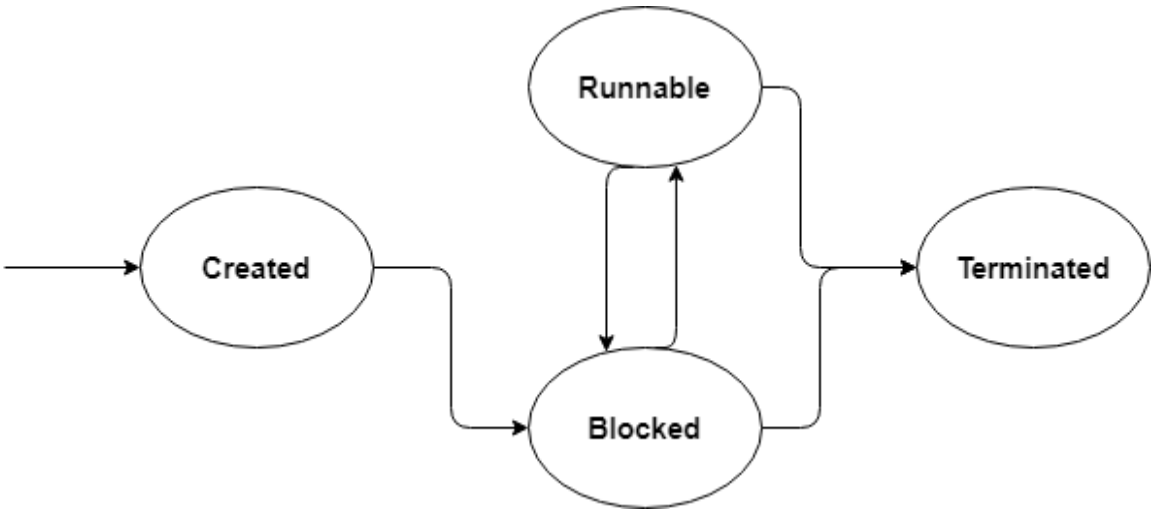

Figure 4-5. Threads states and transitions

Thread Coordination

There are many problems that require threads with different roles to coordinate their activities. Imagine a collection of threads that each accept documents from users, do some processing on the documents (e.g. generate a pdf), and then send the processed document to a shared printer pool. Each printer can only print one document at a time, so they read from a shared print queue, grabbing and printing documents in the order they arrive.

This printing problem is an illustration of the classic producer-consumer problem. Producers generate and send messages via a shared FIFO buffer to consumers. Consumers retrieve these messages, process them, and then ask for more work from the buffer. A simple illustration of this problem is in Figure 4-6. It's a bit like a 24 hour, 365 day buffet restaurant - the kitchen keeps producing, the wait staff collect the food and put it in the buffet, and hungry diners help themselves. Forever.

Like virtually all real resources, the buffer has a limited capacity. Producers generate new items, but if the buffer is full, they must wait until some item(s) have been consumed before they can add the new item to the buffer. Similarly, if the consumers are consuming faster than the producers are producing, then they must wait if there are no items in the buffer, and somehow get alerted when new items arrive.

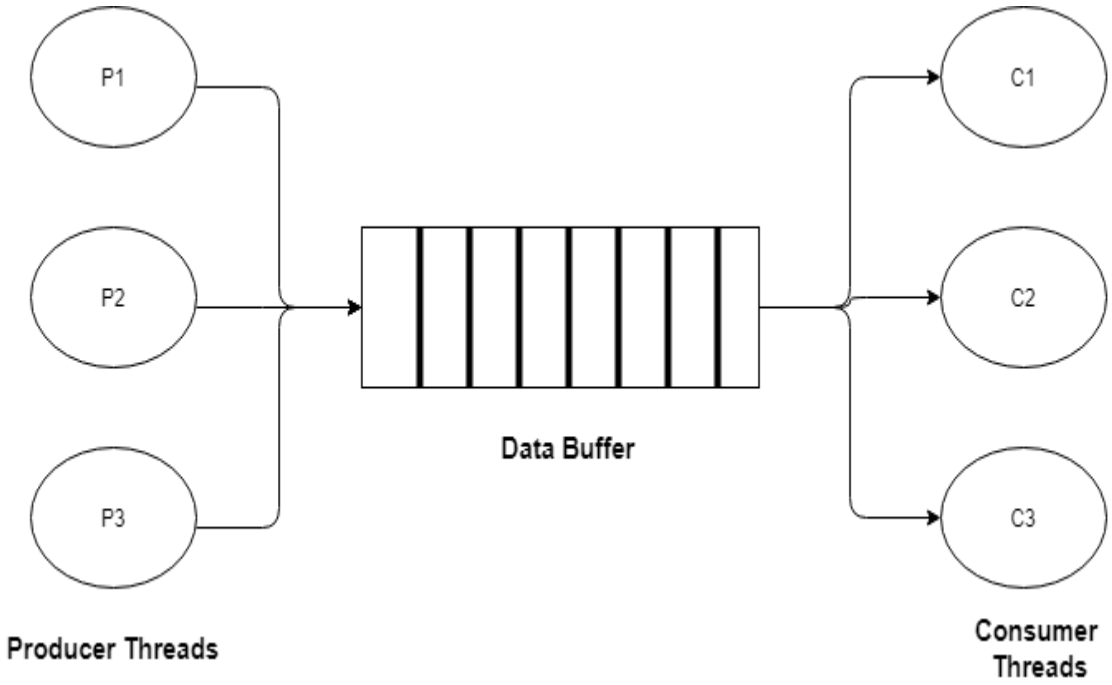

Figure 4-6. The Producer Consumer Problem

One way for a producer to wait for space in the buffer, or a consumer to wait for an item, is to keep retrying an operation. A producer could sleep for a second, and then retry the put operation until it succeeds. A consumer could do likewise.

This solution is called polling, or busy waiting. It works fine, but as the second name implies, each producer and consumer are using resources (CPU, memory, maybe network?) each time it retries and fails. If this is not a concern, then cool, but in scalable systems we are always aiming to optimize resource usage, and polling can be wasteful.

A better solution is for producers and consumers to block until their desired operation, put or get respectively, can succeed. Blocked threads consume no resources and hence provide an efficient solution. To facilitate this, thread programming models provide blocking operations that enable threads to 'signal' to other threads when an event occurs. With the producer-consumer problem, the basic scheme is as follows:

- When a producer adds an item to the buffer, it sends a signal to any blocked consumers to notify them that there is an item in the buffer

- When a consumer retrieves an item from the buffer, it sends a signal to any blocked producers to notify them there is capacity in the buffer for new items.

In Java, there are two basic primitives, namely `wait()` and `notify()`, that can be used to implement this signaling scheme. Briefly, they work like this:

- A thread may call wait() within a synchronized block if some condition it requires to hold is not true. For example, a thread may attempt to retrieve a message from a buffer, but if the buffer has no messages to retrieve, it calls wait() and blocks until another thread adds a message, sets the condition to true, and calls notify() on the same object.
- notify() wakes up a thread that has called wait() on the object.

These Java primitives are used to implement *guarded blocks*. Guarded blocks use a condition as a guard that must hold before a thread resumes the execution. The code snippet below shows how the guard condition, *empty*, is used to block a thread that is attempting to retrieve a message from an empty buffer.

```
while (empty) {
  try {
    System.out.println("Waiting for a message");
    wait();
  } catch (InterruptedException e) {}
}
```

When another thread adds a message to the buffer, it executes `notify()` as in the code fragment below.

```
// Store message.
this.message = message;
empty = false;
// Notify consumer that message is available
notify();
```

The full implementation of this example is given in the code examples in the book git repository. There are a number of variations of the `wait()`

and `notify()` methods, but these go beyond the scope of what I can cover in this overview. And luckily, Java provides us with thread-safe abstractions that hide this complexity from your code.

An example that is pertinent to the producer-consumer problem is the `BlockingQueue` interface in `java.util.concurrent.BlockingQueue`. A `BlockingQueue` implementation provides a thread-safe object that can be used as the buffer in a producer-consumer scenario. There are 5 different implementations of the BlockingQueue interface. I'll use one of these, the `LinkedBlockingQueue`, to implement the producer-consumer. This is shown in Example 4-13.

Example 4-13. Producer-Consumer with a LinkedBlockingQueue

```
class ProducerConsumer {
   public static void main(String[] args)
     BlockingQueue buffer = new LinkedBlockingQueue();
     Producer p = new Producer(buffer);
     Consumer c = new Consumer(buffer);
     new Thread(p).start();
     new Thread(c).start();
   }
 }

class Producer implements Runnable {
   private boolean active = true;
   private final BlockingQueue buffer;
   public Producer(BlockingQueue q) { buffer = q; }
   public void run() {

     try {
       while (active) { buffer.put(produce()); }
     } catch (InterruptedException ex) { // handle exception}
   }
   Object produce() { // details omitted, sets active=false }
 }

 class Consumer implements Runnable {
   private boolean active = true;
   private final BlockingQueue buffer;
   public Consumer(BlockingQueue q) { buffer = q; }
   public void run() {
```

```
      try {
        while (active) { consume(buffer.take()); }
      } catch (InterruptedException ex) { // handle exception }
    }
    void consume(Object x) {  // details omitted, sets active=false
}
 }
```

This solution absolves the programmer from being concerned with the implementation of coordinating access to the shared buffer, and greatly simplifies the code.

The `java.util.concurrent`[6] package is a treasure trove for building multithreaded Java solutions. In the following sections, I will briefly highlight a few of these powerful and extremely useful capabilities.

Thread Pools

Many multithreaded systems need to create and manage a collection of threads that perform similar tasks. For example, in the producer-consumer problem, we can have a collection of producer threads and a collection of consumer threads, all simultaneously adding and removing items, with coordinated access to the shared buffer.

These collections are known as thread pools. Thread pools comprise several worker threads, which typically perform a similar purpose and are managed as a collection. We could create a pool of producer threads which all wait for an item to process, write the final product to the buffer, and then wait to accept another item to process. When we stop producing items, the pool can be shut down in a safe manner, so no partially processed items are lost through an unanticipated exception.

In the `java.util.concurrent package`, thread pools are supported by the `ExecutorService` interface. This extends the base Executor interface with a set of methods to manage and terminate threads in the pool. A simple producer-consumer example using a fixed size thread pool is shown in Example 4-14 and Example 4-15. The `Producer` class in Example 4-14 is a `Runnable` that sends a single message to the buffer and

then terminates. The `Consumer` simply takes messages from the buffer until an empty string is received, upon which it terminates.

Example 4-14. Producer and Consumer for thread pool implementation

```
class Producer implements Runnable {

  private final BlockingQueue buffer;
  public Producer(BlockingQueue q) { buffer = q; }
  @Override
  public void run() {

  try {
    sleep(1000);
    buffer.put("hello world");

  } catch (InterruptedException ex) {
    // handle exception
  }
 }
}
class Consumer implements Runnable {
  private final BlockingQueue buffer;
  public Consumer(BlockingQueue q) { buffer = q; }
  @Override
   public void run() {
      boolean active = true;
      while (active) {
          try {
             String  s = (String) buffer.take();
             System.out.println(s);
             if (s.equals("")) active = false;
          } catch (InterruptedException ex) {
              / handle exception
          }
      } /
      System.out.println("Consumer terminating");
    }
 }
```

In Example 4-15, we create a single consumer to take messages from the buffer. We then create a fixed size thread pool of size 5 to manage our producers. This causes the JVM to pre-allocate five threads that can be used to execute any `Runnable` objects that are executed by the pool.

In the `for()` loop, we then use the `ExecutorService` to run 20 producers. As there are only 5 threads available in the thread pool, only a maximum of 5 producers will be executed simultaneously. All others are placed in a wait queue which is managed by the thread pool. When a producer terminates, the next `Runnable` in the wait queue is executed using any available thread in the pool.

Once we have requested all the producers to be executed by the thread pool, we call the shutdown() method on the pool. This tells the `ExecutorService` not to accept any more tasks to run. We next call the `awaitTermination()` method, which blocks the calling thread until all the threads managed by the thread pool are idle and no more work is waiting in the wait queue. Once `awaitTermination()` returns, we know all messages have been sent to the buffer, and hence send an empty string to the buffer which will act as a termination value for the consumer.

Example 4-15. Thread pool-based Producer Consumer solution

```
public static void main(String[] args) throws InterruptedException
  {
    BlockingQueue buffer = new LinkedBlockingQueue();

    //start a single consumer
    (new Thread(new Consumer(buffer))).start();

    ExecutorService producerPool = Executors.newFixedThreadPool(5);
    for (int i = 0; i < 20; i++)
      {
        Producer producer = new Producer(buffer) ;
        System.out.println("Producer created" );
        producerPool.execute(producer);
      }

      producerPool.shutdown();
      producerPool.awaitTermination(10, TimeUnit.SECONDS);

      //send termination message to consumer
      buffer.put("");
    }
```

Like most topics in this chapter, there's many more sophisticated features in the `Executor` framework that can be used to create multithreaded

programs. This description has just covered the basics. Thread pools are important as they enable our systems to rationalize the use of resources for threads. Every thread consumes memory, for example the stack size for a thread is typically around 1MB. Also, when we switch execution context to run a new thread, this consumes CPU cycles. If our systems create threads in an undisciplined manner, we will eventually run out of memory and the system will crash. Thread pools allow us to control the number of threads we create and utilize them efficiently.

I'll discuss thread pools throughout the remainder of this book, as they are a key concept for efficient and scalable management of the ever increasing request loads that servers must satisfy.

Barrier Synchronization

I had a high school friend whose family, at dinner times, would not allow anyone to start eating until the whole family was seated at the table. I thought this was weird, but many years later it serves as a good analogy for the concept known as barrier synchronization. Eating commenced only after all family members arrived at the table.

Multithreaded systems often need to follow such a pattern of behavior. Imagine a multithreaded image processing system. An image arrives and a distinct segment of the image is passed to each thread to perform some transformation upon – think Instagram filters on steroids. The image is only fully processed when all threads have completed. In software systems, we use a mechanism called barrier synchronization to achieve this style of thread coordination.

The general scheme is shown in Figure 4-7. In this example, the `main()` thread creates four new threads and all proceed independently until they reach the point of execution defined by the barrier. As each thread arrives, it blocks. When all threads have arrived at this point, the barrier is released, and each thread can continue with its processing.

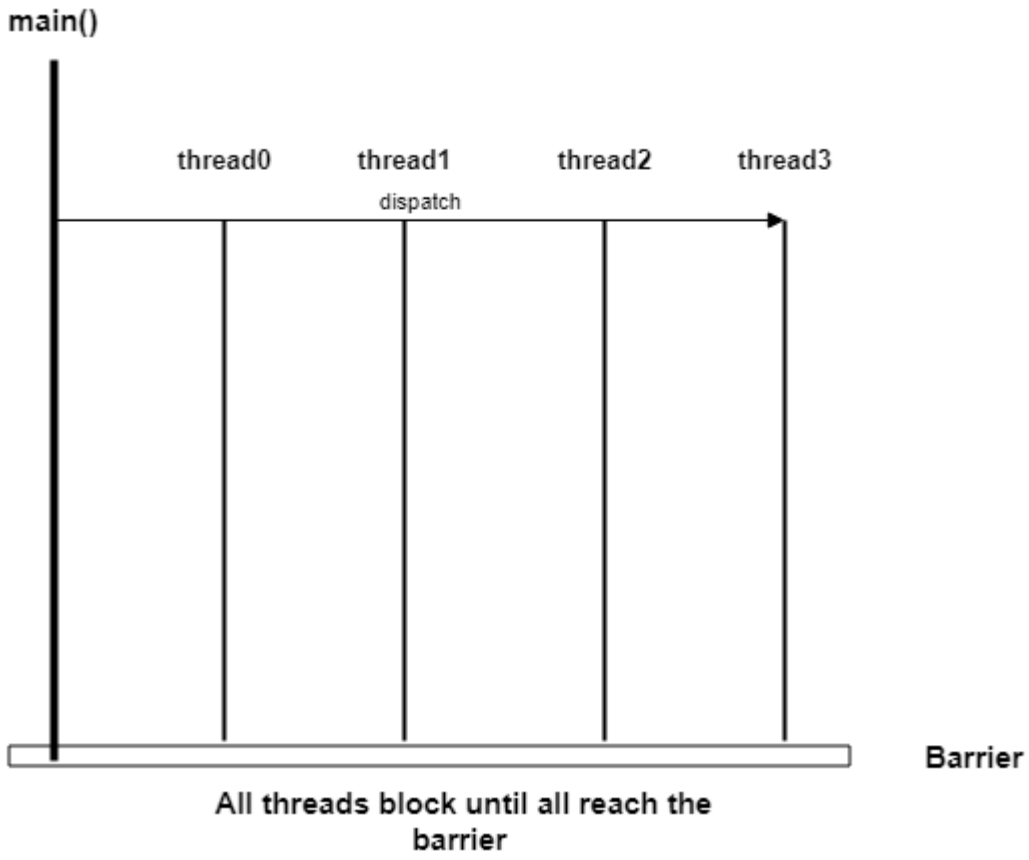

Figure 4-7. Barrier Synchronization

Java provides three primitives for barrier synchronization. I'll show here how just one of the three, namely the `CountDownLatch`, works. The basic concepts apply to other barrier synchronization primitives.

When you create a `CountDownLatch`, you pass a value to its constructor that represents the number of threads that must block at the barrier before they are all allowed to continue. This is called in the thread which is managing the barrier points for the system – in Figure 4-7 this would be main().

```
CountDownLatch  nextPhaseSignal = new CountDownLatch(numThreads);
```

Next you create the worker threads that will perform some actions and then block at the barrier until they all complete. To do this, you need to pass each thread a reference to `CountDownLatch`.

```
for (int i = 0; i < numThreads; i++) {
            Thread worker = new Thread(new
```

```
WorkerThread(nextPhaseSignal));
            worker.start();
        }
```

After launching the worker threads, the `main()` thread will call the `.await()` method to block until the latch is triggered by the worker threads.

```
nextPhaseSignal.await();
```

Each worker thread will complete its task and before exiting call the `.countDown()` method on the latch. This decrements the latch value. When the last thread calls `.countDown()` and the latch value becomes zero, all threads that have called `.await()` on the latch transition from the *blocked* to the *runnable* state. At this stage we are assured that all workers have completed their assigned task.

```
nextPhaseSignal.countDown();
```

Any subsequent calls to `.countDown()` will return immediately as the latch has been effectively triggered. Note `.countDown()` is non-blocking, which is a useful property for applications in which threads have more work to do after reaching the barrier.

This example illustrates using a `CountDownLatch` to block a single thread until a collection of threads have completed their work. You can invert this use case with a latch however if you initialize its value to one. Multiple threads could call `.await()` and block until another thread calls `.countDown()` to release all waiting threads. This example is analogous to a simple gate, which one thread opens to allow a collection of others to continue.

`CountDownLatch` is a simple barrier synchronizer. It's a single use tool, as the initializer value cannot be reset. More sophisticated features are provided by the `CyclicBarrier` and `Phaser` classes in Java. Armed with the knowledge of how barrier synchronization works from this section, these will be straightforward to understand.

Thread-Safe Collections

Many Java programmers, once they delve into the wonders of multithreaded programs, are surprised to discover that the collections in the `java.util` package[7] are not thread safe. Why, I hear you ask? The answer, luckily, is simple. It is to do with performance. Calling synchronized methods incurs overheads. Hence to attain faster execution for single threaded programs, the collections are not thread-safe.

If you want to share an `ArrayList`, `Map` or 'your favorite data structure' from `java.util` across multiple threads, you must ensure modifications to the structure are placed in critical sections. This approach places the burden on the client of the collection to safely make updates, and hence is error prone – a programmer might forget to make modifications in a `synchronized` block.

It's always safer to use inherently thread-safe collections in your multithreaded code. For this reason, the Java collections framework provides a factory method that creates a thread-safe version of `java.util` collections. Here's an example of creating a thread-safe list.

```
List<String> list = Collections.synchronizedList(new ArrayList<>
());
```

What is really happening here is that you are creating a wrapper around the base collection class, which has `synchronized` methods. These delegate the actual work to the original class, in a thread-safe manner of course. You can use this approach for any collection in the `java.util` package, and the general form is:

```
Collections.synchronized….(new collection<>())
```

where "…." is `List, Map, Set`, and so on.

Of course, when using the synchronized wrappers, you pay the performance penalty for acquiring the monitor lock and serializing access from multiple threads. This means the whole collection is locked while a single thread makes a modification, greatly limiting concurrent performance (Amdahl's Law again). For this reason, Java 5.0 included the concurrent collections

package, namely `java.util.concurrent`. It contains a rich collection of classes specifically designed for efficient multithreaded access.

In fact we've already seen one of these classes – the `LinkedBlockingQueue`. This uses a locking mechanism that enables items to be added to and removed from the queue in parallel. This finer grain locking mechanism utilizes the `java.util.concurrent.lock.Lock` class rather than the monitor lock approach. This allows multiple locks to be utilized on the same collection, hence enabling safe concurrent access.

Another extremely useful collection that provides this finer-grain locking is the `ConcurrentHashMap`. This provides the similar methods as the non-thread safe `HashMap`, but allows non-blocking reads and concurrent writes based on a `concurrencyLevel` value you can pass to the constructor (the default value is 16).

```
ConcurrentHashMap (int initialCapacity, float loadFactor,
                     int concurrencyLevel)
```

Internally, the hash table is divided into individually lockable segments, often known as shards. Locks are associated with each shard rather than the whole collection. This means updates can be made concurrently to hash table entries in different shards of the collection, increasing performance.

Retrieval operations are non-blocking for performance reasons, meaning they can overlap with multiple concurrent updates. This means retrievals only reflect the results of the most recently completed update operations at the time the retrieval is executed.

For similar reasons, iterators for a `ConcurrentHashMap` are what is known as weakly consistent. This means the iterator contains a copy of the hash map that reflects its state at the time the iterator is created. While the iterator is in use, new nodes may be added and existing nodes removed from the underlying hash map. However, these state changes are not reflected in the iterator.

If you need an iterator that always reflects the current hash map state while being updated by multiple threads, then there are performance penalties to pay, and a `ConcurrentHashMap` is not the right approach. This is an example of favoring performance over consistency – a classic design trade-off.

Summary and Further Reading

I'll draw upon the major concepts introduced in this Chapter throughout the remainder of this book. Threads are inherently components of the data processing and database platforms that we use to build scalable distributed systems. In many cases, you may not be writing explicitly multithreaded code. However, the code you write will be invoked in a multithreaded environment, which means you need to be aware of thread-safety. Many platforms also expose their concurrency through configuration parameters, meaning that to tune the system's performance, you need to understand the effects of changing the various threading and thread pool settings. Basically, there's no escaping concurrency in the world of scalable distributed systems.

Finally, it is worth mentioning that while concurrent programming primitives vary across programming languages, the foundational issues don't change, and carefully designed multithreaded code to avoid race conditions and deadlocks is needed. Whether you grapple with the `pthreads`[8] library in C/C++, or the classic Communicating Sequential Processes[9] (CSP)-inspired Go concurrency model[10], the problems you need to avoid are the same. The knowledge you have gained from this chapter will regardless stand you in good stead, whatever language you are using.

This chapter has only brushed the surface of concurrency in general and its support in Java. The best book to continue learning more about the basic concepts of concurrency is the classic *Java Concurrency in Practice (JCiP)* by Brian Goetz et al. If you understand everything in JCiP, you'll be writing pretty great concurrent code.

Java concurrency support has moved on considerably however since Java 5. In the world of Java 12 (or whatever version is current when you read this), there are new features such as `CompleteableFutures`, lambda expressions and parallel streams. The functional programming style introduced in Java 8.0 makes it easy to create concurrent solutions without directly creating and managing threads. A good source of knowledge for Java 8.0 features is *Mastering Concurrency Programming with Java 8* by Javier Fernández González.

Other excellent sources include:

Doug Lea, *Concurrent Programming in Java: Design Principles and Patterns*, 2nd Edition

Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft, *Java 8 in Action: Lambdas, Streams, and functional-style programming*, Manning Publications, 1st Edition, 2014.

---

1 http://www.computinghistory.org.uk/det/6192/Introduction-of-Intel-386/

2 http://www.golang-book.com/books/intro/10

3 https://erlang.org/doc/getting_started/conc_prog.html

4 https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/

5 The correct way to handle these problems, namely barrier synchronization, is covered later in this chapter.

6 https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html

7 Except Vector and HashTable, which are legacy classes, thread safe and slow!

8 https://en.wikipedia.org/wiki/POSIX_Threads#:~:text=POSIX%20Threads%2C%20usually%20referred%20to,work%20that%20overlap%20in%20time.

9 https://en.wikipedia.org/wiki/Communicating_sequential_processes

10 https://www.golang-book.com/books/intro/10

Chapter 5. Application Services

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

At the heart of any system lies the unique business logic that implements the application requirements. In distributed systems, this is exposed to clients through Application Programming Interfaces (APIs) and executed within a runtime environment designed to efficiently support concurrent remote calls. An API and its implementation comprise the fundamental elements of the services an application supports.

In this chapter, I'm going to focus on the pertinent issues for achieving scalability for the services tier in an application. I'll explain API and service design and describe the salient features of application servers that provide the execution environment for services. I'll also elaborate on topics such as horizontal scaling, load balancing and state management that I introduced briefly in Chapter 2.

Service Design

In the simplest case, an application comprises one Internet facing service that persists data to a local data store, as shown in Figure 5-1. Clients

interact with the service through its published API, which is accessible across the Internet.

Let's look at the API and service implementation in more detail.

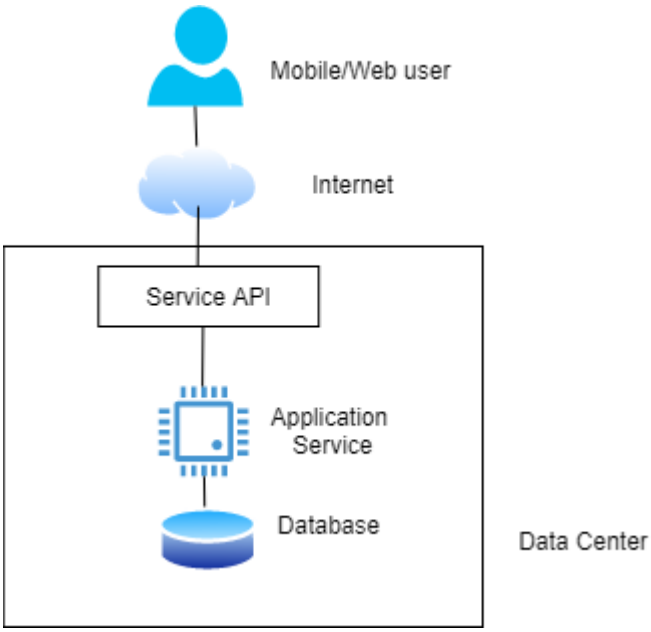

Figure 5-1. A Simple Service

Application Programming Interface (API)

An API defines a contract between the client and server. The API specifies the types of requests that are possible, the data that is needed to accompany the requests, and the results that will be obtained. APIs have many different variations, as I explained in RPC/RMI discussions in Chapter 3. While there remains some API diversity in modern applications, the predominant style relies on HTTP APIs. These are typically, although not particularly accurately, classified as RESTful.

REST is actually an architectural style that was defined by Roy Fielding in his PhD thesis[1]. A great source of knowledge on RESTful APIs and the various degrees to which Web technologies can be exploited is *REST in Practice* by Jim Webber, et al. (O'Reilly). Here I'll just briefly touch on the HTTP *CRUD* API pattern. This pattern does not fully implement the principles of REST, but it is widely adopted in Internet systems today. It

exploits the four code HTTP verbs, namely POST, GET, PUT, and DELETE.

CRUD stands for *Create, Read, Update, Delete*. A CRUD API specifies how clients perform these operations in a specific business context. For example, a user might *create* a profile (POST), *read* catalog items (GET), *update* their shopping cart (PUT) and *delete* items from their order (DELETE).

An example HTTP CRUD API for the example ski resort system, briefly introduced in Chapter 2, that uses these four core HTTP verbs is shown in Table 5-1 In this example, parameter values are passed as part of the request address and are identified by the {} notation.

Table 5-1. HTTP CRUD Verbs

Verb	Uniform Resource Identifier Example	Purpose
POST	/skico.com/skiers/	Create a new skier profile, with skier details provided in the JSON request payload. The new skier profile is returned in the JSON response

GET	/skico.com/skiers/{skierID}	Get the profile information for a skier, returned in a JSON response payload
PUT	/skico.com/skiers/{skierID}	Update skier profile
DELETE	/skico.com/skiers/{skierID}	Delete a skier's profile as they didn't renew their pass!

Additional parameter values can be passed and returned in HTTP request and response bodies respectively. For example, a successful request to:

```
GET /skico.com/skiers/12345
```

will return an HTTP 200 response code and the following results formatted in JSON:

```
{
    "username": "Ian123",
    "email":"i.gorton@somewhere.com"
    "city": "Seattle"
}
```

To change the skier's city, the client could issue the following PUT request to the same URI along with a request body representing the updated skier profile.

```
PUT  /skico.com/skiers/12345
{
    "username": "Ian123",
    "email":”i.gorton@somewhere.com”
    “city”: “Wenatchee”
}
```

More formally, an HTTP CRUD API applies HTTP verbs on *resources* identified by Uniform Resource Identifiers (URIs). In Table 5-1 for example, a URI that identifies skier 768934 would be:

```
/skico.com/skiers/768934
```

An HTTP GET request to this resource would return the complete profile information for a skier in the response payload, such as name, address, number of days visited, and so on. If a client subsequently sends an HTTP PUT request to this URI, we are expressing the intent to update the resource for skier 768934 – in this example it would be the skier's profile. The PUT request would provide the complete representation for the skier's profile as returned by the GET request. Again, this would be as a payload with the request. Payloads are typically formatted as JSON, although XML and other formats are also possible. If a client sends a DELETE request to the same URI, then the skier's profile will be deleted.

Hence the combination of the HTTP verb and URI define the semantics of the API operation. Resources, represented by URIs, are conceptually like objects in Object Oriented Design (OOD) or entities in Entity-Relationship (ER) model. Resource identification and modeling hence follows similar methods to OOD and ER modeling. The focus however is on resources that need to be exposed to clients in the API. The Further Reading section at the end of this chapter points to useful sources of information for resource design.

HTTP APIs can be specified using a notation called OpenAPI[2]. At the time of writing the latest version is 3.0. A tool called SwaggerHub[3] is the de facto standard to specify APIs in OpenAPI. The specification is defined in YAML, and an example is shown in Figure 5-2. It defines the `GET`

operation on the URI `/resorts`. If the operation is successful, a 200 response code is returned along with a list of resorts in a format defined by a JSON schema that appears later in the specification. If for some reason the query to get a list of resorts operated by `skico.com` returns no entries, a 404 response code is returned along with an error message that is also defined by a JSON schema.

Example 5-1. Figure 5-2 OpenAPI Example

```
paths:
  /resorts:
    get:
      tags:
        - resorts
      summary: get a list of ski resorts in the database
      operationId: getResorts
      responses:
        '200':
          description: successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ResortsList'
        '404':
          description: Resorts not found. Unlikely unless we go
broke
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/responseMsg'
```

API design is a complex topic in itself and delving deeply into this area is beyond the scope of this book. From a scalability perspective, a there are some issues that should however be borne in mind:

- Each API request requires a round trip to a service, which incurs network latency. A common anti-pattern is known as a chatty API, in which multiple API requests are used to perform one logical operation. This commonly occurs when an API is designed following pure object-oriented design approaches. Imagine exposing get() and set() methods for individual resource properties

as HTTP APIs. Accessing a resource would require multiple API requests, one for each property. This is not scalable. Use GET to retrieve the whole resource and PUT to send back an updated resource. You can also use the HTTP PATCH verb[4] to update individual properties of a resource. PATCH allows partial modification of a resource representation, in contrast to PUT which replaces the complete resource representation with new values.

- HTTP APIs that pass large payloads should consider using compression. All modern Web servers and browsers support compressed content using the HTTP `Accept-Encoding` and `Content-Encoding` headers[5]. Specific API requests and responses can utilize these headers by specifying the compression algorithm that is used for the content – for example `gzip`, Compression can reduce network bandwidth and latencies by 50% or more. The trade off cost is the compute cycles to compress and decompress the content. This is typically small compared to the savings in network transit times.

Designing Services

An application server container receives requests and routes them to the appropriate handler function to process the request. The handler is defined by the application service code and implements the business logic required to generate results for the request. As multiple simultaneous requests arrive at a service instance, each is typically[6] allocated an individual thread context to execute the request. The issue of thread handling in application servers is one I'll discuss in more detail later in this chapter.

The sophistication of the routing functionality varies widely by technology platform and language. For example, in Express.js, the container calls a specified function for requests that match an API signature – known as a route path - and HTTP method. The code example below illustrates this with a method that will be called when the client sends a GET request for a specific skier's profile, as identified by the value of `:skierID`.

```
app.get('/skiers/:skierID', function (req, res) {
  // process the GET request
  ProcessRequest(req.params)
})
```

In Java, the widely used Spring framework provides an equally sophisticated method routing technique. It leverages a set of annotations that define dependencies and implement dependency injection to simplify the service code. The code snippet below shows an example of annotations usage:

```
@RestController
public class SkierController {
     @GetMapping("/skiers/{skierID}",
                produces = "application/json")
    public Profile GetSkierProfile(
                        @PathVariable String skierID,
                        ) {
          // DB query method omitted for brevity
        return GetProfileFromDB(skierID);
    }
}
```

These annotations provide the following functionality:

`@RestController`

Identifies the class as a controller that implements an API and automatically serializes the return object into the `HttpResponse` returned from the API.

`@GetMapping`

Maps the API signature to the specific method, and defines the format of the response body

`@PathVariable`

Identifies the parameter as a value that originates in the path for URI that maps to this method

Another Java technology, JEE servlets, also provide annotations, as shown in Figure 5-3, but these are simplistic compared to Spring and other higher-level frameworks. The `@WebServlet` annotation identifies the base pattern for the URI which should cause a particular servlet to be invoked. This is `/skiers` in our example. The class that implements the API method must extend the `HttpServlet` abstract class from the `javax.servlet.http` package and override at least one method that implements an HTTP request handler. The four core HTTP verbs map to methods as follows:

`doGet`

HTTP GET requests

`doPost`

HTTP POST requests

`doPut`

HTTP PUT requests

`doDelete`

for HTTP DELETE requests

Each method is passed two parameters, namely an `HttpServletRequest` and `HttpServletResponse` object. The servlet container creates the `HttpServletRequest` object, which contains members that represent the components of the incoming HTTP request. This object contains the complete URI path for the call, and it is the servlet's responsibility to explicitly parse and validate this, and extract path and query parameters if valid. Likewise, the servlet must explicitly set the properties of the response using the `HttpServletResponse` object.

Servlets therefore require more code from the application service programmer to implement. However, they are likely to provide a more efficient implementation as there is less generated code 'plumbing'

involved in request processing as compared to the more powerful annotation approaches of Spring et al. This is a classic performance versus ease-of-use trade-off. You'll see lots of these in this book.

Example 5-2. JEE Servlet Example

```
import javax.servlet.http.*;
@WebServlet(
    name = "SkiersServlet",
    urlPatterns = "/skiers"
)
public class SkierServlet extends HttpServlet (

protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
  // handles requests to /skiers/{skierID}
  try {
     // extract skierID from the request URI (not shown for
brevity)
     String skierID  = getSkierIDFromRequest(request);
     if(skierID == null) {
        // request was poorly formatted, return error code
        response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
}
     else {
        // read the skier profile from the database
        Profile profile = GetSkierProfile (skierID);
        // add skier profile as JSON to HTTP response and return
200
        response.setContentType("application/json");
        response.getWriter().write(gson.toJson(Profile);
        response.setStatus(HttpServletResponse.SC_OK);
     } catch(Exception ex) {
         response.setStatus
           (HttpServletResponse.SC_INTERNAL_SERVER_ERROR);    }

       }
} }
```

State Management

State management is a tricky, nuanced topic. The bottom line is that service implementations that need to scale should avoid storing conversational state. What on earth does that mean?

Let's start by examining the topic of state management with HTTP.

HTTP is known as stateless protocol. This means each request is executed independently, without any knowledge of the requests that were executed before it from the same client. Statelessness implies that every request needs to be self-contained, with sufficient information provided by the client for the Web server to satisfy the request regardless of previous activity from that client.

The picture is a little more complicated that this simple description portrays, however. For example:

- The underlying socket connection between a client and server is kept open so that the overheads of connection creation are amortized across multiple requests from a client. This is the default behavior for versions HTTP/1 and above.
- HTTP supports cookies, which are known as the HTTP State Management Mechanism[7]. Gives it away really!
- HTTP/2 supports streams, compression, and encryption, all of which require state management.

So, originally HTTP was stateless, but perhaps not anymore? Armed with this confusion (!), I'll move on to state management in application services APIs that are built on top of HTTP.

When a user or application connects to a service, it will typically send a series of requests to retrieve and update information. *Conversational state* represents any information that is retained between requests such that a subsequent request can assume the service has retained knowledge about the previous interactions. I'll explore what this means in a simple example.

In the skier service API, a user may request their profile by submitting a GET request to the following URI:

```
GET /skico.com/skiers/768934
```

They may then use their app to modify their `city` attribute and send a PUT request to a update the resource:

```
PUT /skico.com/skiers/
{
    "username": "Ian123",
    "email":”i.gorton@somewhere.com”
    “city”: “Wenatchee”
}
```

As this URI does not identify the skier, the service must know the unique identifier of the resource to update, namely 768934. Hence for this update operation to succeed, the service must have retained conversational state from the previous GET request.

Implementing this approach is relatively straightforward. When the service receives the initial GET request, it creates a session state object that uniquely identifies the client connection. In reality, this is often performed when a user first connects to or logs in to a service. The service can then read the skier profile from the database and utilize the session state object to store conversational state – in our example this would be `skierID` and likely values associated with the skier profile. When the subsequent PUT request arrives from the client it uses the session state object to look up the `skierID` associated with this session and uses that to update the skier's home city.

Services that maintain conversational state are known as stateful services. Stateful services are attractive from a design perspective as they can minimize the number of times a service retrieves data (state) from the database and reduce the amount of data that is passed between clients and the services.

For services with light request loads they make eminent sense and are promoted by many frameworks to make services easy to build and deploy. For example, JEE servlets support session management using the `HttpSession` object, and similar capabilities are offered by the `Session` object in ASP.NET.

As you scale the service implementations however, the stateful approach becomes problematic. For a single service instance, you have two problems to consider:

1. If you have multiple client sessions all maintaining session state, this will utilize available service memory. The amount of memory utilized will be proportional to the number of clients the service is maintaining state for. If a sudden spike of requests arrives, how can you be certain we will not exhaust available memory and cause the service to fail?

2. You also must be mindful about how long to keep session state available. A client may stop sending requests but not cleanly close their connection to allow the state to be reclaimed. All session management approaches support a default session timeout. If you set this to a short time interval, clients may see their state disappear unexpectedly. If you set the session time out period to be too long, you may degrade service performance as it runs low on resources.

In contrast, stateless services do not assume that any conversational state from previous calls has been preserved. The service should not maintain any knowledge from earlier requests, so that each request can be processed individually. This requires the client to provide all the necessary information for the service to process the request and provide a response. This is in fact how the skier API is specified in Table 5-1, namely:

```
PUT /skico.com/skiers/768934
{
    "username": "Ian123",
    "email":”i.gorton@somewhere.com”
    “city”: “Wenatchee”
}
```

A sequence diagram illustrating this stateless design is shown in Figure 5-2.

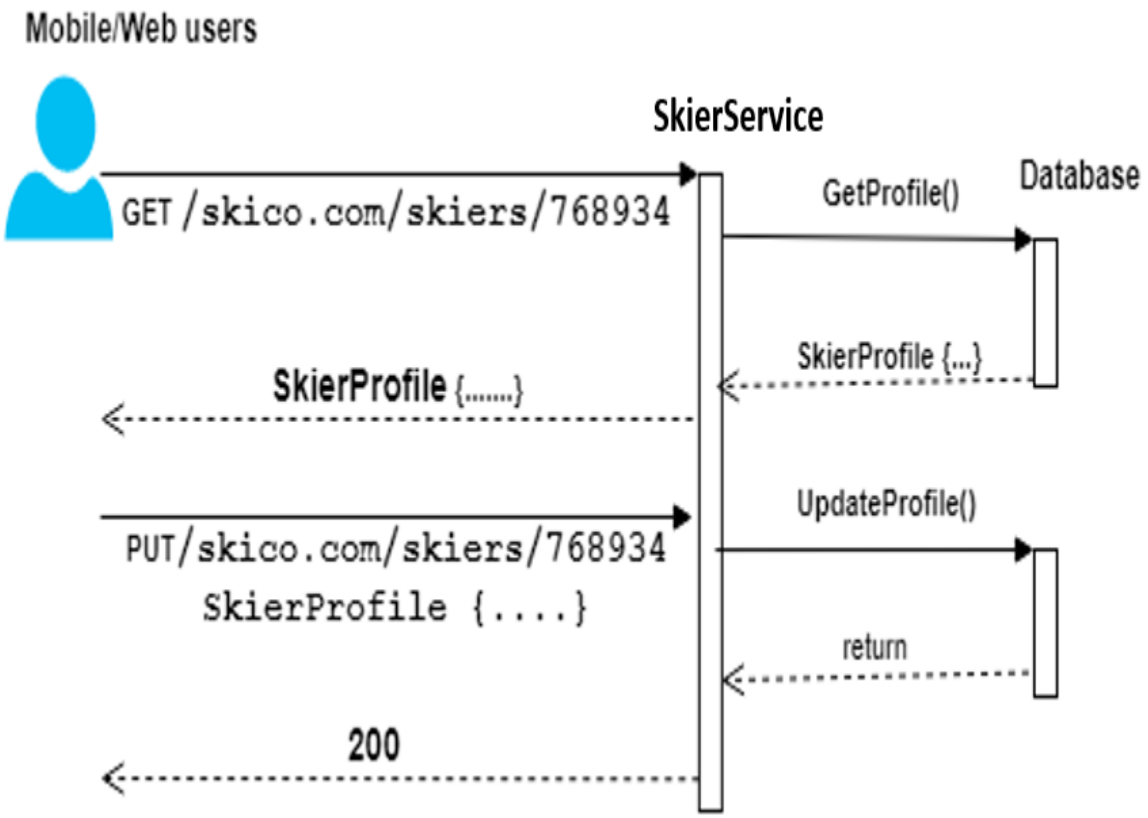

Figure 5-2. Stateless API Example

Any scalable service will need stateless APIs. The reason why will become clear when I explain horizontal scaling later in this chapter. For now, the most important design implication is that for a service that needs to retain state pertaining to client sessions – the classic shopping cart example – this state must be stored externally to the service. This invariably means an external data store.

Applications Servers

Application servers are the heart of a scalable application, hosting the business services that comprise an application. Their basic role is to accept requests from clients, apply application logic to the requests, and reply to the client with the request results. Clients may be external or internal, as in other services in the application that require to use the functionality of a specific service.

The technological landscape of application servers is broad and complex, depending on the language you want to use and the specific capabilities that each offers. In Java, the Java Enterprise Edition (JEE)[8] defines a comprehensive, feature rich standards-based platform for application servers, with multiple different vendor and open source implementations.

In other languages, the Express.js[9] server supports Node, Flask supports Python[10], and in GoLang a service can be created by incorporating the `net/http` package. These implementations are much more minimal and lightweight than JEE and are typically classified as Web application frameworks. In Java, the Apache Tomcat server[11] is a somewhat equivalent technology. Tomcat is an open source implementation of a subset of the JEE platform, namely the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies.

Figure 5-3 depicts a simplified view of the anatomy of Tomcat. Tomcat implements a *servlet container*, which is an execution environment for application-defined servlets. Servlets are dynamically loaded into this container, which provides lifecycle management and a multithreaded runtime environment.

Requests arrive at the IP address of the server, which is listening for traffic on specific ports. For example, by default Tomcat listens on port 8080 for HTTP requests and 8443 for HTTPS requests. Incoming requests are processed by one or more listener threads. These create a TCP/IP socket connection between the client and server. If network requests arrive at a frequency that cannot be processed by the TCP listener, pending requests are queued up in the *Sockets Backlog*. The size of the backlog is operating system dependent. In most Linux versions the default is 100.

Once a connection is established, the TCP requests are marshalled by, in this example, a *HTTP Connector* which generates the HTTP request (`HttpServletRequest` object as in Figure 5-2) that the servlet can process. The HTTP request is then dispatched to an application container thread to process.

Application container threads are managed in a thread pool, essentially a Java `Executor`, which by default in Tomcat is a minimum size of 25 threads and a maximum of 200. If there are no available threads to handle a request, the container maintains them in a queue of runnable tasks and dispatches these as soon as a thread becomes available. This queue by default is size `Integer.MAX_VALUE` – that is, essentially unbounded[12]. By default, if a thread remains idle for 60 seconds, it is killed to free up resources in the Java Virtual Machine.

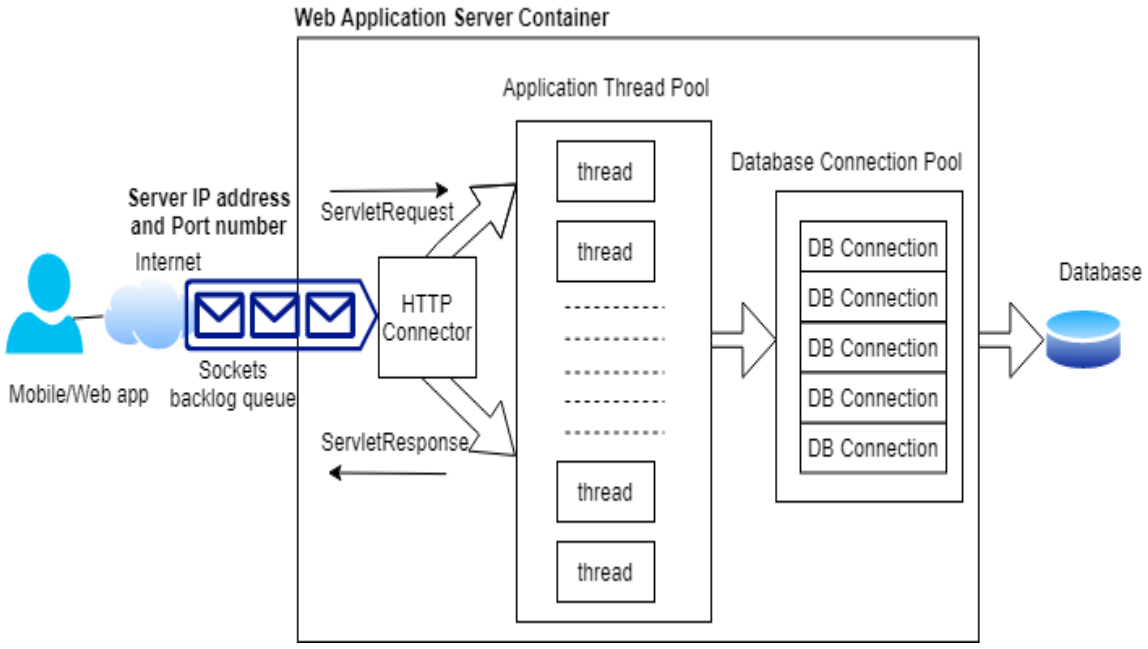

Figure 5-3. Anatomy of a Web application server

For each request, the method that corresponds with the HTTP request is invoked in a thread. The servlet method processes the HTTP request headers, executes the business logic, and constructs a response that is marshalled by the container back into a TCP/IP packet and sent over the network to the client.

In processing the business logic, servlets often need to query an external database. This requires each thread executing the servlet methods to obtain a database connection and execute database queries. In many databases, especially relational ones, connections are limited resources as they consume memory and system resources in both the client and database

server. For this reason, a fixed size database connection pool is typically utilized. The pool hands out open connections to requesting threads on demand.

When a servlet wishes to submit a query to the database, it requests a connection from the pool. If one is available, access to the connection is granted to the servlet until it indicates it has completed its work. At that stage the connection is returned to the pool and made available for another servlet to utilize. As the container thread pool is typically larger than the database connection pool, a servlet may request a connection when none are available. To handle this, the connection pool maintains a request queue and hands out open connections on a FIFO basis, and threads in the queue are blocked until there is availability or a timeout occurs.

An application server framework such as Tomcat is hence highly configurable for different workloads. For example, the size of the thread and database connection pools can be specified in configuration files that are read at startup.

The complete Tomcat container environment runs within a single JVM, and hence processing capacity is limited by the number of vCPUs available and the amount of memory allocated as heap size. Each allocated thread consumes memory, and the various queues in the request processing pipeline consume resources while requests are waiting. This means that request response time will be governed by both the request processing time in the servlet business logic as well as the time spent waiting in queues for threads and connections to become available.

In a heavily loaded server with many threads allocated, context switching may start to degrade performance, and available memory may become limited. If performance degrades, queues grow as requests wait for resources. This consumes more memory. If more requests are received than can be queued up and processed by the server, then new TCP/IP connections will be refused, and clients will see errors. Eventually, an overloaded server may run out of resources and start throwing exceptions and crash.

Consequently, time spent tuning configuration parameters to efficiently handle anticipated loads is rarely wasted. Systems tend to degrade in performance well before they reach 100% utilization.[13] Once any resource - CPU utilization, memory usage, network, disk accesses - gets close to full utilization, systems exhibit less predictable performance. This is because more time is spent, for example thread context switching and memory garbage collecting. This inevitably affects latencies and throughput. Thus, having a utilization target is essential. Exactly what these thresholds should be is extremely application dependent.

Monitoring tools available with Web application frameworks enable engineers to gather a range of important metrics, including latencies, active requests, queue sizes and so on. These are invaluable for carrying out data-driven experiments that lead to performance optimization.

Java-based application frameworks such as Tomcat support the JMX[14] (Java Management Extensions) framework, which is a standard part of the Java Standard Edition platform. JMX enables frameworks to expose monitoring information based on the capabilities of MBeans (Managed Beans), which represent a resource of interest (e.g., thread pool, database connections usage). This enables an ecosystem of tools to offer capabilities for monitoring JMX-supported platforms. These range from JConsole[15] which is available in the Java Development Kit by default, to powerful open source technologies such as JavaMelody[16] and many expensive commercial offerings.

Horizontal Scaling

A core principle of scaling a system is being able to easily add new processing capacity to handle increased load. For most systems, a simple and effective approach is deploying multiple instances of stateless server resources and using a load balancer to distribute the requests across these instances. This is known as horizontal scaling and illustrated in Figure 5-4.

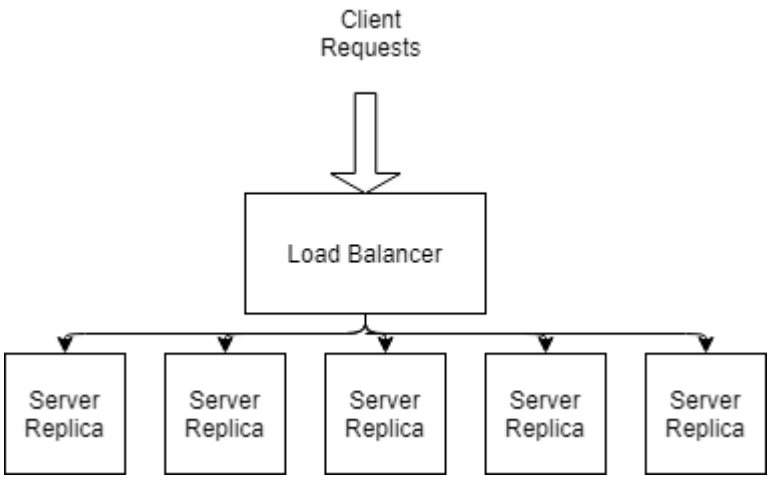

Figure 5-4. Simple Load Balancing Example

These two ingredients, namely stateless service replicas and a load balancer, are both necessary for horizontal scaling. I'll explain why.

Service replicas are deployed on their own (virtual) hardware. Hence if we have two replicas, we double our processing capacity. If we have ten replicas, we have potentially 10x capacity. This enables our system to handle increased loads. The aim of horizontal scaling is to create a system processing capacity that is the sum of the total resources available.

The servers need to be stateless, so that any request can be sent to any service replica to handle. This decision is made by the load balancer, which can use various policies to distribute requests. If the load balancer can keep each service replica equally busy, then we are effectively using the processing capacity provided by the service replicas.

If our services are stateful, the load balancer needs to always route requests from the same server to the same service replica. As client sessions have indeterminate durations, this can lead to some replicas being much more heavily loaded than others. This creates an imbalance and is not effective in using the available capacity evenly across replicas. I'll return to this issue in more detail in the next section on load balancing.

> **NOTE**
>
> Technologies like Spring Session and plugins to Tomcat's Clustering platform allow session state to be externalized in general purpose distributed caches like Redis and Memcached. This effectively makes our services stateless. Load balancers can distribute requests across all replicated services without concern for state management. I'll cover the topic of distributed caches in Chapter 6.

Horizontal scaling also increases availability. With one service instance, if it fails, the service is unavailable. This is known as a single point of failure (SPoF) – a bad thing, and something to avoid in any scalable distributed system. Multiple replicas increase availability. If one replica fails, requests can be directed to any – they are stateless, remember – replica. The system will have reduced capacity until the failed server is replaced, but it will still be available. Which is important. The ability to scale is crucial, but if a system is unavailable, then the most scalable system ever built is still somewhat ineffective!

Load Balancing

Load balancing aims to effectively utilize the capacity of a collection of services to optimize the response time for each request. This is achieved by distributing requests across the available services to ideally utilize the collective service capacity. The objective is to avoid overloading some services while underutilizing others.

Clients send requests to the IP address of the load balancer, which redirects requests to target services, and relays the results back to the client. This means clients never contact the target services directly, which is also beneficial for security as the services can live behind a security perimeter and not be exposed to the Internet.

Load balancers may act at the *network level* or the *application level*. These are often called *Layer 4* and *Layer 7* load balancers, respectively. The names refer to network transport layer at Layer 4 in the Open Systems Interconnection (OSI) Reference Model[17], and the application layer at

Layer 7. The OSI model defines network functions in seven abstract layers. Each layer defines standards for how data is packaged and transported.

Network level load balancers distribute requests at the network connection level, operating on individual TCP or UDP packets. Routing decisions are made on the basis of client IP addresses. Once a target service is chosen, the load balancer uses a technique called Network Address Translation (NAT). This changes the destination IP address in the client request packet from that of the load balancer to that of the chosen target. When a response is received from the target, the load balancer changes the source address recorded in the packet header from the target's IP address to its own. Network load balancers are relatively simple as they operate on the individual packet level. This means they are extremely fast, as they provide few features beyond choosing a target service and performing NAT functionality.

In contrast, application level load balancers reassemble the complete HTTP request and base their routing decisions on the values of the HTTP headers and on the actual contents of the message. For example, a load balancer can be configured to send all POST requests to a subset of available services, or distribute requests based on a query string in the URI. Application load balancers are sophisticated reverse proxies. The richer capabilities they offer means they are slightly slower than network load balancers, but the powerful features they offer can be utilized to more than make up for the overheads incurred.

To give you some idea of the raw performance difference between network and application layer load balancers, Figure 5-5 compares the two in a simple application scenario. The load balancing technology under test is the AWS Application and Network Elastic Load Balancers[18]. Each load balancer routes requests to one of 4 replicas. These execute the business logic and return results to the clients via the load balancer. Client load varies from a lightly loaded 32 concurrent clients to a moderate 256 concurrent clients. Each client sends a sequence of requests with no delay between receiving the results from one request and sending the next request to the server.

You can see from Figure 5-5 that the network load balancer delivers on average around 20% higher performance for the 32, 64, and 128 client tests. This validates the expected higher performance from the less sophisticated network load balancer. For 256 clients, the performance of the two load balancers is essentially the same. This is because the capacity of the 4 replicas is exceeded and the system has a bottleneck. At this stage the load balancers make no difference to the system performance. You need to add more replicas to the load balancing group to increase system capacity, and hence throughput.

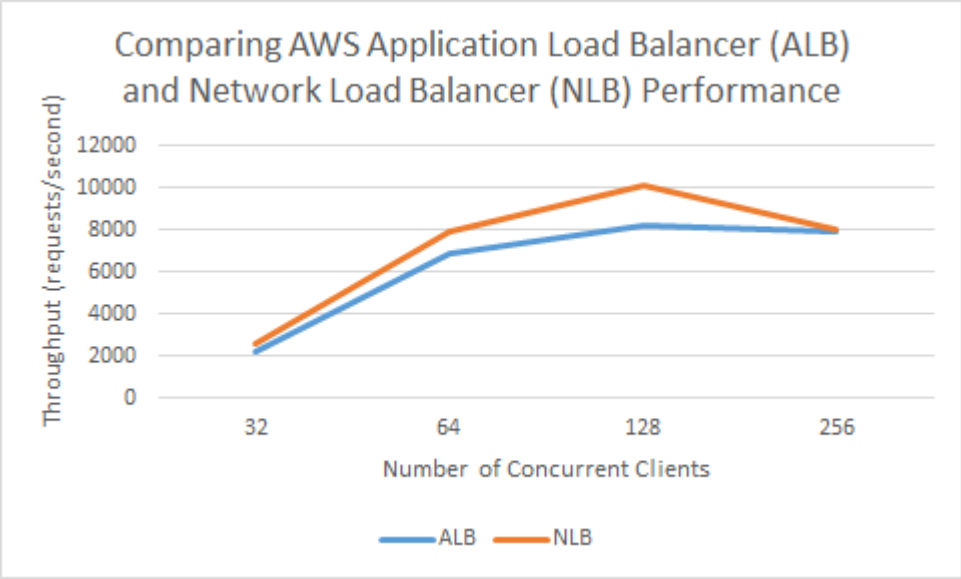

Figure 5-5. Comparing Load Balancer Performance[19]

In general, a load balancer has the following features that will be explained in the following sections:

- Load distribution policies
- Health monitoring
- Elasticity
- Session affinity

Load Distribution Policies

Load distribution policies dictate how the load balancer chooses a target service to process a request. Any load balancer worth its salt will offer several load distribution policies – HAProxy offers 10 in fact[20]. The following are four of the most commonly supported across all load balancers:

round-robin

: The load balancer distributes requests to available servers in a round-robin fashion

least connections

: The load balancer distributes new requests to the server with the least open connections

HTTP header field

: The load balancer directs requests based on the contents of a specific HTTP header field. For example all requests with the header field X-Client-Location:US,Seattle could be routed to a specific set of servers.

HTTP operation

: The load balancer directs requests based on the HTTP verb in the request

Load balancers will also allow services to be allocated weights. For example, standard service instances in the load balancing pool may have 4 vCPUs and each is allocated a weight of 1. If a service ereplica running on 8 vCPUs is added, it can be assigned a weight of 2 so the load balancer will send twice as many requests its way.

Health Monitoring

A load balancer will periodically send pings and attempt connections to test the health of each service in the load balancing pool. These tests are called health checks. If a service becomes unresponsive or fails connection

attempts, it will be removed from the load balancing pool and no requests will be sent to that host. If the connection to the service has experienced a transient failure, the load balancer will reincorporate the service once it becomes available and healthy. If, however, it has failed, the service will be removed from the load balancer target pool.

Elasticity

Spikes in request loads can cause the service capacity available to a load balancer to become saturated, leading to longer response times and eventually request and connection failures. *Elasticity* is the capability of an application to dynamically provision new service capacity to handle an increase in requests. As load increases, new replicas are started and the load balancer directs requests to these. As load decreases the load balancer stops services that are no longer needed.

Elasticity requires a load balancer to be tightly integrated with application monitoring, so that scaling policies can be defined to determine when to scale up and down. Policies may specify for example that capacity for a service should be increased when the average service CPU utilization across all instances is over 70%, and decreased when average CPU utilization is below 40%. Scaling policies can typically be defined using any metrics that are available through the monitoring system.

An example of elastic load balancing is the Amazon Web Services (AWS) Auto-Scaling groups. An Auto Scaling group is a collection of service instances available to a load balancer that is defined with a minimum and maximum size. The load balancer will ensure the group always has the minimum numbers of services available, and the group will never exceed the maximum number. This scheme is illustrated in Figure 5-6.

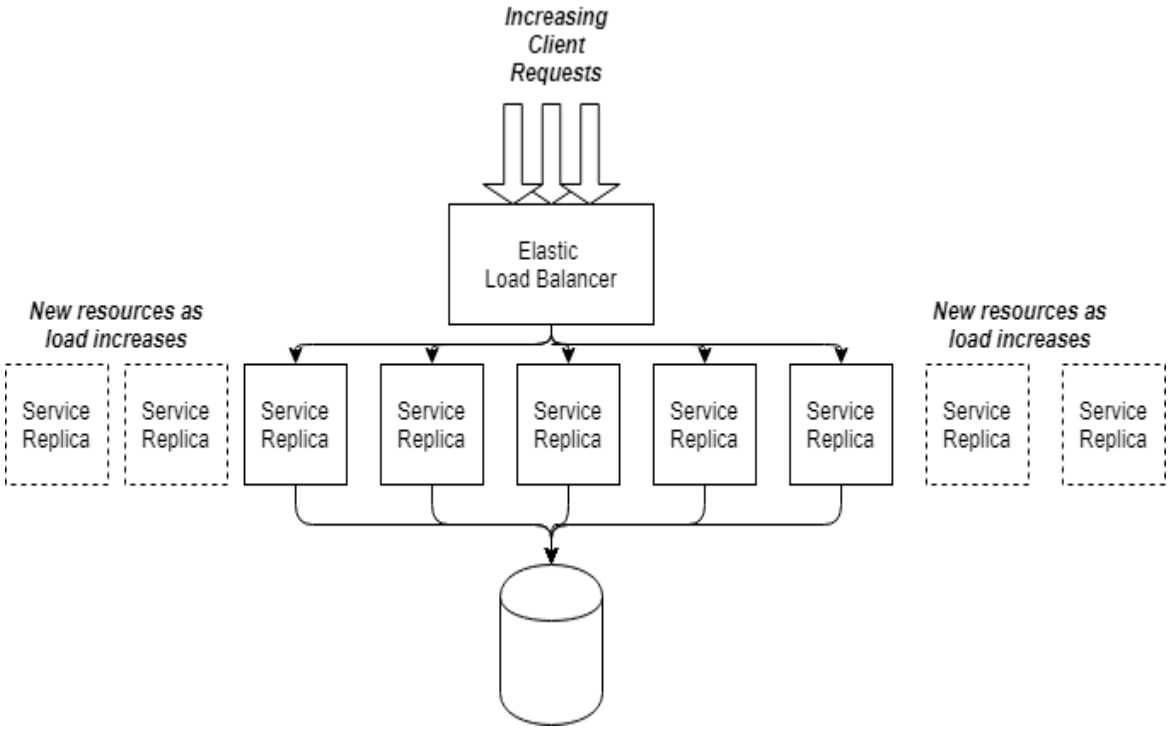

Figure 5-6. Elastic Load Balancing

Typically, there are two ways to control the number of replicas in a group. The first is based on a schedule, when the request load increases and decreases are predictable. For example, you may have an online entertainment guide and publish the weekend events for a set of major cities at 6pm on Thursday. This generates a higher load until Sunday at noon. An Auto Scaling group could easily be configured to provision new services at 6pm Thursday and reduce the group size to the minimum at noon Sunday.

If increased load spikes are not predictable, elasticity can be controlled dynamically by defined scaling policies based on application metrics such as average CPU and memory usage and number of messages in a queue. If the upper threshold of the policy is exceeded, the load balancer will start one or more new service instances until performance drops below the metric threshold. Instances need time to start – often a minute or more - and hence a *warmup* period can be defined until the new instance is considered to be contributing to the group's capacity. When the observed metric value drops below the lower threshold defined in the scaling policy, *scale in* or *scale down* commences and instances will be automatically stopped and removed from the pool.

Elasticity is a key feature that allows services to scale dynamically as demand grows. For highly scalable systems with fluctuating workloads it is pretty much a mandatory capability for providing the necessary capacity at minimum costs.

Session Affinity

Session affinity, or sticky sessions, are a load balancer feature for stateful services. With sticky sessions, the load balancer sends all requests from the same client to the same service instance. This enables the service to maintain in-memory state about each specific client session.

There are various ways to implement sticky sessions. For example, HAProxy provides a comprehensive set of capabilities to maintain client requests on the same service in the face of service additions, removals and failures[21]. AWS Elastic Load Balancing generates an HTTP cookie that identifies the service replica a client's session is associated with. This cookie is returned to the client, which must send it in subsequent requests to ensure session affinity is maintained.

Sticky sessions can be problematic for highly scalable systems. They lead to a load imbalance problem, in which, over time, clients are not evenly distributed across services. This is illustrated in Figure 5-7, where two clients are connected to one service while another service remains idle.

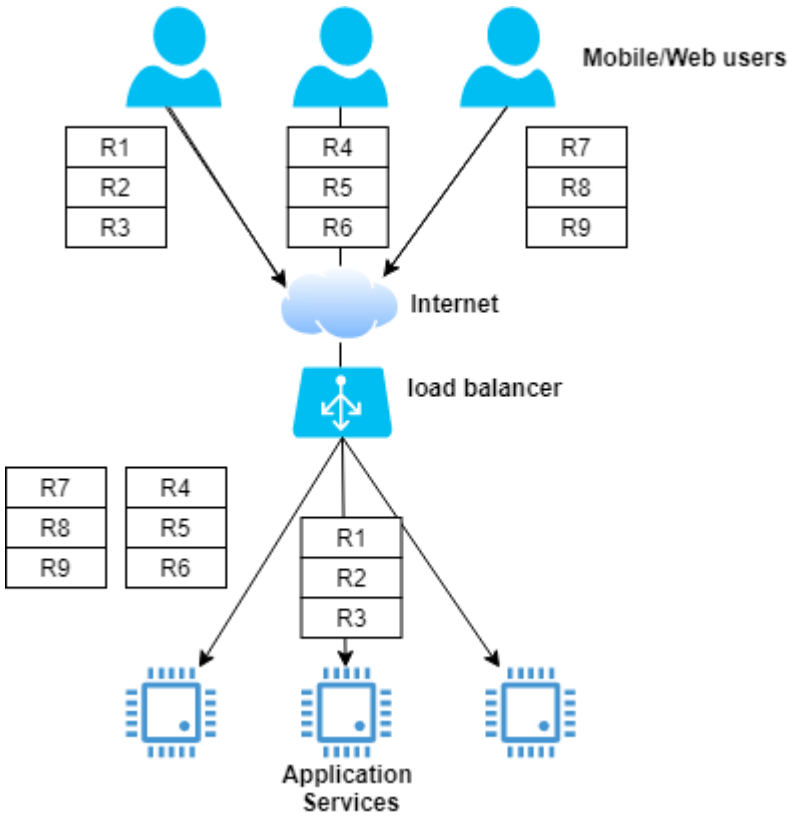

Figure 5-7. Load Imbalance with Sticky Sessions

Load imbalance occurs because client sessions last for varying amounts of time. Even if sessions are evenly distributed initially, some will terminate quickly while others will persist. In a lightly loaded system, this tends to not be an issue. However, in a system with millions of sessions being created and destroyed constantly, load imbalance is inevitable. This will lead to some service replicas being underutilized, while others are overwhelmed and may potentially fail due to resource exhaustion. To help alleviate load imbalance, load balancers usually provide policies such as sending new sessions to instances with the least connections or fastest response times. These help direct new sessions away from heavily loaded services.

Stateful services have other downsides. When a service inevitably fails, how do the clients connected to that server recover the state that was being managed? If a service instance becomes slow due to high load, how do clients respond? In general stateful servers create problems that in large scale systems can be difficult to design around and manage.

Stateless services have none of these downsides. If one fails, clients get an exception and retry, with their request routed to another live service replica. If a service is slow due to a transient network outage, the load balancer takes it out of the service group until it passes health checks or fails. All application state is either externalized or provided by the client in each request, so service failures can be handled easily by the load balancer.

Stateless services enhance scalability, simplify failure scenarios and ease the burden of service management. For scalable applications, these advantages far outweigh the disadvantages, and hence their adoption in most major large scale Internet sites such as Netflix.[22]

Finally, bear in mind that scaling one collection of services through load balancing may well overwhelm downstream services or databases that the load balanced services depend on. Just like with highways, adding 8 traffic lanes for 50 miles will just cause bigger traffic chaos if the highway ends at a set of traffic lights with a one lane road on the other side. We've all been there, I'm sure. I'll address these issues in Chapter 9.

Summary and Further Reading

Services are the heart of a scalable software system. They define the contract as an API that specifies their capabilities to clients. Services execute in an application server container environment that hosts the service code and routes incoming API requests to the appropriate processing logic. Application servers are highly programming language dependent, but in general provide a multithreaded programming model that allows services to process many requests simultaneously. If the threads in the container thread pool are all utilized, the application server queues up requests until a thread becomes available.

As request loads grow on a service, we can scale it out horizontally using a load balancer to distribute requests across multiple instances. This architecture also provides high availability as the multiple service configuration means the application can tolerate failures of individual instances. The service instances are managed as a pool by the load balancer,

which utilizes a load distribution policy to choose a target service replica for each request. Stateless services scale easily and simplify failure scenarios by allowing the load balancer to simply resend requests to responsive targets. Although most load balancers will support stateful services using a feature called sticky sessions, stateful services make load balancing and handling failures more complex. Hence, they are not recommended for highly scalable services.

API design is a topic of great complexity and debate. An excellent overview of basic API design and resource modeling is https://www.thoughtworks.com/insights/blog/rest-api-design-resource-modeling.

The Java Enterprise Edition (JEE) is an established and widely deployed server-side technology. It has a wide range of abstractions for building rich and powerful services. The Oracle tutorial is an excellent starting place for appreciating this platform - https://docs.oracle.com/javaee/7/tutorial/.

Much of the knowledge and information about load balancers is buried in the documentation provided by the technology suppliers. You choose your load balancer and then dive into the manuals. For an excellent, broad perspective on the complete field of load balancing, *Server Load Balancing* by Tony Bourke (O'Reilly) is a good resource.

---

1 Fielding, Roy Thomas (2000). "Architectural Styles and the Design of Network-based Software Architectures". *Dissertation*. University of California, Irvine

2 https://app.swaggerhub.com/help/tutorials/openapi-3-tutorial

3 https://app.swaggerhub.com

4 https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/PATCH

5 https://developer.mozilla.org/en-US/docs/Web/HTTP/Compression

6 Node.js is a notable exception here as it is single threaded. However, it employs an asynchronous programming model for blocking input-output that supports handling many simultaneous requests.

7 https://tools.ietf.org/html/rfc6265

8 https://www.oracle.com/java/technologies/java-ee-glance.html

9 https://expressjs.com/

10 https://palletsprojects.com/p/flask/

11 http://tomcat.apache.org/

12 See https://tomcat.apache.org/tomcat-9.0-doc/config/executor.html for default Tomcat Executor configuration settings

13 https://sre.google/sre-book/monitoring-distributed-systems/

14 https://www.oracle.com/java/technologies/javase/javamanagement.html

15 https://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html

16 https://github.com/javamelody/javamelody/wiki

17 https://www.networkworld.com/article/3239677/the-osi-model-explained-and-how-to-easily-remember-its-7-layers.html

18 https://aws.amazon.com/elasticloadbalancing/?whats-new-cards-elb.sort-by=item.additionalFields.postDateTime&whats-new-cards-elb.sort-order=desc

19 Experimental results due to Ruijie Xiao, from Northeastern University's MS in Computer Science in Seattle.

20 http://cbonte.github.io/haproxy-dconv/2.3/intro.html#3.3.5

21 http://cbonte.github.io/haproxy-dconv/2.3/intro.html#3.3.6

22 https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

Chapter 6. Distributed Caching

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 6th chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Caches exist in many places in an application. The CPUs that run your applications have multi-level, fast hardware caches to reduce relatively slow main memory accesses. Database engines can make use of main memory to cache the contents of the data store in memory so that in many cases queries do not have to touch relatively slow disks.

Distributed caching is an essential ingredient of a scalable system. Caching makes the results of expensive queries and computations available for reuse by subsequent requests at low cost. By not having to reconstruct the cached results for every single request, the capacity of the system is increased, and it is hence able to scale to handle greater workloads.

There's two flavors of caching that I'll cover in this chapter. Application caching requires business logic that incorporates the caching and access of precomputed results using distributed caches. Web caching exploits mechanisms built into the HTTP protocol to enable caching of results within the infrastructure provided by the Internet. When used effectively, both will protect your services and databases from heavy read traffic loads.

Application Caching

Application caching is designed to improve request responsiveness by storing the results of queries and computations in memory so they can be subsequently served by later requests. For example, think of an online newspaper site where readers can leave comments. Once posted, articles change infrequently, if ever. New comments tend to get posted soon after an article is published, but the frequency drops quickly with the age of the article. Hence an article can be cached on first access and reused by all subsequent requests until the article is updated, new comments are posted, or no one wants to read it anymore.

In general, caching relieves databases of heavy read traffic, as many queries can be served directly from the cache. It also reduces computation costs for objects that are expensive to construct, for example those needing queries that span several different databases. The net effect is to reduce the computational load on our services and databases and create head room, or capacity for more requests.

Caching requires additional resources, and hence cost, to store cached results. However, well designed caching schemes are low-cost compared to upgrading database and service nodes to cope with higher request loads. As an indication of the value of caches, approximately 3% of infrastructure at Twitter is dedicated to application level caches.[1] At Twitter scale, operating hundreds of clusters, that is a lot of infrastructure!

Application level caching exploits dedicated distributed cache engines. The two predominant technologies in this area are memcached[2] and Redis.[3] Both are essentially distributed in-memory hash tables designed for arbitrary data (strings, objects) representing the results of database queries or downstream service API calls. Common use cases for caches are storing user session data, dynamic web pages and results of database queries. The cache appears to application services as a single store, and objects are allocated to individual cache servers using a hash function on the object key.

The basic scheme is shown in Figure 6-1. The service first checks the cache to see if the data it requires is available. If so, it returns the cached contents as the results – this is known as a *cache hit*. If the data is not in the cache – a *cache miss* - the service retrieves the requested data from the database and writes the query results to the cache so it is available for subsequent client requests without querying the database.

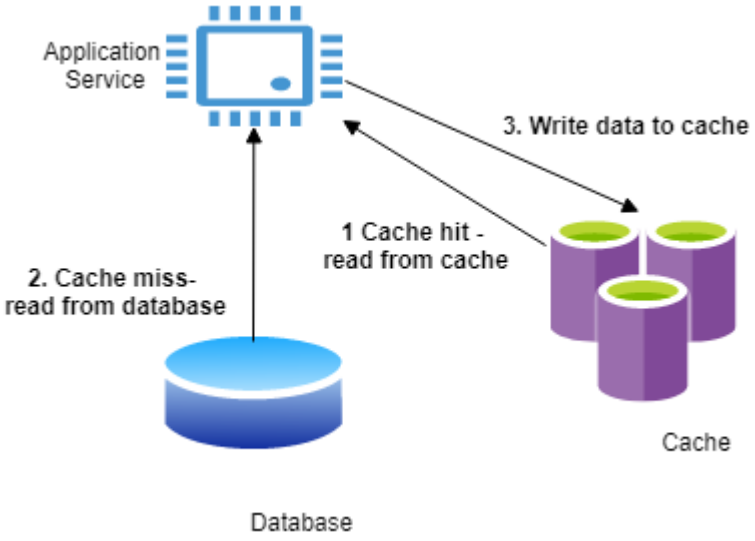

Figure 6-1. Application Level Caching

For example, at a busy winter resort, skiers and boarders can use their mobile app to get an estimate of the lift wait times across the resort. This enables them to plan and avoid congested areas where they will have to wait to ride a lift for say 15 minutes (or sometimes more!).

Every time a skier loads a lift, a message is sent to the company's service that collects data about skier traffic patterns. Using this data, the system can estimate lift wait times from the number of skiers who ride a lift and the rate they are arriving. This is an expensive calculation, taking maybe a second or more at busy times, as it requires aggregating potentially 10's of thousands of lift ride records and performing the wait time calculation. For this reason, once the results are calculated, they are deemed valid for five minutes. Only after this time has elapsed is a new calculation performed and results produced.

Figure 6-2 shows an example of how a stateless `LiftWaitService` might work. When a request arrives, the service first checks the cache to see if the latest wait times are available. If they are, the results are immediately returned to the client. If the results are not in the cache, the service calls a downstream service which performs the lift wait calculations and returns them as a `List`. These results are then stored in the cache and then returned to the client.

Cache access requires a key with which to associate the results with. In this example the key is constructed with the string "`liftwaittimes:`" concatenated with the resort identifier that is passed by the client to the service. This key is then hashed by the cache to identify the server where the cached value resides. I'll describe how the hash algorithm for most distributed caches typically works in Chapter 13, as the same approach is commonly used in distributed databases for key distribution.

When a new value is written to the cache, a value of 300 seconds is passed as a parameter to the `put` operation. This is known as a *time to live*, or *TTL* value. It tells the cache that after 300 seconds this key-value pair should be evicted from the cache as the value is no longer current – also known as stale.

While the cache value is valid, all requests will utilize it. This means there is no need to perform the expensive lift wait time calculation for every call. A cache hit on a fast network will take maybe a millisecond – much faster than the lift wait times calculation. When the cache value is evicted after 300 seconds, the next request will result in a cache miss. This will result in the calculation of the new values to be stored in the cache. Hence if we get N requests in a 5 minute period, N-1 requests are served from the cache. Imagine if N is 10000? This is a lot of expensive calculations saved, and CPU cycles that your database can use to process other queries.

Example 6-1. Figure 6-2 Caching Example

```
public class LiftWaitService {
  public List getLiftWaits(String resort) {
    List liftWaitTimes = cache.get("liftwaittimes:" + resort);
      if (liftWaitTimes == null) {
         liftWaitTimes = skiCo.getLiftWaitTimes(resort);
```

```
        // add result to cache, expire in 300 seconds
        cache.put("liftwaittimes:" + resort, liftWaitTimes, 300);
      }
    return liftWaitTimes;
     }
  }
```

Using an expiry time like the *TTL* is a common way to invalidate cache contents. It ensures a service doesn't deliver stale, out of date results to a client. It also enables the system to have some control over cache contents, which are typically limited. If cached items are not flushed periodically, the cache may fill up. In this case, a cache will adopt a policy such as *least recently used* or *least accessed* to choose cache entries to evict and create space for more current, timely results.

Application caching can provide significant throughput boosts, reduced latencies, and increased client application responsiveness. The key to achieving these desirable qualities is to satisfy as many requests as possible from the cache. The general design principle is to maximize the cache hit rate and minimize the cache miss rate. When a cache miss occurs, the request must be satisfied through querying databases or downstream services. The results of the request can then be written to the cache and hence be available for further accesses.

There's no hard and fast rule on what the cache hit rate should be, as it depends on the cost of constructing the cache contents and the update rate of cached items. Ideal cache designs have many more reads than updates. This is because when an item must be updated, the application needs to invalidate cache entries that are now stale because of the update. This means the next request will result in a cache miss.[4]

When items are updated regularly, the cost of cache misses can negate the benefits of the cache. Service designers therefore need to carefully consider query and update patterns an application experiences, and construct caching mechanisms that yield the most benefit. It is also crucial to monitor the cache usage once a service is in production to ensure the hit and miss rates are in line with design expectations. Caches will provide both management utilities and APIs to enable monitoring of the cache usage characteristics.

For example, memcached makes a large number of statistics available, including the hit and miss counts as shown in the snippet of output below.

```
STAT get_hits 98567
STAT get_misses 11001
STAT evictions 0
```

Application level caching is also known as the *cache-aside* pattern.[5] The name references the fact that the application code effectively bypasses the data storage systems if the required results are available in the cache. This contrasts with other caching patterns in which the application always reads from and writes to the cache. These are known as *read-through*, *write-through* and *write-behind* caches as explained below:

Read- through

> The application satisfies all requests by accessing the cache. If the data required is not available in the cache, a loader is invoked to access the data systems and load the results in the cache for the application to utilize.

Write- through

> The application always writes updates to the cache. When the cache is updated, a writer is invoked to write the new cache values to the database. When the database is updated, the application can complete the request.

Write- behind

> Like write-through, except the application does not wait for the value to be written to the database from the cache. This increases request responsiveness at the expense of possible lost updates if the cache server crashes before a database update is completed. This is also known as a write-back cache, and internally is the strategy used by most database engines.

The beauty of these caching approaches is that they simplify application logic. Applications always utilize the cache for reads and writes, and the cache provides the 'magic' to ensure the cache interacts appropriately with the backend storage systems. This contrasts with the cache-aside pattern, in which application logic must be cognizant of cache misses.

Read-through, write-through and write-behind strategies require a cache technology that can be augmented with an application-specific handler to perform database reads and writes when the application accesses the cache. For example, NCache[6] supports *provider interfaces* that the application implements. These are invoked automatically on cache misses for read-through caches and on writes for write-through caches. Other such caches are essentially dedicated database caches, and hence require cache access to be identical to the underlying database model. An example of this is Amazon's DynamoDB Accelerator (DAX).[7] DAX sits between the application code and DynamoDB, and transparently acts as a high-speed in memory cache to reduce database access times.

One significant advantage of the cache-aside strategy is that it is resilient to cache failure. In such circumstances, as the cache is unavailable, all requests are essentially handled as a cache miss. Performance will suffer, but services will still be able to satisfy requests. In addition, scaling cache-aside platforms such as Redis and Memcached is straightforward due to their simple, distributed hash table model. For these reasons, the cache-aside pattern is the primary approach seen in massively scalable systems.

Web Caching

One of the reasons that Web sites are so highly responsive is that the Internet is littered with Web caches. Web caches store a copy of a given resource, for example a Web page or an image, for a defined time period. The caches intercept client requests and if they have a requested resource cached locally, they return the copy rather than forwarding the request to the target service. Hence many requests can be satisfied without placing a

burden on the service. Also, as the caches are physically closer to the client, the requests will have lower latencies.

Figure 6-2 gives an overview of the Web caching architecture. Multiple levels of caches exist, starting with the client's Web browser cache and local organization-based caches. Internet Service Providers will also implement general web proxy caches, and reverse proxy caches can be deployed within the application services execution domain. Web browser caches are also known as private caches (for a single user). Organizational and ISP proxy caches are shared caches that support requests from multiple users.

Edge caches, also known as CDNs (Content Delivery Networks), live at various strategic geographical locations globally, so that they cache frequently accessed data close to clients. For example a video streaming provider may configure an edge cache in Sydney, Australia to serve video content to Australasian users rather than streaming content across the Pacific Ocean from US-based origin servers. Edge caches are deployed globally by CDN providers. For example Akamai, the original CDN provider, has over 2000 locations and delivers up to 30% of Internet traffic globally.[8] For media rich sites with global users, edge caches are essential.

Caches typically store the results of GET requests only, and the cache key is the URI of the associated GET. When a client sends a GET request, it may be intercepted by one or more caches along the request path. Any cache with a fresh copy of the requested resource may respond to the request. If no cached content is found, the request is served by the service endpoint, which is also called in Web technology parlance as the *origin server*.

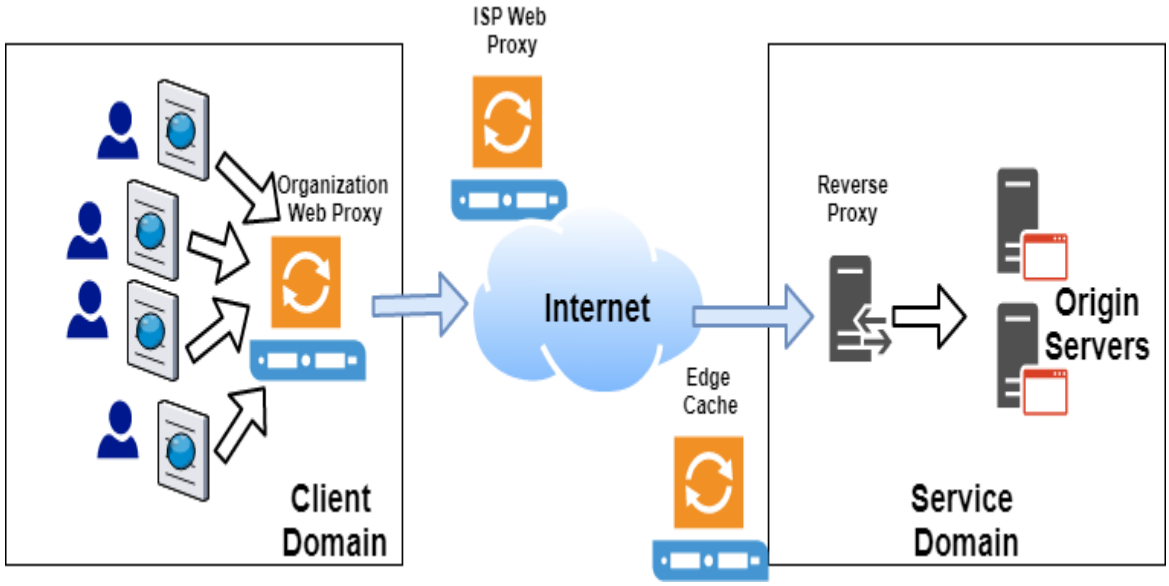

Figure 6-2. Web Caches in the Internet

Services can control what results are cached and for how long they are stored by using HTTP caching directives. Services set these directives in various HTTP response headers, as shown in the simple example in Figure 6-4. I will describe these directives in the following subsections.

Example 6-2. Figure 6-4 Example HTTP Response with caching directives

```
Response:
HTTP/1.1 200 OK Content-Length: 9842
Content-Type: application/json
Cache-Control: public
Date: Fri, 26 Mar 2019 09:33:49 GMT
Expires: Fri, 26 Mar 2019 09:38:49 GMT
```

Cache-Control

The `Cache-Control` HTTP header can be used by client requests and service responses to specify how the caching should be utilized for the resources of interest. Possible values are:

`no-store`

> Specifies that a resource from a request response should not be cached. This is typically used for sensitive data that needs to be retrieved from the origin servers each request.

no-cache

Specifies that a cached resource must be revalidated with an origin server before use. I discuss revalidation in the `Etag` subsection below.

Private

Specifies a resource can be cached only by a user-specific device such as a Web browser

Public

Specifies a resource can be cached by any proxy server

max-age

Defines the length of time in seconds a cached copy of a resource should be retained. After expiration, a cache must refresh the resource by sending a request to the origin server.

Expires and Last-Modified

The `Expires` and `Last-Modified` HTTP headers interact with the `max-age` directive to control how long cached data is retained.

Caches have limited storage resources and hence must periodically evict items from memory to create space. To influence cache eviction, services can specify how long resources in the cache should remain valid, or *fresh*. When a request arrives for a fresh resource, the cache serves the locally stored results without contacting the origin server. Once any specified retention period for a cached resource expires, it becomes stale and becomes a candidate for eviction.

Freshness is calculated using a combination of header values. The "`Cache-Control: max-age=N`" header is the primary directive, and this value specifies the freshness period in seconds.

If max-age is not specified, the `Expires` header is checked next. If this header exists, then it is used to calculate the freshness period. The `Expires` header specifies an explicit date and time after which the resource should be considered stale. For example:

```
Expires: Wed, 26 Oct 2022 09:39:00 GMT
```

As a last resort, the `Last-Modified` header can be used to calculate resource retention periods. This header is set by the origin server to specify when a resource was last updated, and uses the same format as the `Expires` header. A cache server can use `Last-Modified` to determine the freshness lifetime of a resource based on a heuristic calculation that the cache supports. The calculation uses the `Date` header, which specifies the time a response message was sent from an origin server. A resource retention period subsequently becomes equal to the value of the `Date` header minus the value of the `Last-Modified` header divided by 10.

Etag

HTTP provides another directive that can be used to control cache item freshness. This is known as an `Etag`. An `Etag` is an opaque value that can be used by a Web cache to check if a cached resource is still valid. I'll explain this using another winter sports example.

Going back to our winter resort example, the resort produces a weather report at 6am every day during the winter season. If the weather changes during the day, the resort updates the report. Sometimes this happens two or three times each day, and sometimes not at all if the weather is stable. When a request arrives for the weather report, the service responds with a maximum age to define cache freshness, and also an `Etag` that represents the version of the weather report that was last issued. This is shown in Figure 6-5, which tells a cache to treat the weather report resource as fresh for at least 3600 seconds, or 60 minutes. The `Etag` value, namely "`blackstone-weather-03/26/19-v1`", is simply generated using a label that the service defines for this particular resource. In this example,

the `Etag` represents the first version of the report for the Blackstone resort on the 26th March, 2019. Other common strategies are to generate the `Etag` using a hash algorithm such as MD5.

Example 6-3. Figure 6-5 HTTP Etag Example

```
Request:GET /skico.com/weather/Blackstone

Response:
HTTP/1.1 200 OK Content-Length: ...
Content-Type: application/json
Date: Fri, 26 Mar 2019 09:33:49 GMT
Cache-Control: public, max-age=3600
ETag: "blackstone-weather-03/26/19-v1"
<!-- Content omitted -->
```

For the next hour, the Web cache simply serves this cached weather report to all clients who issue a GET request. This means the origin servers are freed from processing these requests – the outcome that we want from effective caching. After an hour though, the resource becomes stale. Now, when a request arrives for a stale resource, the cache forwards it to the origin server with a `If-None-Match` directive along with the `Etag` to enquire if the resource, in our case the weather report, is still valid. This is known as revalidation.

There are two possible responses to this request.

- If the `Etag` in the request matches the value associated with the resource in the service, the cached value is still valid. The origin server can therefore return a `304 (Not Modified)` response, as shown in Figure 6-6. No response body is needed as the cached value is still current, thus saving bandwidth, especially for large resources. The response may also include new cache directives to update the freshness of the cached resource.
- The origin server may ignore the revalidation request and respond with a `200 OK` response code, a response body and `Etag` representing the latest version of the weather report.

Example 6-4. Figure 6-6 Validating an Etag

```
Request:
GET /upic.com/weather/Blackstone
If-None-Match: “blackstone-weather-03/26/19-v1"
Response:
HTTP/1.1 304 Not Modified
Cache-Control: public, max-age=3600
```

In the service implementation, a mechanism is needed to support revalidation. In our weather report example, one strategy is as follows:

1. Generate new daily report: The weather report is constructed and stored in a database, with the Etag as an attribute.
2. GET requests: When any GET request arrives, the service returns the weather report and the Etag. This will also populate Web caches along the network response path.
3. Conditional GET requests: For conditional requests with the If-None-Match: directive, lookup the Etag value in the database and return 304 if the value has not changed. If the stored Etag has changed, return 200 along with the latest weather report and a new Etag value.
4. Update weather report: A new version of the weather report is stored in the database and the Etag value is modified to represent this new version of the response.

When used effectively, Web caching can significantly reduce latencies and save network bandwidth. This is especially true for large items such as images and documents. Further, as Web caches handle requests rather than application services, this reduces the request load on origin servers, creating additional capacity.

Proxy caches such as Squid[9] and Varnish[10] are extensively deployed on the Internet. Web caching is most effective when deployed for static (images, videos and audio streams) and infrequently changing data such as weather reports. The powerful facilities provided by HTTP caching in conjunction with proxy and edge caches are therefore invaluable tools for building scalable applications.

Summary and Further Reading

Caching is an essential component of any scalable distribution. Caching stores information that is requested by many clients in memory and serves this information as the results to client requests. While the information is still valid, it can be served potentially millions of times without the cost of recreation.

Application caching using a distributed cache is the most common approach to caching in scalable systems. This approach requires the application logic to check for cached values when a client request arrives and return these if available. If the cache hit rate is high, with most requests being satisfied with cached results, the load on backend services and databases can be considerably reduced.

The Internet also has a built in, multilevel caching infrastructure. Applications can exploit this through the use of cache directives that are part of HTTP headers. These directives enable a service to specify what information can be cached, for how long it should be cached, and employ a protocol for checking to see if a stale cache entry is still valid. Used wisely, HTTP caching can significantly reduce request loads on downstream services and databases.

Caching is a well-established area of software and systems, and the literature tends to be scattered across many generic and product specific sources. A great source of *all-things-caching* is Gerardus Blokdyk's *Memcached Third Edition, 2018.* While the title gives away the product-focused content, the knowledge contained can be translated easily to cache designs with other competing technologies.

A great source of information on HTTP/2 in general is *Learning HTTP/2: A Practical Guide for Beginners 1st Edition, O'Reilly Media, 2017* by Stephen Ludin and Javier Garza. And while dated, W*eb Caching, O'Reilly Media, 2001*, by Duane Wessels contains enough generic wisdom to remain a very useful reference.

CDN's are a reasonably complex, vendor-specific topic in themselves. They come into their own for media-rich websites with a geographically dispersed group of users that require fast content delivery. For a highly readable overview, Ogi Djuraskovic site (https://firstsiteguide.com/cdn-guide/) is well worth checking out.

---

1 https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html

2 https://memcached.org/

3 https://redis.io/

4 Some application use cases may make it possible for a new cache entry to be created at the same time an update is made. This can be useful if some keys are 'hot' and will have a great likelihood of being accessed again before the next update. This is known as an eager cache update.

5 https://www.ehcache.org/documentation/3.3/caching-patterns.html#cache-aside

6 https://www.alachisoft.com/resources/docs/ncache/prog-guide/server-side-api-programming.html

7 https://aws.amazon.com/dynamodb/dax/

8 https://www.globaldots.com/resources/blog/content-delivery-network-explained/

9 http://www.squid-cache.org/

10 https://varnish-cache.org/

Chapter 7. Asynchronous Messaging

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Inevitably for a distributed systems book, I've spent a fair bit of time in the preceding chapters discussing communications issues. Communication is fundamental to distributed systems, and it is a major issue that architects need to incorporate into their system designs.

So far, these discussions have assumed a synchronous messaging style. A client sends a response and waits for a server to respond. This is how most distributed communications are designed to occur, as the client requires an instantaneous response to proceed.

Not all systems have this requirement. For example, when I return some goods I've purchased online, I take them to my local UPS or FedEx store. They scan my QRCode, and I give them the package to process. I do not then wait in the store for confirmation that the product has been successfully received by the vendor and my payment returned. That would be dull and unproductive. I trust the shipping service to deliver my

unwanted goods to the vendor and expect to get a message a few days later when it has been processed.

We can design our distributed systems to emulate this behavior. Using an asynchronous communications style, clients, known as producers, send their requests to an intermediary messaging service. This acts as a delivery mechanism to relay the request to the intended destination, known as the consumer, for processing. Producers *fire and forget* the requests they send. Once a request is delivered to the messaging service, the producer moves on to the next step in their logic, confident that the requests it sends will eventually get processed. This improves system responsiveness, in that producers do not have to wait until the request processing is completed.

In this chapter I'll describe the basic communication mechanisms that an asynchronous messaging system supports. I'll also discuss the inherent trade-offs between throughput and data safety – basically making sure your systems don't lose messages. I'll also cover three key messaging patterns that are commonly deployed in highly scalable distributed systems.

To make these concepts concrete, I'll describe RabbitMQ (https://www.rabbitmq.com/), a widely deployed open-source messaging system. After introducing the basics of the technology, I'll focus on the core set of features you need to be aware of in order to design a high throughput messaging system.

Introduction to Messaging

Asynchronous messaging platforms are a mature area of technology, with multiple products in the space.[1] The venerable IBM MQ Series appeared in 1993 and is still a mainstay of enterprise systems. The Java Messaging Service, an API level specification, is supported by multiple Java Enterprise Edition vendor implementations. RabbitMQ, which I'll use as an illustration later in this chapter, is arguably the most widely deployed open-source messaging system. In the messaging world, you will never be short of choice.

While the specific features and APIs vary across all these competing products, the foundational concepts are pretty much identical. I'll cover these in the following subsections, and then describe how they are implemented in RabbitMQ in the next section. Once you appreciate how one messaging platform works, it is relatively straightforward to understand the similarities and differences inherent in the competition.

Messaging Primitives

Conceptually, a messaging system comprises the following:

- **Message queues**: queues store a sequence of messages
- **Producers:** send messages to queues
- **Consumers**: retrieve messages from queues
- **Message broker:** manages one or more queues

This scheme is illustrated in Figure 7-1.

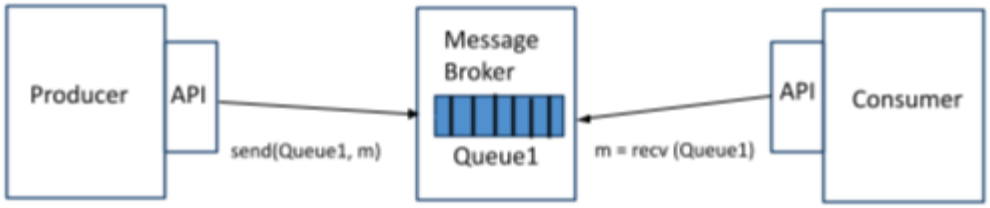

Figure 7-1. A Simple Messaging System

A message broker is a service that manages one or more queues. When messages are sent from producers to a queue, the broker adds messages to the queue in the order they arrive – basically a FIFO approach. The broker is responsible for efficiently managing message receipt and retention until one or more consumers retrieve the messages, which are then removed from the queue. Message brokers that manage many queues and many requests can effectively utilize many vCPUs and memory to provide low latency accesses.

Producers send messages to a named queue on a broker. Many producers can send messages to the same queue. A producer will wait until an acknowledgement message is received from the broker before the send operation is considered complete.

Many consumers can take messages from the same queue. Each message is retrieved by exactly one consumer. There are two modes of behavior for consumers to retrieve messages, known as *pull* or *push*. While the exact mechanisms are product-specific, the basic semantics are common across technologies.

- In pull mode, also known as polling, consumers send a request to the broker, which responds with the next message available for processing. If there are no messages available, the consumer must poll the queue until messages arrive.
- In push mode, a consumer informs the broker that it wishes to receive messages from a queue. The consumer provides a callback function that should be invoked when a message is available. The consumer then blocks (or does other work) and the message broker delivers messages to the callback function for processing when they are available.

Generally, utilizing the push mode when available is much more efficient and recommended. It avoids the broker being potentially swamped by requests from multiple consumers and makes it possible to implement message delivery more efficiently in the broker.

Consumers will also acknowledge message receipt. Upon consumer acknowledgement, the broker is free to mark a message as delivered and remove it from the queue. Acknowledgement may be done automatically or manually.

If automatic acknowledgement is used, messages are acknowledged as soon as they are delivered to the consumer, and before they are processed. This provides the lowest latency message delivery as the acknowledgement can be sent back to the broker before the message is processed.

Often a consumer will want to ensure a message is fully processed before acknowledgement. In this case it will utilize manual acknowledgements. This guards against the possibility of a message being delivered to a consumer but not being processed due to a consumer crash. It does of course increase message acknowledgement latency. Regardless of the acknowledgement mode selected, unacknowledged messages effectively remain on the queue and will be delivered at some later time to another consumer for processing.

Message Persistence

Message brokers can manage multiple queues on the same hardware. By default, message queues are typically memory based, in order to provide the fastest possible service to producers and consumers. Managing queues in memory has minimal overheads, as long as memory is plentiful. It does however risk message loss if the server were to crash.

To guard against message loss, known as data safety, queues can be configured to be persistent. When a message is placed on a queue by a producer, the operation does not complete until the message is written to disk. This scheme is depicted in Figure 7-2. Now, if a message broker should fail, on reboot it can recover the queue contents to the state they existed in before the failure, and no messages will be lost. Many applications can’t afford to lose messages, and hence persistent queues are necessary to provide data safety and fault tolerance.

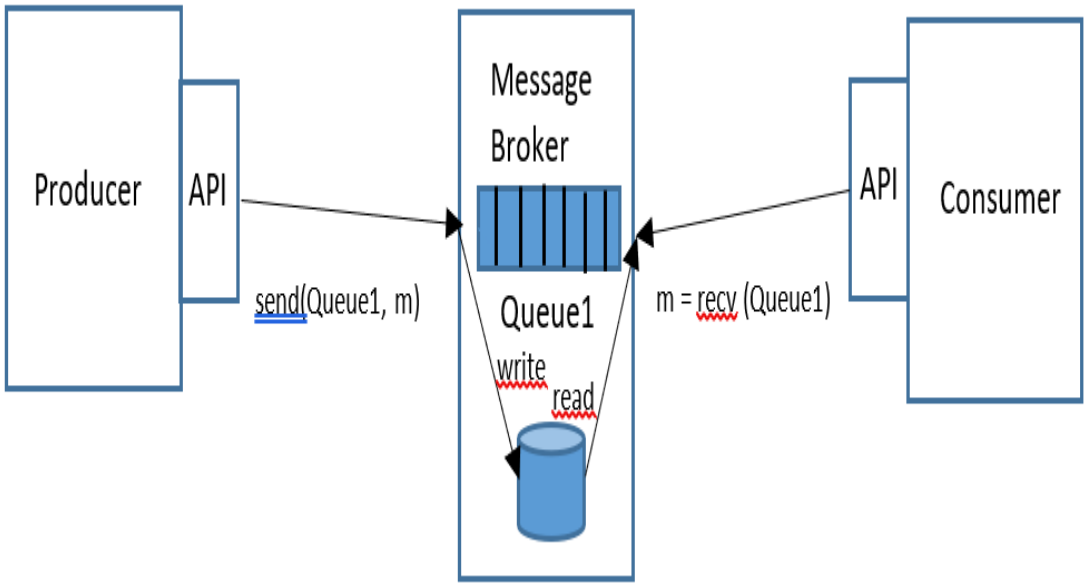

Figure 7-2. Persisting Messages to Disk

Persistent queues have an inherent increase in the response time for send operations, with the tradeoff being enhanced data safety. Brokers will usually maintain the queue contents in memory as well as on disk so messages can be delivered to consumers with minimal overhead during normal operations.

Publish-Subscribe

Message queues deliver each message to exactly one consumer. For many use cases, this is exactly what you want—my online purchase return needs to be consumed just once by the originating vendor—so that I get my money back.

Let's extend this use case. Assume the online retailer wants to do analysis of all purchase returns so it can detect vendors who have a high rate of returns and take some remedial action. To implement this, you could simply deliver all purchase return messages to the respective vendor *and* the new analysis service. This creates a one-to-many messaging requirement, which is known as a publish-subscribe architecture pattern. In publish-subscribe

systems, message queues are known as *topics*. A topic is basically a message queue that delivers each published message to one of more subscribers, as illustrated in Figure 7-3.

With publish-subscribe, you can create highly flexible and dynamic systems. Publishers are decoupled from subscribers, and the number of subscribers can vary dynamically. This makes the architecture highly extensible as new subscribers can be added without any changes to the existing system. It also makes it possible to perform message processing by a number of consumers in parallel, thus enhancing performance.

Publish-subscribe places an additional performance burden on the message broker. The broker is obliged to deliver each message to all active subscribers. As subscribers will inevitably process and acknowledge messages at different times, the broker needs to keep messages available until all subscribers have consumed each message. Utilizing a push model for message consumption provides the most efficient solution for publish-subscribe architectures.

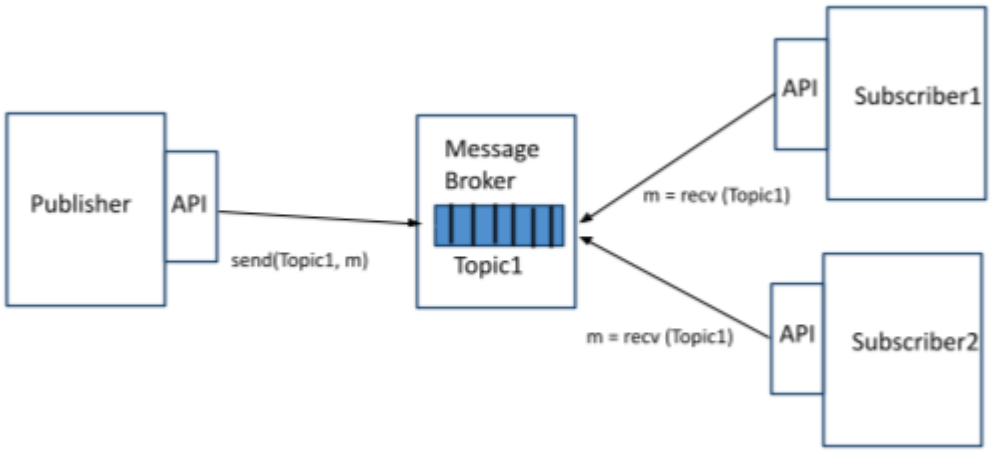

Figure 7-3. A Publish-Subscribe Broker Architecture

Publish-subscribe messaging is a key component for building distributed, event-driven architectures. In event-driven architectures, multiple services can publish events related to some state changes using message broker topics. Services can register interest in various event types by subscribing to

a topic. Each event published on the topic is then delivered to all interested consumer services. I'll return to event-driven architectures[2] when microservices are covered in Chapter 9.

Message Replication

In an asynchronous system, the message broker is potentially a single point of failure. A system or network failure can cause the broker to be unavailable, making it impossible for the systems to operate normally. This is rarely a desirable situation.

For this reason, most message brokers enable logical queues and topics to be physically replicated across multiple brokers, each running on their own node. If one broker fails, then producers and consumers can continue to process messages using one of the replicas. This architecture is illustrated in Figure 7-4. Messages published to the leader are mirrored to the follower, and messages consumed from the leader are removed from the follower.

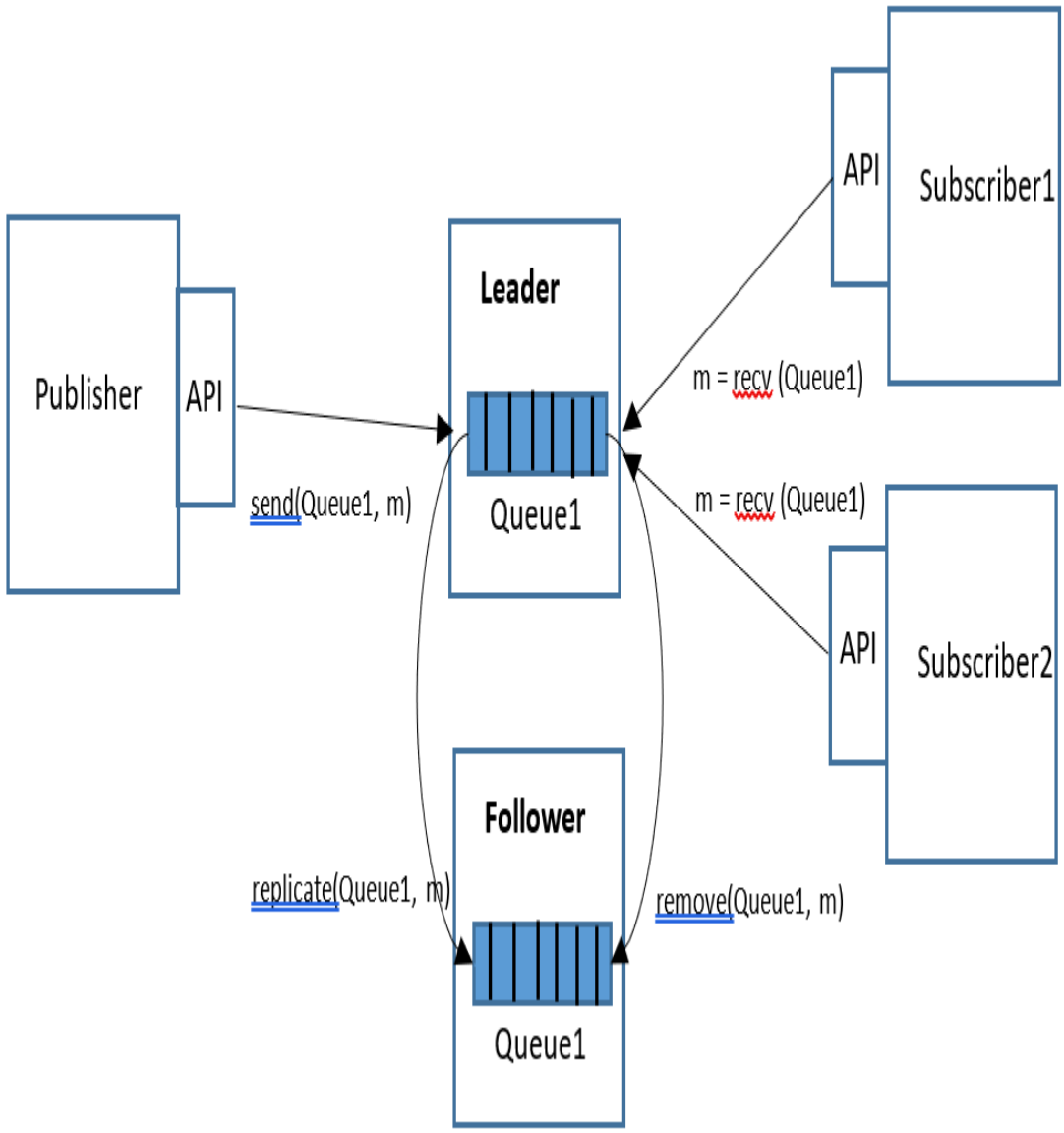

Figure 7-4. Message Queue Replication

The most common approach to message queue replication is known as a leader-follower architecture. One broker is designated as the leader, and producers and consumers send and receive messages respectively from this leader. In the background, the leader replicates, or mirrors all messages it receives to the follower, and removes messages that are successfully delivered. This is shown in Figure 7-4 with the *replicate* and *remove* operations. How precisely this scheme is implemented and the effects it has

on broker performance is inherently implementation, and hence product dependent.

With leader-follower message replication, the follower is known as a hot standby, basically a replica of the leader that is available if the leader fails. In such a failure scenario, producers and consumers can continue to operate by switching over to accessing the follower. This is also called *fail over*. Fail over is implemented in the client libraries for the message broker, and hence occurs transparently to producers and consumers.

Implementing a broker that performs queue replication is a complicated affair. There are numerous subtle failure cases that the broker needs to handle when duplicating messages. I'll start to raise these issues and describe some solutions in Chapters 10 and 11 when discussions turn to scalable data management.

> **NOTE**
>
> Some advice - don't contemplate *rolling your own* replication scheme, or any other complex distributed algorithm for that matter. The software world is littered with failed attempts to build application-specific distributed systems infrastructure, just because the solutions available 'don't do it quite right for our needs' or 'cost too much'. Trust me – your solution will not work as well as existing solutions and development will cost more than you could ever anticipate. And you will probably end up throwing your code away. These algorithms are really hard to implement correctly at scale.

Example: RabbitMQ

RabbitMQ is one of the most widely utilized message brokers in distributed systems. You'll encounter deployments in all application domains, from finance to telecommunications and building environment control systems. It was first released around 2009 and has developed into a fully featured, open source distributed message broker platform with support for building clients in most mainstream languages.

The RabbitMQ broker is built in Erlang, and primarily[3] provides support for the Advanced Message Queuing Protocol (AMQP) open standard.

AMQP emerged from the finance industry as a cooperative protocol definition effort. It is a binary protocol, providing interoperability between different products that implement the protocol. Out of the box, RabbitMQ supports AMQP v0-9-1, with v1.0 support via a plugin.

Messages, Exchanges, and Queues

In RabbitMQ, producers and consumers use a client API to send and receive messages from the broker. The broker provides the store-and-forward functionality for messages, which are processed in a FIFO manner using queues. The broker implements a messaging models based on a concept called exchanges, which provide a flexible mechanism for creating messaging topologies.

An exchange is an abstraction that receives messages from producers and delivers them to queues in the broker. Producers only ever write messages to an exchange. Messages contain a message payload, and various attributes known as message metadata. One element of this metadata is the *routing key*, which is a value used by the exchange to deliver messages to the intended queues.

Exchanges can be configured to deliver a message to one or more queues. The message delivery algorithm depends on the exchange type and rules called binding, which establish a relationship between an exchange and a queue using the routing key. The three most commonly used exchange types are shown in Table 7-1.

Table 7-1. Exchange Types

Exchange Type	Message Routing Behavior
	Delivers a message to a queue based on matching the value of a routing key

Direct	which is published with each message
Topic	Delivers a message to one or more queues based on matching the routing key and a pattern used to bind a queue to the exchange.
Fanout	Delivers a message to all queues that are bound to the exchange, and the routing key is ignored.

Direct exchanges are typically used to deliver each message to one destination queue based on matching the routing key[4]. Topic exchanges are a more flexible mechanism based on pattern matching that can be used to implement sophisticated publish-subscribe messaging topologies. Fanout exchanges provide a simple one-to-many broadcast mechanism, in which every message is sent to all attached queues.

Figure 7-5 depicts how a direct exchange operates. Queues are bound to the exchange by consumers with three values, namely “France”, “Spain” and “Portugal”. When a message arrives from a publisher, the exchange uses the attached routing key to deliver the message to one of the three attached queues.

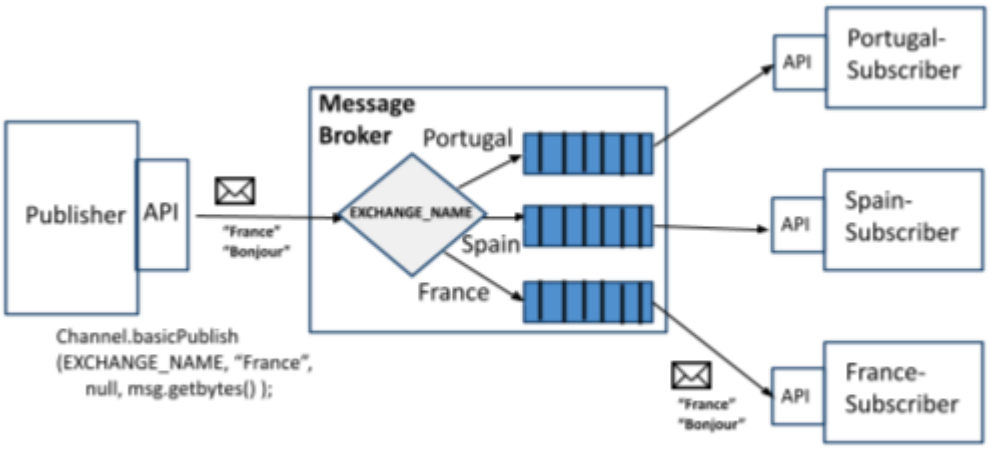

Figure 7-5. An Example of a RabbitMQ Direct Exchange

The following code shows an excerpt of how a direct exchange is configured and utilized in Java. RabbitMQ clients, namely producer and consumer processes, use a *channel* abstraction to establish communications with the broker (more on channels in the next section). The producer creates the exchange in the broker and publishes a message to the exchange with the routing key set to “France”. A consumer creates an anonymous queue in the broker, binds the queue to the exchange created by the publisher, and specifies that messages published with the routing key “France" should be delivered to this queue.

Producer:

```
channel.exchangeDeclare(EXCHANGE_NAME, "direct");
channel.basicPublish(EXCHANGE_NAME, “France”, null,
message.getBytes());
```

Consumer:

```
String queueName = channel.queueDeclare().getQueue();
channel.queueBind(queueName, EXCHANGE_NAME, “France”);
```

Distribution and Concurrency

To get the most from RabbitMQ in terms of performance and scalability, you must understand how the platform works under the covers. The issues

of concern relate to how clients and the broker communicate, and how threads are managed.

Each RabbitMQ client connects to a broker using a RabbitMQ connection. This is basically an abstraction on top of TCP/IP, and can be secured using user credentials or TLS. Creating connections is a heavyweight operation, requiring multiple round trips between the client and server, and hence a single long-lived connection per client is the common usage pattern.

To send or receive messages, clients use the connection to create a RabbitMQ channel. Channels are a logical connection between a client and the broker, and only exist in the context of a RabbitMQconnection, as shown in the following code snippet

```
ConnectionFactory connFactory = new ConnectionFactory();
Connection rmqConn = connFactory.createConnection();
Channel channel = rmqConn.createChannel();
```

Multiple channels can be created in the same client to establish multiple logical broker connections. All communications over these channels are multiplexed over the same RabbitMQ (TCP) connection, as shown in Figure 7-6. Creating a channel requires a network round trip to the broker. Hence for performance reasons, channels should ideally be long lived, with channel churn, namely constantly creating and destroying channels, avoided.

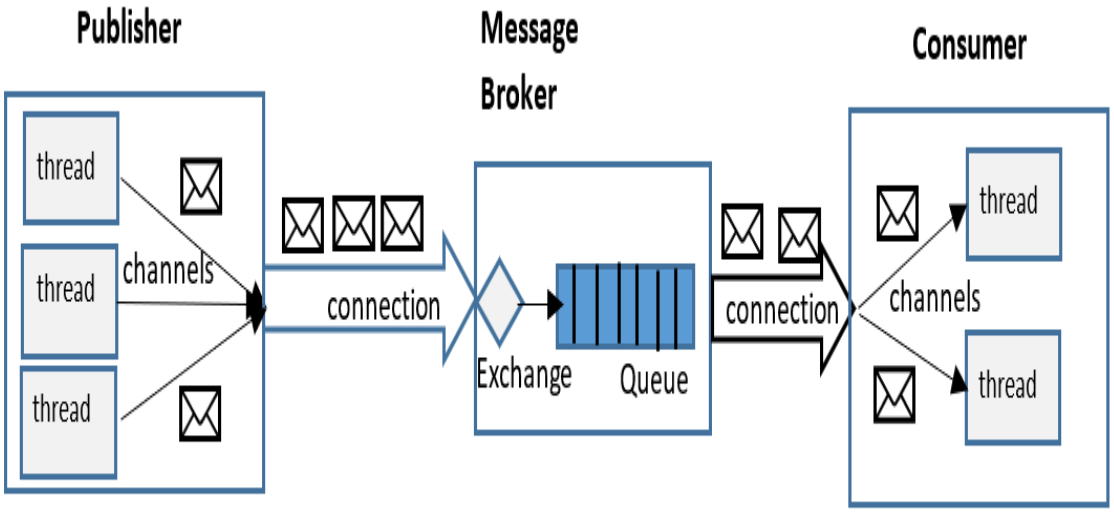

Figure 7-6. RabbitMQ Connections and Channels

To increase the throughput of RabbitMQ clients, a common strategy is to implement multithreaded producers and consumers. Channels, however, are not thread safe, meaning every thread requires exclusive access to a channel. This is not a concern if your client has long lived, stateful threads and can create a channel per thread, as shown in Figure 7-6. You start a thread, create a channel and publish or consume away. This is a channel-per-thread model.

In application servers such as Tomcat or Spring however, the solution is not so simple.The lifecycle and invocation of threads is controlled by the server platform, not your code. The solution is to create a global channel pool upon server initialization. This pre-created collection of channels can be used on demand by server threads without the overheads of channel creation and deletion per request. Each time a request arrives for processing, a server thread takes the following steps:

- Retrieves a channel from the pool
- Sends the message to the broker
- Returns the channel to pool for subsequent reuse

While there is no native RabbitMQ capability to do this, in Java you can utilize the Apache Commons Pool library[5] to implement a channel pool. The complete code for this implementation is included in the accompanying code repository for this book. The following code snippet shows how a server thread uses the `borrowObject()` and `returnObject()` methods of the Apache `GenericObjectPool`[6] class. You can tune the minimum and maximum size of this object pool using setter methods to provide the throughput your application desires.

```
private boolean sendMessageToQueue(JsonObject message) {
  try {
    Channel channel = pool.borrowObject();
    channel.basicPublish(// arguments omitted for brevity)
    pool.returnObject(channel);
      return true;
    } catch (Exception e) {
      logger.info("Failed to send message to RabbitMQ");
      return false;
    }
  }
```

On the consumer side, clients create channels that can be used to receive messages. Consumers can explicitly retrieve messages on demand from a queue using the `basicGet()` API, as shown in the following example:

```
boolean autoAck = true;
GetResponse response = channel.basicGet(queueName, autoAck);
if (response == null) {
    // No message available. Decide what to do …
} else {
    // process message
}
```

This approach uses the *pull* model, also known as polling. Polling is inefficient as it involves busy-waiting, obliging the consumer to continually ask for messages even if none are available. In high performance systems, this is not the approach to use.

The alternative and preferable method is the *push* model. The consumer specifies a callback function that is invoked for each message the RabbitMQ broker sends, or pushes, to the consumer. Consumers issue a call to the `basicConsume()` API. When a message is available for the

consumer from the queue, the RabbitMQ client library on the consumer invokes the callback in another thread associated with the channel. The following code example shows how to receive messages using an object of type `DefaultConsumer` that is passed to `basicConsume()` to establish a connection:

```
boolean autoAck = true;
channel.basicConsume(queueName, autoAck, "tag",
     new DefaultConsumer(channel) {
         @Override
         public void handleDelivery(String consumerTag,
                                    Envelope envelope,
                                    AMQP.BasicProperties
properties,
                                    byte[] body)
             throws IOException
         {
             // process the message
         }
     });
```

Reception of messages on a single channel is single threaded. This makes it necessary to create multiple threads and allocate a channel-per-thread or channel pool in order to obtain high message consumption rates. The following Java code extract shows how this can be done. Each thread creates and configures its own channel and specifies the callback function – `threadCallback()` – that should be called by the RabbitMQ client when a new message is delivered.

```
Runnable runnable = () -> {
      try {
        final Channel channel = connection.createChannel();
        channel.queueDeclare(QUEUE_NAME, true, false, false, null);
        // max one message per receiver

        final DeliverCallback threadCallback = (consumerTag,
delivery)
         -> {
             String message =
                 new String(delivery.getBody(),
StandardCharsets.UTF_8);
  // process the message
        };
        channel.basicConsume(QUEUE_NAME,
```

```
                                false, threadCallback, consumerTag ->
{});
          //
        } catch (IOException e) {
          logger.info(e.getMessage());
        }
```

Another important aspect of RabbitMQ to appreciate in order to obtain high performance and scalability is the thread model used by the message broker. In the broker, each queue is managed by a single thread. This means you can increase throughput on a multi-core node if you have at least as many queues as cores on the underlying node. And conversely, if you have many more highly utilized queues than cores on your broker node, you are likely to see some performance degradation.

Like most message brokers, RabbitMQ performs best when consumption rates keep up with production rates. When queues grow long, in the order of 10,000s of messages, the thread managing a queue will experience more overheads. By default, the broker will utilize 40% of the available memory of the node it is running on. When this limit is reached, the broker will start to throttle producers, slowing down the rate at which the broker accepts messages, until the memory usage drops below the 40% threshold. The memory threshold is configurable and again this is a setting that can be tuned to your workload to optimize message throughput.[7]

Data Safety and Performance Trade-offs

All messaging systems present a dilemma around a performance versus reliability trade-off. In this particular case, the core issue is the reliability of message delivery, commonly known as data safety. You want your messages to transit between producer and consumer with minimum latency, and of course you don’t want to lose any messages along the way. Ever. If only it were that simple. These are distributed systems, remember.

When a message transits from producer to consumer, there are multiple failure scenarios you have to understand and cater for in your design. These are:

- Producer sends a message to broker and message is not successfully accepted by the broker
- A message is in a queue and the broker crashes
- A message is successfully delivered to the consumer but the consumer fails before fully processing the message

If your application can tolerate message loss, then you can choose options that maximize performance. It probably doesn't matter if occasionally you lose a message from an instant messaging application. In this case your system can ignore message safety issues and run full throttle. This isn't the case for say a purchasing system. If purchase orders are lost, the business loses money and customers. You need to put safeguards in place to ensure data safety.

RabbitMQ, like basically all message brokers, has features that you can utilize to guarantee end-to-end message delivery. These are:

Publisher-confirms

A publisher can specify that it wishes to receive acknowledgements from the broker that a message has been successfully received. This is not default publisher behavior and must be set as a channel attribute by calling the `confirmSelect()` method. Publishers can wait for acknowledgements synchronously, or asynchronously by registering a callback function.

Persistent messages and message queues

If a message broker fails, all messages stored in memory for each queue are lost. To survive a broker crash, queues need to be configured as persistent, or durable. This means messages are written to disk as soon as they arrive from publishers. When a broker is restarted after a crash, it recovers all persistent queues and messages. In RabbitMQ, both queues and individual messages need to be configured as persistent to provide a high level of data safety.

Consumer manual acknowledgements

> A broker needs to know when it can consider a message successfully delivered to a consumer so it can remove the message from the queue. In RabbitMQ, this occurs either immediately after a message is written to a TCP socket, or when the broker receives an explicit client acknowledgement. These two modes are known as automatic and manual acknowledgements respectively. Automatic acknowledgements risk data safety as a connection or a consumer may fail before the consumer processes the message. For data safety, it is therefore important to utilize manual acknowledgements to make sure a message has been both received and processed before it is evicted from the queue.

In a nutshell, you need publisher acknowledgements, persistent queues and messages, and manual consumer acknowledgements for complete data safety. Your system will almost certainly take a performance hit, but you won't lose messages.

Availability and Performance Trade-offs

Another classic messaging system trade-off is between availability and performance. A single broker is a single point of failure, and hence the system will be unavailable if the broker crashes or experiences a transient network failure. The solution, as is typical for increasing availability, is broker and queue replication.

RabbitMQ provides two ways to support high availability, known as mirrored queues and quorum queues. While the details in implementation differ, the basics are the same, namely:

- Two or more RabbitMQ brokers need to be deployed and configured as a cluster.
- Each queue has a leader version, and one or more followers.

- Publishers send messages to the leader, and the leader takes responsibility for replicating each message to the followers.
- Consumers also connect to the leader, and when messages are successfully acknowledged at the leader, they are also removed from followers.
- As all publisher and consumer activity is processed by the leader, both quorum and mirrored queues enhance availability but do not support load balancing. Message throughput is limited by the performance possible for the leader replica.

There are numerous differences in the exact features supported by quorum and mirrored queues.[8] The key difference however revolves around how messages are replicated and how a new leader is selected in case of leader failure. Quorum in this context essentially means a majority. If there are 5 queue replicas, then at least 3 replicas – the leader and 2 followers - need to persist a newly published message. Quorum queues implement an algorithm known as RAFT to manage replication and electing a new leader when a leader becomes available. I'll discuss RAFT in some detail in Chapter 12.

Quorum queues must be persistent, and are therefore designed to be utilized in use cases when data safety and availability take priority over performance. They have other advantages over the mirrored queue implementation in terms of failure handling. For these reasons, the mirrored queue implementation will be deprecated in future versions.

Messaging Patterns

With a long history of usage in enterprise systems, a comprehensive catalog of design patterns[9] exist for applications that utilize messaging. While many of these are concerned with best design practices for ease of construction and modification of systems, and message security, a number apply directly to scalability in distributed systems. I'll explain three of the most commonly utilized patterns in this section.

Competing Consumers

A common requirement for messaging systems is to consume messages from a queue as quickly as possible. With the competing consumers[10] pattern, this is achieved by running multiple consumer threads and/or processes that concurrently processes messages. This enables an application to scale out message processing by horizontally scaling the consumers as needed. The general design is shown in Figure 7-7.

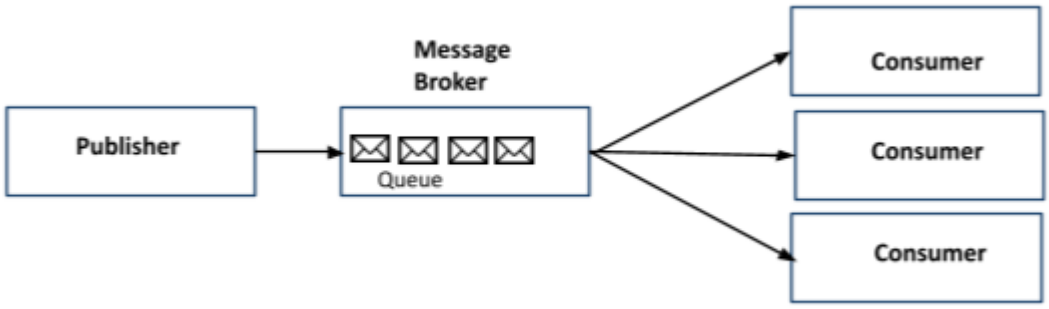

Figure 7-7. The Competing Consumers Pattern

Using this pattern, messages can be distributed across consumers dynamically using either the push or a pull model. Using the push approach, the broker is responsible for choosing a consumer to deliver a message to. A common method, which for example is implemented in RabbitMQ and ActiveMQ, is a simple round robin distribution algorithm. This ensures an even distribution of messages to consumers.

With the pull approach, consumers simply consume messages as quickly as they can process them. Assuming a multi-threaded consumer, if one consumer is running on an 8 core node and another on a 2 core node, we'd expect the former would process approximately 4 times the amount of messages of the latter. Hence load balancing occurs naturally with the pull approach.

There are three key advantages to this pattern, namely:

Availability

If one consumer fails, the system remains available, and its share of messages is simply distributed to the other competing consumers.

Failure handling

If a consumer fails, unacknowledged messages are delivered to another queue consumer.

Dynamic load balancing

New consumers can be started under periods of high load, and stopped when load is reduced, without the need to change any queue or consumer configurations.

Support for competing consumers will be found in any production-quality messaging platform. It is a powerful way to scale out message processing from a single queue.

Exactly-Once Processing

As I discussed in Chapter 3, transient network failures and delayed responses can cause a client to resend a message. This can potentially lead to duplicate messages being received by a server. To alleviate this issue, we need to put in place measures to ensure idempotent processing.

In asynchronous messaging systems, there are two sources for duplicate messages being processed. The first is duplicates from the publisher, and the second is consumers processing a message more than once. Both need to be addressed to ensure exactly once processing of every message.

The publisher part of the problem originates from a publisher retrying a message when it does not receive an acknowledgement from the message broker. If the original message was received and the acknowledgement lost or delayed, this may lead to duplicates on the queue. Fortunately, some message brokers provide support for this duplicate detection, and thus ensure duplicates do not get published to a queue. For example, the ActiveMQ Artemis release can remove duplicates[11] that are sent from the

publisher to the broker. The approach is based on the solution I described in Chapter 3, using client generated, unique *idempotent-key* values for each message. Publishers simply need to set a specific message property to a unique value, as shown in the following code:

```
ClientMessage msg = session.createMessage(true);
UUID idKey = UUID.randomUUID();  // use as idempotence key
msg.setStringProperty(HDR_DUPLICATE_DETECTION_ID, idKey.toString()
);
```

The broker utilizes a cache to store idempotent-key values and detect duplicates. This effectively eliminates duplicate messages from the queue, solving the first part of your problem.

On the consumer side, duplicates occur when the broker delivers a message to a consumer, which processes it and then fails to send an acknowledgement (consumer crashes or the network loses the acknowledgement). The broker therefore redelivers the message, potentially to a different consumer if the application utilizes the competing consumer pattern.

It's the obligation of consumers to guard against duplicate processing. Again, the mechanisms I described in Chapter 3, namely maintaining a cache or database of idempotent-keys for messages that have been processed. Most brokers will set a message header that indicates if a message is a redelivery. This can be used in the consumer implementation of idempotence. It doesn't guarantee a consumer has seen the message already. It just tells you that the broker delivered it and the message remains unacknowledged.

Poison Messages

Sometimes messages delivered to consumers can't be processed. There are numerous possible reasons for this. Probably most common are errors in producers that send messages that cannot be handled by consumers. This could be for reasons such as a malformed JSON payload or some unanticipated state change, for example a *StudentID* field in a message for a student who has just dropped out from the institution and is no longer active

in the database. Regardless of the reason, these *poison messages* have one of two effects:

- They cause the consumer to crash. This is probably most common in systems under development and test. Sometimes though these issues sneak into production, when failing consumers are sure to cause some serious operational headaches.
- They cause the consumer to reject the message as it is not able to successfully process the payload.

In either case, assuming consumer acknowledgements are required, the message remains on the queue in an unacknowledged state. After some broker-specific mechanism, typically a timeout or a negative acknowledgement, the poison message will be delivered to another consumer for processing - with predictable, undesirable results.

If poison messages are not somehow detected, they can be delivered indefinitely. This at best takes up processing capacity and hence reduces system throughput. At worst it can bring a system to its knees by crashing consumers every time a poison message is received.

The solution to poison message handling is to limit the number of times a message can be redelivered. When the redelivery limit is reached, the message is automatically moved to a queue where problematic requests are collected. This queue is traditionally and rather macabrely known as the *dead-letter queue*.

As you no doubt expect by now, the exact mechanism for implementing poison message handling varies across messaging platforms. For example, Amazon's Simple Queueing Service (SQS) defines a policy that specifies the dead-letter queue that is associated with an application-defined queue. The policy also states after how many redeliveries a message should be automatically moved from the application queue to the dead-letter queue. This value is known as the `maxReceiveCount`.

In SQS, each message has `a ReceiveCount` attribute, which is incremented when a message is not successfully processed by a consumer.

When the `ReceiveCount` exceeds the defined `maxReceiveCount` value for a queue, SQS moves the message to the dead-letter queue. Sensible values for redelivery vary with application characteristics, but a range of three to five is common.

The final part of poison message handling is diagnosing the cause for messages being redirected to the dead-letter queue. First, you need to set some form of monitoring alert that sends a notification to engineers that a message has failed processing. At that stage, diagnosis will comprise examining logs for exceptions that caused processing to fail and analyzing the message contents to identify producer or consumer issues.

Summary and Further Reading

Asynchronous messaging is an integral component of scalable system architectures. Messaging is particularly attractive in systems that experience peaks and troughs in request. During peak times, producers can add requests to queues and respond rapidly to clients, without having to wait for the requests to be processed.

Messaging decouples producers from consumers, making it possible to scale them independently. Architectures can take advantage of this by elastically scaling producers and consumers to match traffic patterns and balance message throughput requirements with costs. Message queues can be distributed across multiple brokers to scale message throughput. Queues can also be replicated to enhance availability.

Messaging is not without its dangers. Duplicates can be placed on queues, and messages can be lost if queues are maintained in memory. Deliveries to consumers can be lost, and a message can be consumed more than once if acknowledgements are lost. These data safety issues require attention to detail in design so that tolerance for duplicate messages and message loss is matched to the system requirements.

If you are interested in acquiring a broad and deep knowledge of messaging architectures and systems, the classic book *Enterprise Integration Patterns*

by Gregor Hohpe and Bobby Woolf (Addison-Wesley Professional) should be your first stop. Other excellent sources of knowledge tend to be messaging platform specific, and as there are a lot of competing platforms, there's a lot of books to choose from. My favorite RabbitMQ books for general messaging wisdom and RabbitMQ specific information are *RabbitMQ Essentials, 2nd Edition* by Lovisa Johansson and David Dosset (Packt) and *RabbitMQ In Depth* by Gavin Roy (Manning).

On a final note, the theme of asynchronous communications and the attendant advantages and problems will permeate the remainder of this book. Messaging is a key component of microservice-based architectures (Chapter 9) and is foundational to how distributed databases function. And you'll certainly recognize the topics of this chapter when I cover streaming systems and event-driven processing in Part 4.

---

1 Overview of messaging technologies landscape (https://www.g2.com/categories/message-queue-mq)

2 Chapter 14 of *Fundamentals of Software Architecture* by Mark Richards and Neal Ford is an excellent source of knowledge for Event-Driven architectures.

3 Other protocols such STOMP and MQTT are supported via plugins.

4 Consumers can call queueBind() multiple times to specify that their destination should receive messages for more than one routing key value. This approach can be used to create one-to-many message distribution. Topic exchanges are more powerful for one to many messaging.

5 Documentation for the Apache Commons library (https://commons.apache.org/proper/commons-pool/index.html)

6 Generic Object Pool documentation (https://javadoc.io/doc/org.apache.commons/commons-pool2/latest/org/apache/commons/pool2/impl/GenericObjectPool.html)

7 A complete description of how the RabbitMQ server memory can be configured (https://www.rabbitmq.com/memory.html)

8 Detailed Quorum Queues feature descriptions (https://www.rabbitmq.com/quorum-queues.html#feature-comparison)

9 Messaging patterns main Web site (https://www.enterpriseintegrationpatterns.com/patterns/messaging/)

10 An general overview of the Competing Consumers pattern (https://www.enterpriseintegrationpatterns.com/patterns/messaging/CompetingConsumers.html

)

11 A full description of how ActiveMQ nandles duplicates can be found at this link https://activemq.apache.org/components/artemis/documentation/latest/duplicate-detection.html

Chapter 8. Serverless Processing Systems

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at vwilson@oreilly.com.

Scalable systems experience widely varying patterns of usage. For some applications, load may be high during business hours and low or non-existent during non-business hours. Other applications, for example an online concert ticket sales system, might have low background traffic for 99% of the time. But when tickets for a major series of shows are released, the demand can spike by 10000 times of average load for a number of hours before dropping back down to normal levels.

Elastic load balancing, as described in Chapter 5, is one approach for handling these spikes. Another is serverless computing, which I'll examine in this chapter.

The Attractions of Serverless

The transition of major organizational IT systems from on-premise to public cloud platforms deployments seems inexorable. Organizations from startups to

government agencies to multinationals see clouds as digital transformation platforms and a foundational technology to improve business continuity.

Two of the great attractions of cloud platforms are their pay-as-you-go billing and ability to rapidly scale up (and down) virtual resources to meet fluctuating workloads and data volumes. This ability to scale of course doesn't come for free. Your applications need to be architected to leverage the scalable services[1] provided by cloud platforms. And of course, as I discussed in Chapter 1, cost and scale are indelibly connected. The more resources a system utilizes for extended periods, the larger your cloud bills will be at the end of the month.

Monthly cloud bills can be big. Really big. Even worse, unexpectedly big! Cases of 'sticker shock' for significant cloud overspend are rife – in one survey[2] 69% of respondents regularly overspent on their cloud budget by more than 25%. One well known case spent $500K on an Azure[3] task before it was noticed. Reasons attributed for overspending are many, including lack of deployment of auto-scaling solutions, poor long-term capacity planning, and inadequate exploitation of cloud architectures leading to bloated system footprints.

On a cloud platform, architects are confronted with a myriad of architectural decisions. These decisions are both broad, in terms of the overall architectural patterns or styles the systems adopts – for example microservices, n-tier, event driven – and narrow, specific to individual components and the cloud services that the system is built upon.

In this sense, architecturally significant decisions pervade all aspects of the system design and deployment on the cloud. And the collective consequences of all these decisions are highly apparent when you receive your monthly cloud spending bill.

Traditionally, cloud applications have been deployed on an Infrastructure-as-a-Service (IaaS) platform utilizing virtual machines (VMs). In this case, you pay for the resources you deploy regardless of how highly utilized they are. If load increases, elastic applications can spin up new virtual machines to increase capacity, typically using the cloud-provided load balancing service. Your costs are essentially proportional to the type of VMs you choose, the duration they are deployed for, and the amount of data the application stores and transmits.

Major cloud providers offer an alternative to explicitly provisioning virtual processing resources. Known as *serverless* platforms, they do not require any compute resources to be statically provisioned. Using technologies such as AWS Lambda or Google App Engine (GAE), the application code is loaded and executed on demand, when requests arrive. If there are no active requests, there are essentially no resources in use and no charges to meet.

Serverless platforms also manage autoscaling (up and down) for you. As simultaneous requests arrive, additional processing capacity is created to handle requests and, ideally, provide consistently low response times. When request loads drop, additional processing capacity is decommissioned, and no charges are incurred.

Every serverless platform varies in the details of its implementation. For example, a limited number of mainstream programming languages and application server frameworks are typically supported. Platforms provide multiple configuration settings that can be used to balance performance, scalability and costs. In general, costs are proportional to the following factors:

- The type of processing instance chosen to execute a request,
- The number of requests and processing duration for each request,
- How long each application server instance remains resident on the serverless infrastructure.

However the exact parameters used vary considerably across vendors. Every platform is proprietary and different in subtle ways. The devil lurks, as usual, in the details. So let's explore some of those devilish details specifically for the Google App Engine and AWS Lambda platforms

Google App *Engine*

Google App Engine (GAE) was the first offering from Google as part of what is now known as the Google Cloud Platform. It has been in general release since 2011, and enables developers to upload and execute HTTP-based application services on Google's managed cloud infrastructure.

The Basics

GAE supports developing applications in Go, Java, Python, Node.js, PHP, .NET, and Ruby. To build an application on GAE, developers can utilize common HTTP-based application frameworks that are built with the GAE runtime libraries provided by Google. For example, in Python, applications can utilize Flask, Django and web2py, and in Java the primary supported platform is servlets built on the Jetty JEE web container.

Application execution is managed dynamically by GAE, which launches compute resources to match request demand levels. Applications generally access a managed persistent storage platform such as Google's Firestore (https://cloud.google.com/firestore) or Google Cloud SQL (https://cloud.google.com/sql), or interact with a messaging service like Google's Cloud PubSub (https://cloud.google.com/pubsub).

GAE comes in two flavors, known as the standard environment and the flexible environment. The basic difference is that the standard environment is more closely managed by GAE, with development restrictions in terms of language versions supported. This tight management makes it possible to scale services rapidly in response to increased loads. In contrast, the flexible environment is essentially a tailored version of Google's Compute Engine service, which runs applications in Docker containers (https://www.docker.com/) on VMs. As its name suggests, it gives more options in terms of development capabilities that can be used, but is not as suitable for rapid scaling.

In the rest of this chapter, I'll focus on the highly scalable standard environment.

GAE Standard Environment

In the standard environment, developers upload their application code to a GAE project that is associated with a base project URL. This code must define HTTP endpoints that can be invoked by clients making requests to the URL. When a request is received, GAE will route it to a processing instance to execute the application code. These are known as resident instances for the application and are the major component of the cost incurred for utilizing GAE.

Each project configuration can specify a collection of parameters that control when GAE loads a new instance or invokes a resident instance. The two simplest settings control the minimum and maximum instances that GAE will have resident at any instant. The minimum can be zero, which is perfect for applications that have long periods of inactivity, as this incurs no costs.

When a request arrives and there are no resident instances, GAE dynamically loads an application instance and invokes the processing for the endpoint. Multiple simultaneous requests can be sent to the same instance, up to some configured limit (more on this when I discuss auto-scaling later in this chapter). GAE will then load additional instances on demand until the specified maximum instance value is reached. By setting the maximum, an application can put a lid on costs, albeit with the potential for increased latencies if load continues to grow.

Standard environment applications can be built in Go, Java, Python, Node.js, PHP, and Ruby. As GAE itself is responsible for loading the runtime environment for an application, it restricts the supported versions[4] to a small number per programming language. The language used also affects the time to load a new instance on GAE. For example, a lightweight runtime environment such as Go will start on a new instance in less than a second. In comparison, a more bulky Java Virtual Machine is of the order of 1-3 seconds on average. This load time is also influenced by the number of external libraries that the application incorporates.

Hence, while there is variability across languages, loading new instances is relatively fast. Much faster than booting a virtual machine anyway. This makes the standard environment extremely well suited for applications that experience rapid spikes in load. GAE is able to quickly add new resident instances as request volumes increase. Requests are dynamically routed to instances based on load, and hence assume a purely stateless application model to support effective load distribution. Subsequently, instances are released with little delay once the load drops, again reducing costs.

GAE's standard environment is an extremely powerful platform for scalable applications, and one I'll explore in more detail in the case study later in this chapter.

AutoScaling

Autoscaling is an option that you specify in an `app.yaml` file that is passed to GAE when you upload your server code. An autoscaled application is managed by GAE according to a collection of default parameter values, which you can override in your `app.yaml`. The basic scheme is shown in Figure 8-1.

GAE basically manages the number of deployed processing instances for an application based on incoming traffic load. If there are no incoming requests, then GAE will not schedule any instances. When a request arrives, GAE deploys an instance to process the request.

Deploying an instance can take anywhere between a few 100 milliseconds to a few seconds depending on the programming language you are using.[5] This means latency can be high for initial requests if there are no resident instances. To mitigate this instance loading latency effects, you can specify a minimum number of instances to keep available for processing requests. This of course costs money.

As the request load grows, the GAE scheduler will dynamically load more instances to handle requests. Three parameters control precisely how scaling operates, namely:

Target CPU Utilization

> Sets the CPU utilization threshold above which more instances will be started to handle traffic. The range is 0.5 (50%) to 0.95 (95%). The default is 0.6 (60%).

Maximum Concurrent Requests

> Sets the maximum number of concurrent requests an instance can accept before the scheduler spawns a new instance. The default value is 10, and the maximum is 80. The documentation[6] doesn't state the minimum allowed value, but presumably 1 would define a single-threaded service.

Target Throughput Utilization

> This is used in conjunction with the value specified for maximum concurrent requests to specify when a new instance is started. The range is 0.5 (50%) to 0.95 (95%). The default is 0.6 (60%). It works like this: when

the number of concurrent requests for an instance reaches a value equal to maximum concurrent requests value multiplied by the target throughput utilization, the scheduler tries to start a new instance.

Got that? As is hopefully apparent, these three settings interact with each other, making configuration somewhat complex. By default, an instance will handle 10 x 0.6 = 6 concurrent requests before a new instance is created. And if these 6 (or less) requests cause the CPU utilization for an instance to go over 60%, the scheduler will also try to create a new instance.

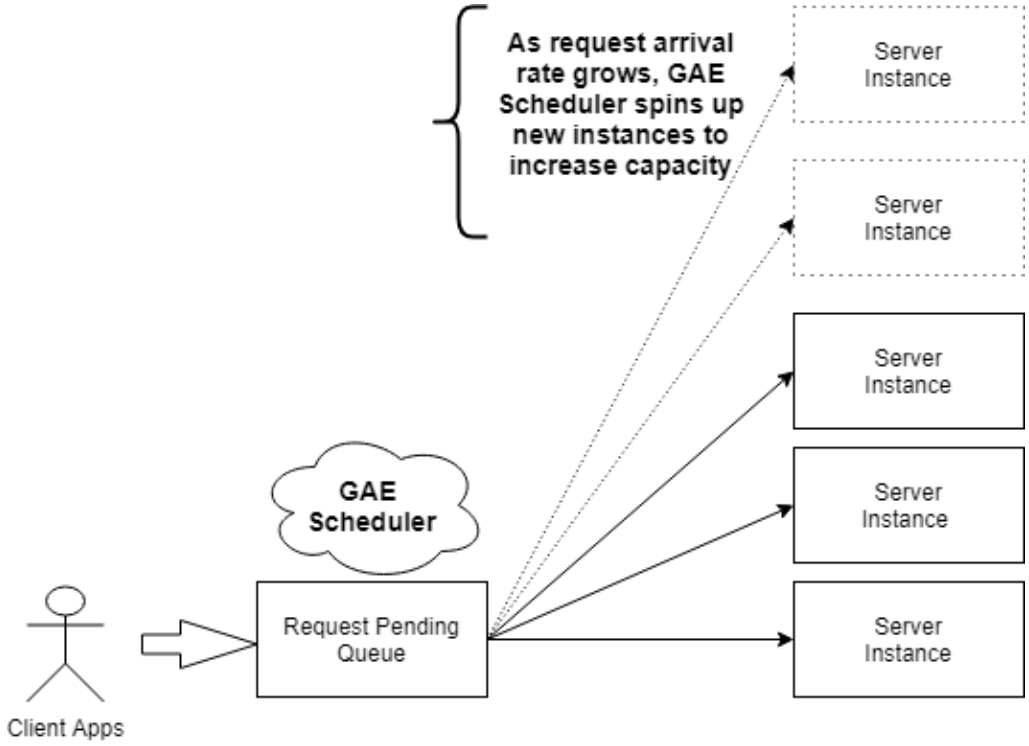

Figure 8-1. GAE Autoscaling

But wait, there's more!

You can also specify values to control when GAE adds new instances based on the time requests spend in the request pending queue - see Figure 8-1 - waiting to be dispatched to an instance for processing. The `max-pending-latency` parameter specifies the maximum amount of time that GAE should allow a request to wait in the pending queue before starting additional instances to handle requests and reduce latency. The default value is 30ms. The lower the

value, the quicker an application will scale. And the more it will probably cost you.[7]

These auto scaling parameter settings give us the ability to fine tune a service's behavior to balance performance and cost. How modifying these parameters will affect an application's behavior is of course dependent on the precise functionality of the service. The fact that there are subtle interplays between these parameters makes this tuning exercise somewhat complicated, however. I'll return to this topic in the case study section later in this chapter, and explain a simple, platform-agnostic approach you can take to service tuning.

AWS Lambda

AWS Lambda is Amazon's serverless platform. The underlying design principles and major features echo that of GAE and other serverless platforms. Developers upload code which is deployed as services known as Lambda functions. When invoked, Lambda supplies a language-specific execution environment to execute the function code.

A simple example of a Python Lambda function is shown in the following code. This function simply extracts a message from the input event and returns it unaltered as part of a HTTP 200 response. In general, you implement a function that takes an event and a context parameter. The event is a JSON-formatted document encapsulating data for a Lambda function to process. For example, if the Lambda function handles HTTP requests, the event will contain HTTP headers and the request body. The context contains metadata about the function and runtime environment, such as the function version number and available memory in the execution environment.

```
import json
def lambda_handler(event, context):
     event_body = json.loads(event['body'])
     response = {
        'statusCode': 200,
        'body': json.dumps({ event_body['message'] })
    }
    return response
```

Lambda functions can be invoked by external clients over HTTP. They can also be tightly integrated with other AWS services. For example, this enables

Lambda functions to be dynamically triggered when new data is written to the AWS S3 storage service or a monitoring event is sent to the AWS CloudWatch service. If your application is deeply embedded in the AWS ecosystem, Lambda functions can be of great utility in designing and deploying your architecture.

Given the core similarities between serverless platforms, in this section I'll just focus on the differentiating features of Lambda from a scalability and cost perspective.

Lambda Function Lifecycle

Lambda functions can be built in a number of languages and support common service containers such as Spring for Java and Flask for Python. For each supported language, namely Node.js, Python, Ruby, Java, Go and .NET-based code, Lambda supports a number of run time versions. The run time environment version is specified at deployment time along with the code, which is uploaded to Lambda in a compressed format.[8]

Lambda functions must be designed to be stateless so that the Lambda runtime environment can scale the service on demand. When a request first arrives for the API defined by the Lambda function, Lambda downloads the code for the function, initializes a runtime environment and any instance specific initialization (e.g. create a database connection), and finally invokes the function code handler.

This initial invocation is known as a cold start, and the time taken is dependent on the language environment selected, the size of the function code, and time taken to initialize the function. Like in GAE, lightweight languages such as Node.js and Go will typically take a few hundred milliseconds to initialize, whereas Java or .NET are heavier weight and can take a second or more.

Once an API execution is completed, Lambda can use the deployed function runtime environment for subsequent requests. This means cold start costs are not incurred. However, if a burst of requests arrive simultaneously, multiple runtime instances will be initialized, one for each request.[9] Unlike GAE, Lambda does not send multiple concurrent requests to the same runtime

instance. This means all these simultaneous requests will incur additional response times due to cold start costs.

If a new request does not arrive and a resident runtime instance is not immediately reutilized, Lambda *freezes* the execution environment. If subsequent requests arrive, the environment is *thawed* and reused. If more requests do not arrive for the function, after a platform-controlled number of minutes Lambda will deactivate a frozen instance so it does not continue to consume platform resources.[10]

Cold start costs can be mitigated by using *provisioned concurrency*. This tells Lambda to keep a minimum number of runtime instances resident and ready to process requests with no cold start overheads. The 'no free lunch' principle applies of course, and charges increase based on the number of provisioned instances. You can also make a Lambda function a target of an AWS Application Load Balancer, in a similar fashion to that discussed in Chapter 5. For example, a load balancer policy can be defined that increases the provisioned concurrency for a function at a specified time, in anticipation of an increase in traffic.

Execution Considerations

When you define a Lambda function, you specify the amount of memory that should be allocated to its runtime environment. Unlike GAE, you do not specify the number of vCPUs to utilize. Rather, the computation power is allocated in proportion to the memory specified, which is between 128MB and 10GB.

Lambda functions are charged for each millisecond of execution. The cost per millisecond grows with the amount of memory allocated to the runtime environment. For example, at the time of writing the costs per millisecond for a 2GB instance are double a 1GB instance.[11] Lambda does not specify precisely how much more compute capacity this additional memory buys your function, however. Still, the larger the amount of memory allocated, then the faster your Lambda functions will likely execute.[12]

This situation creates a subtle trade-off between performance and costs. Let's examine a simple example based on the costs for 1GB and 2GB instances

mentioned above, and assume that 1 millisecond of execution on a 1GB instance incurs 1 mythical cost unit, and a millisecond on a 2GB instance incurs 2 units.

With 1GB of memory, I'll assume this function executes in 40 milliseconds, thus incurring 40 cost units. With 2GB of memory allocated, and a commensurately more CPU allocation, the same function takes 10 milliseconds, meaning you part with 20 cost units from your AWS eWallet. Hence your bills will be reduced by 50% and you will get 4 times faster execution by allocating more memory to the function. Tuning can surely pay dividends.

This is obviously very dependent on the actual processing your Lambda function performs. Still, if your service is executed several billion times a month, this kind of somewhat non-intuitive tuning exercise may result in a significant cost savings and greater scalability.

Finding this 'sweet spot' that provides faster response times at similar or lower costs is a performance tuning experiment that can pay high dividends at scale. Lambda makes this a relatively straightforward experiment to perform as there is only one parameter, namely memory allocation, to vary. The case study later in this chapter will explain an approach that can be used for platforms such as GAE, which have multiple interdependent parameters that control scalability and costs.

Scalability

As the number of concurrent requests for a function increases, Lambda will deploy more runtime instances to scale the processing. If the request load continues to grow, Lambda reuses available instances and creates new instances as needed. Eventually, when the request load falls, Lambda scales down by stopping unused instances. That's the simple version anyway. In reality, it is a tad more complicated.

All Lambda functions have a built-in concurrency limit for request bursts. Interestingly, this default burst limit varies depending on the AWS region where the function is deployed. For example, in US West (Oregon), a function

can scale up to 3000 instances to handle a burst of requests, whereas in Europe (Frankfurt) the limit is 1000 instances.[13]

Regardless of the region, once the burst limit is reached, a function can scale at a rate of 500 instances per minute. This continues until the demand is satisfied and requests start to drop off. If the request load exceeds the capacity that can be processed by 500 additional instances per minute, Lambda throttles the function and returns a HTTP 429 to clients, who must retry the request.

This behavior is depicted in Figure 8-2. During the request burst, the number of instances grows rapidly up to the Region-defined burst limit. After that, only 500 new instances can be deployed per minute. During this time, requests that cannot be satisfied by the available instances are throttled. As the request load drops. Instances are removed from the platform until a steady state of traffic resumes.

Precisely how many concurrent client requests a function can handle depends on the processing time for the function. For example, assume we have 3000 deployed instances, and each request takes on average 100 milliseconds to process. This means that each instance can process 10 requests per second, giving a maximum throughput of:

(3000x10) = 30000 requests per second.

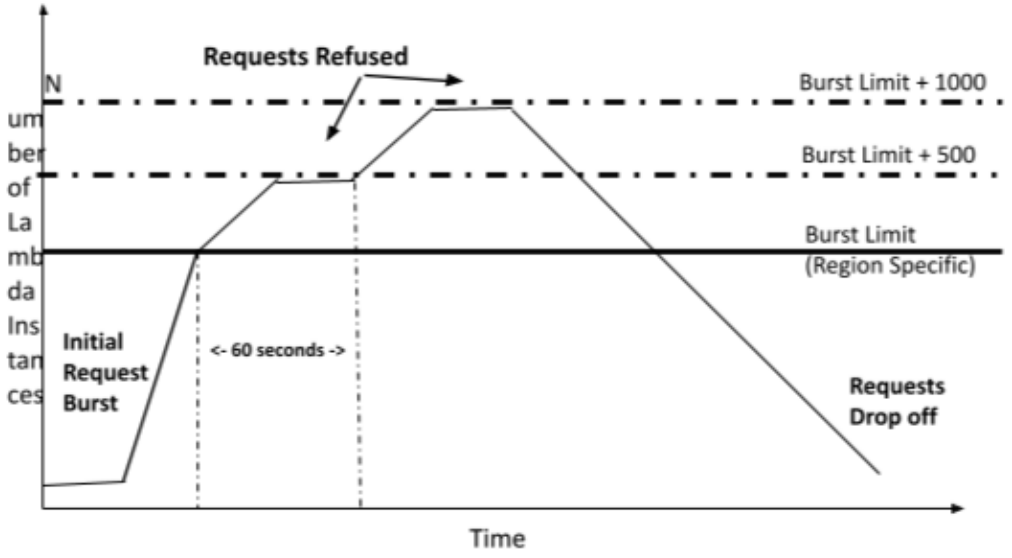

Figure 8-2. Scaling an AWS Lambda Function

To complete the picture, you need to be aware that the burst concurrency limit actually applies to all functions in the Region associated with a single AWS account. So, if you deploy 3 different Lambda functions in the same region under one account, their *collective number* of deployed instances is controlled by the burst limit that determines the scaling behavior. This means if one function is suddenly unexpectedly heavily loaded, it can consume the burst limit and negatively impact the availability of other functions that wish to scale at the same time.

To address this potential conflict, you can fine tune the concurrency levels associated with each individual Lambda function deployed under the same AWS account in the same Region.[14] This is known as *reserved concurrency*. Each individual function can be associated with a value that is less than the burst limit.[15] This value defines the maximum number of instances of that function that can be executed concurrently.

Reserved concurrency has two implications:

- The Lambda function with reserved concurrency always has execution capacity available exclusively for its own invocations. It cannot be unexpectedly starved by concurrent invocations of other functions in the Region.
- The reserved capacity caps the maximum number of resident instances for that function. Requests that cannot be processed when the number of instances is at the reserved value fail with a HTTP 429 error.

As should be apparent from this discussion, AWS Lambda provides a powerful and flexible serverless environment. With care, the runtime environment can be configured to scale effectively to handle high volume, bursty request loads. It has become an integral part of the AWS toolbox for many organizations internal and customer-facing applications.[16]

Case Study: Balancing Throughput and Costs

Getting the required performance and scalability at lowest cost from a serverless platform almost always requires tweaking of the runtime parameter settings. When your application is potentially processing many millions of

requests per day, even a 10% cost reduction can result in significant monetary savings. Certainly, enough to make your boss and clients happy.

All serverless platforms vary in the parameter settings you can tune. Some are relatively straightforward, such as AWS Lambda in which choosing the amount of memory for a function is the dominant tuning parameter. The other extreme is perhaps Azure Functions, which has multiple parameter settings and deployment limits that differ based on which of three hosting plans are selected.[17]

Google App Engine sits between these two, with a handful of parameters that govern autoscaling behavior. I'll use this as an example of how to approach application tuning.

Choosing Parameter Values

There are three main parameters that govern how GAE autoscales an application, as I explained earlier in this chapter. Table 8-1 lists these parameters along with possible values ranges.

Table 8-1. Google App Engine Autoscaling P

a
r
a
m
e
t
e
r
s

Parameter name	Minimum	Maximum	Default
target_throughput_utilization	0.5	0.95	0.6
target_cpu_utilization	0.5	0.95	0.6
max_concurrent_requests	1	80	10

Given these ranges, the question for a software architect is, simply, how do you choose the parameter values that provide the required performance and scalability at lowest cost? Probably the hardest part is figuring out where to start.

Even with three parameters, there is a large combination of possible settings that, potentially, interact with each other. How do you know that you have parameter settings that are serving both your users and your budgets as close to optimal as possible? There's some good general advice available,[18] but you are still left with the problem of choosing parameter values for your application.

For just the three parameters listed in Table 8-1, there are approximately 170K different configurations. You can't test all of them. If you put your engineering hat on, and just consider values in increments of 0.05 for throughput and cpu utilization, and increments of 10 for maximum concurrent requests, you still end up with around 648 possible configurations. That is totally impractical to explore, especially as we really don't know a priori how sensitive our service behavior is going to be to any parameter value setting. So, what can you do?

One way to approach tuning a system is to undertake a parameter study.[19] Also known as a parametric study, the approach comprises three basic steps:

1. nominate the parameters for evaluation
2. define the parameter ranges and discrete values within those ranges
3. analyze and compare the results of each parameter variation

To illustrate this approach, I'll lead you through an example based on the three parameters in Table 8-1. The aim is to find the parameter settings that give ideally the highest throughput at the lowest cost. The application under test was a GAE Go service that performs reads and writes to a Google FireStore database. The application logic was straightforward, basically performing 3 steps:

1. input parameter validation
2. database access
3. formatting and returning results

The ratio of write to read requests was 80%-20%, hence defining a write heavy workload. I also used a load tester that generated an uninterrupted stream of requests from 512 concurrent client threads at peak load, with short warm up and cool down phases of 128 client threads.

GAE Autoscaling Parameter Study Design

For a well defined parameter study, you need to:

- Choose the parameter ranges of interest.
- Within the defined ranges for each parameter, choose one or two intermediate values

For the example Go application with simple business logic and database access, intuition seems to point to the default GAE CPU utilization and concurrent request settings to be on the low side. Therefore, I chose these two parameters to vary, with the following values:

target_cpu_utilization: {0.6, 0.7. 0.8},

max_concurrent_requests: {10, 35, 60, 80}

This defines 12 different application configurations, as shown by the entries in Table 8-2.

Table 8-2. Parameter Study Selected V

a
l
u
e
s

Parameter name	max_concurrent_requests			
cpu_utilization				
0.6	10	35	60	80
0.7	10	35	60	80
0.8	10	35	60	80

The next step is to run load tests on each of the 12 configurations. This was straightforward and took a few hours over 2 days. Your load testing tool will capture various test statistics. In this example you are most interested in overall average throughput obtained and the cost of executing each test. The latter

should be straightforward to obtain from the serverless monitoring tools available

Now, I'll move on to the really interesting part – the results.

Results

Table 8-3 shows the mean throughput for each test configuration. The highest throughput of 6178 requests per second is provided by the (*CPU80, max10)* configuration. This value is 1.7% higher than that provided by the default settings (CPU60, max10), and around 9% higher than the lowest throughput of 5605 requests per second. So the results show a roughly 10% variation from lowest to highest throughput. Same code. Same request load. Different configuration parameters.

Table 8-3. Mean Throughput for reach test

tconfiguration

Throughput	max10	max35	max60	max80
CPU60	6006	6067	5860	5636
CPU70	6064	6121	5993	5793
CPU80	6178	5988	5989	5605

Now I'll factor in cost. In Table 8-4, I've normalized the cost for each test run by the cost of the default GAE configuration {CPU60, max10}. So for example, the cost of the {CPU70, max10} configuration was 18% higher than the default, and the cost of the {CPU80, max80} configuration was 45% lower than the default.

Table 8-4. Mean Cost for each test configuration

g
u
r
a
t
i
o
n

n
o
r
m
a
l
i
z
e
d

t
o

d
e
f
a
u
l
t
c
o
n
f
i
g
u

r
a
t
i
o
n

c
o
s
t

Normalized Instance Hours	max10	max35	max60	max80
CPU60	100%	72%	63%	63%
CPU70	118%	82%	63%	55%
CPU80	100%	72%	82%	55%

There are several rather interesting observations we can make from these results:

- The default settings {CPU60, max10} give neither the highest performance nor lowest cost. This configuration makes Google happy but maybe not your client.
- We obtain 3% higher performance with the {CPU80, max10} configuration at the same cost of the default configuration.
- We obtain marginally (approximately 2%) higher performance with 18% lower costs from the {CPU70, max35} configuration as compared to the default configuration settings.
- We obtain 96% of the default configuration performance at 55% of the costs with the {CPU70, max80} test configuration. That is a pretty decent cost saving for slightly lower throughput.

Armed with this information, you can choose the configuration settings that best balance your costs and performance needs. With multiple, dependent configuration parameters, you are unlikely to find the 'best' setting through intuition and expertise. There are too many intertwined factors at play for that to happen. Parameter studies let you quickly and rigorously explore a range of parameter settings. With two or three parameters and three or four values for each, you can explore the parameter space quickly and cheaply. This enables you to see the effects of the combinations of values and make educated decisions on how to deploy your application.

Summary and Further Reading

Serverless platforms are a powerful tool for building scalable applications. They eliminate many of the deployment complexities associated with managing and updating clusters of explicitly allocated virtual machines. Deployment is as simple as developing the service's code, and uploading it to the platform along with a configuration file. The serverless platform you are using takes care of the rest.

In theory anyway.

In practice of course, there are important dials and knobs that you can use to tune the way the underlying serverless platforms manage your functions. These are all platform specific, but many relate to performance and scalability, and ultimately the amount of money you pay. The case study in this chapter illustrated this relationship and provided you with an approach you can utilize to find that elusive 'sweet spot' that provides the required performance at lower costs than the default platform parameter settings provide.

Exploiting the benefits of serverless computing requires you to buy into a cloud service provider. There are many to choose from, but all come with the attendant vendor lock-in and downstream pain and suffering if you ever decide to migrate to a new platform.

There are open source serverless platforms such as Apache OpenWhisk (https://openwhisk.apache.org/) that can be deployed to on-premise hardware or cloud-provisioned virtual resources. There are also solutions such as the Serverless Framework (https://www.serverless.com/) that are provider-independent. These make it possible to deploy applications written in Serverless to a number of mainstream cloud providers, including all the usual suspects. This delivers code portability but does not insulate the system from the complexities of different provider deployment environments. Inevitably, achieving the required performance, scalability and security on a new platform is not going to be a walk in the park.

A great source of information on serverless computing is Jason Katzer's *Learning Serverless* (2020, O'Reilly Media). I'd also recommend two extremely interesting articles that discuss the current state of the art and future possibilities for serverless computing. These are:

- D. Taibi, J. Spillner and K. Wawruch, "Serverless Computing-Where Are We Now, and Where Are We Heading?," in IEEE Software, vol. 38, no. 1, pp. 25-31, Jan.-Feb. 2021, doi: 10.1109/MS.2020.3028708.
- J. Schleier-Smith et al, 2021. What serverless computing is and should become: the next phase of cloud computing. Commun. ACM 64, 5 (May 2021), 76–84.

Finally, serverless platforms are a common technology for implementing *microservices architectures*. Microservices are an architectural pattern for

decomposing an application into multiple independently deployable and scalable ‘parts’. This design approach is highly amenable to a serverless-based implementation, and conveniently, are the topic we cover in the next chapter.

---

1 A brief overview of scalable services can be found at this link https://medium.com/@i.gorton/six-rules-of-thumb-for-scaling-software-architectures-a831960414f9

2 A survey of cloud bills is an interesting read - https://www.prnewswire.com/news-releases/survey-enterprises-overspending-by-millions-on-cloud-bills-300968100.html

3 An example of unexpected cloud bills is at https://www.infotech.com/software-reviews/research/unexpected-cloud-costs-are-rocking-the-enterprise

4 This link details the supported runtime environments - https://cloud.google.com/appengine/docs/the-appengine-environments

5 A GAE language comparison illustrates the load times - see https://medium.com/dev-genius/scalability-and-cost-analysis-for-cloud-based-software-systems-part-1-472012435b26

6 See this link for the documentation - https://cloud.google.com/appengine/docs/standard/java/config/appref#automatic_scaling_max_concurrent_requests

7 There’s also an optional min-pending-latency parameter, with a default value of zero. If you are brave, how the minimum and maximum values work together is explained here. https://cloud.google.com/appengine/docs/standard/python/config/appref#scaling_elements

8 As of 2021, Lambda also supports services that are built using Docker containers. This gives the developer the scope to choose language runtime when creating the container image.

9 A good description of cold start behavioris at this link https://medium.com/hackernoon/im-afraid-you-re-thinking-about-aws-lambda-cold-starts-all-wrong-7d907f278a4f

10 This experiment describes how long idle function are kept resident - https://acloudguru.com/blog/engineering/how-long-does-aws-lambda-keep-your-idle-functions-around-before-a-cold-start?utm_source=medium_blog&utm_medium=redirect&utm_campaign=medium_blog

11 https://aws.amazon.com/lambda/pricing/

12 “At 1,769 MB, a function has the equivalent of one vCPU (one vCPU-second of credits per second).” from https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html

13 Established customers can negotiate with AWS to increase these limits.

14 Alternatively, if the lambda usage is across different applications, it could be separated into different accounts. AWS account design and usage is however outside the scope of this book.

15 Actually, this maximum reserved concurrency for a function is the (Burst Limit -100). AWS reserves 100 concurrent instances for all functions that are not associated with explicit concurrency limits. This ensures that all functions have access to some spare capacity to execute.

16 See https://serverlessfirst.com/real-world-serverless-case-studies/ for an interesting set of curated case studies from Lambda users.

17 Scaling Azure functions is covered in this link - https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale

18 https://www.youtube.com/watch?v=eUXUY7QFfAI

19 This is link is a good overview of parameter studies - http://wiki.analytica.com/Parametric_analysis

About the Author

Ian Gorton has 30 years experience as a software architect, author, computer science professor and consultant. He has focused on distributed technologies since his days in graduate school, and has worked on large scale software systems in areas such as banking, telecommunications, government, health care and scientific modeling and simulation. During this time, he has seen software systems evolve to the massive scale they routinely operate at today.

Ian has written 3 books, including *Essential Software Architecture* and *Data Intensive Computing*, and is the author of 200+ scientific and professional publications on software architecture and software engineering. At the Carnegie Mellon Software Engineering Institute he led R&D projects in big data and massively scalable systems, and has continued working, writing and speaking on these topics since joining Northeastern University as a Professor of Computer Science in 2015. He has a PhD from Sheffield Hallam University, UK and is a Senior Member of the IEEE Computer Society.