Deployment and Operations for Software Engineers

Second Edition

A DevOps Engineering Textbook

Deployment and Operations for Software Engineers

Second Edition

A DevOps Engineering Textbook

Len Bass and John Klein

Cover photo credit to Tanya Bass

ISBN: 9798801454825

Third printing, January 2023

Acknowledgements

We would like to thank Hasar Yasar for his suggestions and insights.

Contents

Preface to 2nd Edition

The first edition of this book has been used as a textbook for a course in DevOps engineering targeted at first year graduate students. This use exposed several flaws in the first edition. One significant flaw was that there was no overall discussion of DevOps and the problems that DevOps is intended to solve. Without this understanding, the motivation for some of the DevOps processes is more difficult to understand. The 2nd edition adds a chapter called “What is DevOps?’ that puts the technical activities in context and includes a comparison of DevOps processes to Agile processes.

A second flaw is the unevenness of background of the students. Questions about how VMs boot, what is the difference between stateful and stateless computation are common. The 2d edition has two introductory chapters – one for the platform and one for DevOps where material known to some students and not to others is located. As a student, you should look at the topics in these introductory chapters and decide whether they can be skipped. As an instructor, you will have to judge the background of your students to decide which of this material you should cover.

New material in this edition includes a broader treatment of DevOps including DevSecOps, service meshes, chaos engineering, and a better treatment of version skew. The discussion of Kubernetes for orchestration and the use of a service mesh are much improved. Measurement is treated separately in Part I. Not only logging and metrics but tracing and spans are introduced as end-to-end measurement methods.

Part II introduces DevOps and includes a new chapter on the future as well as extending the material in the first edition. Vault is described as a method for managing credentials. The discussion on postproduction has been extended to include site reliability engineering.

Writing a second edition necessitates re-reading the first edition and several terminological inconsistencies were discovered. These, hopefully, have been removed or clarified. A glossary has been added to the beginning of each chapter that gives fundamental terms and their definitions. This should reduce the number of terminological problems the students have.

The first edition had many virtues as well as the defects mentioned. It was focused on DevOps engineering and the associated concepts rather than being tool specific. DevOps tools were treated as a class and their principles emphasized. Hopefully, the 2nd edition retains those virtues.

The instructor’s slides for this book can be found at **https://github.com/len-bass/dosebook**

Preface to the first edition

Three recent trends are dramatically extending the responsibilities of software engineers and, hence, the requisite skill set. The widespread adoption of the cloud is one of these trends. Software engineers need to be familiar with virtualization both of hardware and operating systems, with how networks work, and with managing failure in the cloud, scaling, and load balancing.

Driven by the adoption of the cloud, release cycles have shortened dramatically. It is no longer sufficient to release once a quarter. Monthly, weekly, daily, and hourly releases are common with the release time dependent more on business considerations than on technical ones. Short release cycles are supported by the migration to microservices, the proliferation of tools for a deployment pipeline, and by the growth of container orchestration tools. Yet rapid release cycles also depend on architectural decisions and disciplined use of the available tools.

Today, a software engineer is often expected to be responsible for their portion of the system during creation, deployment, and execution. This, together with the architecture of the cloud, requires an engineer to understand operations concepts such as business continuity and incident handling. They also must understand logging, monitoring, and alerts.

A software engineer, typically, learns this material on the job, requiring much work going through blogs, tutorials, and system documentation. This is a difficult and time-consuming process. Some organizations have training courses for new hires where an organization’s specific processes and tools are covered. Usually, however, these courses do not provide a broad context for the processes and tools.

This book is aimed at software engineers (and prospective software engineers) who are juniors/seniors or first-year graduate students in computer science or software engineering. It provides the context within which modern engineers work and the skills necessary to operate within that context.

This book also supports the teaching of this material by an instructor. Each chapter discusses the theory associated with its topics and then has hands-on exercises to apply the theory. There are also discussion questions that can be answered in a classroom setting or in discussion groups to enable the students to gain a better understanding of the theory and its implications. The modular nature of the presentation of the material makes it easy to skip sections that cover familiar material.

Our goal in this book, admittedly ambitious, is to distill the knowledge required of modern software engineers beyond programming languages and a three-tiered architecture. We cover six major topics: Virtualization and the cloud, networks and distribution, microservice architecture, deployment, security, and operations including business continuity and monitoring. We make no claim that the reader of this book will emerge as an expert in any of these areas, but they will have a basic understanding of the topics and some hands-on exercises to cement this understanding. This provides the basis for an engineer to become productive in an organizational setting more quickly.

Since the beginning of computer science education, there has been a debate about the extent to which a student should understand what is below a given level of abstraction. Higher level languages mask machine language–how much does the student need to know about machine language? Optimizing compilers mask certain reorganizations of computation–how much does the student need to know about these reorganizations? Java masks memory allocation–how much does the student need to know about memory allocation? And so on. In almost every case, the answer has been some, but not too much.

This is the view we are taking in this book. The software engineer needs to know something about virtualization, networking, the cloud, and the other topics we cover, but they do not need to know the gritty details. Computer science and software engineering curricula have courses in distributed networks (the cloud), security, and software architecture. These courses go into more detail than the typical software engineer needs. This leaves a gap that we are attempting to fill with this book. Enough knowledge of these topics for the software engineer but not so much knowledge that an engineer is overwhelmed.

Introduction to Part 1: The Platform Perspective

You have just graduated and started a new job and been given your first assignment. Your company is running a promotion and you will write a service that calculates the discount based on a set of rules involving other customer purchases and the organization’s loyalty program. You breathe a sigh of relief; it doesn’t look too hard, maybe a little complicated to understand the discount rules, but certainly manageable. Then your manager says: “You should probably set the scaling rules at 5 seconds, allocate a new VM at 80% utilization, and keep your service stateless. Also, when you containerize your service, use our private Docker repository and not the public Docker Hub. After you generate SSH keys, send your public key to Mary and she’ll put it on our DMZ servers. I’ve got to run–over the weekend, we had a spike in IDS alarms that I need to review.”

Huh? If these instructions are bewildering, then Part 1 is for you. Part 1 describes the cloud as a platform for development and focuses on those things that are out of your control as a developer.

Today’s developers need to understand much more than programming languages and business rules. The widespread adoption of cloud computing means that your organization is probably utilizing a large, distributed network for its basic computing platform. This platform can be public or private, but in either case a developer needs to understand enough about virtualization and distributed computing to operate effectively in this environment.

We expect the readers of this book to have different backgrounds. Chapter 1 Platform Preliminaries covers some material that many of you will be familiar with but that some of you will not. You should scan the topics in Chapter 1 to see whether you should read those sections in detail.

A cloud data center has tens of thousands of computers communicating over a network and is accessible externally via the internet. A system running in the cloud ideally acquires the resources that it needs (CPU, network, disk) for only as long as they are needed. These resources–virtual machines or containers–run on physical machines. They are either allocated automatically and dynamically by the system being or reserved, in advance, by the system developer. In Chapter 2 Virtualization we introduce virtual machines. Hypervisors allow virtual machines to share physical hardware. More recently, containers that virtualize the operating system have been in widespread use. We introduce containers in Chapter 2 Virtualization and discuss them in more detail in Chapter 5 Container Orchestration.

The physical and virtual machines in the cloud communicate over networks–both internal to the cloud and external using the internet. Communication over networks transmits data using messages. Two concerns immediately arise when using messages as the communication mechanism–how the messages are delivered to the correct recipient and what protocol is used to deliver them. By “correct recipient,” we mean not only the virtual machine intended as the target but also the service within that machine. Furthermore, networks have structure and sub-structure. Chapter 3 Networking is a discussion of networks.

With virtualization and networks as building blocks, Chapter 4 The Cloud discusses the cloud. We will see how the cloud allocates resources. When tens of thousands of computers are involved, it is inevitable that some will fail, and handling failures permeates the design of services that run on the cloud. In Chapter 4 The Cloud we introduce the possibility of failure, which will be elaborated in future chapters. As workloads change, virtual machines become overloaded; however, the cloud provides technology to handle overload by distributing the load over multiple instances of a virtual machine. These instances can be allocated dynamically: load balancers and autoscaling manage the creation and destruction of instances. Multiple service instances and multiple clients introduce the problems of state management and distributed coordination. We discuss the motivation for distributed systems and explain how the complications of distributed coordination systems are hidden behind understandable interfaces.

In Chapter 5 Container Orchestration we return to containers. Containers are managed by different mechanisms than virtual machines, although there is a relation. Container images can be stored in a repository, container instances can be scaled similarly to virtual machines, and containers are used in *serverless architecture*. All these topics are discussed in this chapter.

Systems running in the cloud must have observable characteristics. Chapter 6 Measurement discusses the types of measurements that can be made from a running system and how those measurements are made available to an obsesrver.

Finally, in Part 1 we discuss the basic concepts of security. In today’s environment, security is a concern of every software engineer. Some elements of securing systems are provided by the platform and the history of the internet. Chapter 7 Infrastructure Security discusses these elements. We provide a brief overview of cryptography and present several protocols and systems that use elements of cryptography to create robust, secure solutions. We finish the chapter with a discussion of technology used to detect cyberattacks against your virtual computers and your networks.

Chapter 1 Platform Preliminaries

This chapter introduces several concepts that are important for the remainder of the book. These concepts are in three categories: those specific to a single physical computer such as your laptop, those related to distributed computing, and those quality attributes that dominate the discussion of distributed computing platforms.

When you finish this chapter you will know

- How the boot process for a computer works
- How computer resources are isolated and shared
- What discovery, scaling, message communication patterns, and failure recovery mean in a distributed computing system
- Definitions of performance and availability.

1.2 Coming to Terms

Availability– a property of software that it is there and ready to carry out its task when you need it to be.

BIOS,(Basic Input/Output System,)—a computer program that is used by the CPU to perform start-up procedures when the computer is turned on.

Communication pattern- the sequence of computers through which a message passes to reach its recipient.

Component – a unit of software that can be deployed independently.

Discovery – a process that provides the mapping between a name and its network address.

Message – A collection of information passed over a network from one component to another.

Operating System(OS)–system software that manages computer hardware, software resources, and provides common services for computer programs.

Performance – a component's ability to meet timing requirements – either latency or throughput.

Scaling – creating additional instances of a component to enhance either performance or availability.

1.2 Basic Computer Concepts

This section will be new for some and a review for others. We discuss how the bits in a computer can be interpreted and the meaning of stateful and stateless. We also discuss the resources in a computer and how they are shared.

1.2.1 Bits are Bits

Consider the hexadecimal number “41FC3456”. Is this an address, an instruction, or a data item? The answer is “yes” and the use depends on context. If this number is fetched by the processor, then it is interpreted as an instruction. This number can also be the address of a location in memory. Finally, this number can be a data item that is read or written to disk or a memory location.

This flexibility means that, for example, a value can be read into memory as data and be used by the Central Processing Unit (CPU) as its next instruction. This sequence – read data and interpret it as instructions – is the basis for using virtual machine images which we will discuss in Chapter 2 Virtualization. It is also the basis for the boot process. During boot, a set of bits is read from a known location—persistent memory, a disk or network—and then control is transferred to that set of bits. This boot program—Basic Input Output System (BIOS)– then loads other software that enables your computer to execute as you expect. See Figure 1.1 for a representation of this process.

1.2.3 Stateful and Stateless

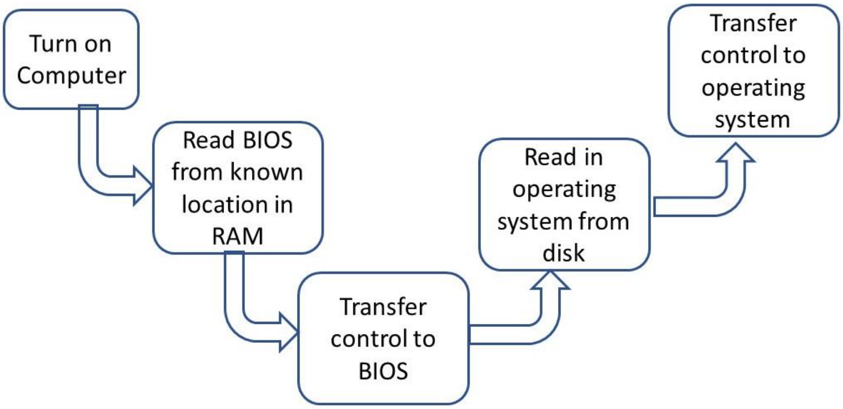
For a system to execute it needs both instructions and data (state). A service can be either stateful – maintaining state from one invocation to the next – or stateless – no state is maintained from one invocation to another. Stateless services must have state to execute, and this state is provided either through input parameters or retrieved from a repository.

1.2.3 Computer resources

There are four basic hardware resources in a computer:

1. Central processing unit. Modern computers have multiple central processing units, or CPUs (and each CPU can have multiple processing cores), and may have one or more graphics processing units (GPUs) or other special-purpose processors, such as a tensor processing unit (TPU).

Figure 1.1: The boot process


Turn on Computer
Read BIOS from known location in RAM
Transfer control to BIOS
Read in operating system from disk
Transfer control to operating system

2. Memory. The physical computer has a fixed amount of physical memory. The contents of memory hould be assumed to be destroyed when a reboot occurs.

3. Disk. Disks provide persistent storage for instructions and data, across reboots and shutdowns of the computer. A physical computer has one or more attached disks, each with a fixed amount of storage capacity. Disks can be either rotating or solid state.

4. Network connection. Networks were a later addition to the set of resources of a computer, emerging in the 1970s. Today, every non-trivial physical computer has one or more network connections through which all messages pass. These include both outbound messages and inbound messages.

These resources are managed by specialized software called the *operating system*. A system acquires or accesses a resource by requesting an operation on that resource through the operating system.

1.2.4 Sharing and Isolating Computer Resources

In the 1960s, the computing community was frustrated by the problem of sharing resources such as memory, disk, I/O channels, and user input devices on one host among several independent systems. The inability to share resources meant that only one system could be run at a time. Computers at that time cost millions of dollars and most systems used only a fraction of the available resources, so this constraint had a significant effect on computing costs.

Several mechanisms emerged to deal with sharing. The goal of these mechanisms was to isolate one system from another, while still sharing resources. Isolation allows developers to write systems as if they are the only ones using the computer, while sharing resources allows multiple systems to run on the computer at the same time. Since the systems are sharing one physical computer with a fixed set of resources, there are limits to the illusion that isolation creates. If, for example, one system consumes all the CPU resources, then the other systems cannot execute. For most purposes, however, these mechanisms, with some modification, have been sufficient.

Now we turn to the isolation and sharing mechanisms.

- Processor sharing is achieved through the thread-scheduling mechanism. The scheduler selects and assigns a thread to an available processor core, and that thread maintains control until the processor core is rescheduled. No other thread can gain control of this processor core without going through the scheduler. Rescheduling occurs when the thread yields control of the core, when a fixed time interval expires, or when an interrupt occurs.
- As systems grew, all the code and data would not fit into physical memory and virtual memory technology emerged. This memory management hardware partitions a process's address space into *pages*, and swaps pages between physical memory and disk storage as needed. The pages that are in physical memory can be accessed immediately, and other pages are stored on the disk until they are needed. Every memory reference, either a fetch or a store, is checked to see whether the referenced memory location is currently in physical memory. If it is, then the reference proceeds. If it is not, a page-fault interrupt is generated, and the relevant page is fetched from disk and placed into memory. Then the memory reference is repeated. This technology evolved to tag pages with the process ID, providing isolation between processes in a host machine. Virtualization extends this tagging to isolate memory pages between VMs in a host, and between processes within each VM.
- Disk sharing and isolation are achieved using several mechanisms. First, the physical disks can be accessed only through a disk controller that

ensures that the data streams to and from each thread are delivered in sequence. Also, the operating system may tag executing threads and disk content such as files and directories with information such as a userid and group, and restrict visibility or access by comparing the tags of the thread requesting access and the disk content.

- Network isolation is achieved through the identification of messages. Every virtual machine (VM) has an address used to identify messages to or from that VM. We discuss networking in detail in Chapter 3 Networking..
- Memory sharing. In some cases, processes whose address spaces are isolated from each other may wish to share information without using a disk. Operating systems provide services that allow memory to be shared among multiple processes. Each process accesses shared memory using a different address space and the references are resolved through hardware mechanisms.

1.3 Distributed Computing Issues

Today, most systems execute in the Cloud. The Cloud, as we will see in Chapter 4 The Cloud, is a distributed computing platform involving multiple computers that communicate through messages. Distributed computing platforms have several issues that are not present in a single computing platform. This section discusses these issues. Solutions will vary depending on how systems are packaged, and we will discuss those solutions in subsequent chapters. This section identifies the issues so that when we come to various solutions, you will recognize them as solving problems specific to distributed computing.

A system executing in a distributed computing environment consists of a collection of components each of which is executing on a different physical or virtual computer. These components communicate by sending messages from one computer to another. This basic statement – one component sends messages to another component – raises two immediate questions– how is the recipient component identified and how does the message arrive at the recipient?

Several secondary questions arise as well. What happens when the recipient component is overloaded with messages and how is the failure of the recipient managed?

These are some issues that must have solutions for any distributed system. We will identify each of these issues here although the mechanisms for solution will be deferred until later chapters.

1.3.1 Discovery

Discovery is the process of finding an address to go with a particular name. Components, typically, have names. Names are easier to remember than numbers. The address of a component, however, consists of numbers. Discovery allows you to map the name to its associated number. Discovery mechanisms are discussed in Chapter 3 Networking, Chapter 4 The Cloud, and Chapter 5 Container Orchestration

1.3.2 Scaling

A request for service from a component arrives in the form of a message. A component has limited capacity and could get overloaded because it is asked to service more messages than it has the capacity to process. Horizontal scaling of a component (the only type of scaling we discuss in this book) is the process of creating additional instances of the component and distributing the messages requesting service among these instances. One aspect of scaling is how are resources for the new instance allocated. Scaling is discussed in Chapter 4 The Cloud and Chapter 5 Container Orchestration.

1.3.3 Message communication patterns

A message sent from one component to another must arrive at the designated recipient. Partially, this is a matter of using the correct numeric identifier as we discussed in Section 1.3.1 Discovery but it is also a matter of passing through intermediaries that may modify the numeric identifier in a controlled fashion. We will see a number of these intermediaries in Chapter 3 Networking, Chapter 4 The Cloud, and Chapter 5 Container Orchestration. Each use of an intermediary determines a communication pattern.

1.3.4 Failed component recovery

As we will discuss in Chapter 4 The Cloud, the computers on which a component is executing may fail. We will discuss failure detection and recovery in Chapter 4 The Cloud and Chapter 5 Container Orchestration

1.4 Quality Attributes

A quality attribute is a measure of a system in some dimension. Lists of quality attributes are extensive but for our purposes in Part I, two are important: performance and availability. Security is also an important quality and we discuss that in detail later.

1.4.1 Performance

Performance is a property of a component that deals with the time it takes to perform operations or its capacity to perform operations. Two different measures of performance are relevant: latency and throughput. Latency is how long does it take to perform an operation. The operation could be sending a message or it could be computation. If the operation is user input, the latency until a response is visible to the user may be called response time.

Throughput is the number of operations performed per unit time. Requests processed per second, for example. Another example is bits transferred over a network per unit time.

There is no definitive relationship between latency and throughput. You might expect with high throughout you would get low latency but that is not necessarily true. Latency is a function not only of throughput but also the type of requests and the scheduling strategy used to allocate the resource being requested.

1.4.2 Availability

Availability is a property of software that it is there and ready to carry out its task when you need it to be. It is usually measured as the percentage of time a system is available for service. You will see references to 4 9's, for example, meaning that a system or component is available 99.9999 percent of the time. Availability is achieved through redundancy. A spare can be kept active in case of component failure or the spare can be activated when a component fails. Failed component recovery– activating a spare – is an issue that must be considered in distributed computing. The mechanisms for achieving availability of components will be discussed in Chapter 4 The Cloud and Chapter 5 Container Orchestration.

1.5 Summary

Bits in a computer can be used for instructions, as an address, or as a data value. Reads and writes are in terms of bits and the flexibility of data allows for saving and restoring images of software.

State is another term for data. A computation needs both computational instruction and data on which to operate.

Computers have four resources – processors, memory, disks, and networks. Mechanisms exist for the isolation and sharing of these resources.

Distributed computation introduces issues of discovery, scaling, communication patterns, and recovery.

Important quality attributes are performance and availability.

1.6 Exercises

1 Use an operating system simulator to construct an instruction, store it in memory, and transfer control to it.

2 Create memory that is shared between two processes.

1.7 Discussion Questions

1 Find a discussion of a M/M/1 queueing model. What factors influence latency. What factors influence throughput.

2 Find a discussion of the history of virtual memory. When was the first computer built that used virtual memory? How long did it take for a commercial virtual memory computer to become available?

Chapter 2 Virtualization

Much computing these days is done on virtual machines. Even systems such as automobiles rely heavily on virtualization. When you finish this chapter and do the exercises, you will understand

- The goals of virtualization
- How virtualization works
- The distinction between a virtual machine instance and a virtual machine image

2.1 Coming to Terms

Client– A client of a service is another service that directly depends on the original service.

Container – a container is an executable image that packages a service and its dependent libraries.

Host – a physical computer.

Hypervisor -an operating system that manages virtual machines.

Node – a physical computer or a virtual machine.

Service– a coherent collection of functionality.

Serverless architecture—a computing model in which cloud providers maintain a pool of servers and you provide a stand alone function packaged as a container.

Shared memory – memory shared by multiple processes.

State – Data in a computer system.

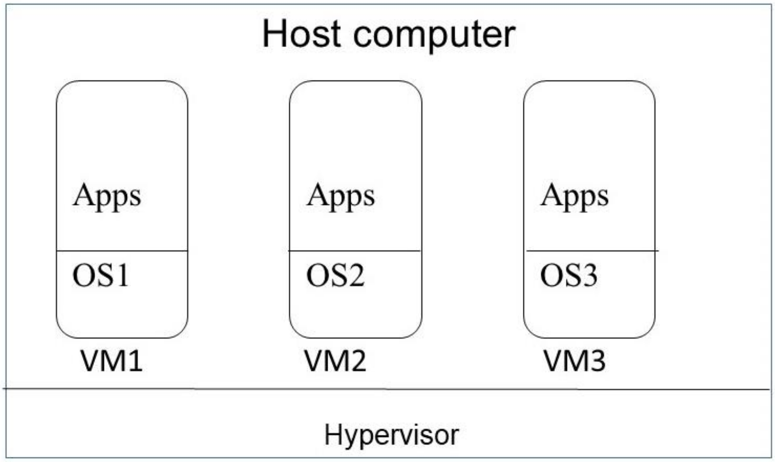
Virtual machine—a software construct that exposes the resources CPU, memory, disk, and network connection to loaded software making it appear to the loaded software that it is a physical computer.

2.2 Virtual Machines

A Virtual Machine (VM) is a software construct that exposes the resources of CPU, memory, disk, and network connection to loaded software making it appear to the loaded software that it is executing on a physical computer. A VM runs under the control of a specialized operating system called a hypervisor.

Figure 2.1 shows several VMs residing in a physical computer. The physical computer is called the “host computer” and the VMs are called “guest computers.” Figure 2.1 also shows a hypervisor which is an operating system for the virtual machines. This hypervisor runs directly on the host hardware and is often called a *bare-metal* or *Type 1* hypervisor. The VMs that it hosts implement systems and services. Bare-metal hypervisors typically run on a data center or cloud.

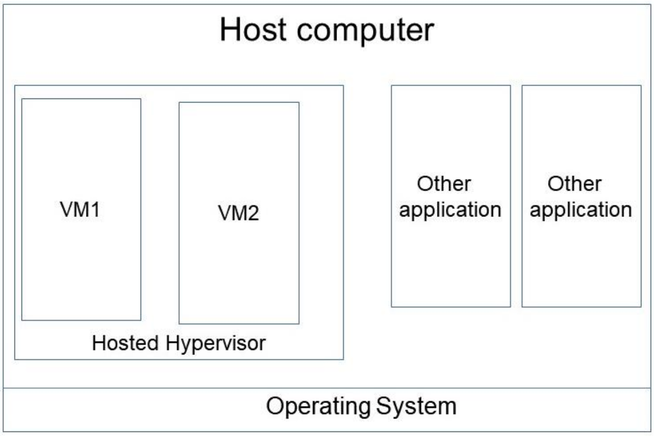
Figure 2.1: Bare-Metal Hypervisor and VMs



There is another type of hypervisor, called a *hosted* or *Type 2* hypervisor, which is shown in Figure 2.2. In this case, the hypervisor runs as a service on top of a host operating system, and the hypervisor in turn hosts one or more VMs. Hosted hypervisors are typically used on desktop or laptop computers. They allow users to run and test systems that are not compatible with the computer’s host operating system (e.g., run Linux systems on a Windows computer). They are also used to replicate a production environment on a development computer even if the operating system is the same on both the development computer and the production environment.

A hypervisor requires that its guest VMs use the same instruction set as the underlying physical CPU–the hypervisor does not translate or simulate instruction execution. For example, if you have a VM for a mobile or embedded device that uses an ARM processor, you cannot run that virtual machine on a hypervisor that uses an x86 processor. There is another technology, related to hypervisors, for cross-processor execution, called an *emulator*. An emulator reads the binary code for the target or guest processor and simulates execution of guest instructions on the host processor. The emulator often also simulates guest I/O hardware devices. For example, the open-source QEMU emulator[1] can emulate a full PC system, including BIOS, x86 processor and memory, floppy disk drive, sound card, and graphics card.

Figure 2.2: Hosted Hypervisor



Both hosted/Type 2 hypervisors and emulators allow a user to interact with the guest system through an on-screen display, keyboard, and mouse/touchpad. Developers working on desktop systems or working on specialized devices, such as mobile platforms or devices for the Internet of Things (IoT), may use a hosted/Type 2 hypervisor and/or an emulator as part of their build/test/integrate toolchain. We will not discuss emulators in this book and will be focused on VMs that use the same instruction set as the host.

[1] https://www.qemu.org

A hypervisor performs two main functions: (1) managing the code running in each VM, and (2) managing the VMs themselves. To elaborate,

1. Code that communicates outside the VM by accessing a virtualized disk or network interface is intercepted by the hypervisor and executed by the hypervisor on behalf of the VM. This allows the hypervisor to tag these external requests so that the response to these requests can be routed to the correct VM.

   The response to an external request to an I/O device or the network is an asynchronous interrupt. This interrupt is initially handled by the hypervisor. Since multiple VMs are operating on a single physical host computer and each VM may have I/O requests outstanding, the hypervisor must have a method for forwarding the interrupt to the correct VM. This is the purpose of the tagging mentioned above.

   One additional type of instruction that is neither a normal instruction nor an external request is a *system call*. In this case, code operating in user mode inside the VM asks its internal operating system for a service and this, frequently, involves changing the mode of the code being executed in the VM to kernel mode. Depending on the hardware architecture, a system call may generate an interrupt. In this case the interrupt is initially fielded by the hypervisor and forwarded to the appropriate VM, just as in the case of an asynchronous interrupt from an I/O device or the network.

2. VMs must be managed. For example, they must be created and destroyed, among other things. Managing VMs is a function of the hypervisor. The hypervisor does not decide on its own to create or destroy a VM. It acts on instructions from a user or, more frequently, from a cloud infrastructure. We will explore how this works in Chapter 4 The Cloud. The process of creating a VM involves loading a VM image, and we discuss this in the next section.

   In addition to creating and destroying VMs, the hypervisor monitors them. Health checks and resource usage are part of the monitoring. The hypervisor is also in the defensive security perimeter of the VMs with respect to attacks.

   Finally, the hypervisor is responsible for ensuring that a VM does not exceed its resource utilization limits. Each VM has limits on CPU utilization, memory, and disk and network I/O bandwidth. Before starting a VM, the hypervisor first ensures that there are sufficient physical resources available to satisfy that VM's needs, and then the hypervisor enforces those limits while the VM is running.

A VM is booted just as a *bare metal* host is booted. When the machine begins executing, it automatically reads a special program called the *boot loader* (BIOS) from a disk drive,[2] either a disk drive internal to the computer or a disk drive connected through a network. The boot loader reads the operating system code from disk into memory, and then transfers execution to the operating system. In the case of a physical computer, the connection to the disk drive is made during the power up process. In the case of the VM, the connection to the disk drive is established by the hypervisor when it starts the VM.

From the perspective of the operating system and service software inside a VM, it appears that the software is executing inside of a bare host. The VM provides a CPU, memory, I/O devices, and a network connection. In Chapter 3 Networking, we will see that each VM also has its own IP address. In fact, it is quite difficult for an operating system to detect whether it is running in a VM or physical machine–for example, it can look for evidence in log files, but the detection approaches are *ad hoc* and specific to each hypervisor.

All these concerns mean that the hypervisor is a complicated piece of software. One concern with virtual machines is the overhead introduced by the sharing and isolation needed for virtualization. That is, how much slower does a service run on a virtual machine, compared to running directly in a bare physical host? This turns out to be a complicated question since it depends on the characteristics of the service and on the virtualization technology used. For example, services that perform more disk and network I/O incur more overhead than services that do not share these host resources. Virtualization technology is improving all the time, but overheads of around 10% have been reported by Microsoft on their Hyper-V hypervisor[3].

2.3 VM Images

Just as a physical computer without software merely consumes power and generates heat, a virtual machine without software is not very useful. A physical computer loads software by reading directly from the BIOS, a built-in disk drive, an

[2] We use the term "disk drive" to refer to either a rotating hard disk drive (HDD) device or a solid-state disk drive (SSD) device, recognizing that SSDs have neither disks nor any moving parts to drive.

[3] https://docs.microsoft.com/en-us/biztalk/technical-guides/system-resource-costs-on-hyper-v

external disk drive, or a storage access network (SAN).[4] The same is true for a VM, except that instead of direct hardware or network access it accesses its potential sources of software through the hypervisor.

We call the contents of the disk drive that we boot a VM from an *image.* This image contains the bits that represent the instructions and data that make up the software that we will run (i.e., operating system and services). The bits are organized into files and directories according to the file system used by your operating system. The image also contains the boot load program, stored in its pre-determined location.

There are several ways to create a new image. One common approach is to find a node that is already running the software we want and make a snapshot copy of the bits in that node's memory.

Another approach is to start from an existing image and add additional software. There are repositories of machine images (usually containing open-source software) providing a variety of minimal images with just OS kernels, other images that include complete systems, and everything in between. These provide efficient starting points and support quickly trying out a new package or program. However, there are some issues when pulling down and running an image that you (or your organization) did not create: there are relatively minor issues, such as inability to control the versions of the OS and software, and there are more significant issues related to security. The image may have software that contains vulnerabilities or that is not configured securely, or worse, the image may include malware. In Chapter 14 Secure Development, we discuss security issues related to the download of software from the web.

A third approach, creating an image from scratch, is a bit more complicated. First consider creating an image on a host–you may have done this yourself if you have a new laptop computer. You start by obtaining *install media* for your chosen operating system. This is a bootable CD, DVD, or USB memory stick that contains a program to create the image you want. You boot your new node from the install media, and it formats the node disk drive, copies the operating system onto the drive, and adds the boot loader in the pre-determined location. You remove the install media, restart your machine, and it boots from your newly created image.

[4] https://en.wikipedia.org/wiki/Storage_area_network

You can then add whatever services you want to the image, and maybe make a snapshot copy to use on other nodes.

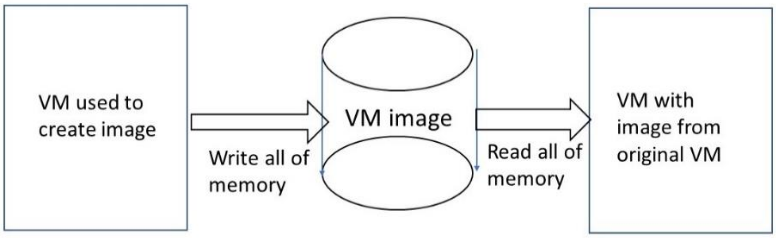
You can follow a similar approach to create an image for a virtual machine. Here, start by using a Type 2 or hosted hypervisor, so that you can interactively control the hypervisor and have a GUI for the VM. You command the hypervisor to start a new VM with an empty virtual disk drive and boot the VM from your install media. The installer program copies the operating system files to the VM's virtual disk drive. You may need to have the hypervisor add its customized boot loader to your virtual disk (some hypervisors require a special boot loader). You can then make a snapshot copy of the bits on your virtual disk and use that image to boot other VMs using Type 1 or Type 2 hypervisors.

This manual approach to creating a VM image is easy to understand but requires a person to attend to each of the tasks. As we will see in Chapter 14 Secure Development, you must update your images frequently as patches become available for your operating system and other software. To operate efficiently, you need to automate the creation of new images. We will discuss this in Section 10.2 Infrastructure as Code.

You might have noticed that we did not mention installing services on the newly created image. While you could easily install services when creating an image, this would lead to a unique image for every version of every service. Aside from the storage cost, this proliferation of images becomes difficult to keep track of and manage. It is customary to create images that contain only the operating system and other essential programs, and then add services to these images after the VM is booted, in a process called *configuration*.

We summarize the process of creating and loading a virtual machine image in Figure 2.3, which shows a disk file being generated from one VM and subsequently loaded by the hypervisor into another VM.

Figure 2.3: The Process of Creating, Saving, and Loading an Image



2.4 Containers

VMs solve the problem of sharing resources and maintaining isolation that was posed in the 1960s. However, VM images can be large, and transferring VM images around the network is time consuming. Suppose you have an 8 GB(yte) VM image. You wish to move this from one location on the network to another. In theory, on a 1 Gb(it) per second network, this will take 64 seconds. However, in practice a 1 Gbps network operates at around 35% efficiency. Thus, transferring an 8 GB VM image will take around 3 minutes. There are approaches that can reduce this transfer time, but the result is still on the order of minutes. After the image is transferred, the VM must boot the operating system and start your services, which takes still more time.

Containers are a mechanism to maintain the advantages of virtualization—isolation of its actions from other containers—while reducing image transfer time and startup time. A container is an executable image that packages a service and its dependent libraries. It runs under the control of a container engine. Container engines use several Linux features to provide isolation. Linux control groups set resource utilization limits, and Linux namespaces prevent a container from seeing other containers.

Re-examining Figure 2.2, we see that a VM executes on virtualized hardware under the control of the hypervisor. In Figure 2.4, we see several containers operating under the control of a container engine which, in turn, is running on top of a fixed operating system. By analogy with VMs, containers run on a virtualized operating system. Just as all VMs on a physical host share the same underlying physical hardware, all containers share the same operating system kernel (and through the operating system, they share the same underlying physical hardware). The operating system can be loaded either onto a bare-metal physical machine or a virtual machine.

This sharing of the operating system gives us one source of performance improvement when transferring images. If the target machine has a container engine running on it, there is no need to transfer the operating system as part of the container image. Since modern operating systems are on the order of 1 GB(yte) in size, this saves a substantial amount of time.

The second source of performance improvement is the use of layers in container images. To understand this, we describe how a container image is constructed. We illustrate the construction of a container to run the *LAMP stack*, and we will build the image in layers. LAMP is Linux, Apache, MySQL, and PHP. LAMP is a widely used stack for constructing web systems.

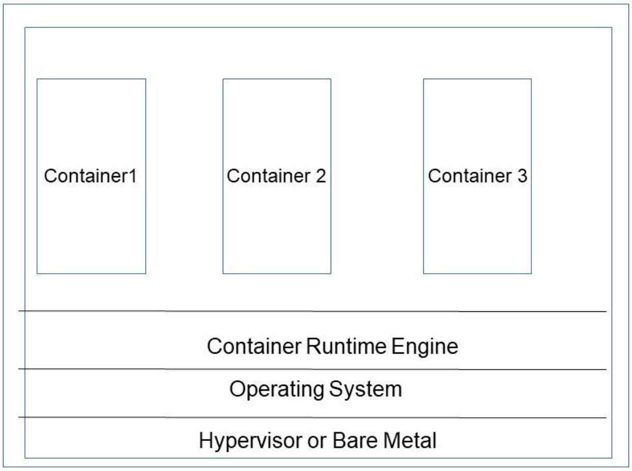
The process begins with creating a container image containing a Linux distribution. We'll choose Ubuntu for this example. This image can be downloaded from a library using the container management system. Once you have created the Ubuntu container image and identified it as an image, you execute it, i.e., make a container, and you use that container to load Apache using features of Ubuntu. Now you exit the container and inform the container management system that this is a second image. You execute this second image and load MySQL. Again, you exit the container and give the image a name. Repeating this process one more time and loading PHP, you end up with a container image holding the LAMP stack. Because this image was created in steps and you told the container management system to make each step an image, the final image is considered by the container management system to be made up of layers.

Now you move the LAMP stack container image to a different location for production use. The initial move requires moving all the elements of the stack, so the time it takes is the time to move the total stack. Suppose, however, you update PHP to a newer version and move this revised stack into production. The container management system knows that only PHP was revised and moves only the PHP layer of the image. This saves the movement of the rest of the stack. Since changing a software component within an image happens much more frequently than initial image creation, placing a new version of the container into production becomes a much faster process than it would be using a VM. Whereas loading a VM takes on the order of minutes, loading a new version of a container takes on the order of microseconds or milliseconds.

Layers reflect changes in the image creation and, when recreating an image, the build process does not build lower layers if they have not changed. The build begins where the first layer with a change is encountered. Thus, when updating an existing image, if only the top layer has changed, only the top layer needs to be updated.

A container image is self-contained. That is, when you load software into a container, dependent functions are also loaded. These dependent functions may have vulnerabilities. Scanning tools can examine the dependencies within a container and compare the sources to known vulnerabilities. Tools such as Docker Bench, Clair, and Anchore are open-source and freely available. We discuss this in more detail when we discuss Software Bill of Materials (SBOM) in Section 11.5 Integration environment.

Figure 2.4: Containers on Top of a Container Runtime Engine on Top of an Operating System on Top of a Hypervisor (or Bare Metal)



There is one more wrinkle to this example using PHP. Since PHP is an interpreted language (see Section 8.2 for a discussion of the distinction between compiled and interpreted languages), suppose your cloud provider had a LAMP stack pool. All that is needed to make an executable is a file containing a PHP program that can be used as input. Pointing an instance of the LAMP stack pool to the file containing a new version of your PHP will put the new version into execution within microseconds.

You can script the creation of a container image through a file. This file is specific to the tool you are using to create the container image. Such a file allows you to specify what pieces of software are to be loaded into the container and saved as an image. Using version control on the specification file ensures that each member of your team can create an identical container image and modify the specification file as needed. We discuss these practices in more detail in Section 10.2 Infrastructure as Code.

Containers interact with a container runtime. There are several container runtime providers, most notably Docker, Kubernetes, and Mesos. Each of these packages provides capabilities to create container images and to allocate and execute container instances. The interface between the container runtime and the container has been standardized by the Open Container Initiative, allowing a container created by one vendor’s package, say Docker, to be executed on a container runtime provided by another vendor, say Kubernetes.

This means that you can develop a container on your development computer and deploy it to a production computer and have it execute. The resources available, of course, will be different and so deployment is still not trivial. If you specify all the resource specifications as configuration parameters, the movement of your container into production is simplified. We return to the topic of configuration parameters in Chapter 10 Basic DevOps Tools.

Sidebar: Container Images

The terminology distinction between a VM and a VM image carries over to containers. That is, a container image is a set of bits that when executed becomes a container.

However, this terminology is not always followed in practice. The term container is frequently used to refer to either a container image or a container, depending on context. For example, in Section 2.6, we will discuss container registries. In fact, a container registry is actually a repository for container images. Understanding the distinction between containers and container images is important when you use containers but maintaining this distinction when discussing or writing about containers is cumbersome.

2.5 Containers and VMs

As we noted earlier, a VM virtualizes the physical hardware–CPU, storage, and network I/O. The software that you run on the VM includes an entire operating system, and you can run almost any operating system in a VM. You can also run almost any program in a VM (unless it must interact directly with the physical hardware). This is important when working with legacy or purchased software. Having the entire operating system also allows you to run multiple services in the same VM, which is desirable when the services are tightly coupled, share large data sets, or if you want to take advantage of the efficient interservice communication and coordination that are available when the services run within the context of the same operating system. The hypervisor ensures that the operating system starts, monitors its execution, and restarts the operating system if it crashes. Containers are monitored, started and stopped by the run time engine.

In addition, there is a distinction between VMs and containers in how they can be connected to a network. We discuss this in Section 3.6 Bridged and NAT networks

2.6 Container Registries

A container image is created by choosing a suitable *base image*, for example an image that has the runtime and core libraries for the programming language that you are using, and then adding your service-specific code and dependent packages. The base images are stored in a *container registry*. When you use the container runtime to build the full container image for your service, that image is stored locally on your development machine. Typically, you will then place that image back into a registry so that it can be accessed from other machines for test, integration, and deployment to production. This is the same model used in version control systems where modifications are stored and tested locally until you are ready to commit them to the version control system.

Container registries have an interface that is similar to the interface of software version control systems such as Git. We discuss version control systems in Chapter 10 Basic DevOps Tools. You retrieve a container image through a “pull” command and you store a container image through a “push” command. Furthermore, the images can be tagged with a version identifier, which allows you to retrieve the latest version of an image or a specific previous version. This provides the same benefits as controlling versions of source code. You can easily roll back to a previous image, and you can differentiate images being used to fix bugs in production from images being used to develop new features.

Like a source code repository, a container registry can be public, for example Docker Hub, or private to your organization. Public registries provide access to a broad collection of base images. Since the format of the images is standardized, you do not need to be concerned about which tool was used to create the image. You do need to be concerned that you are not introducing malware into your system. Some organizations ensure that entries into their private repositories have been scanned for security vulnerabilities.

Private registries can be hosted by a cloud service provider (for example, Amazon Elastic Container Registry), or maintained as an infrastructure service within your organization's network. Private registries

- allow your organization to control which base images are used within the organization. Such as
    - only containers with no known vulnerabilities are entered into the repositories.
    - only authorized personnel can push or pull entries.
- Private registries also standardize the operational environment and allow your organization to protect the intellectual property in its completed service container images.

2.7 Serverless Architecture

We saw an example with PHP where the cloud provider maintained a pool of PHP servers and you needed only to provide a PHP program go begin execution. *Serverless architecture* generalizes this to be any executable container. Suppose, for example, you wish to send a welcome message when a new user signs up. You can package this welcome message as a function in a container. It only needs to know the details of the new user sufficient for a welcome message, e.g. first name and email address. You now tell the cloud provider about this container. When a new user arrives, a message is sent to the welcome message container. This triggers the cloud provider to activate the container and send the message to it.

Since load times a container are very short, taking just milliseconds, the time to allocate, load, and start the container can be very fast—on the order of microseconds. Since placing the container into execution is fast, it is not necessary to leave the container running. You can afford to reallocate a new container instance for every request. When your service completes processing of a request, instead of looping back to take another request, it exits, and the container stops running and is deallocated.

Serverless architectures do, in fact, have servers, which host container runtimes. The servers and container runtimes are a portion of the cloud infrastructure. You, as a developer, are not responsible for allocating or deallocating them. The cloud service provider features that support allocation and invocation are called *function-as-a-service (FaaS)*.

A consequence of the dynamic allocation and deallocation in response to individual requests is that these short-lived containers cannot maintain any state. That is, the containers must be stateless. Any state used for coordination must be stored in an infrastructure service delivered by the cloud provider.

Cloud providers impose some practical limitations on FaaS features. One limitation is that the “cold-start” time, when your container is allocated and loaded the first time, can be several seconds. Subsequent requests are handled nearly instantaneously, as your container image is cached on a node. Finally, the execution time for a request is limited–your service must process the request and exit within the provider’s time limit, or it will be terminated. Cloud providers do this for economic reasons, so that they can tailor the pricing of FaaS compared to other ways of running containers, and to ensure that no FaaS user consumes too much of the resource pool. Some designers of serverless systems devote considerable energy to working around or defeating these limitations; for example, pre-starting services to avoid cold-start latency, making dummy requests to keep services in cache, and forking or chaining requests from one service to another to extend the effective execution time.

2.8 Summary

Hardware virtualization allows the creation of several virtual machines sharing the same host. It does this while enforcing isolation of CPU, memory, disk storage, and network. This allows the resources of the physical machine to be shared among several VMs and reduces the number of physical machines that an organization must purchase or rent.

A virtual machine image is the set of bits that are loaded into a VM to enable its execution. Virtual machine images are created by various techniques for provisioning including using operating system functions or loading a pre-created image.

Containers are virtualized operating systems and provide performance advantages over VMs. Containers constructed in terms of layers are faster to deploy when a component changes.

Serverless architectures are stand-alone container runtimes that can load and execute a container in microseconds.

2.9 Exercises

1. Create a virtual manager on your laptop. This can be done by downloading VirtualBox or using the facilities of your operating system.
2. Create a virtual machine running Ubuntu.
3. Load LAMP into your virtual machine using apt-get. What are the errors that you made?
4. Create a Docker container image with the LAMP stack in four layers. Test the image by executing a simple PHP program.

2.10 Discussion Questions

1. Where do you find the Ubuntu distribution? What are the steps that take a Linux kernel distribution and creates an Ubuntu distribution? Who provides the funding for the creation of the Ubuntu distribution?
2. Enumerate different package managers used by the different distributions of Linux. What are the differences between them?
3. Two VMs hosted on the same physical device are isolated, but it is still possible for one VM to affect the other VM. How can this happen?
4. We’ve focused on isolation among VMs that are running at the same time on a hypervisor. VMs shut down and stop executing, and new VMs start up. What does a hypervisor do to maintain isolation, or prevent leakage, between virtual machines running at different times? Hints: memory, disk, virtual MAC.
5. How does the container management system know that only one layer has been changed so that it needs to transport only one layer?
6. Can a WM image created on AWS be executed on the Google Cloud?

7. How does the cloud provider know that a container image is intended to be serverless?

Chapter 3 Networking

Along with virtualization, networking is the second pillar of modern computing infrastructure. The two of them come together to enable cloud computing. In this chapter we explore the topic of networking. Cloud computing will be covered in the next chapter.

When you have completed this chapter, you will be familiar with IP addresses and the IP protocol suite. You will also be familiar with ports and TCP/IP. The domain name system (DNS) allows you to find an IP address given a Uniform Resource Locator. Other networking topics discussed are *subnets, gateways, proxies* and *firewalls* as well as *tunneling* through firewalls.

3.1 Coming to Terms

Bridge network– a single, aggregate network from multiple communication networks or network segments

Cache– a partial copy of a repository kept locally to improve performance.

Checksum – a small amount of data that is derived from a larger set of data. It is used to detect errors in the larger set.

Device – the term device as used in this book means either a physical computer, a virtual machine, a container, or some piece of electronics that is connected to the internet.

Firewall– monitors incoming and outgoing network traffic and permits or blocks data packets. Firewalls are configured by specifying rules:

- *allow or disallow message based on ip address of recipient or sender*
- *route allowed message based on port number. E.g. messages specifying port 80 are sent to http server.*

Gateway – A gateway connects two networks and converts information, data or other communications between the protocols used by these two networks.

Message – a message is a collection of information passed from one component to another.

Network Address Translation (NAT) – maps one IP address to another. Firewalls, gateways, and proxies frequently perform NAT.

Proxy– A proxy server makes web request on your behalf, collects the response from the web server, and forwards the results back to your system. The effect is to hide your IP address from the outside world.

3.2 Introduction

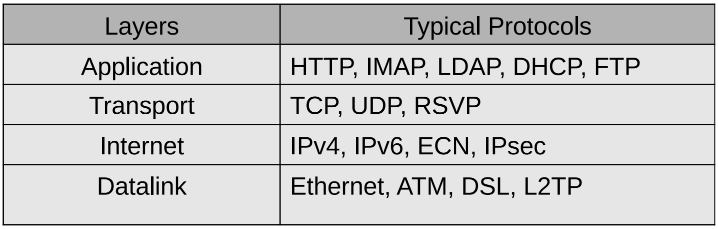
Networking enables us to send messages from one device to another device, or more precisely, from a service running on one device to a service running on another device. The devices can be physical or virtual. They can be nodes which act as general-purpose computers, or they can be special purpose electronics such as a thermostat. In Chapter 2 Virtualization, we mentioned virtualization of network interfaces, and we will discuss more of those details in this chapter.

When devices are in the same room or building and directly connected to each other, it is conceptually easy to send a message: We send the bits of the message, prefixed with the address of the intended recipient device, onto the network cable. Every device watches all the messages on the network, and when it sees a message addressed to one of its services, it takes the message off the network and delivers the message to the service. In practice, there are many complicated details to this seemingly simple process. What if two services put messages on the network at exactly the same time? How do we assign addresses to devices? How do we handle electrical noise that might corrupt the contents of your message? Local area network (LAN) protocols such as the Ethernet protocol solve this complexity–the core Ethernet protocol specification is 3,315 pages long!

It isn’t practical to create a network where every message sent must be seen by every potential message receiver. This led to the invention of the Internet Protocol, or IP. The term *internet* is an abbreviation for internetwork, and IP allows us to efficiently connect smaller networks together. Instead of requiring every device to monitor every message on the network to pluck out the messages that it should receive, IP delivers messages by passing each message through a series of special devices called *routers*. Each *hop* in this series gets the message closer to its destination, until the final router in the sequence delivers the message to the intended recipient. The techniques used by the routers to determine the next recipient are clever. A message sender does not need to know the sequence of routers needed to reach a particular recipient. Instead, the original sender, and each router in the sequence, only needs to make a best guess to choose a next router that might get the message closer to the recipient. Because each message is usually sent only to the next router,[5] IP makes efficient use of available bandwidth. We will not go into detail into the mechanisms used to determine the next router in the sequence.

As a developer creating services that use IP networking, you need to understand the basics of the internet protocols. The internet protocols are organized into four layers, as shown in Figure 3.1 The lowest layer, *Datalink*, includes the Ethernet that we mentioned above. The next layer, *Internet*, includes the Internet Protocol and several other protocols that allow us to address and route messages from a source device to a destination device. The *Transport* layer provides the capability to connect a service on the source device to a service on the destination device. Finally, the *Application* layer protocols define the structure of messages between services.

Figure 3.1: Layers of the Internet



Layers	Typical Protocols
Application	HTTP, IMAP, LDAP, DHCP, FTP
Transport	TCP, UDP, RSVP
Internet	IPv4, IPv6, ECN, IPsec
Datalink	Ethernet, ATM, DSL, L2TP

Ethernet typically provides the physical layer between the source device and the first router in the sequence, and between the last router in the sequence and the destination device. The connections between the other routers in the sequence may use other physical layer protocols for long distance communication, such as Asynchronous Transfer Mode (ATM) or Synchronous Optical Networking (SONET).

In the following sections, we will discuss the Networking and Transport layers. There are many Application layer protocols. Some are used by infrastructure services, and we will discuss these later in this book. Other application protocols, such as the IMAP4 protocol for email, are outside the scope of this book.

[5] IP does use special broadcast messages that go to more than one computer, but these are usually infrequent.

Sidebar: internet or WWW?

One question frequently asked is, “What is the difference between the internet and the World Wide Web (WWW)?” There are three different answers for that question. The first is that the internet has existed since the 1980s and the WWW since the 1990s. The WWW is built on top of the internet. A second answer is that the internet consists of the three bottom layers of Figure 3.1 and the WWW lives in the Application layer. A final answer is that each is governed by a different organization. The internet is governed by the Internet Engineering Task Force (IETF), which can be found at http://www.ietf.org/ and the WWW is governed by the World Wide Web Consortium (W3C), which can be found at https://www.w3.org/. What both organizations have in common is that they develop standards for their respective areas of control. These standards are developed using an open, transparent process, and you can find the discussions leading to the standards at each organization’s home websites.

We begin our discussion of the Networking layer by mentioning some common false assumptions that might be made about networking.

The fallacies of distributed computing are a set of assertions made by L Peter Deutsch and others at Sun Microsystems describing false assumptions that programmers new to distributed systems invariably make.

1. The network is reliable. In fact, messages are not guaranteed to be delivered. This is the motivation for including acknowledgements in some protocols.
2. Latency is zero. Transferring messages takes time. The amount of time will vary depending on multiple factors including the hardware used in the network and the distance between endpoints.
3. Bandwidth is infinite. Bandwidth is limited by the hardware used to transport the messages.
4. The network is secure. Any system with access to a network can eavesdrop and modify the traffic on that network.
5. Technology does not change. As with everything in the computer field, changes in technology are ubiquitous.

6. There is one administrator. Different portions of a network can be administered by different individuals using different systems.

7. Transport cost is zero. Data is transported by hardware and takes time. Both factors have costs.

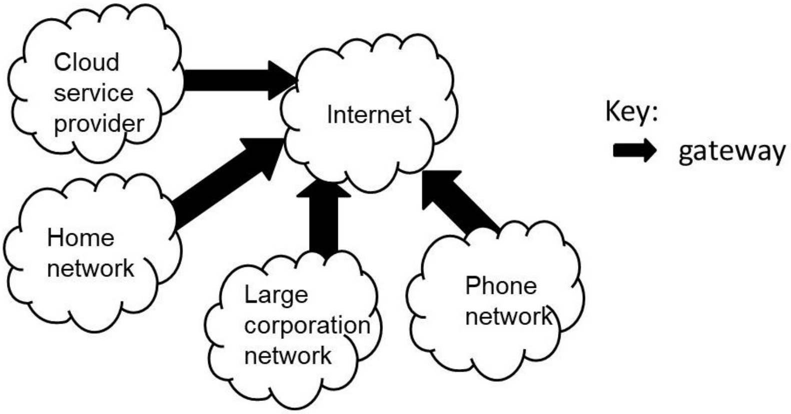
8. The network is homogeneous. Different portions of a network can be constructed from different hardware and can use different software.

Keep these fallacies in mind as you continue in this section. They may help you understand some of the complexities in the various protocols. Now we turn to the identification of individual devices.

3.3 IP Addresses

As we noted above, the internet connects networks together. Figure 3.2 shows several typical types of networks, connected by the internet. Every device that can connect to the internet is assigned an IP address. This address enables messages to be sent to that device.

Figure 3.2: Different Types of Networks Connected to the Internet



Internet Protocol Version 4 (IPv4) was the first widely used version of the Internet Protocol addressing scheme. It uses 32 bits to represent an address. This is typically written using *dot decimal format*. Each octet (8-bit chunk) of the address is represented by a decimal number between 0 and 255, with a dot between each number, e.g., 4.31.198.44.

Websites such as whatismyip.com will report the IP address of the device you use to access it. 32 bits allows somewhat over 4 billion addresses. When even toasters have IP addresses, this is not enough and all the available IPv4 addresses have been assigned.

Because the exhaustion of IPv4 addresses was anticipated, a different IP addressing scheme, called IPv6, was approved in 1996.[6] The major difference between IPv4 and IPv6 is that addresses in IPv6 are 128 bits long. This will provide sufficient addresses for a long time into the future. IPv6 addresses are represented as eight groups of four hexadecimal digits, with the groups separated by a colon. By convention, the leading zeros in each group are suppressed, and a string of all-zero groups is replaced by two colons ("::"). For example, the IPv4 address 4.31.198.44 has an IPv6 address of 2001:1900:3001:11::2c.

These addressing schemes use similar routing strategies, and these strategies are based on how the addresses are assigned. As we write this book, IPv4 addresses continue to dominate traffic on the internet: For example, fewer than 40% of Google searches were made from systems using IPv6. To simplify our discussion, for the rest of this chapter we will limit our examples to IPv4.

3.3.1 Assigning IP Addresses

The Internet Corporation for Assigned Names and Numbers (ICANN) is a non-profit organization with the responsibility for assigning domain names and IP addresses. In this section, we are concerned with the numbers portion of their responsibility. We will discuss domain names when we discuss the Domain Name System in the next section.

A division of ICANN, the Internet Assigned Numbers Authority (IANA), allocates numbers through a hierarchy. The first level of the hierarchy allocates addresses in large blocks (X.0.0.0 through X.255.255.255). A recipient of one these large blocks is authorized by ICANN to split up the block and further allocate the addresses in smaller blocks. The recipients of these large blocks are, for the most part, non-profit organizations managing internet addresses for a region such as North America, Europe, Africa, or Asia-Pacific. (A few of these large address blocks were allocated directly to large telecommunications and computer hardware companies,

[6] In case you are wondering what happened to IPv5, that designation was used for an experimental protocol for streaming over the internet, called "ST." It used the same address scheme as IPv4.

and several blocks were allocated to the United States Department of Defense.[7]) In turn, the regional authority allocates these large blocks (or a part of a block) to an ISP (Internet Service Provider), such as AT&T, BT, or DT, or to large businesses or other organizations. In each case, the ISP or business then assigns addresses to the devices on its network.

Two attributes of the IP address deserve special attention.

1. *Public or Private.* The IP address number your device is assigned can be kept private to your network or can be known publicly throughout the internet. Keeping the IP address number private serves three purposes. First, keeping the address private to a local network allows the same IP address number to be used in multiple different networks. This is one technique that is used to make the IPv4 addressing scheme extend to more devices than the 2^{32} unique addresses available. Second, it allows an organization to add and remove devices from its network without coordinating with any higher authority in the address assignment hierarchy. You usually need to pay an administrative fee to receive a public IP address, allocated through the IANA hierarchy. Finally, it also allows organizations to hide the structure of their internal network from the internet, using a NAT device between the private network and the public internet. NAT devices are usually part of a network device called a *firewall*, which restricts IP message exchanges between two networks. Together, NAT and firewalls improve security and privacy at a network perimeter. We will return to these devices later in this chapter.

   IANA has designated the following IPv4 address ranges as private:

10.0.0.0	to	10.255.255.255
172.16.0.0	to	172.31.255.255
192.168.0.0	to	192.168.255.255

   Every local network is free to assign these numbers to devices on the local network. If you see a device with an IP number in one of these ranges, it is a private IP number and is visible and accessible only from within its local network.

[7] The U.S. Department of Defense funded the development and operation of the internet (then called ARPANET) until the early 1980s.

One other range of IP addresses is private and given special treatment. This is the 127.xxx.xxx.xxx block. These IP addresses are called “reserved addresses.” The address 127.0.0.1 is the most used reserved address. When your local network interface sees this address as a destination, it does not send the message out to the network, but immediately loops the message back as though it was received from the network. Conventionally, 127.0.0.1 is called “localhost.” Thus, if you wish to test a web page locally, one technique is to put localhost/test page into your browser and the request will be sent back to your device and, if you have correctly set up your environment, test page will be displayed.

Given an IP address, there is no way to tell with 100% certainty whether it is a public or private address. An address is private or public based only on how the network infrastructure is configured. Certainly, an IP address that is in one of the designated private ranges is private, however we have encountered networks where other address ranges were configured as private networks.

2. *Static or Dynamic.* The ISP or organization managing your local network can allocate an IP address to your device for permanent use, or your device can request a new, temporary address every time it boots and connects to the network. Having a permanent, or *static*, IP address allows those devices that send messages to your device to be configured to send messages to that address and not have to discover it. This is useful for routers, servers, and other elements of the network hardware infrastructure but is not common for devices running services or for client devices. Those devices are given a temporary, or *dynamic*, IP address with each boot and new connection to the network. For example, using dynamic IP addresses is essential to efficiently operating wireless networks, allowing you to connect your laptop to the WiFi network at a coffee shop or internet café without having to request a static IP address and then reconfigure your network settings. Dynamic addresses are also essential for cloud computing.

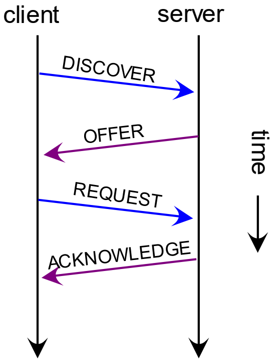
   The DHCP (Dynamic Host Communication Protocol) is used to manage dynamic IP addresses. Every local network has a DHCP server, which is often built into the router. When you boot your device and it connects to the network, it broadcasts a message to its local network. That message is understood by the DHCP server on the local network, and the DHCP protocol defines how your device and the DHCP server exchange messages to establish your device's presence on the local network. Figure 3.3 shows the interactions between a client and the DHCP server.

Note that a complete static or dynamic configuration includes other parameters beyond just the device's IP address, such as the range of addresses that are considered local to the network and the address of the router used to send messages to addresses outside local network.

Given only an IP address, there is no way to determine whether the address is static or dynamic. You must inspect the configuration of the network interface, which will either have a static IP address as a configuration parameter or be configured to use DHCP.

These two dimensions can be combined to characterize how a device is addressed. Private dynamic IP addresses are ubiquitous–if you are reading this on an internet-connected device, it probably has a private dynamic IP address. Private static IP addresses are used for the network infrastructure devices on a private network. Public dynamic IP addresses are typically assigned by ISPs to their subscribers. As noted above, dynamic configuration includes other parameters beyond IP address, which eases management of these very large networks. Finally, public static addresses are essential for internet-level infrastructure elements such as the DNS servers that we discuss below.

Figure 3.3: Interactions between a client and a DHCP server[8]



[8] By Gelmo96 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=38179484

3.3.2 Message Delivery

We discussed how messages are delivered on a local network. Messages sent by a device on one local network to a destination device on another network using the Internet Protocol (either IPv4 or IPv6) must pass through one or more routers. A distinguishing characteristic of a router is that it has more than one network interface, each connected to a different network and having an IP address on that network. A router receives a message over a network interface connected to one network and sends the message out over a network interface connected to a different network. We do not go into detail about this network of routers (it gets complicated) but there are several points you need to understand:

- Networks are noisy and errors occur. Bits may get lost through electrical interference, through loose connections, and even due to rats chewing on the wires. This results in incorrect messages being delivered or messages being dropped. We will see mechanisms to compensate for these faults when discussing the IP and the TCP protocols.
- Individual routers may fail. All computers fail eventually. Since a router is a computer there is some probability of it failing while it is delivering your message. Messages may be replicated and sent multiple times through different routes. This means that the recipient must recognize that it may already have received a copy of your message.
- Messages take time to be delivered. Messages are moved from one router to another one bit at a time. Although delivery times are fast–nanoseconds if the two routers are close to each other or microseconds if they are further away–it still takes time. This will become important when we discuss coordination across distributed computers.
- The message is routed to the operating system of the recipient. This may involve going through a hypervisor.

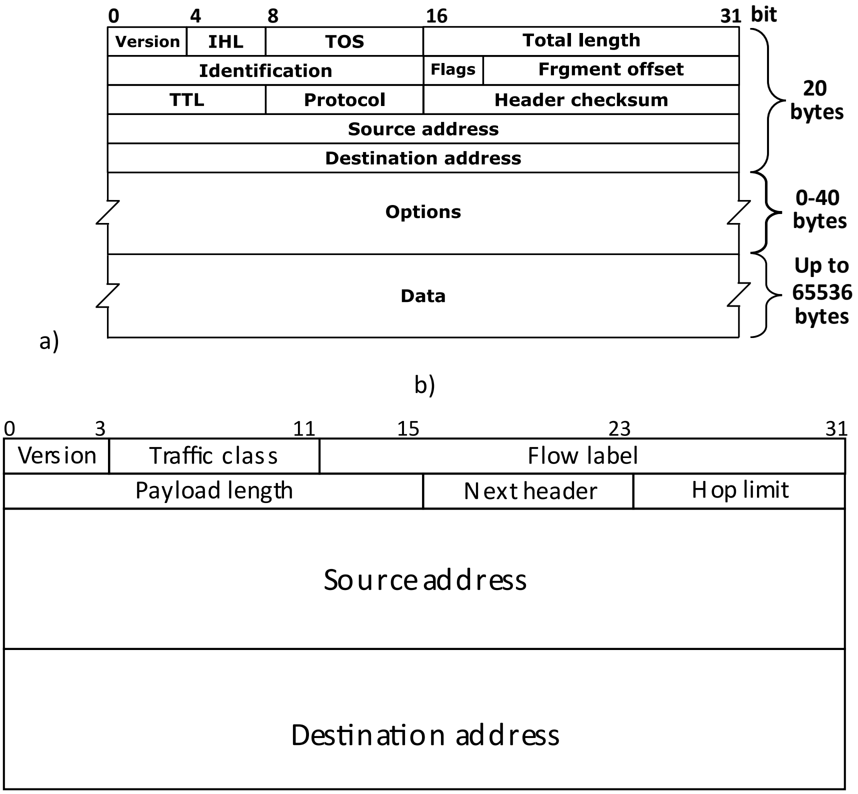
3.3.3 Internet Protocol (IP)

The IP address is used to direct messages to devices, but interpreting the messages requires that there be agreement between the sender and receiver about the formatting and meaning of the various bits in the message. This is the purpose of the Internet Protocol. It specifies that a message consists of two portions: the header and the payload, and it specifies the structure and meaning of the bits in the header. The header specifies such things as the IP addresses of the sender and the recipient and provides instructions to the routers that pass the message along.

The format and meaning of the payload is a matter for the sender and receiver to agree on.

As you might expect, the headers for IPv4 and IPv6 differ. Figure 3.4 shows the headers for the two versions.

Figure 3.4: Headers of IPv4(a) and IPv6(b)



Some of the fields of the header of IPv4 are

- *Version.* This is 4 bits and for IPv4 is always 4.
- *Internet Header Length (IHL)*. This field specifies the size of the header (this also coincides with the offset to the payload). The minimum value for the header is 20 bytes. The maximum value is 60 bytes.

- *Identification.* This is a field used to uniquely identify this message. We mentioned that copies of a message might be delivered to its destination through multiple routes, and so a recipient may receive a particular message multiple times. The identification field is used to detect multiple deliveries of the same message.

- *Differentiated Services Code Point (DSCP).* New technologies are emerging, such as real-time data streaming, which require special treatment of IP messages. The DCSP field identifies the routing policy to be used for this message. An example is Voice over IP (VoIP), which is used for interactive data-voice exchange.

- *Total Length.* The minimum size is 20 bytes (header with no payload data) and the maximum is 65,535 bytes.

- *Fragmentation Information.* The maximum message length in IPv4 is 65,535 bytes. This is not large enough for some payloads so IPv4 has fields that allow for splitting a payload into fragments and then recovering the original payload from the fragments, which may be received out of order.

- *Time to Live (TTL).* An eight bit time-to-live field specifies the maximum number of routers a message can traverse on its way to the destination.

- *Protocol.* This field defines the protocol used in the payload data portion of the message. The Internet Assigned Numbers Authority maintains a list of IP Protocol numbers. TCP, which we will see shortly, is one such protocol.

- *Header Checksum.* The 16 bit checksum field is used to detect bit corruption in the header. When a message arrives at a router, the router calculates the checksum of the header and compares it to the checksum field. If the values do not match, the router discards the message. Errors in the payload must be handled by the encapsulated protocol. When a message arrives at a router, the router decrements the TTL field. Consequently, the router must calculate a new checksum before sending the message out on the next hop.

- *Source Address.* This field is the 32 bit IPv4 address of the sender of the message. Below, we discuss how this address may be changed in transit.

- *Destination Address.* This field is the 32 bit IPv4 address of the receiver of the message. As with the source address, this may be changed in transit.

Some observations about the IPv4 IP message format:

- Note the limited size of the fields. IPv4 was defined in an era where communication over networks was slow and, hence, smaller messages were important. These size limitations, however, are why IPv4 became outmoded and is being replaced by IPv6.
- Note the use of the *Time to Live* field. This is intended to prevent messages from circulating forever in the network. For both IPv4 and IPv6, this is the number of routers a message goes through. Below, we will see that DNS records also have a field called Time to Live that has a totally different meaning and use. The multiple meanings of the name is confusing.
- Note also the allowance for error. IP runs on top of a datalink layer that can be wired Ethernet but could also be a satellite link or long-distance radio link, which are more prone to data corruption. The header checksum is an error-detecting mechanism.

The header for IPv6 is much simpler. It has the following fields:

- *Version.* In this case, it is 6.
- *Traffic Class.* This is like the *Differentiated Services Code Point* of IPv4.
- *Flow Label.* IPv6 does not support fragmented payloads, and this label is used to tell routers to keep messages with the same label on the same path so that they will not get out of order.
- *Payload Length*. The default payload size is limited to 65,536 bytes, which is the same as IPv4. However, IPv6 will allow payloads of up to 1 GB, if supported by the network infrastructure.
- *Next Header.* This specifies where the payload begins in the message, like the IHL field in IPv4.

- *Hop Limit.* This specifies the maximum number of routers a message can traverse on its way to the destination, like the TTL field in IPv4.
- *Source Address.* The 128 bit address of the device that generated the message. As with IPv4 this may be modified during the life of a message.
- *Destination Address.* The 128 bit address of the intended destination. As with IPv4 this may be modified during the life of the message.

Some differences between IPv4 and IPv6 are

- The size of the address space: 32 bits versus 128 bits.
- The IPv4 concept of private addresses is replaced by the IPv6 concept of *unique local addresses*. IPv6 reserves the block from fd00::0 to fdff:ffff:ffff:ffff:ffff:ffff:ffff:ffff to be allocated within a local network without coordination with any higher numbering authority.
- The IPv6 localhost address is ::1 (all zeros, with the least significant bit set to one).
- The payload size by default is the same although IPv6 does allow a much larger payload.
- With IPv4 the possibility for fragmentation of the payload is built into the protocol. With IPv6, any fragmentation of payload is the responsibility of the sender and recipient to negotiate and is external to the protocol.
- IPv6 does not include a checksum on the header. The protocol designers noted that both the lower level datalink protocols (e.g., Ethernet) and the system level protocols (e.g., TCP) included error checking, making this checksum unnecessary. Since, as we discussed above, the IPv4 checksum must be recalculated for each hop, removing the checksum improves router performance.

Although IPv6 was standardized in 1996, adoption has been slow. High performance routers perform part of the message processing in hardware, and the difference in address size means that these routers and transmission devices must be modified to be IPv6 compatible. In addition, the use of private IP addresses has extended the life of IPv4 by decades.

The payload of an Internet Protocol message, whether v4 or v6, will likely include other protocols. We will see this with TCP, which is embedded into the payload of the Internet Protocol message. We will also see further nesting when we discuss tunneling. The ultimate content payload may be nested two or three layers deep into the Internet Protocol payload.

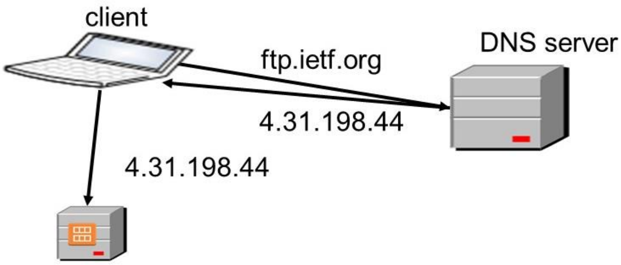
Now we turn our attention to the Domain Name System (DNS).

3.4 DNS

IP addresses are not suitable for most end-use systems. First, they are hard to remember, especially IPv6 addresses. Secondly, and more importantly, in most cases they are not static–we have seen how the IP address for a device may change every time it boots and connects to the network. Finally, we may wish to access an arbitrary member of a collection of servers with different IP addresses but the same logical function.

Logical names are better suited to identify services. This is the purpose of domains and the internet Domain Name System (DNS). DNS is an example of a discovery service. Although it can be used for containers as well as other types of devices, it is not really suitable for containers and we will see discovery mechanisms for containers in Chapter 5 Container Orchestration. You can think of the DNS as a table with two columns: hostnames and IP addresses. We show a simplified view of DNS in Figure 3.5. A hostname identifies a logical resource: a web server, a database server, a file server, or some other service with an API.

Figure 3.5: A Simplified View of DNS



A hostname forms part of the Uniform Resource Locator (URL) that you use in your web browser. For example, https://www.ietf.org/about/ will take you to a page that describes the Internet Engineering Task Force. The hostname is the www.ietf.org and “/about/” is a path within that hostname. It is also possible to add parameters to the path. Your service or browser sends the hostname part of the URL to the DNS and gets back one or more IP addresses. For now, we assume a single IP address is returned and we will deal with the other case later in this section. This IP address is then used to send messages to the web server or other service running on the device at that IP address.

In reality, of course, the DNS is much more complicated both logically and physically than shown in Figure 3.5. We will discuss some of the logical complications but omit most of the discussion of physical complexity. For our purposes, you should realize that the DNS system consists of a large collection of distributed and replicated nodes scattered around the world.

3.4.1 Hostname Structure

Hostnames are the names of the various servers that you may wish to access, and have a structure with the elements separated by “.”s. Although hostnames have a similar structure to IPv4 addresses (elements separated by “.”), there is no correspondence between hostname fields and IPv4 address elements.

Consider the hostname www.amazon.com, which is a website where you could order this book. Hostnames are *resolved* by a series of *lookups*, or requests to a DNS server. The resolution process works from right to left. in this case, we start with the field “.com”. There is a portion of the DNS that knows all the .com domains. This information is stored in the .com DNS servers. So, to resolve the hostname to an IP address, you start by asking the .com DNS server for the address of “amazon.”

If you are paying attention, you realize that we are jumping the gun. How do we know the address of a “.com” DNS server? In our discussion above about static and dynamic IP configuration, we noted that the configuration includes parameters beyond the IP address. One of those parameters is the address of the local DNS server. This server acts as our entry to the global hierarchy of DNS servers. We make our request to our local DNS server, which then starts at the root of the global DNS hierarchy. This root is a list of 13 DNS root name servers at fixed IP addresses, which is managed by the Internet Assigned Numbers Authority (IANA). Although there are 13 IP addresses for root name servers, there are more than a thousand physical servers[9], and the network infrastructure performs load balancing and failover from the 13 root addresses to this set of servers. These root servers maintain the IP address of the DNS servers for the Top-Level Domains (TLDs). Historically, the TLDs were limited to .com, .edu, .org, .net, .gov, and .mil. As the internet has grown, there are more than 3000 TLDs that include country code domains (e.g., .uk, .au, or .jp), and sponsored top-level domains that are administered for narrow communities of interest (e.g., .jobs and .travel).

Thus, the process to start resolving the IP address for the hostname www.amazon.com is to ask a root server for the address of the .com DNS server and then ask the .com DNS server for the address of the amazon DNS server and then ask that server for the address of www. This yields the IP address for www.amazon.com.

3.4.2 Time to Live

If every name resolution on the internet had to start at a DNS root name server every time, the entire internet would come to a halt. Name resolution results are cached at every level in the process: within the networking stack on your device, at your local DNS server, and at each level of the DNS hierarchy. When using a service, we usually send a series of messages to the same destination, so caching the destination address has significant performance benefits. Also, IP addresses for nodes (even dynamically assigned ones) do not change very frequently, so caching is relatively safe. However, addresses do change, and we need a strategy to invalidate the cached information. Each server in the DNS hierarchy includes a Time to Live (TTL) for every response. As we noted earlier, DNS TTL is different from the TTL in the IP Protocols. A DNS TTL is the time that each requestor can assume that the IP address it has requested will be valid. The TTL for a TLD server, for example, is 24 hours. Thus, your local DNS server can cache the IP address of a TLD server and assume that it is valid for 24 hours. If the address of the TLD server changes, it would take 24 hours for that change to propagate throughout the internet.

Servers at each level of the DNS hierarchy may set the TTLs to different values, typically becoming smaller as you get closer to the actual server that you are trying to access. TTLs are set in seconds so that smallest TTL is 1 second. This is important when

[9] See https://root-servers.org/ has a map of the current instances of root servers.

you wish to change a DNS entry to have a different IP address. It means the new mapping from domain name to IP address cannot be effective until old values in caches have expired. One second is too long when dealing with some types of containers where IP addresses might change in milliseconds. We will see other discovery methods in Chapter 5 Container Orchestration.

3.4.3 Using DNS to Handle Overload and Failure

We have sketched a pretty simple picture–you ask the DNS for the address of a host, you get back an IP address, and you send a message to that address. What could go wrong with such a simple scenario?

Computers fail. What happens if the device to which you sent a message does not respond? Its lack of response could be due to its failure, or it could be because it is overloaded and too busy to respond, or it could be a problem with the network connection between your device and the device to which you are sending the message.

In message-based systems such as we are discussing in this book, the main failure detection mechanism is a timeout. That is, if you send a message and a response is not sent in a timely fashion, the recipient is assumed to have failed. Typically, you would retry the message several times before deciding that failure has occurred.

DNS can help a client handle a request timeout. Rather than the DNS server returning a single IP address in response to a resolution request, it may return a list. The list contains the addresses for several distinct devices that are running the service identified by the hostname. Then you would send your message to the first address on the list, and if you don’t get a response, you would send the message to the second address, and so forth.

Devices also get overloaded. One technique for managing overloading is to use DNS-based load balancing. The DNS server can balance the requests by rotating the list of returned addresses it returns to each client. That is, the first query would return IP1, IP2, IP3 and the second query would return IP2, IP3, IP1. Then the requests would be distributed among the replicas of the service. This approach is referred to as *round robin DNS.*

3.5.4 DNS Security

Since the DNS is such an integral portion of the internet, its security is important. We discuss two of the many security mechanisms. First is physical security.

1. Physical security of the DNS root servers is extensive. The following item is taken from the IETF requirements for root servers.[10]

   "The specific area in which the root server is located MUST have positive access control, i.e. the number of individuals permitted access to the area MUST be limited, controlled, and recorded. At a minimum, control measures SHOULD be either mechanical or electronic locks. Physical security MAY be enhanced by the use of intrusion detection and motion sensors, multiple serial access points, security personnel, etc."

2. Security DNS messages. Security extensions to DNS (DNSSEC) strengthen authentication in DNS using digital signatures based on public key cryptography. We discuss these two topics in Chapter 7 Infrastructure Security. With DNSSEC, it's not DNS queries and responses themselves that are cryptographically signed, but rather DNS data itself is signed by the owner of the data.

Now you know how to get a message to a device, but it has still not gotten to the service you wanted to send it to. This is the role of ports and TCP.

3.5 Ports

Figure 3.6 shows an old-fashioned telephone switchboard for a business. All calls came to the operator through a single telephone number, and the operator would ask the caller for the extension that the caller wished to reach, and then the operator would plug the cord into the receptacle for that extension. The phone at the extension would ring and the person for whom the call was intended would pick up their phone and begin speaking to the caller.

[10] [10] https://datatracker.ietf.org/doc/html/rfc2870

Figure 3.6: An Old-Fashioned Telephone Switchboard[11]

Computer systems use the same architecture to deliver messages. The IP address gets the message to the desired device and then the *port number* gets it to the appropriate service. Each service is listening on one or several ports so that when a message for it arrives, it will receive the message and act on it.

As with domain names and IP addresses, the IANA maintains a list of port numbers that are reserved for particular systems. Port numbers 0 through 1023 are reserved for use by internet standard protocols, for example,

- 22 for Secure Shell (SSH)
- 25 for mail service
- 53 for DNS
- 80 for HTTP (hypertext transfer protocol)
- 443 for HTTPS (secure hypertext transfer protocol)

Port numbers 1024 through 49151 are assigned by IANA for other types of services. For example,

- 2181 for the Apache Zookeeper client

[11] https://commons.wikimedia.org/wiki/File:Jersey_Telecom_switchboard_and_operator.jpg#file

- 5432 for the PostgreSQL database
- 9092 for Apache Kafka messaging

Finally, port numbers 49152 through 65535 are *dynamic* or *ephemeral* ports, used temporarily by services for secondary connections.

Some of the assigned port numbers are quite specialized and somewhat surprising. For example, port 17 is reserved for the "quote of the day." Some are clearly historical legacies such as port 5190 for AOL Instant Messenger.

3.6 Bridged and NAT networks

Our discussion of networking has been framed in the context of physical network connections among physical devices. In Chapter 2 Virtualization , we learned that one function of a hypervisor was to share a physical network connection across multiple virtual machines. There are two ways to accomplish this, *bridge* or *NAT*. In each case, the hypervisor creates a small network within the physical machine. This network exists only in software.

NAT maps one IP address into another. Typically, a public IP address is used for a collection of machines and NAT will take an outgoing message from a VM and change the source IP address so that the return comes to the NAT. When the return message arrives, the destination IP address is changed to be the IP address of the originating VM. Externally to the NAT, there appears to be one IP address for the network.

A bridge network connects two local networks so that they appear to be one network externally. Each VM will retain its IP address and a machine on the bridged network will have its IP address available from outside the bridged network. These IP addresses can still be public or private but the gateway that manages private IP addresses is outside of the bridged network.

Type 1, or bare-metal hypervisors (see Chapter 2 Virtualization) typically use bridge networking. From the Networking Layer (IP) viewpoint, it appears that each VM is on the external local network. When a VM starts, it uses DHCP to get an IP address and network configuration. The DHCP request is broadcast on the external local network, and the IP address and network configuration are provided by a DHCP server on that network. The IP address will be private or public, depending on the DHCP server configuration. Each VM can be accessed directly from the external local network and so can deliver services to clients on that network. Messages from the local network go through the bridge on the hypervisor, which routes them to the appropriate VM.

Type 2, or hosted hypervisors, typically use NAT networking because the host machine must manage services other than the VM. In this configuration, the hypervisor creates an IP subnet. This subnet exists only within the host computer, and each VM is connected to this subnet. The hypervisor provides a DHCP server so that when a VM starts and requests an IP address and network configuration, it receives appropriate parameters to connect to the subnet, including a private, dynamic IP address. Messages between subnets must pass through a router, so the hypervisor provides a routing service between the VM subnet and the physical device connection to the external local network. Finally, because the VM subnet IP addresses are private, the hypervisor provides a NAT service for all messages sent to the external network.

Recall from Chapter 2 Virtualization that one of the use cases for hosted hypervisors is to allow users to run different operating systems on a single workstation computer. These computers are often connected to corporate networks that include security measures such as *network admission control*, where DHCP servers are configured to ignore requests from unknown computers. Or, the computer may be connected to the corporate network through a VPN connection. In each of these cases, the bridged configuration will not work: Either the DHCP server will not recognize the new VM as a known device, or each VM would require its own VPN credentials. In these scenarios, the NAT network approach allows each VM to appear to the external network to be (more or less) just another program running on the user's workstation.

The networking services provided by a container engine are similar to those provided by a hypervisor, however the labels for each configuration can be confusing.

Container engines such as Docker provide *bridge* networking, which allows each container to expose and map its ports to ports on the host's network interface. This acts somewhat like a firewall, in that only explicitly mapped ports on the container are accessible from the external local network, reached using the host's IP address and the port number that the container port is mapped to.

Docker bridge networking also creates a subnet within the host, and each container receives a private IP address on this subnet and can communicate directly to other containers on the host over the bridge subnet. In fact, you can create multiple bridge subnets and assign containers to these subnets to isolate one group of containers from another group of containers on the same host. This begins to look a bit like NAT networking on a hypervisor. As we said, the labels are confusing.

Because containers share the same operating system, container networking capabilities are more limited than hypervisor networking. For example, if you have two web servers and run each in a VM on a bare-metal hypervisor, then each VM gets its own IP address. Web servers usually run the HTTP protocol on port 80, and since each web server is uniquely identified by its IP address on the local network, there is no conflict and clients can access both services over the usual HTTP port number. However, if you want to run these two web servers in containers on the same host, then both containers are exposed to the local network using the host's IP address, and only one of the containers can map its web server service to the host's port 80. You could map the other container to use port 8080 (registered with IANA as an alternative to using port 80 for HTTP), but then clients may have to be modified to use this port. And what if you have more than two web servers? There are workarounds: In this scenario, you could add a third container that runs an HTTP forwarding proxy and map this container to host port 80. Based on the HTTP request, the proxy container would forward the request to one of the two original web servers over an internal bridge subnet within the host. This gets complicated and illustrates that no single infrastructure technology is right for every problem.

Container instances share an operating system. The operating system must be compatible with the container runtime, which limits the software that can run on a container. The container runtime starts, monitors, and restarts the service running in a container. The container runtime can start and monitor only one program in a container instance. If that one program completes and exits normally, execution of that container ends. For this reason, containers generally run a single service (although that service can be multithreaded). Furthermore, one of the benefits of using containers is that the size of the container image is small, including only the programs and libraries necessary to support the service we want to run. Multiple services in a container could bloat the image size, increasing the container startup time and runtime memory footprint. As we will see in Chapter 5 Container Orchestration, we can group container instances running related services so that they will execute on the same physical machine and can communicate efficiently using the container runtime's bridge network. Some container runtimes even allow containers within a group to share memory and coordination mechanisms such as semaphores.

Also, a service receives requests on a port of the service’s IP address. Ports used by services in one VM cannot conflict with ports used by services in another VM running on the same hypervisor, because every VM gets its own IP address that is bridged to the local network. On the other hand, containers that run on the same pod (we will discuss pods later in this chapter) share a single externally visible IP address. The container runtime creates a bridge network that connects the containers on the machine to each other and uses network address translation (NAT) to allow the containers to share the single external IP address. As a result, container-based services running on the same machine can have port conflicts. For example, if there are two web servers running in containers on the same machine, each web server will try to use port 80 to receive requests. The first web server to claim the port will be successful and the second one will fail to initialize. When delivering your services in containers, you must design your own service *and* you must also design how your service will be grouped with other services to avoid port conflicts and take advantage of efficient bridge networking.

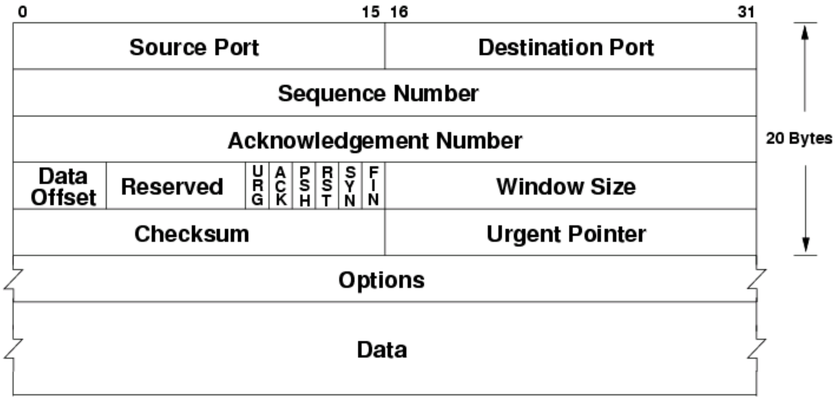
3.7 TCP

Just as IP addresses required a protocol such as IPv4 to define how to interpret messages, ports have several protocols that enable interpretation at the port level. The most common port-based protocol is the Transmission Control Protocol (TCP). TCP works at the Transport layer, as shown in Figure 3.1, between the Network layer provided by the IP and the Application layer. A system sends a series of messages to its TCP layer, which uses the Network and Datalink layers to communicate with the receiver’s TCP layer. Together, the sender’s and receiver’s TCP layers provide reliable,[12] ordered, and error-checked delivery of messages from the sending system to the receiving system.

Like the IP protocol, a TCP message has a header and a payload. Before going into details of the TCP header, we will discuss the reliable, ordered, and error checking aspects. Figure 3.7 shows the header for the TCP protocol.

[12] In computer networking, the term *reliable* means that a protocol can detect message-delivery failures. It does not mean that messages are always delivered. Similarly, *unreliable* means that delivery failures cannot be detected, and does not mean that the protocol itself is untrustworthy.

Figure 3.7: Header for the TCP protocol



Every TCP message has a sequence number associated with it. This sequence number is used to ensure that messages are delivered to the receiving system in the order that the sender intended. The sequence number is also used as a portion of the acknowledgement (ACK) by the destination.

A sending TCP layer sends a message with a sequence number and then waits for an ACK from the receiving TCP layer. If the ACK does not arrive in a timely fashion (i.e., a *timeout* occurs), the message is resent by the TCP layer. The destination will discard the second sending of a packet if it has already received the first sending. Thus, sequence numbers are the means for both reliability and for ordering of TCP packets.

The TCP header also includes a checksum that is used for error detection. The IP checksum that we discussed above is computed only over the IP header and will not detect errors in the IP payload that contains the TCP header and TCP payload. The TCP checksum is computed over the TCP header *and* the TCP payload to detect errors in transmission. If the checksum indicates that an error occurred in transmission, the receiving TCP layer discards the message and does not send back an ACK. Thus, after the timeout, the sender sends the message a second time.

There are optimization techniques that may reduce the number of ACKs required so that every message may not generate an ACK.

From this, we see that a TCP connection is reliable and error checked, since if a message is not delivered or is corrupted, we will eventually see an ACK timeout for that message. Furthermore, messages are delivered to the receiving service in the same order as they were sent by the sending service. However, if you are a service developer, these TCP features may not be good enough to meet your system's needs. For example, the exact value of the TCP ACK timeout varies across operating systems, but it is usually in the range of 75 to 100 seconds. Messages may be re-sent three times by the TCP layer before it notifies the sending Application layer of an error, so several minutes may pass before a network failure or destination service failure is detectable by the sending Application layer. The receiving TCP layer must deliver the messages to the destination service in order, so if one message is lost, then no later messages will be delivered by the TCP layer until the lost message is retransmitted and received. For these reasons, many services choose to implement Application-level message ordering and acknowledgement rather than relying on TCP, so that failures can be detected more quickly and appropriate actions can be taken; for example, re-directing requests and notifying interactive users.

In addition to the sequence number and checksum, the header of the TCP packet includes a source port and a destination port. Just as the service that is receiving the packet has a port on which it is listening, the service that is sending the packet is also listening on a particular port. Thus, both the sender and the destination ports are necessary. Often, the sender port number is a dynamic or ephemeral port, selected automatically by the sender's TCP layer.

As with IP addresses, the port numbers in a TCP header may be changed by a device on the network, while the message is *en route* to its destination.

3.8 Structuring your network

3.8.1 Subnets

A subnet is a group of devices that are connected directly to each other. This is the type of network we described in the introduction to this chapter: Any message sent on a subnet can be seen by all devices. Messages between devices on a subnet do not pass through an IP router.

One reason for creating a subnet is network performance. If your local network grows and performance begins to suffer because of excessive traffic, you should define subnets within your local network.

Another reason for creating a subnet is for security. A firewall for the subnet can verify that accesses to the resources on that subnet are allowed.

A subnet is also a logical division of the IP address space. By grouping IP addresses into ranges and arranging those blocks hierarchically, we can look at part of the address and decide which router we should receive a message to get the message closer to its final destination. This greatly simplifies the processing that a router must perform and reduces the memory needed for routing tables, which reduces cost and speeds up message delivery.

A subnet is created within an address space by making the first portion of the IP address (the *subnet prefix*) the same for all the devices in the subnet. The remaining portion of the IP address will identify which device in the subnet receives a message.

For example, suppose as a system administrator you are given a block of IP addresses that begins with 71.229.83.xxx You can assign any IP address that starts with these numbers to the various devices under your control, e.g., 71.229.83.1, 71.229.83.2, etc.

Each subnet has a *subnet mask*. This is a binary number that is logically ANDed with an IP address to get the subnet prefix. In our example, the subnet mask is 255.255.255.0. This mask, when ANDed with an IP address for a device in the subnet will yield the subnet prefix, 71.229.83.0. An alternate notation for representing a subnet mask is called CIDR notation, because it was introduced in the specification of the Classless Inter-Domain Routing protocol. CIDR notation appends a slash and number to an IP address, where the number represents the count of leading ones in the subnet mask. In our example, the subnet mask of 255.255.255.0 has 24 leading ones, and so we would represent one of our IP addresses in CIDR notation as 71.229.81.1/24. Given the length of an IPv6 address, this notation is much more convenient than explicitly specifying the subnet mask, and so IPv6 subnetting uses CIDR notation exclusively. Finally, note that subnet masks do not need to end on 8 bit boundaries. Historically, IPv4 subnet masks were either /8, /16, or /24, however, there is no need

to restrict subnet masks to those values. For example, AWS cloud services use IPv4 subnet masks that include /17, /23, and /26.

3.8.2 Partitioning your network

One reason to partition networks is to improve performance, using subnets as described above. Other reasons why your organization structures its network are protection and logical simplification to improve manageability. The structuring mechanisms in this case are specialized devices that are in the path of messages into, and possibly out of, the organization’s network. These specialized gatekeeper devices have a variety of names–firewall, NAT, proxy server, gateway–but they all serve to monitor and control network traffic into and out of the local network. They sit between the local network and other networks. In practice, these terms may be used interchangeably. Technically, however, they have slightly different definitions. Definitions for these terms are given in Section 3.1. These functions may also be combined into a single computer system.

In theory, every device on your network can have a static public IP address, be registered in the internet DNS, and be accessible directly from the internet. In practice, your organization wishes to restrict access to some or all of these devices for reasons of security and privacy.

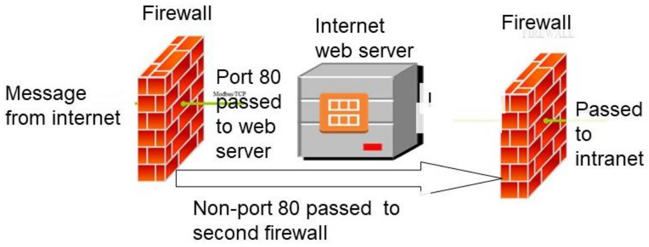
A message from outside the local network intended for a device inside the network will go through a screening gatekeeper. The basic screening gatekeeper is called a *firewall*, and it is often part of the router at the edge of the network. The default behavior of a firewall is to block messages from entering the network. To allow certain messages to enter the network, the firewall *rules* must be configured to open a hole. The most basic configuration uses the IP address of the message sender and the IP address and port of the message recipient. The configuration rules can use source blacklists (accept all messages except those from listed addresses) or source whitelists (reject all messages except those from listed addresses).

The rules are not necessarily static. For example, a device may be put on a temporary blacklist because it has been sending too many messages through the firewall. In this case, the source device may be labelled as possibly being part of a distributed denial-of-service attack and then it will be blocked for some period.

More sophisticated firewalls examine the content of the payload, using techniques such as stateful packet inspection (SPI) to look for known types of malware and attack patterns. The examination of the message content is limited because the payload may be encrypted. In this case, the firewall can be configured to accept (or deny) such messages.

Organizations frequently organize their network into several zones. The simplest division is a publicly visible portion that can be reached from the internet, and a private portion (an intranet) with restricted access. This is shown in Figure 3.8. The public portion is where, for example, an organization's external facing web server or API server might be located. When you access the organization's web page, it is served by the public-facing web server. This type of structure is often called a demilitarized zone (DMZ) or perimeter network and is defined by firewalls on each side of the zone. The rules for the internet-facing firewall in the public zone are generally relaxed to allow broad access to the public-facing resources. They also use a whitelist to restrict access to server ports used for management and monitoring only to addresses inside the private zone. The rules for the intranet zone firewall are very restrictive, whitelisting the addresses only of certain servers in the public zone, and allowing access from the whitelisted addresses only to very limited resources within the intranet.

Figure 3.8: Firewalls used to create a DMZ



Firewalls may also be used between subnets within the intranet both for security reasons and for to enforce organizational policies. Sometimes, policy or regulations require that one part of the organization cannot share information with another part of the organization. Separating such groups with a firewall allows network traffic between members of these groups to be monitored, both to prevent others from seeing the information they generate and for forensic purposes in case a breach occurs.

A firewall can also restrict the visibility of certain types of failures. Suppose a device on your local network misbehaves and spews out messages in an uncontrolled fashion. If that device is on a portion of the intranet isolated by a firewall, the damage it can do is contained. For organizations that operate integration and staging environments within their intranets (we discuss more about these environments in Chapter 11 Deployment Pipeline) these environments are usually isolated by firewalls so that the behavior of the software under test does not affect the rest of the organization.

Organizations may also wish to keep their individual device IP addresses and network topology invisible from the broader internet. This is one purpose of a NAT device. Like firewall devices, NAT devices are often part of a router. Recall that a TCP message header includes the sender's IP address and port number. A service inside your network that is sending a message to a service on the internet fills these fields with its private IP address and a port number. As the message leaves your network, the NAT device replaces the source address and port number with the public IP address of the NAT device, and a unique port number. The unique port number is used to direct return messages from the internet (e.g., TCP ACKs) to the right service on the intranet. Recall that when we discussed IP and TCP, we said that the source or destination address may be modified while a message is in transit. This is one example of this modification.

Networks using NAT cannot support certain types of protocols. For example, in many video conferencing protocols, each side of the video call tries to open a direct connection to the other side of the call to control the call and deliver a video stream. If either side of the call is behind a NAT device, it cannot be directly reached from outside the local network. This has led to *NAT traversal protocols*, where both sides push their streams to a publicly reachable service, and each pulls the other side's stream from that service, so that each side is connecting only *out* from its network.

In contrast to firewalls, which require careful configuration, NAT device operation is simple and generally requires no configuration.

Using the network model shown in Figure 3.1, we see that firewall devices operate mostly at the Networking Layer, with some impact on the Transport Layer, and NAT devices operate mostly at the Transport Layer. There is another type of gatekeeper device, which operates at the Application Layer and provides some features that are similar to those provided by firewalls and NAT devices. These devices are usually called *system proxy servers*, with the HTTP proxy probably being the most common. All HTTP requests to services outside the local network, say from a web browser or to a service API, are sent to the HTTP proxy server. The proxy server, like a firewall, can be configured using blacklists and whitelists to filter the request based on the destination. HTTP proxy servers perform NAT functions, and the proxy server can also modify parameters in the HTTP protocol header to improve privacy.

Proxy servers are configured separately from firewalls, so there may be configuration conflicts. For example, a firewall may permit messages to be sent to a destination IP address that is restricted by the proxy server, or the proxy server may allow messages to be sent that are blocked by the outgoing firewall.

Tunneling is a technique to pass messages through the firewall without being rejected.

3.8.3 Tunneling

You have probably heard of and used a VPN (Virtual Private Network). A VPN allows you to access an organization’s private network from the public internet. VPNs work by using a tunneling protocol. The term *tunneling* is derived from the fact that your messages on a tunneled network are isolated and protected from the rest of the traffic on the internet just as a tunnel isolates and protects you from seawater.

Tunneling takes advantage of the DMZ arrangements of firewalls shown in Figure 3.8. Inside of the DMZ is not only your organization’s web server but also a tunneling server. We will use VPN for concreteness but other tunneling protocols exist.

The sequence then proceeds as follows:

- You are using a device that is on the internet, possibly a mobile device.
- You access the VPN server through the internet and supply your credentials.
- These credentials are validated either by the VPN server itself or by the VPN server accessing a service in the intranet to validate your credentials.
- After your credentials have been validated, the VPN tunneling protocol is initiated between software on your device and software within the organization. This protocol takes your entire message, including the IP header that you created for the message, and encrypts it and puts it into

the payload portion of a TCP message that passes across the VPN connection, through the firewall.

- The VPN server removes the TCP payload and decrypts your message, including the IP header that you created, and then sends the message onto the network into your intranet. If your destination IP address is a device within your organization, the message is delivered over your organization’s intranet. If your destination IP address is outside your organization, your message will be sent out to the internet through your organization’s NAT and firewall, which provides monitoring and security.

There are a variety of different tunneling protocols with different levels of security. Some protocols require encrypted payloads. Some protocols use TLS (Transport Layer Security). We will discuss encryption and the handshake used in TLS in Chapter 7 Infrastructure Security. For now, just be aware that if you are providing VPN access to your network, security is an overriding concern.

3.9 Summary

Networks are one of the pillars of modern computing environments. Devices on the internet have IP addresses and typically communicate over networks using messages encoded with the IP and TCP protocols.

IP addresses are unwieldy and subject to change, and hostnames are usually used to identify remote resources. The DNS is used to convert a hostname to an IP address. DNS resolves hostnames from right to left: The rightmost parts of a name are controlled by regional authorities and the leftmost elements are controlled by ISPs and large organizations. The DNS also can implement a form of load balancing to spread requests across different devices providing the same service.

IP addresses will be routed to a device but not to a service. That is the function of ports. A message goes first to a device and then to a port being listened to by a service. The TCP protocol is used to encode messages sent to services.

Containers are limited somewhat in what network connection they use because ports are managed by the local operating system and containers using the same runtime engine use only a single instance of the operating system.

Subnets are used to segment the IP addresses that an organization must control. They are a collection of IP addresses with the same prefix. Their purpose is to provide an organizational mechanism for a system administrator and to reduce congestion on the local network.

Firewalls are a mechanism to protect an organization’s local network from intruders coming from the internet. One use of firewalls is to establish a DMZ of publicly available servers. A DMZ or perimeter network separates an organization’s devices and networks into a controlled area that is open to the internet, and a private area.

Firewalls, in conjunction with tunneling protocols, can be used to implement VPN for remote access to an organization’s intranet.

3.10 Exercises

1. Create a virtual machine as in Chapter 2 Virtualization and display the IP addresses. Categorize the different addresses according to the public/private/reserved categories we discussed.
2. Create two virtual machines and ping one from the other.
3. Create two containers and ping one from the other.
4. Examine a packet sent to your machine. Decode the fields of the IP header and the TCP header. There are a variety of tools to enable this, such as TCPdump and Wireshark.

3.11 Discussion Questions

1. Draw an organization chart of the ICANN with the responsibilities of each portion.
2. We did not discuss the hardware topology of the internet in this chapter. Look at the major cables (https://www.submarinecablemap.com/) and identify who owns the cables that serve your area. Are there any surprises?
3. Examine the network architecture for your department. How many firewalls are there? Do you have a DMZ? Can you VPN into your local network?
4. You send a message from behind a NAT device out to the internet. The NAT device replaces the source IP address with its own address. This means that the response goes to the NAT device, not to you. How does the NAT device know how to forward the message to you as opposed to some other device in your intranet?

5. NAT networking is very complicated compared to bridged networking and is less efficient since there is an extra NAT and router. Why is it used?
6. A dynamic IP address makes it more difficult for the ISP’s subscribers to expose services, even though the address is public. Why? What is dynamic DNS and how does it work around these issues?

Chapter 4 The Cloud

Virtualization and networking are 20th century inventions. We now move into the 21st century with a discussion of the cloud.

At the end of this chapter you should know

- How the cloud manages scaling
- The communication patterns for the cloud
- Why you need to be concerned about availability for your services
- How messages are routed to instances of VMs
- How VMs or containers can coordinate to share data
- How the choice of where to keep state affects computations.
- How to manage security policies in the cloud.

4.1 Coming to Terms

Availability zone– isolated locations within data center regions from which public cloud services originate and operate

Autoscaler – a cloud service that automatically allocates or deallocates resources.

Data center – a facility that centralizes an organization’s shared IT operations and equipment

IaaS – Infrastructure as a Service.

Load balancer a service that distributes requests among a collection of servers all executing the same software.

Security policy – **set of rules organizations adhere to that helps ensure safe and secure operations in the cloud**

PaaS – Platform as a Service

Region – the geographic region where a collection of cloud resources are located.

SaaS – Software as a Service

3.2 Structure

We use the term *the cloud* to refer to a wide range of computing capabilities. You might say, “All my photos are backed up to the cloud.” In this case, “the cloud” means something like “on someone else’s computers, accessible over the internet.” This use demonstrates several aspects of the more general use of the term “cloud.”

- You pay only for what you use.
- The storage service is *elastic*, meaning that it can grow or shrink as your needs change.
- Your use of the cloud is *self-provisioned*: You create an account and can immediately begin using it to store your photos.

The computing capabilities delivered from the cloud range from systems such as the photo storage example to fine-grained services exposed through APIs (e.g., text translation or currency conversion), to low level infrastructure services such as processors, network, and storage virtualization. In this chapter, we will focus on how to use infrastructure services from the cloud to deliver the higher-level services that you develop. We will discuss how the cloud provides and manages virtual machines.

Public clouds are owned and provided by cloud service providers. These are organizations that provide infrastructure services to anyone who agrees to the terms of service and can pay for use of the services. In general, the services you build in this infrastructure are accessible on the public internet, although you can use mechanisms such as firewalls that we discussed in Chapter 3 Networking to restrict visibility and access. Some organizations operate a *private cloud*. A private cloud is owned and operated by an organization for the use of members of that organization. An organization might choose to operate a private cloud because of concerns such as control, security, and cost. In this case the cloud infrastructure and the services developed on it are visible and accessible only within the organization’s network. A mixed model is the *hybrid cloud* approach, running some workloads in a private cloud and other workloads in a public cloud. A hybrid cloud might be used during a migration from a private cloud to a public cloud, it might be used because some data are legally required to be subject to greater control and scrutiny than is possible with a public cloud, or it might be used to enhance availability.

From a service developer's perspective, there is not much difference between private clouds and public clouds, so we will focus our discussion on public infrastructure-as-a-service clouds.

A typical public cloud data center has tens of thousands of physical devices, closer to 100,000 than 50,000. The limiting factor on the size of a data center is the electric power it consumes and the amount of heat that the equipment produces: There are practical limits to bringing electrical power into the buildings, distributing it to the equipment, and removing the heat that the equipment generates. Figure 4.1 shows one rack of a typical cloud data center. One rack consists of upwards of 25 computers (each with multiple CPUs) depending on power and cooling available. The data center consists of rows and rows of such racks, with high-speed network switches connecting the racks. In Chapter 3 Networking, we said that message latency is on the order of nanoseconds when the source and destination are "close," which equates to "in the same rack." On the other hand, sending a message from Europe to California takes approximately 150 milliseconds.[13] These delays become important when you need to coordinate actions across separated devices, as we will see later in this chapter.

When you access a cloud from a public cloud provider, you are accessing data centers scattered around the globe. The cloud provider organizes its data centers into regions. A cloud region is both a logical and physical construct. Since the services you develop and deploy to the cloud are accessed over the internet, cloud regions can help you be sure that your service is physically close to its users, reducing the network delay to access the service. Also, there may be regulatory constraints. E.g., General Data Protection Regulation [GDPR], as we will discuss in Chapter 14 Secure Development, that restrict transmitting certain types of data across national borders, and cloud regions help you comply with these regulations. A cloud region has many data centers that are physically distributed with different sources for electrical power and internet connection. The data centers within a region are grouped into *availability zones* such that the probability of all data centers in two different availability zones failing at the same time is extremely low. Choosing the cloud region that your service will run on is an important design decision. When you allocate a new VM, you may specify which region the VM will run on. The availability zone may be chosen automatically, but you often want to

[13] Electrical signals travel at nearly the speed of light, which results in a latency of about 1 nanosecond per foot of separation. There is additional latency introduced by the networking computers – routers, firewalls, etc.

also choose the availability zone yourself, as we will see when we discuss availability and business continuity in 15 Disaster Recovery.

Figure 4.1: A Cloud Data Center Rack[14]

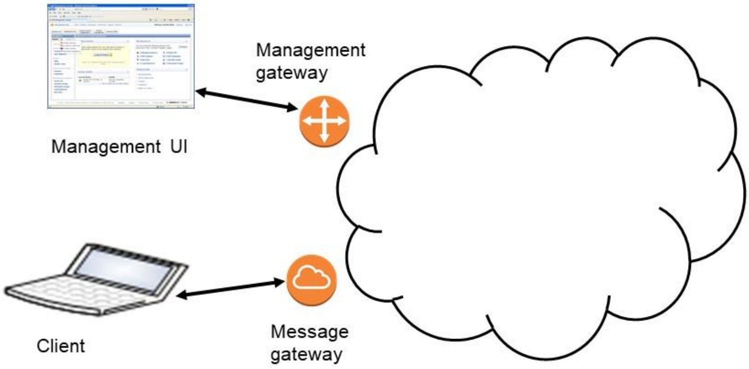
All access to a public cloud is over the internet. There are two main gateways into a cloud–a management gateway and a message gateway. Figure 4.2 shows these two gateways. The message gateway is a router, discussed in Chapter 3 Networking, so here we focus on the management gateway.

Suppose you wish to allocate a virtual machine in the cloud. You send a request to the management gateway asking for a new VM instance. The request has many parameters, but three essential parameters are the cloud region where the new instance will run, the instance type (e.g., CPU and memory size), and the ID of a

[14] By Jfreyre– Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2817411

VM image. The management gateway is responsible for tens of thousands of physical computers, and each physical computer has a hypervisor that manages the VMs on it. The management gateway will ask each hypervisor whether it can manage an additional VM of the type you have selected–is there enough unallocated physical CPU and memory capacity available on that physical computer to meet your needs? It will select one of the positive responses and ask that hypervisor to create an additional VM and return its IP address to the management gateway. The management gateway then returns that IP address to you. Note that the cloud provider always ensures that there is enough physical hardware available in its data centers so that your request will never fail due to insufficient physical resources.

Figure 4.2: Gateways into a Public Cloud



The management gateway not only returns the IP address for the newly allocated VM, it also returns a hostname. We discussed the structure of a hostname in Chapter 3 Networking, and pointed out that the leftmost fields in hostname are controlled by the local organization. The hostname returned after allocating a VM reflects the fact that the IP address has been added to the cloud DNS server.

There are several points to note here:

- This is a simplified conceptual description of how a new VM is allocated. The actual interactions between the management gateway and the individual hypervisors are more complicated than we have just described. For example, the management gateway does not actually poll all hypervisors for available capacity every time a new VM is allocated. For our purposes, however, this simplified description is sufficient to understand the function of the management gateway.

- The VM image used to create the new VM instance can be any VM image. It could be a simple service or it could be one step in the deployment process to create a complex system.
- The hypervisor does not generate the IP address for the new VM instance, it asks an IP address manager within the cloud for an available IP address. The IP address manager is one of the many infrastructure services provided by the cloud that is invisible to the service you are developing. We will see others.

The management gateway performs other functions in addition to allocating new VMs. It allows for the deletion of a VM and for collecting billing information about the VM, and it provides monitoring capability for the VM.

The management gateway is accessed through messages over the internet to its API. These messages can be from another service, such as a deployment service, or from a command line program on your computer (allowing you to script operations). The management gateway can also be accessed through a web-based system operated by the cloud service provider; however, this interactive interface is not efficient for more than the most trivial operations.

The management gateway also allows you to manage the virtual network that connects your VMs. The terminology and capabilities for network virtualization varies somewhat across public cloud providers. Most have a concept called a virtual private cloud (VPC). A VPC is, as its name suggests, is a collection of VMs that are isolated from other VMs in the cloud. A VPC acts like a private network. VMs created in a VPC are assigned private IP addresses, and the VMs are not reachable from outside that VPC, unless you specifically request a public IP address for a VM. You can create subnets within a VPC, with firewalls between the subnets to implement a DMZ topology, as we described in Chapter 3 Networking. A VPC may be created with restrictions on which users can create VMs in the VPC. This is essential for securing the services running in the VPC and allows organizations to enforce accountability and allocate spending in the cloud. Suppose, for example, that you or your organization wish to separate the resources used by the system group to do development from the resources used by the financial group to manage billing and payroll. Each group could have its own VPC. Access controls can ensure that only members of the appropriate group can perform actions that incur costs in each VPC. Cloud provider features such as billing tags allow the organization to separate the actual resource costs for each VPC. This is one approach to allow an organization to impose limits and controls on the unrestricted self-provisioning of most public cloud providers. We continue with this topic in Section 4.8 on Security Policies.

4.3 Service Models

The three most popular services that the cloud provides to its customers are Infrastructure as a Service, Platform as a Service, and Software as a Service. These definitions are taken from a report by the U. S. National Institute of Science and Technology.[15]

1. Infrastructure as a Service (IaaS). The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and systems. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed systems; and possibly limited control of select networking components (e.g., host firewalls).
2. Platform as a Service (PaaS). The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired systems created using programming languages, libraries, services, and tools supported by the provider.3 The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed systems and possibly configuration settings for the system-hosting environment.
3. Software as a Service (SaaS). The capability provided to the consumer is to use the provider's systems running on a cloud infrastructure2 . The systems are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating

[15] https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

systems, storage, or even individual system capabilities, except for limited user specific system configuration settings.

4.4 Failure in the Cloud

When a data center contains 100,000 or more physical computers, failure of one or more of them every day is highly probable. Amazon reports that in a data center with around 64,000 computers, each with two spinning disk drives, approximately 5 computers and 17 discs will fail every day[16]. Google reports similar statistics[17]. In addition to computer and disk failures, network switches can fail, the data center can overheat causing all the computers to fail, etc. Thus, although your cloud provider will have relatively few total outages, the physical computer your VM is running on may fail. If availability is important to your service, you need to think carefully about what level of availability you wish to achieve and how to achieve it. Below, we discuss load balancers as one way to improve availability, and we will see other approaches when we discuss microservices.

In a distributed system, timeouts are used to detect failure. There are two consequences that follow from using timeouts.

1. You can’t distinguish between a failed node or broken network connection and a slow reply to a message that exceeds the timeout period. This causes you to label some slow responses as failures.
2. A timeout will not tell you where the failure or slowness occurs. Many times, a request to a service triggers that service to make requests to other services, which make more requests. Although each of the responses in this chain may have a latency that is close to the expected average response time, when we add all the latencies together some of the overall responses may be a little slow, and some may be very slow.

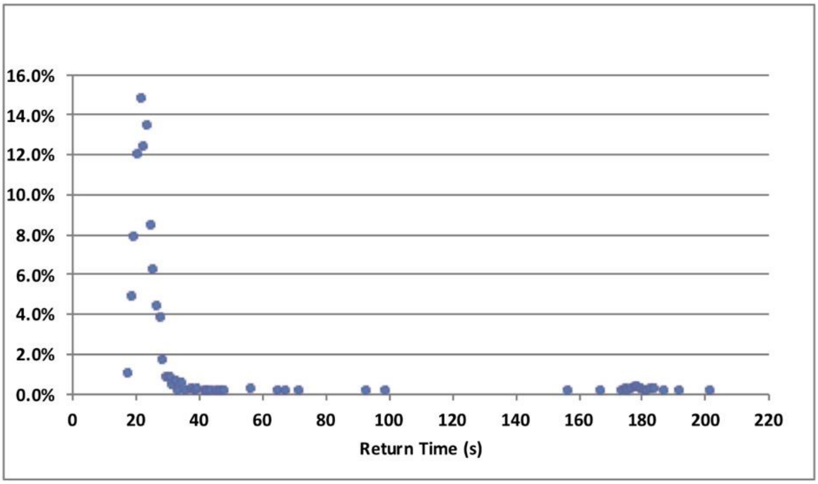
The response to your original request may exhibit what is called long tail latency. Figure 4.3 shows a histogram of the latency of 1000 “launch instance” requests to Amazon Web Services (AWS). Notice that some requests took a very long time to satisfy. When evaluating measurements such as this one, you must be careful which statistic you use to characterize the data set. In this case, the histogram peaks at a latency of 22 seconds, however the average latency over all the measurements is 28 seconds, and the median latency (half the requests are

[16] https://www.slideshare.net/AmazonWebServices/cpn208-failuresatscale-aws-reinvent-2012

[17] https://static.googleusercontent.com/media/research.google.com/en//people/jeff/stanford-295-talk.pdf

completed with latency less than this value) is 23 seconds. Notably, even after a latency of 57 seconds, 5% of the requests have still not been completed (i.e., the 95th percentile is 57 seconds). So, although the mean latency for each service-to-service request to a cloud-based service may be within tolerable limits, a significant number of these requests can have much greater latency–in this case, more than 2 times longer but sometimes 5 to 10 times longer than the average. These are the measurements in the long tail on the right side of the histogram. Long tail latencies are a result of congestion or failure somewhere in the path of the service request. There are many possible contributors to congestion–server queues, hypervisor scheduling, or others–but the cause of the congestion is out of your control as a service developer. Your monitoring techniques and your programming techniques must reflect the possibility of a long tail distribution. We return to this topic in Section 12.7.3 Protecting Against Failure

Figure 4.3: Long Tail Distribution of 1000 “Launch Instance” Requests to AWS



4.5 Scaling Service Capacity

When a service receives more requests than it can process within the required latency, the service is overloaded. This can occur because there is insufficient I/O bandwidth, CPU cycles, memory, or some other resource. We mentioned that when allocating a VM, you specify an instance type. This defines the resource limits for a VM instance. In some cases, we can resolve a service overload issue by running the service in a different VM instance type that provides more of the resource that we need. This approach is simple–we don't need to change the design of our service, we just run it on a larger virtual machine. This approach is called *vertical scaling* or *scaling up*.

There are limits to what we can achieve with vertical scaling–there may not be a large enough instance type to support your workload. In this case, *horizontal scaling* or *scaling out* provides more resources of the type needed. Horizontal scaling involves having multiple copies of the same service and using a load balancer to distribute requests among them. We have already seen one example of load balancing–using round-robin DNS to distribute requests to a list of machines rather than a single machine. Using DNS for load balancing is both cumbersome and limited. This is not the main purpose of DNS, and so having a distinct load balancer provides more control. Load balancers can be standalone systems or bundled with other functions such as proxy. Load balancers define the communication patterns for the cloud that we identified in Chapter 1 Platform Preliminaries as one of the issues associated with distributed computing. They also are responsible for health checks for the instances of the system – another of the issues we identified. We divide our discussion into two main portions: how load balancers work and how services that sit behind a load balancer must be designed to manage service state. Then we discuss health management and how load balancers can improve availability.

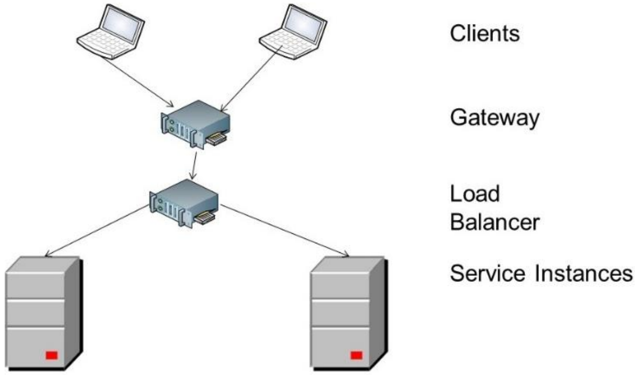
4.5.1 How Load Balancers Work

A load balancer solves the following problem: There is a single instance of a service running on a VM, and there are too many requests arriving at this instance for it to provide acceptable latency. One solution is to have multiple instances of the service and distribute the requests among them. The distribution mechanism is a separate service hosted on a distinct node–the load balancer. Figure 4.4 shows a load balancer distributing requests between two instances. We show messages for the load balancer going through a gateway.

As a side note, what are values for “too many requests” and “reasonable response time”? We'll come back to these questions later in this chapter when we discuss *autoscaling*. For now, let's stay focused on how a load balancer works.

In Figure 4.4, we show two clients responding to requests. Each request is sent to a load balancer. The load balancer is managing two instances of the same service. Each service instance was started from the same virtual machine image and performs the same functions. The load balancer will send request 1 to instance 1, request 2 to instance 2, request 3 back to instance 1, and so forth. This sends half of the requests to each instance, balancing the load between the two instances. Hence, the term "load balancer."

Figure 4.4: A Load Balancer Distributing Requests From Two Clients Between Two Instances



Consider the load balancer in terms of IP addresses. The load balancer has an IP address of IP_{lb}, service instance 1 has an IP address of IP_{a1} and service instance 2 has an IP address of $IP_{a2.}$ The client sends a message to IP_{lb} (the load balancer) and the load balancer then replaces the destination portion of the IP header with either IP_{a1} or IP_{a2} and sends the message back onto the network.

The service instance, whether 1 or 2, will send its response message to the IP address in the source field of the message. This could be either the cloud gateway or the load balancer if it is acting as a proxy and hiding the IP addresses of the service instances.

Some observations about this simple example of a load balancer:

- The algorithm we provided–alternate the messages between the two instances–is called "round robin." This algorithm balances the load uniformly across the service instances only if every request consumes roughly the same resources to respond. Other algorithms for distributing the messages exist for cases where resource consumption to process requests is variable.

- From a client's perspective, the service's IP address is the address of the load balancer. This address may be associated with a hostname in DNS. The client does not know, or need to know, how many instances of the service exist or the IP address of any of those service instances.
- Multiple clients may coexist. Each client sends its messages to the load balancer, which does not care about the message source. The load balancer distributes the messages as they arrive. (There is a concept called *sticky sessions* or *session affinity* that we ignore for the moment.)
- Load balancers may get overloaded. In this case, the solution is to balance the load of the load balancer, sometimes referred to as *global load balancing.* That is, a message goes through a hierarchy of load balancers before arriving at the service instance.
- The load balancer must know which service instances belong to it. That is, it has an interface that accepts the IP addresses of the service instances to which it is distributing requests. The instances may be statically allocated and the load balancer configured at system initialization, or the service instances may be added dynamically such as from the autoscaler, as we discuss below.

4.5.2 Autoscaling

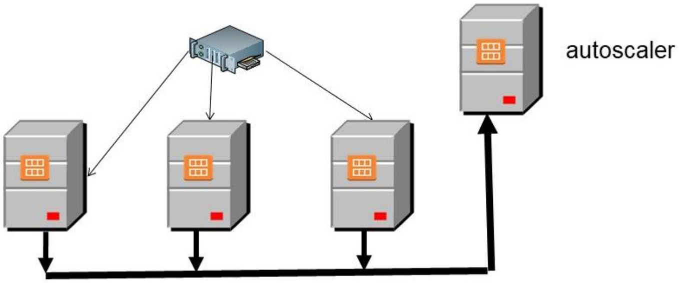
Let's step back and consider a traditional data center, where your organization owns all the physical resources. In this environment, you need to allocate enough physical hardware to a system to handle the peak of the largest workload you expect to see. When the workload is less than the peak, much of the hardware capacity allocated to the system is idle. Compare this to a cloud environment. Two of the defining features of the cloud are that you pay only for the resources you use and that you can easily and quickly add and release resources– *elasticity*. Together, these features allow you to create systems that have the capacity to handle your workload, and you don't pay for any excess capacity.

You can use elasticity at different time scales. Some systems see relatively stable workloads, in which case you might consider manually reviewing and changing resource allocation on a monthly or quarterly time scale to match this slowly changing workload. Other systems see more dynamic workloads with rapid increases and decreases in the rate of requests, and these systems need a way to automate the adding and releasing of service instances.

Autoscaling is an infrastructure service that many cloud providers offer to automatically create new instances when needed and release surplus instances when they are no longer needed. Autoscaling usually works in conjunction with load balancing to grow and shrink the pool of service instances behind a load balancer.

Returning to Figure 4.4, suppose that the two clients generate more requests than can be handled by the two service instances shown. Autoscaling creates a third instance, based on the same virtual machine image that was used for the first two instances. The new instance is registered with the load balancer so that subsequent requests are distributed among three instances rather than two. Figure 4.5 shows the autoscaler monitoring the utilization of the server instances. Once it creates a new server instance, it notifies the load balancer of the new IP address so that the load balancer can distribute requests to the new server instance in addition to the requests it distributes to the other instances.

Figure 4.5: An Autoscaler Monitoring the Utilization



Since the clients do not know how many instances exist or which instance is serving their requests, autoscaling activities are invisible to service clients. Furthermore, if the client request rate decreases, an instance can be removed from the load balancer pool, halted, and deallocated, again without the client's knowledge.

Now we go more deeply into the workings of the autoscaler. First, the autoscaler is a service attached to, but distinct from, the load balancer. Since the load balancer participates in every message request, it must maintain high performance. Hence, it should not be bothered with autoscaling activities. On the other hand, the autoscaler and the load balancer must coordinate. When the autoscaler creates a new instance, it must inform the load balancer of the existence of that instance. Figure 4.5 shows a load balancer and an autoscaler. For simplicity, we show a one-to-one relationship between the autoscaler and load balancer, but in practice an autoscaler can operate with multiple load balancers and multiple services.

Notice in Figure 4.5 that the autoscaler is monitoring the service instances. The instances are passive in the autoscaling process, unaware of the metrics that the autoscaler is monitoring. This means that the autoscaler can monitor only the metrics that are collected by the cloud infrastructure, which typically include CPU utilization and network I/O request rates. You set up a collection of rules for the autoscaler that govern its behavior. The configuration information you provide to the autoscaler includes

- The virtual machine image to be launched when a new instance is created, and any instance configuration parameters required by the cloud provider such as security settings.
- The CPU utilization threshold for any instance above which a new instance is launched.
- The CPU utilization threshold for any instance below which an instance is shut down.
- The network I/O bandwidth thresholds for creating and deleting instances.
- The minimum and maximum number of instances you want in this group.

The autoscaler does not create or remove instances based on instantaneous values of the CPU utilization or network I/O bandwidth metrics. First, these metrics have spikes and valleys and are meaningful only when averaged over some time interval. Second, allocating and starting a new VM takes a relatively long time, on the order of minutes. The VM image must be loaded and connected to the network, and the operating system must boot before it will be ready to process messages. Consequently, autoscaler rules typically are of the form, “Create a new VM when CPU utilization is above 80% for 5 minutes.”

In addition to creating and destroying VMs based on utilization metrics, you can also set rules to provide a minimum or maximum number of VMs or to create VMs based on a time schedule. During a typical week, for example, load may be heavier during work hours, and knowing this, you can allocate more VMs before the beginning of a work day and remove some after the work day is over. On the other hand, some services may be more heavily used outside of work hours. These scheduled allocations should be based on historical data about the pattern of usage of your services.

When the autoscaler removes an instance, it cannot just shut down the VM. First, it must notify the load balancer to stop sending requests to the service instance. Next, the instance may be in the process of servicing a request, so the autoscaler notifies the instance that it should terminate its activities and shut down, after which it can be destroyed. This process is called *draining* the instance. As a service developer, you are responsible for implementing the appropriate interface to receive instructions to terminate and drain an instance of your service.

4.5.3 Detecting and Managing Service Instance Failures

So far, our discussion of load balancers has focused on increasing scale. Here, we will consider how load balancers also serve to increase availability of services.

Figure 4.4 shows messages from clients passing through the load balancer but does not show the return messages. Return messages may go directly from the service instances to the cloud gateway (determined by the “from” field in the IP message header) bypassing the load balancer. This means the load balancer has no information about whether a message was processed by a service instance, or how long it took to process a message. More specifically, the load balancer does not know whether any service instance is alive and processing, or if the instance has failed.

Health checks are a mechanism that allow the load balancer to determine whether an instance is performing properly. The load balancer will periodically check the health of the instances assigned to it. If an instance fails to respond to a health check, it is marked as unhealthy, and no further messages are sent to it. Health checks can be pings from the load balancer to the instance, opening a TCP connection to the instance or even sending a message for processing. In the latter case, the return IP address is the load balancer.

It is possible for an instance to move from healthy to unhealthy and back again. Suppose, for example, that the instance has an overloaded queue. It may not respond to the load balancer’s health check, but once the queue has been drained, it may be ready to respond again. For this reason, the load balancer checks multiple times before moving an instance to an unhealthy list and then periodically checks the unhealthy list to determine whether an instance is again responding. In other cases, there is a hard failure or crash, in which case the failed instance may restart and re-register with the load balancer, or a new replacement instance may be started and register with the load balancer, to maintain overall service delivery capacity.

Sidebar: Timeouts

We use a *timeout*–a decision that a response has taken too long–to detect a failure. A timeout cannot isolate whether the failure is due to a failure in the software of the requested service, the virtual or physical machine that the service is running on, or the network connection to the service. In most cases, the cause is not important: You made a request, or you were expecting a periodic keep-alive or heartbeat message, and did not receive a timely response, and now you need to take appropriate action.

This seems simple, but in real systems, it can be more complicated. There is usually a cost, such as a time or latency penalty, for a recovery action. You may need to start a new VM, which could take minutes before it is ready to accept new requests. You may need to establish a new session with a different service instance, which may affect the usability of your system. As we noted above, there can be considerable variation in response times in cloud systems. If you jump to a conclusion that there was a failure, when there was just a one-time delay, then you pay the cost of recovery when you don’t need to.

Distributed system designers generally parameterize the timeout detection mechanism so that it can be tuned for a particular system or infrastructure. One parameter is the timeout interval–how long to wait before deciding that a response has failed. Most systems do not trigger failure recovery after a single missed response. A typical approach is to look for some number of missed responses over a longer time interval. The number of missed responses is a second parameter for the timeout mechanism. For example, we might set a timeout to 200 milliseconds, and trigger failure recovery if we see 3 missed messages over a 1 second interval.

For systems running with a single data center, we can set our timeouts and thresholds aggressively, since network delays are minimal and missed responses are likely due to software crashes or hardware failures. On the

> other hand, if your system is operating over a wide area network, a cellular radio network, or even a satellite link, you need to be more thoughtful about setting the parameters, as these systems may experience intermittent but longer network delays, and so you may relax parameters to reflect this and avoid triggering unnecessary recovery actions.

A load balancer with health checking improves availability by hiding the failure of a service instance from clients, and the pool of service instances can be sized to accommodate some number of simultaneous service instance failures while still providing enough overall service capacity to handle the required volume of client requests within the desired latency. However, even when using health checking, there can be cases where a service instance starts processing a client request but never returns a response. Clients must be designed so that a client resends a request if it does not receive a timely response, allowing the load balancer to distribute the request to a different service instance.

The load balancer that we've discussed here uses a *push* approach. The load balancer pushes, or sends, messages to one of the service instances in its pool. There is another scheme that uses an infrastructure service called a *message queue service*. In contrast to the push approach, this approach can guarantee that every message is received and processed by a service instance. Client requests are placed into a message queue. When a service instance is ready to process a request, it pulls a request from a queue, and the message queue makes that request invisible to other service instances. After the service instance completes message processing and has sent its response, it asks the message queue to delete the request from the queue. If the instance does not ask for the request to be deleted within a time interval, the message queue makes the request visible again to other service instances, and the next pull by an instance will return that request. It is possible that the first instance actually processed the request, however it was slow, and so the request may be processed twice. Message queues provide *at-least-once* delivery of messages–*exactly-once* delivery has been proven mathematically to be impossible.[18] Systems using a message queue this way must be *idempotent* – a second request generates the same response as the first request. Furthermore, not all message queue services provide a feature to make unprocessed requests invisible, and using that feature means that the service instance design cannot depend on client requests appearing in first-in, first-out

[18] In the field of software development, there are few things that are mathematically impossible. This is one of them.

order. For example, a service instance failure during the processing of a client's first request in a sequence might trigger a retry, and the first request will be successfully processed after a later request by the client has been processed.

Thus far, we have seen three design implications of using load balancers.

1. A client must use timeouts to determine if a service has failed and retry a request multiple times before deciding that the service has failed.
2. A client or the service must be designed to accept multiple responses to the same request.
3. If a message queue is being used to distribute requests to multiple service instances, the service instances must interact correctly with the message queue system to gather messages from the queue and remove messages when they have been processed.

4.6 State Management

We introduced the topic of state in Chapter 2 Virtualization. Here we elaborate on that topic.

State refers to information internal to a service that affects the computation of a response to a client request. State, or more precisely, the collection of the values of the variables or data structures that store the state, depends on the history of requests to the service.

Management of state becomes important when a service can process more than one client request at the same time, either because a service instance is multithreaded, or because there are multiple service instances behind a load balancer, or both. The key issue is where the state is stored. The three options are: in each service instance; in the clients of the service; or external to the service instances. As an example, consider a service that counts how many times it is called. We will show pseudocode for three implementations of the service instances. When reasoning about state, it is helpful to use precise terminology: A *service* is a collection of one or more *service instances*. A client sees only the service, while a particular client request is processed by a single service instance.

Variant 1: The service instance stores the count from one call to another.

```
int i;                         // i is our state variable, initialized to 0 when the service
instance starts

int countv1()
{
i = i + 1;          //add 1 to the last value of i
                    return i;
}
```

Variant 2: The service instance does not store the count from one call to the next and relies on its client to provide the value.

```
int countv2(int i)
{
int a;
                    a = i + 1; //add 1 to the last value of i
                    return a;
}
```

Variant 3: The count is stored externally to both the client and the service instance. Note that we must lock the database so that only one service instance at a time can read and write the “count” value.

```
int countv3()
{
  int a;
                    a = dbase_get (“count”);   //retrieve current value from an
                                               // external database
                    a = a + 1;                 //add 1 to the last value of a
                    dbase_write(“count”, a);   //save current value back to the
                                               // database
                    return a;
}
```

Now we examine these three variants in two different cases. In each of these cases there are two clients sending requests to the service.

Case 1: Two clients, one service instance:

In variant 1, the service counts the number of calls it receives, regardless of which client calls it. Both clients see the total number of requests made to the service.

In variant 2, each client counts the number of calls it makes. Each client sees only the number of requests that it has made, not the total number of requests by all clients.

In variant 3, the database contains the number of calls the service receives, regardless of which client calls it and which service instance processes it. Both clients see the total number of requests made to the service.

Case 2: Two clients, two service instances:

In variant 1, each service instance counts the number of calls it receives. A client sees a different count, depending on which service instance processes its request. A client never sees the total number of requests processed by all service instances.

In variant 2, each client counts the number of calls it makes, regardless of which instance receives the call. Each client sees only the number of requests that it has made, not the total number of requests by all clients.

In variant 3, the database records the number of calls received, regardless of which client made them and which instance received them. All clients see the total number of requests made by all clients, which is the same as the number of requests processed by all service instances.

Which of these responses is correct? That depends on the requirements of your system, and you can imagine use cases where each of these would be correct or incorrect. Our point is: how state is managed is important and affects results.

We saw three different solutions:

1. The history maintained in the service instances. The service is *stateful*.
2. The history maintained in the client. The service is *stateless*.
3. The history persists outside the service, in the database. The service is *stateless*.

Common practice is to design and implement services to be stateless. Stateful services lose their history if they fail, and recovering that state is difficult. Also, new service instances may be created, and designing services to be stateless allows a new service instance to process a client request and produce the same response as any other service instance.

There are cases where it may be difficult or inefficient to design a service to be stateless, and so we might want a series of messages from a client to be processed by the same service instance. We can accomplish this by having the first request in the series handled by the load balancer and distributed to a service instance, and then allowing the client to establish a session directly with that service instance and subsequent requests to bypass the load balancer. Alternatively, some load balancers can be configured to treat certain types of requests as *sticky*, which causes the load balancer to send subsequent requests from a client to the same service instance that handled the last message from this client. These approaches–direct sessions and sticky messages–should be used only under very special circumstances because of the possibility of failure of the instance and the possibility that the instance to which the messages are sticking may get overloaded.

4.7 Sharing distributed data

Frequently, there is a need to share information across all instances of a service or across services. This information may be state information or other information that is needed for the service instances to work together efficiently, for example, the IP address of the load balancer or message queue for the service. Three options exist to manage the sharing of information among distributed instances.

1. An external database. External databases can share large amounts of data, have access controls, and availability mechanisms. They are also relatively slow involving multiple reads/writes to disks. We will not discuss them further here.
2. Distributed caching system. A distributed caching system such as Memcached is suitable for sharing small amounts of data where availability of the data is not an issue. Use cases include session data, web page data, API caching and caching of objects, images, and metadata.
3. Distributed coordination system. A distributed coordination system such as Zookeeper, etcd, or Consol is suitable for caching small amounts of data where availability is important. Use cases include distributed locks, scheulers,, discovery for services, and health checks.

We now discuss distributed caching systems and distributed coordination systems.

4.7.1 Distributed caching systems

We use Memcached as an example system although other distributed caching systems exist. Memcached has the following characteristics.

- Key value store. Data has a simple structure. A key is used to retrieve a data value. The data value itself is unstructured and less than 1Mbyte. Any structure is the responsibility of the clients and they must agree on the interpretation of the data.
- Multiple disconnected servers. Memcached has multiple servers, but they do not communicate with each other. There is no replication of data across servers.
- Each client stores data in a server and it or any other clients with which it is sharing must access that server.

Because data Is stored on a single server, if that server fails, Memcached cannot recover the data. These two characteristics—data is stored in a single server and no recovery of data if a server fails—distinguish a distributed caching system from a distributed coordination system.

4.7.2 Distributed Coordination

Leslie Lamport described a distributed system such as the cloud: "A distributed system is one in which the failure of a computer you didn't even know existed can render you own computer unusable."[19]

As an example, consider the problem of creating a resource lock to be shared across distributed machines: There is some critical resource that is being accessed by service instances on two VMs or containers running on two distinct nodes. We assume this critical resource is a data item, for example, your bank account balance. Changing the account balance requires reading the current balance, adding or subtracting the transaction amount, and then writing back the new balance. If we allow two service instances to operate independently on this data item, there is the possibility of a race condition, such as two simultaneous deposits overwriting each other. The standard solution in this situation is to lock the data item: A service cannot access your account balance until it gets the lock. We avoid a race condition because Service Instance 1 is granted a lock on your bank account and can work in isolation to make its deposit until it yields the lock. Then Service

[19] https://www.microsoft.com/en-us/research/publication/distribution/

Instance 2, which has been waiting for the lock to become available, can lock the bank account and make the second deposit.

This solution using a shared lock is easy when the services are processes running on a single node and requesting and releasing a lock are simple memory access operations that are very fast and cannot fail. However, in a distributed system, there are two problems with this scheme. First, the two-phase commit protocol traditionally used to acquire a lock requires multiple messages across the network. In the best case, this just adds delay to the actions, but in the worst case any of these messages may fail to be delivered. Second, Service Instance 1 may fail after it has acquired the lock, preventing Service Instance 2 from proceeding.

The solution to these problems involves complicated distributed coordination algorithms. Leslie Lamport, quoted above, developed one of the first such algorithms, which he named "Paxos"[20]. Paxos and other distributed coordination algorithms rely on a consensus mechanism to allow participants to reach agreement even when there are computer or network failures. These algorithms are notoriously complicated to design correctly, and even implementing a proven algorithm is difficult due to subtleties in programming language and network interface semantics. In fact, distributed coordination is one of those problems that you should *not* try to solve yourself. Using one of the existing solution packages is almost always a better idea than rolling your own.

When service instances need to share information, they store it in a service that uses a distributed coordination mechanism to ensure that all services see the same values. Apache Zookeeper, Consul, and etcd are three open-source distributed coordination systems. We will use Zookeeper for our example, but the other systems are similar. A Zookeeper cluster consists of a set of Zookeeper services, each hosted on a set of servers. The members of the Zookeeper cluster choose one Zookeeper service instance to be the *leader*, the others are *followers*. Your service, as a Zookeeper client, connects to one of the followers. All followers hold identical copies of the state information being maintained by the Zookeeper cluster. This is stored in memory for fast access. When your service reads from the Zookeeper cluster, the follower to which you are connected responds to your read request with the value in its memory. When your service writes a new value to the

[20] Lamport's paper described the algorithm through a fictional story about an ancient Greek island called Paxos, hence the algorithm's name. Around the same time, Brian Oki and Barbara Liskov independently developed and published an algorithm called Viewstamped Replication that was later shown to be equivalent to Lamport's Paxos.

Zookeeper cluster, that request is sent to the follower to which you are connected. The follower forwards your request to the leader. If the leader accepts the request, it then propagates the new value to the other followers and sends a response back to your service. It is possible for the leader to reject your request. We will see an example of that momentarily. It will also reject your request if it has received another request to change the same data item and this prior change has not yet been propagated to all followers.

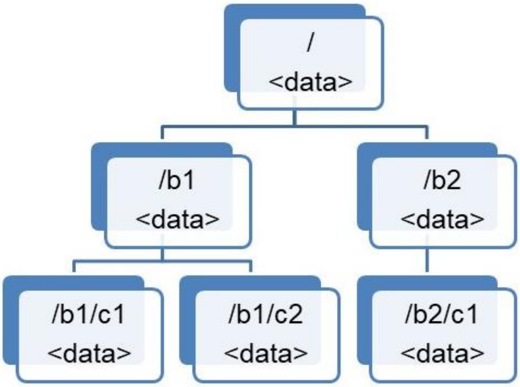
Now consider this sequence when there are failures. If your service fails, your Zookeeper follower detects the failure through a heartbeat mechanism and removes you as a user of any data item. If your follower fails, then your service recognizes it and connects to another follower. At the same time, the Zookeeper leader recognizes that your follower has failed and creates a new follower to replace it and populates the new follower with the necessary state; new clients can then connect to it. The most complicated case comes when the leader fails. In this case, all followers use the distributed coordination consensus algorithm and persistence mechanisms to choose a new leader. We will not go into details of these algorithms; as we noted above, they are quite complicated.

How can we use a Zookeeper cluster to create a distributed lock? Zookeeper organizes its repository as a hierarchy of data nodes (like a directory tree in a file system). See Figure 4.6. For the purposes of this example, the first service instance to request a lock will create a data node in the Zookeeper repository named your_account_lock. When this node is created, it has no children. Any other instance that attempts to create a node named your_account_lock will fail, since the lock has already been created. A service instance that needs to change your account balance creates a child node of the your_account_lock node, naming the node with a unique ID for the service instance. It then asks for a list of children of the your_account_lock node. If the service instance is the first in that list, it assumes that it has the lock, otherwise it does not. The service instance also sets a “watcher” for the your_account_lock node. If the state of this node changes, including adding or removing a child node, watchers are notified through a callback. After a service that has acquired the lock completes its operation on the account balance, the service deletes the child node that it created. If the service fails before deleting its child node, Zookeeper will detect that and delete the node automatically. The other service instances waiting for the lock can do other work while waiting to acquire the lock, since when the child nodes change, all watchers will be notified. Upon notification, each service instance checks the list again to see if it is now the first in the list, and if not, it continues to wait.

Because the Zookeeper server you are dealing with is “close” to your node or container, your interactions with it are fast. Because Zookeeper manages only a limited amount of data, the repository is kept in memory, so it is fast. Interactions with Zookeeper take on the order of milliseconds. Choosing a new leader is more time consuming: First, a new leader must be chosen, and then the new leader must be brought up to date using the persistence mechanisms of the distributed coordination protocol.

In the section on state management above, one of the implementation variants stored shared state in an external database. We assume that this external database would handle the distributed coordination among the multiple service instances and that the state values would persist, using a mechanism such as SQL transactions. Database services providing these features are generally significantly slower than services such as Zookeeper. How should a service designer choose where to store shared information? When the amount of data is small (less than 1MB) and is not required to persist beyond single invocations of a system, a distributed caching or coordination system is usually a better choice than a SQL database for storing shared information.

Figure 4.6: Zookeeper Node Structure



4.8 Security policies

A security policy is a set of rules that govern which accounts or services have access to which resources. The cloud service that provides access control is called Identity and Access Management (IAM). AWS, Azure, and Google Cloud all provide

IAM services. Rules such as “only allow these users access on these dates” or “allow specific user to launch a new instance in a subnet” are specified to the IAM service.

IAM rules are checked by invoking the IAM service when an API call is made to a cloud service.[21] Since all communication is made via cloud services, this includes messages as well as functions such as launching a new VM.

4.9 Summary

The cloud is composed of many distributed data centers, with each data center containing tens of thousands of physical computers. It is managed through a management gateway that is accessible over the internet and is responsible for allocating, deallocating, and monitoring VMs, as well as measuring resource usage and computing billing.

Because of the large number of physical computers, failure of some computer in a data center happens frequently. You as a service developer should assume that at some point, the VMs on which your service is executing will fail. You should also assume that your requests for other services will exhibit a long tail distribution where as many as 5% of your requests will take 5 to 10 times longer than the average request. Thus, as a service developer you must be concerned about the availability of your service.

Because single instances of your service may not be able to satisfy all requests in a timely manner, you may decide to run multiple VMs containing instances of your service. These multiple instances sit behind a load balancer. The load balancer receives requests from clients and distributes the requests to the various instances.

The cloud infrastructure can automatically scale your service by creating new instances when demand grows and removing instances when demand shrinks. You specify the behavior of the autoscaler through a set of rules giving the conditions for creation or deletion of instances.

The existence of multiple instances of your service and multiple clients has a significant impact on how you handle state. Different decisions on where to keep state lead to different results. Common practice is to keep services stateless.

[21] Amazon reports over 400 million calls to the IAM service per second! https://aws.amazon.com/blogs/aws/happy-10th-birthday-aws-identity-and-access-management

Stateless services allow for easier recovery from failure and easier addition of new instances.

Small amounts of data without high availability requirements can be shared among service instances by using a distributed caching service. For data with high availability requirements, a distributed coordination services can be used. They are complicated to implement but there are several proven open-source implementations.

Cloud providers have a service that verifies access control rules that is checked with each call to a cloud API.

4.10 Exercises

1. Allocate a VM within a cloud and display its IP addresses.
2. Examine a message header from a cloud VM and determine where the source IP address in the header comes from.
3. Test the various options given in the section 4.6 State Management and verify they are correct.
4. Install Zookeeper. What went wrong with your installation? What are the restrictions as to the placement and initialization of Zookeeper servers? What did it require you to specify about the leader?
5. Go through the process of creating an autoscaling group for your cloud provider. What additional information did you need to provide other than that enumerated above?
6. Install HAProxy and test it with two instances of LAMP VMs.

4.11 Discussion Questions

1. When you create a VM through the cloud management gateway, it returns a public IP address for the VM. Is this the actual IP address of the VM created for you? Why or why not?
2. Show how the source and destination address fields in the IP header of a message sent by a client through a proxy and then a load balancer are changed. How does the client know that the returning message is a response to the original message it sent?

3. Examine the last program you wrote. Can it easily be divided into a stateful portion and a stateless portion? Can you encapsulate the stateless portion as a service?

4. A context diagram displays an entity and other entities with which it communicates. It separates the responsibilities allocated to the chosen entity from those responsibilities allocated to other entities, and shows the interactions needed to accomplish the chosen entity’s responsibilities. A context diagram is used during the design process to decide what is in and what is out of various pieces of the architecture. Draw a context diagram for a load balancer.

5. What happens if a load balancer fails? How about an autoscaler?

Chapter 5 Container Orchestration

The distributed system cloud mechanisms we have discussed – discovery, load balancing and autoscaling – were discussed in the context of virtual machines. Discovery, load balancing, and autoscaling exist for containers but take a different form.

At the end of this chapter you should know

- How containers are allocated to reduce communication costs
- Common container management tools.
- What a container repository is and how to store container images in them
- What a cluster is and what the orchestration of containers within a cluster is
- Basic security rules for

5.1 Coming to terms

Container orchestration – the management of a container's lifecycle, including provisioning, deployment, scaling, and load balancing

Cluster– a group of hosts linked through a fast network and treated as a single logical unit.

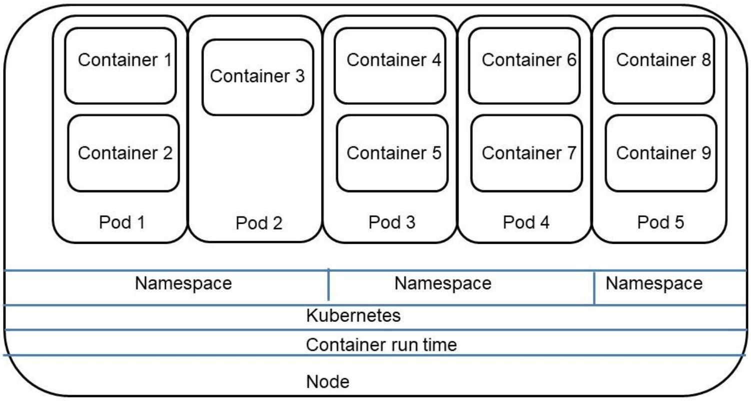
Pod -a collection of containers that are constrained to be allocated together.

Sidecar– a utility container such as logging or monitoring container.

5.2 Pods

Kubernetes, one of the common container runtimes, has an additional element in the container hierarchy – pods. A pod is a group of containers – possibly only one. The Kubernetes hierarchy is that pods contain containers. A pod can hold one or more containers. We will discuss how pods affect container networking. Pods are divided into namespaces for isolation purposes. The namespaces are a Linux construct that Kubernetes uses. Figure 5.1 shows this structure as well as Kubernetes utilizing the services of a container run time.

Figure 5.1: Kubernetes stack including pods



We discuss first the case of a single container in a pod and then the case of multiple containers.

- A single container in a pod. This is the Kubernetes method for managing containers. Kubernetes does not manage containers directly but only through pods. A pod has an IP address and a collection of ports. To communicate with the container inside of a pod, you send an IP message to the IP address of the pod and the port number on which the container listens.
- Multiple containers in a pod. If multiple containers are in a single pod, they should be tightly coupled. That is, there is a strong dependency among them. Sidecars – utility containers – often are placed in a pod with the container for which they provide services. Two containers in a pod have the same lifetime, i.e., the containers in pods are allocated and deallocated together. The pod is allocated to the same host and, consequently, the communication between those two containers uses in memory facilities and does not go onto the network. An example of placing two containers in a pod is to place a logging service in the same pod as the system container for which it provides logging. A consequence of this arrangement is that there may be multiple instances of logging services. One for each pod.

Containers in a pod share an IP address and port space to receive requests from other services. Each container must listen on one port distinct from the ports listened to by other containers. A message from outside the pod is delivered to the pod via the IP address and to the container via their port. Two containers in the same pod an communicate with each other by sending a message to "localhost" (127.0.0.1) with the appropriate port number. The "localhost" routes the message back to the originating pod and the port number routes the message to the destination port. Two containers in the same pod can also communicate with each other using shared memory mechanisms or shared storage volumes.

5.3 Orchestration

Consider the role a conductor has in an orchestra. The conductor plays no instrument but is responsible for choosing the players, arranging them on the stage, setting the tempo, cuing the various sections of the orchestra, and making sure that the various players have an appropriate copy of the score.

This is not a bad analogy for container orchestration. An orchestration system contains no containers but manages a container's lifecycle, including provisioning, deployment, scaling, and load balancing.

Recall the various issues in distributed systems that we saw in Chapter 1 Platform Preliminaries. We saw solutions to these issues for VMs in Chapter 4 The Cloud. VMs were replicated to share the load, load balancers distributed requests and monitored the health of the VMs. Autoscalers created new instances and, with the help of other cloud services, decided where to allocate the new instances. Discovery services provided the IP addresses with which to access the services of the VMs.

When managing containers, these functions are divided between an orchestration system and a service mesh. Some functions could be realized in either of these two options. Fundamental to a container orchestration system are scaling and failure recovery. Fundamental to a service mesh are discovery and communication patterns. It is possible that some functions allocated to the orchestration system can be done by a service mesh and vice versa but we will focus our discussion on this division.

As an example, in Kubernetes, you can set up a specification file that includes the following information.

- Pod name

- Containers included in the pod
- Number of instances of the pod to create
- Scaling rules

After the pods have been created, a portion of Kubernetes will monitor the health of the pods and ensure that one is always active. A different portion of Kubernetes monitors the utilization of the pods and may create additional pods according to rules similar to the rules for autoscaling in the cloud. Thus, Kubernetes manages scaling and failure recovery.

5.4 Container Security

The United States Department of Defense has created a guide to manage container security.[22] The guide contains two sections relevant to container security: container image creation and container deployment. We discuss some of the items in these sections.

5.4.1 Container image creation

1. The Container Image Must Be Built with the SSH Server Daemon Disabled. A container image must only enable the applications needed for the service being implemented by the container image. Allowing an SSH to occur to a container provides the possibility of a malicious attack.
2. The Container Image Must Be Created to Execute as a Non-Privileged User. Containers must run as a non-privileged account. Allowing a container to run as a privileged user leads to containers that can access host system-protected resources and execute privileged commands. Non-privileged user status is set during the build (image creation) process.
3. The Container Image Must Be Clear of Embedded Credentials. A container may require passwords, secrets, or keys to connect to other applications such as backend databases. The credentials must be kept externally, fetched by the application, and not stored in the container image. If the credentials are stored within the container image, anyone with access to the image can parse the credentials. We discuss credential management in Section 14.4 Authorization.

[22] https://dl.dod.cyber.mil/wp-content/uploads/devsecops/pdf/DevSecOps_Enterprise_Container_Image_Creation_and_Deployment_Guide_2.6-Public-Release.pdf

5.4.2 Container deployment

1. The Container Should Have Resource Limits Set. The resource limit setting for a container provides a limit or maximum amount of resources a container can consume. Without this limit, a container can use all available resources, starving other containers of needed resources, causing a Denial of Service (DoS) across all the services executing within the container platform.

2. The Container root filesystem Must Be Mounted as Read-Only. Any attempts to change the root filesystem are usually malicious in nature and can be prevented by making the root filesystem read-only.

3. A Container Must Not Have Access to Operating System Kernel Namespaces. Namespaces provide a straightforward way of isolating containers. Using isolation, services within a container cannot see or access services running within other containers or on the container platform host system. When kernel namespaces are shared to containers, data can shared directly between services bypassing security measures, allowing containers to elevate their privileges.

5.4.3 Container Runtime

It is also possible to monitor Kubernetes for possible malicious activities.   Falco[23] is an open-source tool that reads OS kernel logs, container logs, and platform logs. It has a threat detection rules engine to alert users of malicious activities. E.g. creation of a privileged container or Reading of a sensitive file by an unauthorized user

5.5 Service Mesh

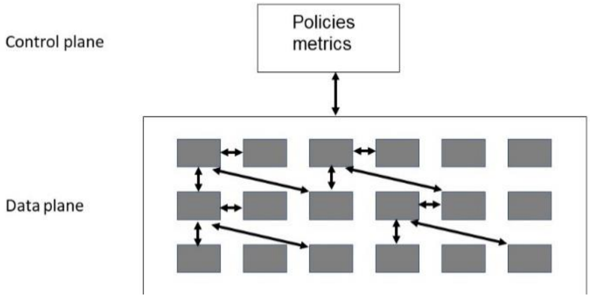
The fundamental purpose of a service mesh is to manage discovery and communication patterns. In addition, a service mesh will manage security activities, resiliency settings, and observability.

Figure 5.2 shows the basic architecture of a service mesh. It is divided into a control plane and a data plane (the terminology comes from networking). The elements of the data plane are containers – or pods in Kubernetes. In theory any container can communicate with any other pod but the communication options for each container are limited. There are a fixed number of other containers with which any container can communicate. The paths of communication form a mesh –

[23] https://falco.org/

hence the name. The construction of the mesh is the responsibility of the control plane. That is, the control plane sets up the possible communication paths and then each container will only be able to communicate with other containers to which it is connected via a communication path. This restricted set of related containers is included in the policy portion of the control plane. It also can limit rates of messages and other managerial functions. The control plane also collects metrics about the message traffic among the containers.

Figure 5.2: Basic architecture of a service mesh



.

In addition, the control plane provides

- Automated key and certificate management
- An API for policy definition and the gathering of telemetry data
- Configuration ingestion for service mesh components

Context rules are specified in the policy section of the control plane. Context includes

- Instances of containers within same pod or same host.
- Containers participating in canary or A/B testing
- Shadowing for availability
- Access control

The data plane component performs three different functions:

1. Secure networking functions. This includes all functions related to the actual routing or communication of messages between services. The functions that come under this category are service discovery, establishing a secure (TLS) session, establishing network paths and routing rules for each microservice and its associated requests, authenticating each request (from a service or user), and authorizing the request.

2. Policy enforcement functions. The data plane enforces the policies defined in the control plane through configuration parameters in the proxies (policy enforcement service. This service is in a container contained in the same pod as the service for which it enforces the policies. An example policy is access control.
3. Observability functions. Observability is achieved through the gathering of telemetry data from the service with which it is associated.

When a container wishes to send a message to another container it uses a discovery service to find the location of that container. The discovery service will only record containers within the set defined for the sending container by the control plane. Each container has a discovery service that it uses, frequently packaged in a pod with the container. For simplicity in this discussion, we will assume that each container has a discovery service that is unique to it and in the same pod. That is, discovery services are not shared among containers. Although, in practice, there are many optimizations to this assumption, we will not go into the complexity here.

Now we discuss how orchestration system and a service mesh interact. The orchestration system creates a new instance of a container. For concreteness we will assume container C is being scaled. The orchestration system informs the control plane of the existence of this new instance of container C. Then the control plane places a copy of the discovery service from an existing instance of container C into the pod containing the new instance. It also places the identity of the new instance into the discovery services that allow communication with any instance of container C. Now the new instance can communicate with any of the containers that the old instances communicated with and any container that communicated with an old instance of C can now communicate with the new instance.

Variants of the process deal with the cases of an instance being deleted or a new instance being created from other than scaling activities.

5.6 The Evolution of Container Technology

Containers are not a new concept, tracing their roots back to pre-Linux Unix systems. However, the adoption of containers as the enabling technology for microservice architectures (we'll talk about microservices in Chapter 12 Design Options) has accelerated the evolution of this technology, and that evolution has in turn led to many uses for containers. Unlike VM hypervisors and hypervisor managers, which began as proprietary technology, container runtimes and orchestration have been open-source technologies from the start, which has also promoted innovation. As we write this book, container management is evolving in two different directions – orchestration and serverless computing. We have discussed each above, and here we will focus on the implications for service designers.

The first direction, container orchestration, uses containers to package and deploy all services. Container groups (e.g., Kubernetes pods) allow you to bundle a set of related services and ensure that they are allocated and that they start, scale up and down, and are terminated and deallocated as a single unit. This allows you to more easily design and analyze scalable and highly available systems. However, as a service designer you still need to be concerned with defining container groups, setting autoscaling rules, and monitoring the performance of the various elements of your system.

The second direction uses containers to deliver serverless computing. As we discussed in Section 2.7, there are servers in this architecture, and there are also containers. However, these are invisible to you as a service developer. The container image is built when your function is invoked for the first time, and the serverless platform providers may optimize the container infrastructure to allow this to complete very quickly; for example, by caching the container base images at every node rather than loading them from a container registry over the network. Because the container allocation logic may not immediately deallocate a container from a node when the function exits, the container can be reused for a subsequent invocation, avoiding the need to rebuild the container. A serverless platform provider could maintain a pool of containers executing the base images and load your function into an already running container.

Each of these directions pulls container technology down a slightly different path. Container orchestration makes much of the container technology and APIs available to service developers. The features must be general and implementations must be robust for many different use cases. The serverless direction hides the container technology from service developers. APIs are used by only the serverless platform providers, and features must be optimized for this narrow use case. Today, the same container technology is being stretched to cover both directions, but this may change in the future.

5.7 Summary

Containers are a packaging mechanism that virtualizes the operating system. A container can be moved from one environment to another if a compatible container runtime is available. The interface to container runtimes has been standardized.

Container images can be stored in repositories. The repositories identify various versions of the container image. Storing a container image in a repository supports coordination among members of a team. It also supports global libraries that anyone can use.

Treating a collection of containers as a pod facilitates scaling and orchestration. The pod manages requests to individual containers and is responsible for scaling, both up and down, of the pods that it manages.

Container orchestration systems manage the scaling and failure recovery distributed systems issues and service meshes manage the discovery and communication pattern issues.

Container orchestration and serverless architecture each prioritize different features in container technology. The same container technology is used for both cases today, but this may change in the future.

5.8 Exercises

1. Create a Docker container image with the LAMP stack in four layers. Test the image by executing a simple PHP program.
2. Store the container you created in Exercise 1in a repository. Have a colleague/classmate retrieve it and execute it.
3. Use Kubernetes to set up a simple pipeline such as described in section 5.3 Orchestration.
4. Use Istio to manage the Bookinfo microserevice system. https://istio.io/latest/docs/examples/bookinfo/

5.9 Discussion Questions

1. How does the container management system know that only one layer has been changed so that it needs to transport only one layer?
2. How are orchestration rules specified in Kubernetes?

3. Locate discussions of using business process models to describe container orchestration. How mature are the tools described? What needs to be done to make these tools production quality?
4. Construct a context diagram for Kubernetes.
5. List the steps that create a new container image and discovery service for a container created from other than scaling.

Chapter 6 Measurement

A distributed system has a great many services and network connections among its elements. To improve performance or troubleshoot failures, it is important to be able to observe the utilization of the different resources (memory, CPU, disks, and networks) as well as the activities of the various services that compromise the system. This means that information that is useful in analyzing the operation of the system must be collected and made visible for analysis. We use the term *measurement* to describe the collection and exposure of a systems behavior during operation. The term *observability* is also used to describe the same activities.

At the end of this chapter, you should know

- The uses of measurements
- The types of measurement – metrics, logs, traces
- Measurement architecture
- Why time is inadequate for sequencing events in a distributed system.

6.1 Coming to Terms

Agent—a system that runs as a background process

Alert—a notification to a first responder that a problem exists

6.2 The types and uses of measures

Operational measurements have three purposes.

1. Generate alerts. Measurements, in particular metrics, are used to generate alerts that indicate a serious problem with a system in operation. Metrics are based on the utilization information collected either by the cloud infrastructure or by the container management system.
2. Forensics. When a problem has occurred, it must be identified. Logs collected by the system in response to events are used to identify the problem.
3. Performance. Logs represent a collection of events. Understanding the end-to-end behavior of a system in response to requests is the purpose of

collecting traces. A single trace shows where time is spent in response to a single request. A collection of traces shows where the system spends time and can be used to detect bottlenecks and improve overall performance

6.3 Measurement architecture

Measurement values are collected from a variety of locations within a system in operation. To synthesize and analyze these values it is useful to have them in a central place. This allows comparative analysis of data from different locations. It also means that an analyst has a single source of information rather than having to track down multiple sources.

For example, suppose an analyst is trying to determine whether a service failure is hardware or software. The analyst knows the service instance that has failed and would like to examine other instances of that service and other services executing on the same host. If other instances of that service have also failed or are having problems, then the source of the problem is likely the service itself. If other services executing on the same host have also failed, then the likelihood is that the host has failed. In any case, having measurement data in a central location – typically a data base – avoids the necessity of the analyst logging into multiple services. This leads to the architecture shown in Figure 6.1. The measurements, from whatever source, are sent to a back-end database from which analysis can be performed.

Figure 6.1: Basic measurement architecture

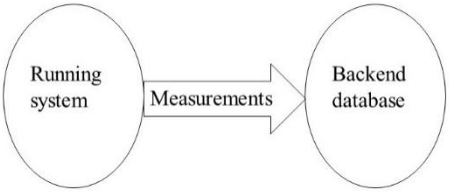


Figure 6.1 does not show where in the running system the measurements originate. It also does not show the activities of the backend database. We now deal with those two items.

6.3.1 Sources of measurement data

The measurement data comes from two sources: the infrastructure that monitors utilization, and the logs produced by the services.

We saw how an autoscaler monitors utilization of VMs. It tracks CPU utilization, network requests, and disk requests. The control plane of the service mesh gathers the same information for containers. These metrics are passed to the back end, either through a push of the information from the infrastructure or from a pull from the back end. Typically, there is a back end specific service that retrieves the infrastructure metrics and places it in the database.

Logs are retrieved in somewhat the same way. A log entry is generated by a service and placed in a known location on the local file system. An agent specific to the backend will retrieve this data and send it to the backend. Log files tend to be voluminous and so the agent may also empty the log file after the data has been copied so that the service does not exhaust available disk space.

6.3.2 Activities of the backend

The backend has several responsibilities dealing with informing an analyst of a potential or current problem. It sends an alert – an alert is a warning of a potential problem – or it sends an alarm – a problem has occurred. It determines whether an alert should be sent based on a collection of rules similar to autoscaling rules.

The backend also provides various database functions. It has a dashboard that allows an analyst to determine whether there is a problem. The dashboard, typically, has a collection of panels that are red, orange, or green depending on the situation. A set of rules determine the thresholds for the various colors.

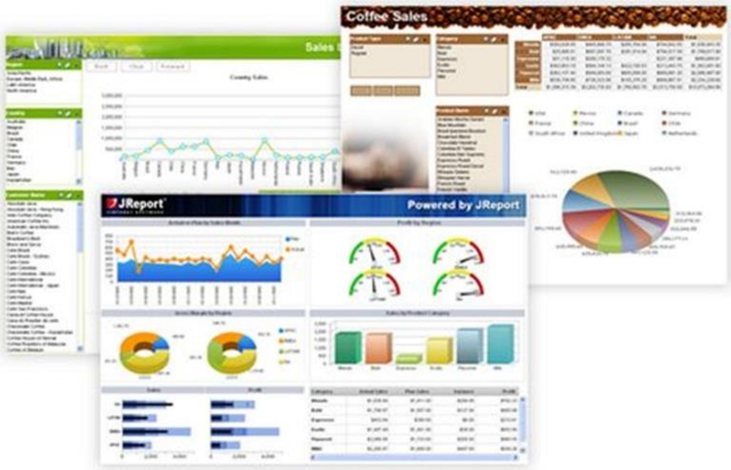
Each panel displays an aggregate and can be drilled into to determine its constituents. An analyst can examine these constituents to determine where the problem is located.

The data stored in the database can be analyzed to determine recurring problems, time to repair problems, and other information useful in improving the system development or repair processes.

There are dozens of systems that perform monitoring and alerting. Some are available as services delivered by your cloud provider, while others are separate packages that you run as your own infrastructure service. The web page https://en.wikipedia.org/wiki/Comparison_of_network_monitoring_systems provides a comparison of some of them. Most likely, you won't have to do any tradeoff analysis and product selection for this infrastructure service, since all systems and infrastructure services within an organization will usually use the same monitoring and alerting systems. This allows the aggregation of metrics across the entire enterprise and facilitates efficient handling of issues that affect multiple systems.

Monitoring systems frequently provide a configurable dashboard user interface, which displays values in a format that allows quick interpretation. For example, the display for a metric can include instantaneous values, short-term averages, long-term trends, and threshold values, and values can be colored green, orange, or red to indicate status of OK, concern, or alerting. Different users can create different dashboard configurations. For example, one company has a large dashboard available to all employees. The dashboard displays an important business metric–sales per minute. As a developer, you are primarily interested in the execution of the software for your service, so your dashboard would show performance information such as processor, network, and disk utilization. Your dashboard would also show the results of health monitoring and metrics related to your service's service-level objectives, or SLOs, which we will discuss in Chapter 13 Post Production. Recall that you may have hundreds of instances of your service in execution, so the dashboard aggregates information for all instances and allows you to drill down, when you need to get more refined information. Figure 6.2 shows a sample dashboard.

Figure 6.2: A Sample Dashboard[24]



6.4 Time Coordination in a Distributed System

We now turn to a discussion of time to see why time is not suitable for determining the sequence of activities in a distributed system.

Maintaining accurate time is a hard problem, even within a single device in your system. Hardware clocks drift over time[25]. A typical laptop will gain or lose one second every 12 days. The clock's precision is primarily determined by manufacturing tolerances, temperature, and physical vibration, so the good news is that unless your device is bouncing around in an automobile or aircraft, your operating system will be able to calibrate for the other factors and your hardware clock will be sufficiently accurate so that relative measurements taken within your device will be accurate–you can measure the start and end times of events within your device to calculate the duration of an activity.

If your device is out in the world, it may have access to a time signal from a Global Positioning System (GPS) satellite. GPS time is not affected by temperature or

[24] By Kate07lyn– Jinfonet Software, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=13309105

[25] Computer clocks are usually biased to run slow, so that all corrections set the clock forward. If the clock is set back then you will see the same time occur twice, which can cause undesired effects, for example, repeating a timed event trigger. Related to this, your code should never check for equality to a specific time, since a clock reset may cause that exact time to be skipped. Instead, check for the clock to be greater than your trigger time.

vibration. It is accurate to within 14 nanoseconds, theoretically, with practical implementations achieving accuracy within 100 nanoseconds, which still provides a clock that has higher resolution than most software can utilize.

Coordinating time across two or more devices is a *very* hard problem, and discussions about this subject quickly shift from engineering issues like the speed of light to metaphysics. For example, my "now" is not your "now": If you create an event source and I observe it, when does the event occur? In your frame of reference, or mine?

We will focus on the engineering side of this discussion: The clock readings from two different devices on the network *will* be different. We will discuss mechanisms to bring the time setting on devices into synchronization, but time cannot be used to determine latency, for example, across different devices. More significantly, time cannot be used to order events that occur on different devices.

Attempts to synchronize clocks across networks began shortly after networks became prevalent. Network Time Protocol (NTP) is used to synchronize time across different devices that are connected over a local or wide area network. It involves exchanging messages between a time server and client devices to estimate the network latency, and then applying algorithms to synchronize a client device's clock to the time server. NTP is accurate to around 1 millisecond on local area networks and around 10 milliseconds on public networks. Congestion can cause errors of 100 milliseconds or more.

If these numbers seem small to you, the financial industry has spent millions of dollars to reduce the latency between Chicago and New York[26], so whether 10 milliseconds is significant depends totally on context.

Cloud service providers provide very precise time references for their time servers. For example, Amazon and Google use atomic clocks, which have virtually unmeasurable drift. Another common approach is to use a GPS time receiver to provide the reference for the time server in a data center. This approach requires a connection to an outdoor antenna, and it isn't practical to put a GPS receiver on every device in the data center.

Having a time server using a high quality time reference inside the data center, along with a well-performing local network, can allow all devices in the data center

[26] https://newswire.telecomramblings.com/2010/04/network-latency-improvement-creates-one-of-the-fastest-routes-between-chicago-and-newarknew-york-city/

to use NTP to synchronize with enough accuracy for many purposes; however as a practical matter, you should assume there is error between clock readings on two different devices. For this reason, most distributed systems are designed so that time synchronization among devices is not required for systems to function correctly. You can use device time to trigger periodic actions, to timestamp log entries, and for a few other purposes where accurate coordination with other devices is not necessary.

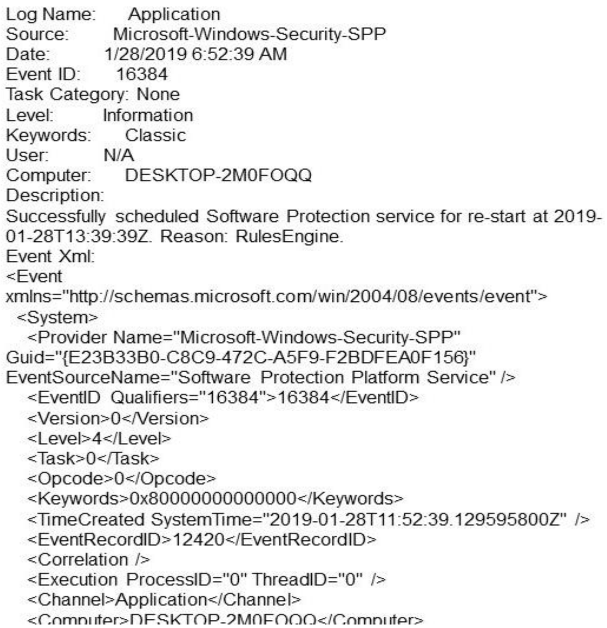
There is a distinction between the sequence that activities occur and the time that they occur. If the time is accurate then you can deduce the sequence, but time is not accurate in a distributed system, and so sequencing is typically used instead. For critical coordination across devices, most distributed systems use mechanisms such as vector clocks (which are not really clocks, but counters that trace actions as they propagate through the services in a system) to determine whether one event happened before another event, rather than comparing times. This ensures that the system can apply the actions in the correct order. We will return to this approach when we discuss traces in Section 6.8.

6.5 Logs

Logs are primarily used for forensic purposes. If something has gone wrong with a service, the analyst will want to know the sequence of events leading to the problem and the values of the parameters passed to the service. This will enable the analyst to work backwards and reconstruct how the problem occurred. The analyst can then create a work around for a quick fix and, subsequently, identify a longer-term fix to prevent the problem from recurring.

Whereas metrics are collected by the infrastructure, generating logs is the responsibility of the service. A log entry is generated by a service when an event of interest occurs. Most common are on entry or exit. Entry and exit from methods within the service are also common reasons for log entries. Figure 6.3 shows a sample log entry from a Windows 10 system.

Figure 6.3: A Portion of a Windows 10 Sample Log Entry



```
Log Name:     Application
Source:       Microsoft-Windows-Security-SPP
Date:         1/28/2019 6:52:39 AM
Event ID:     16384
Task Category: None
Level:        Information
Keywords:     Classic
User:         N/A
Computer:     DESKTOP-2M0FOQQ
Description:
Successfully scheduled Software Protection service for re-start at 2019-
01-28T13:39:39Z. Reason: RulesEngine.
Event Xml:
<Event
xmlns="http://schemas.microsoft.com/win/2004/08/events/event">
  <System>
    <Provider Name="Microsoft-Windows-Security-SPP"
Guid="{E23B33B0-C8C9-472C-A5F9-F2BDFEA0F156}"
EventSourceName="Software Protection Platform Service" />
    <EventID Qualifiers="16384">16384</EventID>
    <Version>0</Version>
    <Level>4</Level>
    <Task>0</Task>
    <Opcode>0</Opcode>
    <Keywords>0x80000000000000</Keywords>
    <TimeCreated SystemTime="2019-01-28T11:52:39.129595800Z" />
    <EventRecordID>12420</EventRecordID>
    <Correlation />
    <Execution ProcessID="0" ThreadID="0" />
    <Channel>Application</Channel>
    <Computer>DESKTOP-2M0FOQQ</Computer>
```

Any logging you do consumes resources. You will use processing time to create the message, disk bandwidth to write the message, storage to retain the message as a log entry, and network bandwidth to copy your log to a central location. Ideally, you want to save just enough information so that you can determine what is wrong when there is a problem with your service. Because in practice, you don't know what problems you will encounter, you will want to include as much information as possible without impacting the operational performance of your service.

Next, we will discuss what events and what information about these events should be recorded, and then we will discuss where that recording is saved.

6.5.1 What Events Should be Recorded?

Your service should log two categories of information: Technical information about the software execution and business information. Technical information will allow you to trace the execution of requests through your service and system, while business information frames your service's execution in terms that product managers and marketers care about, such as the number of completed or abandoned purchase transactions. We will focus on the technical information here, as the relevant business information depends on your system's purpose and your service's role.

At a minimum, you should log a message for every request as it enters your service and as it exits your service and you send a response. Since every log message from your service is timestamped by the same service instance, logging entry and exit either of the service or of methods within the service can be used to calculate latency. You might also choose to log entry and exit of certain functions within your service, which will give insight into where your service is spending its time and help you isolate problems if your total latency is too high. In addition to timing and error information, entry and exit log messages should include the invoking service and the request parameters. On exit, log any exit parameters or return codes. Logging error exits with informative return codes will enable an estimate of availability for downstream services–one cause of an error exit is that a downstream service does not respond in a timely fashion. In Section 12.7 Microservices in Context, we will see that data transferred through Protocol Buffers can be automatically logged by having the proto compiler generate calls to logging software. From a developer's perspective, then, logging entries and exits requires no additional work although, as we said, it does require computational resources.

Beyond entry and exit tracing, other useful information includes echoing your instance's configuration to the log during instance startup. If your service has resources such as a worker-thread pool or temporary object storage, it is often helpful to log resource status at entry/exit or on a periodic basis.

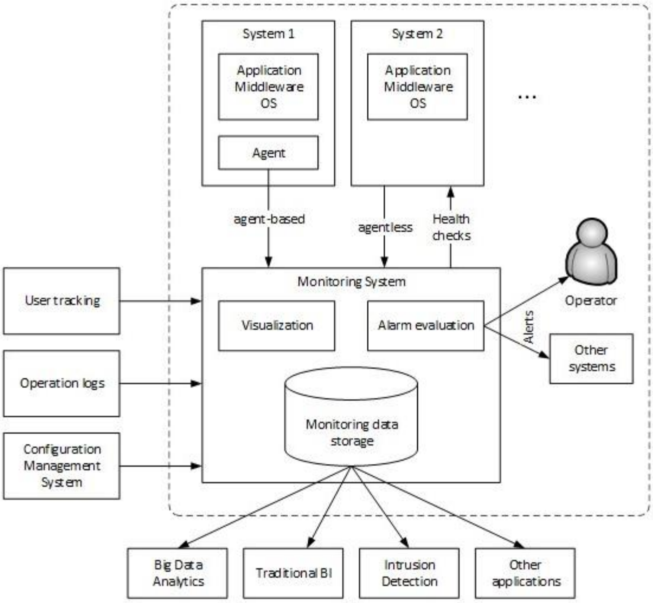
In all distributed systems, there is the challenge in correlating log messages as you trace a request as it fans out from one service to other services. Because you cannot rely on timestamps to precisely align events across machines, all the services in your system must adopt a uniform way of identifying log messages to enable alignment. A common approach is to add a unique identifier to every user request entering the system, and then pass that identifier along as the request fans out to the many services involved in processing it. That identifier should be included in log messages generated by each service in the chain, allowing re-creation of the entire processing sequence from the log messages. The context field in the HTTP header can be used for this purpose.

6.5.2 Where Is the Recording Saved?

Logs are stored in files on your instance's file system. Each service should put its log files in a separate directory. In Linux, this directory should be a subdirectory of /var/log and should be named after your service. Apache logs in Ubuntu, for instance, can be found in /var/log/apache2/error.log.

These local-instance logs are forwarded periodically to a log aggregation infrastructure service that is running a time-series database system. See Figure 6.1 for an overview of this process. Figure 6.4 shows a detailed architecture of a log management system. Commonly used systems are Logstash, Splunk, and Kibana. The log tool requires that an agent executes on your system. This agent is responsible for copying your local log files to the log-database server. This agent should be part of your organization's base image, or part of your deployment pod. Figure 6.4 shows the architecture of a log-management system.

Figure 6.4: The Architecture of a Log Management System



In Figure 6.4, each input entry is a text string (as shown in Figure 6.3) with no fixed interpretation of any part of the string. Within the log-aggregation service, each source of logs has an associated parsing statement that identifies the meaning of each element in the log message and maps log information into the time-series database fields. It says, for example, that columns xx-xx in a log message represent a timestamp, columns xx-xx represent the instance ID, and so forth.

The log-aggregation service stores the parsed log entries in a time-series database. This is a data-storage system that is optimized to store and operate on entries that are indexed by timestamps. A time-series database has built-in support for operating on time ranges and often includes filters to find a minimum or maximum value within a time range or compute an average or other statistic over the range. These functions can be useful for computing metrics, which we discuss next.

6.6 Metrics

Metrics are measures of your service’s activity over some period. For example, the container that hosts your service had less than 10% CPU utilization from 10:00:00 until 10:59:59 today. From this simple example, we can see some fundamental differences between logs and metrics.

- Logs tell what is happening inside the service and metrics tell what is visible from outside the service. This metric tells you that your service is using less than 10% of the CPU, but you cannot determine where the time is being spent within your service.
- Metrics are measurements taken by the operating system or platform. Logs are generated either by your service or by other software packages and need post-collection analysis. Thus, metrics can be used to trigger real-time alerts, whereas logs are used to analyze problems.

As with logs, metrics can be passed on to specific tools for storage and display. It is possible to integrate metric storage and log storage into one location. Merging them allows for analyses such as, “What is the load on the CPU caused by use of a particular feature?” In this case, the logs can identify the periods when the feature is used and the metrics then allow comparisons across periods.

6.7 Tracing

Log entries represent individual events. It is difficult to construct an end-to-end view of a transaction from logs. This end-to-end view is the purpose of tracing.

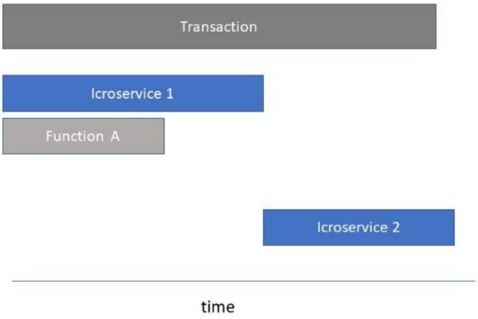
A request is made to the system, either from a user or from another system. This request is given an ID. The ID will be used to identify the processing associated with this request. The ID is propagated through the HTTP headers placed on each message. The World Wide Web Consortium has standardized the format of the header information. [27]

A span is a named segment of a trace. By visualizing the spans across times, an analyst can see where time is being spent and identify bottlenecks. See Figure 6.5 for a sample.

In addition, each trace can have a context associated with it. The context is useful for controlling the behavior of the service or for subsequent analysis. Contexts can be virtually anything. Examples are

- Test version. Use context to affect behavior or routing.
- System. Google might want to know what percentage of their network traffic might be due to search, Gmail, etc
- Traffic prioritization. Give priority to certain requests to maintain quality of service.

Figure 6.5: Display of spans



6.8 Summary

Measurements are used for alerting, forensics, and improving overall performance. Metrics are used for alerting, logs for forensics, and traces for overall performance.

Measurement data is taken from its source and placed into a backend database. This puts the information an analyst needs in one location and avoids the necessity of logging into multiple services across the network.

[27] https://www.w3.org/TR/trace-context/

Time values in a distributed system are going to be different on different hosts. Mechanisms other than time are used to determine the sequence of events across different hosts.

Logs are generated by a service and reflect events of interest including entry and exit values.

Trace IDs are assigned when a request is made to a system. The trace may also have a context that can be used for routing or affecting behavior. Spans are named subsets of a trace. Visualizing spans over time allows an analyst to determine where time is being spent.

6.9 Exercises

1. Use Logstash to capture the log files generated by Bookinfio from exercise 4 in Chapter 5 Container Orchestration. Use Nagios to monitor a virtual machine and demonstrate that it determines when it has been powered down.

2. Ping a system on your local network and determine the time drift between your host and the system you pinged.

6.10 Discussion Questions

1. For one of your current projects, enumerate several contexts that could be of interest.

2. How frequently should a log agent send log files to a backend? What are the tradeoffs?

3. Determine the information collected by AWS CloudWatch or the equivalent from your cloud provider and how that information can be ingested by Logstash.

Chapter 7 Infrastructure Security

Many properties of software systems, such as performance or availability, depend on quality of the code you develop for your services, the infrastructure and technology that the code runs on, and the operations processes you use for your system in production. Security has the same dependencies, with one important distinction: It is likely that malicious actors will attack your system and try to exploit any error in implementation or configuration. Even systems that process no valuable data are subject to attack, to gain control of the computation and network resources and then use those resources to attack other systems (e.g., botnets) or to mine cryptocurrency (cryptojacking).

This chapter discusses the underlying technology and infrastructure needed to deliver secure systems. In Chapter 14 Secure Development, , we discuss how to use this infrastructure to develop and operate secure services. When you finish this chapter, you will have

- Some familiarity with cryptography
- An understanding of the public key infrastructure (PKI)
- An understanding of how to transport data and files securely
- Knowledge of how to control one computer remotely from another in a secure fashion
- Some familiarity with intrusion detection.

7.1 Coming to terms

Certificate – a means of certifying the owner of a web site.

Digital signature—a method of verifying that some text was sent by a known entity and not modified during transmission.

DNSSEC– strengthens authentication in DNS using *digital signatures* based on *public key cryptography*

Encryption – the process of encoding plain text into something not immediately understandable. This can be symmetric – the same key is used for encrypting and decrypting, asymmetric – different keys are used for encryption and decryption, or hashing – a one way encryption that cannot be decrypted.

PKI – Public Key Infrastructure. A system of maintaining public and private keys. These keys can be used for encryption and decryption. They can also be used for authenticating the sender or recipient of a message.

SSH – Secure Shell. A means of allowing remote access without requiring credentials after initialization.

TLS – Transport Level Security. A protocol for secure messaging over the internet. It supersedes SSL (Secure Socket Layer).

7.2 Introduction

A secure system provides three properties, often abbreviated and easily remembered as *CIA*. The system maintains *confidentiality*, so that sensitive information and resources are accessible only to authorized viewers. The second property is *integrity*, which allows users to trust that the information is not corrupted or modified. Finally, *availability* ensures that the information and resources are accessible when they are needed. Note that availability extends the notion of service reachability and responsiveness to a deeper level, such as ensuring that you don’t lose critical system passwords or encryption keys.

Although security is more than simply “keeping secrets,” we will first focus on cryptography, which is essential to achieving confidentiality and is often used to address integrity. We will then discuss how cryptography can be used to create trusted connections between systems and services, both within and between enterprises.

We will discuss how technologies such as cryptography and secure connections function, as this knowledge is necessary to understand how each technology helps and hinders the properties of confidentiality, integrity, and availability. Also, this understanding enables you to correctly use these technologies when developing services and to know what processes need to be in place to operate secure systems in production.

We are NOT discussing this technology so that you can implement your own version of it. In fact, a basic rule of secure software infrastructure is, “Don’t implement your own version of any secure software-infrastructure technology.” Bruce Schneier, an expert in software security, wrote “Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break. It's not even hard. What is hard is creating an algorithm that no one else can break, even after years of analysis. And the only way to prove that is to subject the algorithm to years of analysis by the best cryptographers around.”[28]

Schneier’s quote does not mention an important distinction between the algorithm used to achieve a security function and the implementation of that algorithm. It is not only the algorithm that is significant but also the implementation: Even the most theoretically secure algorithm must be implemented in a fashion that cannot be compromised. The United States National Institute of Science and Technology (NIST) subjects both algorithms and implementations to exhaustive testing, and you should use only software that incorporates tested implementations of tested algorithms.

There are many commercial and open-source packages to perform the functions we discuss here. Use them.

7.3 Cryptography

Cryptography is the process of secret communication using *ciphers*, which are algorithms to encrypt or decrypt data. You start with *plaintext*, e.g., this page of text, and use *encryption* to transform it into *cyphertext*, which cannot be understood by humans. You can then use *decryption* to transform the cyphertext back to plaintext, which you can read and understand.

For example, consider a simple cipher, attributed to Julius Caesar. You take a string of plaintext and encrypt it by replacing each letter with the letter that is a fixed distance later or earlier in the alphabet. You decrypt the ciphertext by reversing the shift.

```
SHIFT = +5

••• GHIJKLMNOP •••          SECRET

                    ↓ encrypt  ↑ decrypt

••• LMNOPQRSTU •••              XJHWJY
```

This cipher is well suited to quick manual operation but is not very “strong.”

[28] *https://www.schneier.com/blog/archives/2011/04/schneiers_law.html

What is meant by the *strength* of a cipher? You assess ciphers using two measures–cost to break and time to break. An attacker can often trade between these, spending more to reduce time, or increasing the time using less costly resources. You want to choose ciphers that ensure that the cost to break the cipher exceeds the value of the encrypted information and/or the time to break the cipher exceeds the useful life of the information. For example, the Data Encryption Standard (DES) cipher was widely used in the 1970s through the late 1990s, until it was cracked in 1998 using hardware costing approximately $250,000 USD and taking 56 hours. (DES was then replaced by other ciphers). Analysis of modern ciphers, such as the Advanced Encryption Standard (AES), shows that brute-force attacks would take billions of years, even considering improvements in hardware.

In addition to brute-force attacks, there are indirect, or *side-channel* attacks that target specific hardware or software implementations of the cipher that inadvertently leak data or information. For example, execution-timing information can be used to infer the internal state of the algorithm. Finally, there are attacks such as *social engineering* and physical theft, which are independent of the cipher. Social engineering is convincing a human to give the attacker passwords or keys. *Phishing*[29] is an example of using social engineering.

Ciphers use a *key* (like a password) for encryption and decryption. In the simple Caesar cipher above, the key is the shift value. Stronger ciphers use longer keys. For example, the key for AES can be 128, 192, or 256 bits long.

There are three types of ciphers. A *symmetric* cipher uses the same key for encryption and decryption, while an *asymmetric* cipher uses different (but mathematically related) keys for encryption and decryption. The third type of cipher, a *hash*, is a one-way operation that performs encryption but no decryption.

Symmetric ciphers are fast, performing at least 4000 times faster than asymmetric ciphers. They are well suited to cases where the same party is performing the encryption and decryption, such as encrypting the file system on the disk of your laptop computer. We refer to data stored on a file system as *data at rest*. These symmetric algorithms can also be used by multiple parties to protect *data in transit* (for example, data passed over a network) if there is a process to securely communicate the shared key between the parties. The weaknesses of symmetric algorithms are that anyone who discovers the key can decrypt the data (i.e.,

[29] https://en.wikipedia.org/wiki/Phishing

eavesdrop on the communication without being detected). Also, unlike asymmetric ciphers, since anyone with the key can correctly perform the encryption, these algorithms do not provide authentication of the party encrypting the data. The DES and AES algorithms mentioned earlier are examples of symmetric ciphers.

Asymmetric ciphers are more complicated. These ciphers are sometimes referred to as *public-key algorithms*, because one of the keys is called a *public key* and the related key is called a *private* or *secret key*. Plaintext encrypted with the public key can be decrypted only with the private key, and vice versa. This allows us to use these algorithms for both communication (you encrypt a secret message using my public key and then only I can read it, using my private key), and for authentication (I encrypt a message using my private key, and anyone can decrypt it using my public key and know that the message came from me). The similarly named topic of *public key infrastructure (PKI)* is the technology that allows trustworthy distribution of public keys among parties in different enterprises.

We have said that the public and private keys are mathematically related. They are generated based on the assumption that if we have a number n that is the product of two large prime numbers p and q:

$$n = p * q$$

then it is computationally very difficult to determine p and q if we know only n. (Note that p and q are not the public and private keys, but p, q, and n are used to generate the keys.) “Computationally very difficult” in this context means that determining p and q is NP hard where NP hard (non-deterministic polynomial time) is a concept in computational complexity.

You generate a large prime number using a random-number generator. Most random-number generators are “pseudorandom.” That is, given the same starting seed, an implementation will always generate the same sequence of random numbers. Pseudorandom number generators are not suitable for implementing security algorithms since two different computers (yours and an attacker’s) may use the same pseudorandom number-generator implementation and the attacker could make a brute-force attack to guess the start key. Some computers include a hardware device that can generate a truly random number based on some physical phenomenon such as environmental noise collected from device drivers. This phenomenon is not repeatable within a specific computer, let alone across computers, and so the stream of numbers generated is truly random and the keys created using those numbers will also be truly random.

Recall that asymmetric ciphers are thousands of times slower than symmetric ciphers, making them impractical for exchanging large messages. In practice, you use an asymmetric cipher to establish the identity of one of the parties. Next the two parties generate a secret symmetric key for the communication session. Secure communication then switches to using a symmetric cipher with the session key. A common example of this sequence is the Transport Layer Security protocol, which we discuss below.

The third type of cipher that we identified above is the hash. A hash is a one-way transformation, which allows encryption but not decryption. A hash maps a variable-length plaintext message to a fixed-length ciphertext using a deterministic and public algorithm. The mapping has the properties that (1) it is not feasible to determine the plaintext given the ciphertext, and (2) any ciphertext output can be produced by one and only one plaintext message (there are no collisions). Hashes are important for providing integrity. For example, a file that will be posted for download on a website may have its hash shown on the download page. After downloading the file, you can compute the hash of the file you received and compare it to the hash shown on the website to verify that the file you retrieved was not tampered with and that the file was not corrupted during the download process.

A hash can also provide confidentiality by allowing us to compare two plaintext messages without exposing the plaintext, by hashing each message and comparing the hashes. This is used, for example, for password storage: Only the hash of a password is stored, a user’s response to a password challenge is hashed, and only the hash is communicated and compared to the stored value. The password itself is not stored and, hence, cannot be compromised.

As with other security mechanisms, hashing algorithms have evolved. The current NIST recommended hashing implementations are based on SHA-3.

7.4 Key Exchange

Before two parties can use a symmetric cipher for confidential communication, they must agree on a shared secret. In the early 1970s, Ralph Merkle devised an approach for two parties communicating over an open channel to agree on a shared secret. This was later formalized and published by Whitfield Diffie and

Martin Hellman in 1976.[30] Although the algorithm is commonly referred to as Diffie-Hellman Key Exchange, the shared secret is never explicitly exchanged. Instead, each party generates a private secret value, and then exchanges information that allows both parties to compute a common, shared secret

The security of the algorithm depends on the difficulty of factoring large integers when using modulo arithmetic. We describe it in terms of colors (following the approach in Wikipedia). This makes it easier to understand. Our two protagonists are Alice and Bob.[31] Figure 7.1 shows Alice and Bob sharing a common color and each having a secret color which they do not share with anyone, including each other. Next. Alice and Bob each mix their secret color with the common color. Alice sends her mixture to Bob and Bob sends his to Alice. This is over an open channel where there may be eavesdroppers. The eavesdropper can see the common color and the two mixtures. It is difficult to determine the color that was added to the common color to produce the mixture. Finally, Alice adds her secret color to the mixture and Bob adds his secret color to the mixture. They both end up with the same color (three components) and this is the shared secret. Secrecy is achieved because any eavesdropper sees only the common color and the mixtures and cannot practically "unmix" them to find the secret colors. The algorithm, in fact, uses large prime numbers and modulo arithmetic but the basic idea is given by the colors; the "unmixing" corresponds to factoring large integers that we mentioned above.

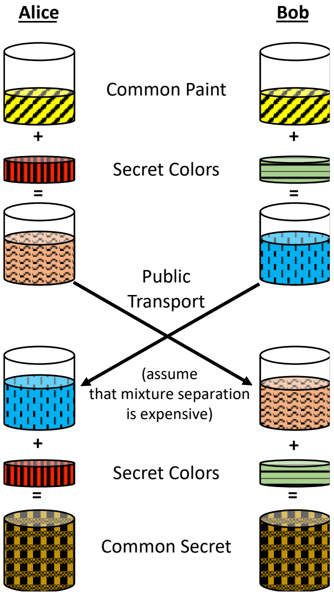
Diffie-Hellman provides a confidentiality property called *forward secrecy*. If implemented properly (e.g., the data value corresponding to the secret color in our simplified description above is immediately cleared from memory), every communications session creates a new *ephemeral* shared secret that cannot be re-created. If one of the parties in the exchange is later compromised, past communications sessions remain confidential. Compare this to using a pre-shared secret, or public/private key. If the pre-shared secret key or a private key is compromised, then every past message encrypted with that key can be decrypted. This is one reason that Diffie-Hellman key exchange is built into many secure communications protocols, such as IPsec for establishing a VPN connection (we

[30] Three members of the British military discovered a similar approach for key exchange in 1969, but the work of James H. Ellis, Clifford Cocks, and Malcolm J. Williamson remained secret until 1997.

[31] In 1978, the authors of the original RSA algorithm named the parties in their examples as Alice and Bob, and this became a convention in this domain. A third party is usually named Carol, Eve is a passive eavesdropper, while Mallory is a malicious attacker. See https://en.wikipedia.org/wiki/Alice_and_Bob.

discussed VPNs in Chapter 3 Networking), and both Transport Layer Security (TLS) and SSH secure shell protocol, which we discuss later in this chapter.

Figure 7.1: Intuitive Picture of Diffie-Hellman[32]



7.5 Public Key Infrastructure and Certificates

Cryptography and ciphers allow us to hide (encrypt) and recover (decrypt) information. This is one part of confidentiality but recall that confidentiality means that we expose information only to *authorized* users. We need a mechanism to identify with whom we are communicating.

Pre-shared keys (PSK) is one solution to authentication. Using pre-shared keys (e.g., passwords) to authenticate clients to your service might seem like a good approach, however this can be difficult to scale up: A new client's key must be pushed to every service that they need to access, and to revoke a client's access, its key must be invalidated or deleted from every service to which it had access. For this reason, enterprises adopt technology such as Kerberos, LDAP, or Vault to centralize authentication–clients and servers establish relationships with the centralized service, which brokers client authentication. Client keys are pre-shared

[32] https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange

only with the centralized service, which makes adding and revoking clients easier. We discuss this further in Chapter 14 Secure Development.

As you move outside of your enterprise, you have two levels of authentication to consider. You must ensure that the machine and service that you connect to is the one that you intended (i.e., you are not connected to an imposter), and then you must exchange information to ensure confidential communication.

Consider this exchange between two parties,[33] Alice and Bob, with malicious Mallory herself in between.

1. Alice sends a message to Bob, which is intercepted by Mallory:

   Alice: *"Hi Bob, it's Alice. Give me your key."* Since the message is intercepted by Mallory, Bob does not get this original message.

2. Mallory relays this message to Bob; Bob cannot tell it is not really from Alice.

3. Bob responds with his encryption key which actually goes to Mallory.

   Mallory replaces Bob's key with her own, and relays this to Alice, claiming that it is Bob's key.

4. Alice encrypts a message with what she believes to be Bob's key, thinking that only Bob can read it.

   However, because the message was encrypted with Mallory's key, Mallory can decrypt it, read it, modify it (if desired), re-encrypt with Bob's key, and forward it to Bob.

This type of attack, referred to as *man-in-the-middle* or *MITM*, is possible in this exchange because we have no way to dependably identify, or authenticate, the party that we are communicating with. For example, Mallory can easily create a legitimate-looking wireless network in a public space like a coffee shop, and intercept all messages sent by anyone who connects to the network. If Bob is a web server for a bank, and Alice a bank customer, then the consequences of Mallory’s actions could be significant.

Sidebar: Domain Name Variation Attacks

[33] Taken from https://en.wikipedia.org/wiki/Man-in-the-middle_attack

Another approach that malicious actors use to insert themselves into communication paths is to register a domain name that is similar in appearance to a legitimate domain name. This is most often used to attack users of a website and may be combined with phishing to trick a user into clicking on a link containing the fraudulent domain name, rather than typing the fraudulent domain name directly into their browser. The attack depends on the user not noticing the slight discrepancies in the fraudulent domain name.

For example, amazon.com is a domain name for a legitimate e-commerce organization. An attacker might register the domain name amason.com or amizon.com. Enough users who are in a hurry to log in and make a purchase might miss the slight difference in the domain name, allowing the attacker to capture their amazon.com login information.

An organization operating websites that process high-value transactions may itself register some of these variations on their domain name, in order to prevent attackers from exploiting the variations.

7.5.1 Digital signature

A digital signature is a method for verifying that some text is sent by a known entity and not changed during transmission. The message is text+encrypted hash of the text. That is, the text is sent in cleartext.The text is then hashed and the hash is encrypted using the private key of the known entity. The main reason for sending the text portion of a digitally signed message in cleartext is efficiency. Hashing the text and then encrypting just the hash is much faster than encrypting the entire message.

7.5.2 Certificates

It is not practical for every pair of parties that need to communicate confidentially to establish a relationship beforehand: Securely identifying another party’s identity and exchanging keys usually requires “out-of-band” communication, for example, speaking by telephone. This has led to the evolution of public key infrastructure technology.

A certificate is issued for a limited time duration (months or years) A Public Key infrastructure (PKI) allows us to accomplish authentication using certificates. X.509

is a standard that establishes formats for the certificates that are exchanged.[34] Both public and private PKIs exist. A private PKI is used by an organization to enable the use of TLS encryption within that organization. For example, Kubernetes uses TLS for communications among the nodes it manages. Our discussion here is about the public PKI.

The public PKI begins with one or more trusted certificate authorities (CAs). A CA is an independent organization that will issue a certificate only to a party (called a *subscriber)* that can verify its identity. This identity validation can be simple *domain validation*, which ensures that the subscriber receiving the certificate actually controls the internet domain name (in the DNS that we discussed in Chapter 3 Networking) specified in the certificate. Alternatively, a certificate may be issued with *extended validation*, which is more stringent and uses offline mechanisms (e.g., published business address, corporate officer's names, and telephone numbers) to verify that the subscriber is truly the business entity identified in the certificate.

A certificate is digitally signed by the CA using the CA's private key. The recipient of a certificate will decrypt the signature using the CA's public key and verify that the hash of the cleartext matches the decrypted signature. The certificate contains an expiration date, the subscriber's name, the internet domain(s) the certificate applies to, and the subscriber's public key. An extended validation certificate will also contain other information about the subscriber, such as the subscriber's business address.

If you trust the CA that issued a certificate and you have a trusted way to receive that CA's public key, then you can use the CA's public key to authenticate a certificate presented by a party (a user, client, or service). You use the CA's public key to decrypt the digital signature. You then inspect the contents of the certificate and verify that the internet domain associated with the certificate matches the domain that you intended to connect to. You can then use the party's public key in the certificate to begin encrypted communication.

Trust is a critical element in the use of PKI. First, you must trust the CA. CAs gain trust through social and legal mechanisms. You need a trusted way to receive the public keys for the CAs that you decide to trust. This is handled by operating-system providers, who embed the public keys for widely trusted CAs directly into the operating-system image. Finally, a PKI depends on the software

[34] https://en.wikipedia.org/wiki/X.509

implementations to correctly use expiration dates and to check with the CA that a certificate has not been revoked. If a subscriber's private key has been compromised, the subscriber should inform the CA and then the CA will revoke the subscriber's certificate that contains the corresponding public key. Revoking a certificate advertises to the world that the certificate should not be trusted.

PKI requires a known CA to sign the certificate. There, typically, is a charge for this. It is also possible to self-sign a certificate. Self-signed certificates are public key certificates that their users issue on their own behalf. They are easy to make and do not cost money. Private PKIs use self-signed certificates.

We have seen two different types of processes for secure communication. Diffie-Hellman is used to establish an ephemeral shared secret between two entities over an open channel with eavesdroppers. It relies on the difficulty of factoring large numbers but not on trust in a third party. PKI relies on trusted CAs and gives the ability to sign messages and exchange certificates that enable the authentication of websites and other services. These two types of processes come together in TLS.

7.6 Transport Layer Security (TLS)

Recall that security requires confidentiality, integrity, and availability, and that cryptography alone provides only part of the solution. PKI allows you to know who is at the other end of your connection. Without authentication, you are vulnerable to *man in the middle* (MITM) attacks, such as the example discussed above.

The man in the middle example above shows that it is relatively easy and inexpensive for Mallory to insert herself into the communication path and intercept messages. Note that Mallory does two things to the messages: She reads them, and she modifies them so that communication between the parties continues without her presence being detected. Use of certificates and a PKI can prevent Mallory from modifying the message without being detected. In the example above, when Bob responds on step 3, he encrypts the message with his private key. This, by itself, is insufficient because Alice does not know his public key. He needs to send his public key as well as encrypting the message. How is Alice to know that the public key is really Bob's? In Step 4, Mallory uses Bob's public key to decrypt the message, replaces Bob's public key with her own, and re-encrypts the message with her key. If Alice and Bob are using certificates and a PKI, then instead of sending his public key directly, Bob sends a certificate along with the encrypted message. The certificate performs two functions. First, it contains Bob's public key and second, it can be verified for authenticity by the CA (i.e., the

CA verifies that the certificate has not been revoked and that it is not expired, and that the presenter of the certificate is truly Bob because Bob had to prove his identity to the CA before it issued the certificate). If Mallory modifies Bob's message, Alice can't decrypt it, because Alice will be using Bob's public key contained in his certificate. If Mallory replaces Bob's certificate with her own, the CA will say, "this is not Bob." Using certificates and a PKI limits Mallory's role to that of an eavesdropper. Now Alice and Bob can use the Diffie-Hellman algorithm to establish a secure communication channel.

Before we go into more detail about TLS, we will present a little background and history. A *socket* is a network abstraction that allows us to exchange data between two services that are running on different physical or virtual hosts. The socket API allows us to open a connection, send and receive data, and close the connection to release the resources when we are finished. The data we send over a socket connection is cleartext without any cryptographic protection. This technology was acceptable when networks were private and there were few malicious attackers, but in most cases, even within an enterprise, this approach has unacceptable risks.

The Secure Sockets Layer (SSL) protocol was introduced by Netscape in 1994 to add security (i.e., confidentiality, integrity, and availability) to data communication between browsers and servers on the World Wide Web. The SSL protocol evolved through several versions, culminating in SSL 3.0, which has further evolved into the TLS protocol, defined as an Internet Standard by the Internet Engineering Task Force (IETF) RFC 5246.

Like many protocols, TLS begins with a *handshake*, which is an exchange of messages between the client initiating the connection and the service. This exchange allows the parties to negotiate to choose a version of the TLS protocol that is compatible with both the client and the service, and to choose cryptographic algorithms that both parties support. Many internet protocols follow this negotiation pattern, which allows heterogeneous and independently evolving systems to interoperate.

During this handshake, the service sends its PKI certificate to the client. Recall that this certificate was digitally signed with the private key of the CA. The client first uses its trusted copy of the CA's public key to decrypt the certificate, which verifies that the certificate was, in fact, issued by the CA. Next, the client verifies that the internet domain name listed in the certificate matches the internet domain name that the client is attempting to communicate with. The client should also check that the certificate has not expired, and that the certificate has not been revoked by the CA. The service can also request that the client provide a certificate and perform these steps to authenticate the client; however, this is not typically done. Instead, other system-level mechanisms (e.g., passwords or API keys) are used to authenticate clients.

The client and service then use the Diffie-Hellman algorithm to agree on a shared secret that will be used to symmetrically encrypt and decrypt their communications for the rest of the session. This does not use either the client's or service's public or private keys. When the connection is terminated, the shared secret is discarded, and the next connection repeats the handshake and secret-agreement process. Recall that the Diffie-Hellman algorithm allows you to generate this ephemeral shared secret that exists only for the duration of the session, providing perfect forward secrecy.

TLS also adds a message digest to each message. The plaintext is encrypted using the ephemeral shared secret. That ciphertext is then hashed, again using the ephemeral shared secret, to produce a message authentication code (MAC). The MAC is appended to the ciphertext to produce the message that is sent. The receiver can use the MAC to verify the integrity of the ciphertext before decrypting it.

TLS underlies the HTTPS (HTTP Secure) protocol used by web browsers and REST APIs, and is used for client connections to email servers, among other uses.

This discussion focused on one operating mode of TLS. To provide backward compatibility with SSL protocol versions and to provide broad interoperability in both PKI and non-PKI environments, the client and service can negotiate among many operating modes, including modes that perform no authentication or that use public keys to agree on a session key (forgoing perfect forward secrecy). Developers must choose appropriate options when using TLS and other cryptographic protocols.

7.7 DNSSEC

The use of DNS assumes that the mapping between URLs and IP addresses is correct. It is possible for an attacker to modify this mapping so that a URL points to a malicious site. The purpose of DNSSEC (Security Extensions to DNS) is to prevent this. DNSSEC attaches a certificate to the data returned by a DNS server. The certificate is signed by the provider of the data, not the server. This means that each data item in a DNS server can be tracked back to an originator and the originator's identity has been verified by a Certificate Authority.

Use of DNSSEC depends on two actors

1. Registrants, who are responsible for publishing DNS information, must ensure their DNS data is DNSSEC–signed.
2. Network operators need to enable DNSSEC validation on their resolvers that handle DNS lookups for users.

7.8 Secure Shell (SSH)

Another essential security protocol is SSH, the Secure Shell. Despite the similarities in their names, SSH has no relationship to SSL, and to add to the confusion, the term "SSH" refers to both a protocol *and* a client-server system.

In old data centers, server administration was performed by looking at a map of the data center to find the row and rack in which the server was housed, and then walking to that rack. Each rack had a drawer that pulled out to expose a keyboard and flip-up monitor, and you turned a dial switch to connect those to the server that you wanted to administer. In today's virtualized infrastructure, you have no way to physically connect to a virtual machine or container. SSH allows you to securely connect and execute commands on a virtual (or physical) server. The early internet used programs called *telnet, rlogin*, and *rsh* for remote command execution, but these are not secure and they should not be used.

The SSH client-server system includes an agent that runs on the virtual machine or container to accept client connections, and a client. These are built into the Linux operating system as the sshd service and the ssh command. Windows clients must use a system such as PuTTY.

SSH uses the same basic approach as SSL/TLS: Public key for server authentication, Diffie-Hellman asymmetric negotiation to agree on a shared secret, and then a symmetric cipher to secure further communication. However, the SSH and SSL/TLS implementations are different, and the options are different. For example, the format of the public/private key pair is different, and the same keys cannot be used for both X.509 and SSH.

An important functional difference is that SSH does not use certificates or rely on a PKI. When connecting to an SSH server for the first time, the SSH client presents you with a hash of the server's public key and asks you to confirm that the connection is to the intended server. You may check that the hashed key matches the known server key, however most users simply accept the hash and continue the connection. The SSH client then saves the hashed key in a list of *known servers* and does not ask you to confirm the server's hash on subsequent connections. However, if you receive this warning later, this is an indication of a mistyped server name or an indication of a man-in-the-middle attack, with a malicious actor impersonating the server.

An operational difference is the use of public keys for client authentication. SSH deployments can use password authentication for clients, but it is much more common to use asymmetric cryptography for client authentication. The client's public key is securely copied to the server. When a client connects and must authenticate for access to the server, the server uses the public key to encrypt a random number and sends it to the client. The client authenticates itself by using its private key to decrypt the message, transforming the random number using the Diffie-Hellman ephemeral shared-session key, and encrypting the result with its private key. The server decrypts this with the client's public key and checks the transformation of the random number it sent.[35]

Recall that although SSL/TLS can use certificates and public keys to authenticate clients, this is less common. SSH servers are usually configured so that a relatively small number of users must be authenticated. This makes it practical to use public key authentication to enable *password-less operation*, which is essential to scripting and automation of actions on remote servers.

7.9 Secure File Transfer

There are two families of secure file-transfer systems: those based on TLS, and those based on SSH.

FTPS (FTP Secure) is based on the internet File Transfer Protocol (FTP) system, adding TLS to secure communications. Like FTP, FTPS is operating-system agnostic, which has the benefit of broad interoperability at the expense of variabilities in features such as directory-listing formats. Most web browsers support FTPS URLs. FTPS also requires the client to explicitly choose between text and binary (or image) transfer modes when interoperating between Unix and Windows systems.

[35] The transformation of the random number with the ephemeral session key is required to preserve the confidentiality of the public/private key pair. Without this transformation, an eavesdropper could observe how the public and private key encrypts the same number. Collecting this information over many sessions could provide an attacker with enough information to break the keys.

Finally, like FTP, FTPS uses two connections from the client and the server–one for control and one for transferring file data. Normally, both are secured, however there is the option to NOT secure the data connection if, for example, the file is already encrypted. Note that plain FTP is not at all secure–usernames and passwords are sent as cleartext, and it should not be used.

SFTP (SSH File Transfer Protocol) is based on SSH Version 2 (SSH2). It is a binary protocol, oriented to transfers between Unix systems (although Windows clients are widely available). SFTP uses a single connection, and directory formats are standardized and can be automatically parsed. Like SSH, password-less operation is possible, making it amenable to scripting and automation.

The SCP (Secure Copy Protocol) was originally based on SSH Version 1.x. It performed only file-copy operations and did not support directory listings or other file operations, such as SFTP. Later versions of SSH, e.g., the OpenSSH implementation of SSH2, replace the SCP implementation with a link to SFTP.

You may have the (correct) impression that there are a lot of protocols and services that implement secure connections and file transfers. This is caused by the legacy of beginning with totally unsecured communication–the original internet–and progressing over the past 30+ years through breaking of various protocols and the increasing power of computers. NIST keeps abreast of these changes and checking the NIST website for their latest publications (https://csrc.nist.gov/publications) is a way to keep up to date on the evolving landscape of security protocols and implementations.

7.10 Intrusion Detection

An intrusion-detection system (IDS) consists of hardware and software that monitor files and network traffic to detect known threats and identify suspicious or atypical behavior. There are two common types of IDS: host-based and network-based. Both look for patterns or signatures that correspond to known threats. An IDS may incorporate adaptive detection, characterizing patterns of “typical” or “steady-state” activity in a system and triggering alerts when activity patterns deviate from the “normal” operating point.

Furthermore, an IDS may be passive or active. A passive IDS monitors and alerts but takes no preventative action. An active IDS can act to prevent an attack or thwart an attack that is in progress.

A host-based IDS, such as an anti-virus system, runs on a physical or virtual host. One form of host-based IDS, known as a virus scanner, scans the file system and matches file signatures (a hash, as discussed above) to a list of signatures of known attacks. Some host based IDSs scan the files in your network. Most host based IDSs are active and will remove ("quarantine") files that match attack signatures. They will also prevent introduction/installation of new files that match attack signatures. The other type of host-based IDS is a container scanner. It is possible that some of the dependencies included in your container have malware inserted. Container scanners look at the manifest of the container images to determine whether any of the dependencies have known vulnerabilities. The manifest is called a Software Bill of Materials (SBOM) and we discuss them further in Chapter 11 Deployment Pipeline.

Because a host-based IDS is typically installed on every node in an enterprise, it should be part of the base image for VMs and containers. The threat-signature database for a host-based IDS is updated frequently (daily or even multiple times each day) and the IDS itself checks for updates and downloads new signatures from a vendor or open-source repository. The practice of completely rebuilding the base image ensures that all images start with the current signature database. Subsequent updates by the IDS are one of the few acceptable deviations from immutable infrastructure practices, where you allow a running instance to update its own configuration directly, instead of rebuilding the image.

A network-based IDS is often a specialized physical machine or an *appliance*– a physical or virtual computer that is optimized for this purpose and includes preconfigured system software. It contains one or more network interfaces that operate in *promiscuous mode*. A network interface operating in promiscuous mode accepts all messages from the subnet it is connected to, even if those messages are not addressed to that network interface. The IDS looks for attack patterns such as port scans, failed login attempts, or other malicious activity. The IDS may also perform adaptive detection, such as sensing outgoing connections from a node that previously had none.

Both host-based and network-based IDSs create alerts that feed into an enterprise's security information and event management (SIEM) system. In addition to simply raising alerts, the SIEM allows event correlation across multiple IDSs and other network resources such as firewalls.

7.11 Summary

Security involves maintaining confidentiality, integrity, and availability of a system.

Cryptographic techniques are used to preserve confidentiality and integrity. Cryptographic techniques rely on using a key to encrypt information and a key to decrypt information. Symmetric ciphers use the same key for encryption and decryption whereas asymmetric ciphers use different keys. Hashing is used for integrity checking but it is a one-way encryption technique and cannot be used for decryption.

The public key infrastructure is based on asymmetric ciphers where one key value is public and the other is private. PKI is the basis of the certificate system that you use to verify that you are connecting to the website you think you are connecting to. PKI is also used to sign messages to verify the authorship of an unencrypted message.

Two different families of secure communication protocols exist: one based on TLS and one based on SSH. SSH is used to enable one computer to control and provision another computer remotely. TLS underlies HTTPS.

Intrusion-detection systems can either scan network traffic or scan files and data you are downloading to your system.

7.12 Exercises

1. Write a program that generates a 256-bit perfectly random number.
2. Install SSH on two virtual machines. Set one up to be able to SSH to the other without use of a password.

7.13 Discussion Questions

1. What are the use cases for SSH? Is it a reasonable tradeoff to not use certificates and PKI?
2. Does the use of TLS influence the effectiveness of a network IDS? What information can you get from a TLS connection without breaking the encryption?
3. Why do we care about perfect forward security? Aren’t modern cryptographic ciphers really secure?
4. Read the first paragraph in the Introduction to Part 1 again. How many of the concepts do you now understand?

Part II Introduction

You have finished Part I, in good shape. Let’s look at interview questions for DevOps Engineers to get some guidance for Part II.

How is DevOps different from agile methodology?

Chapter 9 What is DevOps? Addresses this topic, among others. In this chapter, you will get an overview of DevOps, its motivation, and its impact on an organization’s culture and structure. It discusses metrics that are used to measure progress in adopting DevOps.

What is the role of configuration management in DevOps?

Chapter 10 Basic DevOps Tools discusses configuration management as well as other classes of tools used in DevOps.

What Are Deployment Strategies?

Chapter 11 Deployment Pipeline describes the phases your code goes through on its way to being deployed.

List the advantages of Microservices Architecture.

Chapter 12 Design Options goes into detail about microservices, their advantages as well as other design decisions that affect the architecture of a system.

Can you explain what SLO means and why it's important?

Chapter 13 Post Production discusses alerting thresholds as well as incident handling and chaos engineering.

How often should you perform patch management?

Chapter 14 Secure Development discusses authentication and authorization. It also discusses the software supply chain and vulnerabilities and their management.

In addition to chapters addressing interview questions, there are several other chapters. Chapter 8 DevOps Preliminaries discusses some topics that you may already be familiar with. You should scan this chapter to ensure that your background knowledge is adequate for the rest of Part II.

Disaster Recovery is not a topic you typically see associated with DevOps or, for that matter, with any treatment of software engineering but it is squarely in the Operations side of DevOps. 15 Disaster Recovery deals with this topic.

Finally, 16 Thoughts on the Future gives our predictions as to what will happen to you if you become a DevOps Engineer.

Chapter 8 DevOps Preliminaries

When you finish this chapter you will know

- How an executable is constructed
- How an executable is invoked
- The difference between an imperative and a declarative language.
- The difference between strongly typed and weakly typed languages.
- A definition of the quality attribute of modifiability.

8.1 Coming to terms

Cohesion – a measure of how strongly the responsibilities of a service are elated to each other.

Command line interpreter- a program that translates commands intended for the operating system. It interprets the command and calls the operating system to achieve the desired result.

Compiler – a program that translates another program into machine language.

Coupling– a measure of the overlapping of responsibilities of multiple services.

Declarative language – a language that specifies a desired end state.

Executable – an image that can be invoked and placed into execution.

Imperative language – a language that specifies a sequence of state changes.

Interpreter – a program that takes as input statements from another program and executes them one at a time.

Linker – a program that takes as input object modules and produces as output an executable.

Modifiability – a quality attribute that measures how easy services are to change.

Object module– the machine language output of a compiler.

Path– A description for an operating system informing the OS where to look to find files.

Runtime library– a collection of methods that support standard language activities such as read, write, or memory management.

Strongly typed language – a language in which variables must be explicitly typed.

Weakly typed language – a language in the the run time determines the type of a variable from context.

8.2 Preparing a program for execution

A computer executes machine code. Programmers today typically write in higher level languages and these languages must be translated into machine language to be executed. There are two methods for translating higher level languages into machine language.

1. Compilation is one method translating higher level languages such as C, C++, C#, or Java into machine language. The program is segmented into modules and one module at a time is read by the *compiler* and translated into machine language for the target computer. The output of the compiler is called an *object module*.
2. An *Interpreter* is another method for translating higher level languages such as Python, Perl, JavaScript, Ruby into machine language. An interpreter reads one statement of the program, translates, and executes it. It then reads the next statement to be executed, translates, and executes it. There is no explicit output of the interpreter distinct from the output generated by the program being interpreted.

These two different methods of translating higher level code into machine language means there are two different paths to execution. First, we discuss *run-time libraries*.

When a write statement in your program is executed, whether compiled or interpreted, there is a complicated interaction with the operating system to execute the write. This interaction is accomplished by a write method in the run-time library. The run-time library is a collection of methods that support standard language activities such as read, write, or memory management.

8.2.1 Path to executable for object modules.

When using a compiled language, your code is translated into a set of object modules that contain the machine language equivalent of your code. These object modules must be *linked* together to create a single executable. The linking process involves resolving calls to object modules not included in the object module you just created. This may involve accessing third party libraries as well as the other object modules of your system.

The linking process is typically performed by a separate build tool such as Make. The output of a build tool is an *executable*. The executable requires an operating system to operate. It contains a collection of machine language instructions and calls to operating system functions.

The executable is a set of bits, as we discussed in Chapter 1 Platform Preliminaries. This set of bits is stored on a disk file. When we discuss the deployment pipeline in Chapter 11 Deployment Pipeline, you will see that the executable file can be copied from location to location ending up in a production environment.

8.2.2 Path to executable for interpreted code

When you code for an *interpreter*, the executable is the interpreter. The code that you have written is data for the interpreter. Your code, typically, exists as a file which is read by the interpreter. To execute your code, the interpreter must be given a pointer to the file containing your code. The interpreter also requires an operating system to execute.

To move interpreted code from one location to different destination, both the interpreter and the code must be moved although frequently the interpreter may have been previously located at the destination.

Executing interpreted code is in some ways similar to executing a container. The container requires a container run-time engine which, in turn, requires an operating system. The container run-time engine may have been previously loaded at a destination for the container.

8.3 Invoking an executable

The prior section described how an executable is built and stored on a disk. It still must get into execution. It is invoked by the operating system in response to a request from another program. The other program can be a *command line interpreter* (CLI), a program that reacts to selection of an icon on the screen, or another system.

The invoking program asks the operating system to invoke an executable. The executable is located by name or by disk location, read into memory, and the operating system transfers control to it. Implicit in locating the executable is the concept of *path*. A path description to an operating system tells the OS where to look to find files. In this case, the file is an executable, but it could be any file.

8.4 Imperative and Declarative languages

Languages, whether compiled or interpreted, follow one of two paradigms.

1. Declarative. A declarative language specifies a desired end state. It is left to the language compiler or interpreter to determine how to achieve that end state. HTML is, perhaps, the best-known example. In HTML, you state characteristics of the display such as *title* and *paragraph.* You do not state the font size or spacing of the title or paragraph although that can be specified elsewhere. This allows each browser to determine the characteristics of the display in which the HTML is to be displayed and adjust the font size and spacing to that display.
2. Imperative. An imperative language specifies the steps to go through to achieve a desired state. It does this through a series of statements, each of which specifies an incremental state change. The result of which is, hopefully, the desired state. Common imperative languages are C and its variants, Java, JavaScript, PHP, or Python.

We will see a variety of tools in subsequent chapters and the scripting for these tools is, typically, declarative. Many declarative languages allow escaping to an imperative method when more specific specifications are required.

8.5 Strongly typed and weakly typed languages

Numbers in a computer system may be integers or real numbers. Their internal representations are different. Strings have yet a different representation. Thus, the number “138” may have three different representations. The representation is the *type* of the number. This becomes important when you wish to manipulate the number – adding it to another number, printing it out, or passing it between modules..

In a programming language, you either explicitly specify the type of a variable – a *strongly typed* language – or you allow the compiler or interpreter to determine the type from the context of its use – a *weakly typed* language.

Strongly typed languages have the advantage that errors caused by misinterpreting the type are detected at compile or interpretation time and not at run time. Strongly typed languages have the disadvantage that they require the coder to explicitly specify the types. Weakly typed languages do not require the coder to specify the type but may lead to run time errors caused by misinterpreting the type.

C, C+, and Java are strongly typed languages. Perl and PHP are weakly typed.

We will see echoes of this concept when we discuss communication styles in Chapter 12 Design Options.

8.6 Modifiability

In Chapter 1 Platform Preliminaries, we discuss the qualities of performance and availability. When you design a system, you need to keep the quality of modifiability in mind. Modifiability is a driving force behind the design of a system based on microservice architecture.

A system is modifiable if changes to that system are easy to implement. Performance and availability are run-time qualities. Modifiability is a development time quality. It is measured by the cost of changes in terms of effort or number of services modified.

Some changes can be anticipated but many cannot. Designing for modifiability in the face of unanticipated changes is a matter of increasing the *cohesion* of a service and reducing the *coupling* between services.

A change that only affects one service is easier to make than a change that affects several services. Two services are tightly coupled if their responsibilities overlap such that a change to one service will require a change to the other. Examining dependencies among services is a good technique for determining the coupling.

Cohesion is a measure of how strongly the responsibilities of a service are related to each other. If a service supports two different and disconnected functions, then it will have low cohesion. This increases the possibility of side effects. Side effects occur when a change to one function accidentally has an impact on another function. If these functions are in the same service, then the probability of an impact increases. Having disconnected functions in different services will improve modifiability.

When we discuss microservices, one question will be “how are the services divided?”. Keeping coupling and cohesion in mind when designing a system that uses microservices will make changes to that system easier.

8.7 Summary

A compiled program must be built into an executable by having the compiler output an object module. This object module is then linked with all related object modules.

An interpreted program is built into an executable by using your code as input to the interpreter which then executes your code one statement at a time.

An executable is invoked by a program asking the OS to read the executable into memory and then transfer control to it.

A program in a declarative language specifies a desired end state. A program in an imperative language specifies a sequence of state transformations with the goal of leading to a desired end state.

Strongly typed languages require and explicit definition of the type of variables. Weakly typed languages do not.

Designing microservices to be modifiable will make the inevitable changes easier to implement.

8.8 Exercises

1. Examine your latest program. What might be done to it to reduce the coupling?
2. Compile a module. What object module is created and where is it located?

8.9 Discussion Questions

1. Identify a declarative language with escapes to imperative language.
2. What constraints are there on moving executables? That is, once an executable is moved, what might keep it from executing properly?
3. What is the relationship between modifiability and performance?

Chapter 9 What is DevOps?

This chapter gives an overview of DevOps. When you finish this chapter, you will know

- A definition of DevOps
- The goals of DevOps practices
- An overview of DevOps and the software development life cycle
- Cultural changes brought about by adopting DevOps practices
- Organizational changes brought about by adopting DevOps practices
- Metrics to collect to measure the impact of DevOps practices
- Meaning of the term DevSecOps

9.1 Coming to terms

Deployment time – the time it takes to move through the deployment pipeline from code complete to production.

Environment – a collection of resources that enable development, test, or production.

Incident handling – the process of responding to a problem with a system in production.

IT organization— A synonym for operations staff.

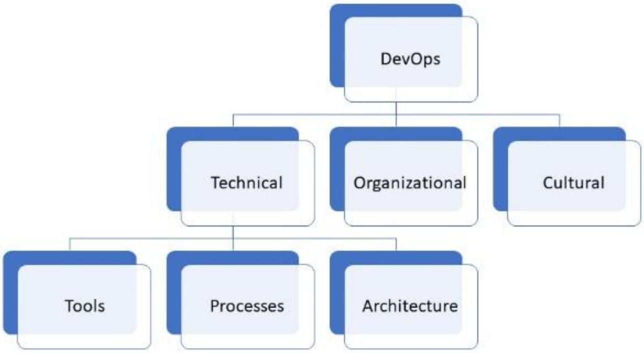
Portmanteau – a word created by combining two other words.

9.2 Introduction

DevOps is a portmanteau of Development and Operations. It is intended to signify the breaking down of barriers between developers and operators. Historically, developers would deliver completed systems to operations which would then be responsible for provisioning hardware and running the system. This division caused problems because of poor communication and differing priorities. Today, however, DevOps has a much broader meaning.

Wikipedia defines DevOps as “a set of practices that aims to shorten the systems development life cycle and provide continuous delivery with high software quality”.

Figure 9.1: DevOps Decomposition



Defining DevOps as a set of practices puts it squarely in the Process Improvement or Business Process Reengineering framework. As with all Process Improvement efforts, there are organizational and cultural implications of process change. In addition, there are metrics that can be used to determine whether improvement is happening. DevOps differs from other Process Improvement efforts in that there are significant technical aspects of DevOps. Figure 9.1 shows a decomposition of DevOps. We will see when we discuss the technical aspects of DevOps that it can be decomposed into tools, process, and architectural aspects. In the remainder of this chapter, we will elaborate on motivation, introduce the technical aspects, and discuss the cultural, organizational, and metrics portions of DevOps.

Recently the term DevSecOps has come into prominence. The intent is to add the Security team to the Development and Operation teams in developing DevOps solution. The effect is that DevOps practices consider security issues as they are implemented. In this book, we will use the term DevOps with the understanding that security must be considered in DevOps practices.

9.3 Motivation

The terms DevOps and DevSecOps represent the desire to break down barriers between Dev, Ops , and Security, but that, by itself, is not enough to motivate organizations to invest in DevOps. The fundamental driver for DevOps is, as stated in the Wikipedia definition, the desire to shorten the software life cycle. In particular, the goal is to shorten two times:

1. The time between the committing of code by the developers – code complete – and the placing of that code into production – deployment time.
2. The time to detect and repair problems after the code goes into production – incident handling.

9.3.1 Deployment time delays

Constructing or modifying large systems requires multiple developers. These developers are divided into teams. A new function or modification to an existing function is analyzed by the project management and portions are allocated to one or more teams. Traditionally, each team develops their portion according to predefined project processes and checks the results into a version control system. After all the teams have completed their portion, the portions are integrated into a complete executable, tested, and placed into production.

Each of these steps has delays and the possibility of errors being introduced.

- Defining project processes takes time. Agreement must be reached on a common language, versions of libraries, interfaces, supporting tooling, and coordination processes. Getting this agreement takes time, documenting the agreement takes time, and not all teams will remember or abide by the agreements.
- Integrating the output of multiple teams exposes errors and misunderstandings. Incorrect use of interfaces, inconsistent library versions, inconsistencies in development environments will all cause delays. Integration cannot be completed until all the teams have finished their portion and until all third-party dependencies are available. Finding and repairing errors and misunderstandings will take time. Waiting until all teams have finished takes time.
- Manual testing also is time consuming. The testers must define the test cases and their expected results.

- Placing the tested executable into production can expose errors related to access control and inconsistencies between the development environment and the production environment.

These steps and the time involved led to scheduled releases. Monthly, quarterly, or yearly were common. DevOps practices have reduced the time and the errors significantly. Many organizations now report multiple daily deployments. The scheduling of deployments has become a business decision rather than being constrained by technical considerations.

9.3.2 Incident handling delays

Traditionally, when an incident occurred, an operator observed or was notified of the incident. If repairing the incident was within the operator's ability, then it was repaired, a report was filed, and the system continued operations. If repairing the incident was not possible for the operator, a ticket describing the incident was filed and the ticket was routed to developers who then took responsibility for repairing the problem. Diagnosing and repairing took time and, for many problems, the repair was included in the next scheduled release.

DevOps practices have changed this sequence as we will see in Chapter 13 Post Production.

9.3.3 Security incidents

You have read about various security breaches. The number of such breaches increases dramatically year to year. The Identity Theft Resource Center reports that over a third of such breaches are caused by configuration errors.[36] These include both mistakes configuring the firewall – allowing attackers to access internal systems they shouldn't have been able to see from outside the organization – as well as cloud systems and servers that were misconfigured to allow unauthorized access.

9.4 Introduction to the technical aspects of DevOps

Figure 9.2 shows the DevOps processes during the software development life cycle. It is expressed as a loop because successful systems are never complete but are continually modified with new features or fixes for problems. The life cycle has the following stages

[36] https://notified.idtheftcenter.org/s/2021-data-breach-report

- Design. The design must accommodate solutions to deployment problems and visibility for monitoring and analysis of the system in production.
- Development. DevOps automation involves a collection of scripts, and these are, typically, created and maintained by the developers.
- Build. An executable image is created automatically when code is checked into a version control system. Errors are returned to the developers.
- Test. Tests should be automated. Errors are returned to the developers.
- Release. Some domains require manual approval of a release. Others allow automatic release.
- Deploy. The executable image is placed in production.
- Operate. The system performs its functions. Data is collected while it is operating.
- Monitoring and Analysis. The data collected during operation is monitored and analyzed. This may result in architectural modifications which are performed by returning to the design stage.
- The Development, Build, Test, and Operate stages all exist in their own environment. An *environment* is a collection of provisioned resources. The resources include an operating system, network connections, memory, CPU, and any software dependencies.

At each stage of these processes, there are security reviews and checks that can be performed. Scanning tools will do some of the checking but having a security presence in the DevOps team also provides the opportunity for an expert pair of eyes to examine scripts and other DevOps code artifacts.

There are other DevOps technical processes such as Infrastructure as Code (IaC) and use of tools for various processes and we will discuss these in Chapter 10 Basic DevOps Tools.

9.5 Culture

The cultural shifts involved in adopting DevOps are extensive. They mostly have their roots in earlier software engineering efforts such as egoless programming in the 1970s and agile development methods in the 2000s. We compare DevOps practices with agile practices and then discuss DevOps specific practices.

9.5.1 DevOps and Agile

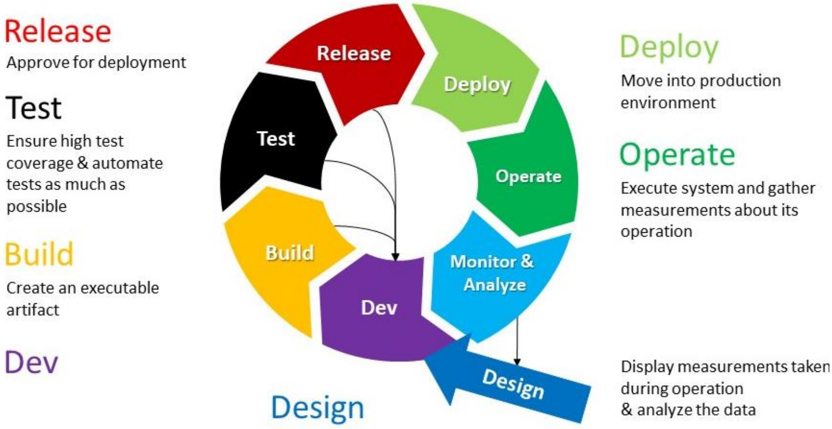
The major difference between DevOps and Agile development is in their focus. DevOps has a focus both on development and operations whereas agile has its focus on development.

This difference in focus is manifested in the various practices advocated by the two movements. Agile focusses on management practices such as Scrum, Kanban, etc. whereas DevOps focuses on tools involved in moving through the DevOps practices loop shown in Figure 9.2.

Agile practices advocate obtaining feedback from customers whereas DevOps practices involve obtaining feedback from measuring the running system.

Both sets of practices advocate a non-blaming, open, sharing culture. All members of the team should have access to the same information, communication channels that provide rapid sharing and feedback. Periodic retrospectives are advocated by both groups and problems are analyzed for the underlying causes without finger pointing and personal comments.

Figure 9.2: DevOps Processes


Release
Approve for deployment
Test
Ensure high test coverage & automate tests as much as possible
Build
Create an executable artifact
Dev
Design
Release
Deploy
Test
Operate
Build
Monitor & Analyze
Dev
Design
Deploy
Move into production environment
Operate
Execute system and gather measurements about its operation
Display measurements taken during operation & analyze the data

9.5.2 DevOps specific cultural practices.

Two cultural practices are specific to DevOps.

- Mindset. Several items make up the DevOps mindset
    a. I if an activity is repeated multiple times, it should be automated. Tools are an essential aspect of DevOps. The automation can consist of scripts or of the adoption of a tool. In any case, a DevOps practitioner should always be alert to the possibility of automation.
    b. Measure and monitor. This is manifested by measuring both the processes and the product. Associated is monitoring the measurements. Taking measurements without monitoring and analyzing the results does not help improve the processes or the products.
    c. Keep security constantly in mind. Vulnerabilities can come from the supply chain, the deployment pipeline, orchestration practices or credential management.
- Incentives. Traditionally, developers were incentivized to produce new releases whereas operators were incentivized to keep systems operating. Introducing new releases threatened the stability of operations. These two sets of incentives produced a tension between the two groups. A cultural shift introduced by the DevOps movement is the development of shared incentives for the two groups. Incentives such as “successfully operating a new release” motivate both the generation of new releases and the stability of such releases.

9.6 Organization

Adopting DevOps practices will cause new roles to be created and add some new responsibilities for existing roles. It may also cause new organizational units to be formed. The extent of the changes will depend on which practices are adopted and the particulars of the organization adopting them. What we present here are changes that have occurred in some organizations. We begin with organizational units.

9.6.1 New and changed organizational units

The quality assurance (QA) unit may disappear, or it may have its responsibilities altered. DevOps practices call for automated testing. Traditionally, the QA unit was responsible for creating and executing tests and deciding whether a system was suitable to be deployed. With automated tests and automated deployment, the responsibilities of the QA unit are substantially reduced if not eliminated.

A new unit may be created to deal with incident handling. System Reliability Engineer (SREs) are the first responders when an incident occurs. They exist in a separate organizational unit that assumes responsibility for the reliability of systems. Google initiated this unit, and it has been adopted in other organizations.

Meta (Facebook) has created a unit to perform Production Engineering. This unit's responsibilities are to "champion the Reliability, Scalability, Performance, and Security posture of production services."

A unit to manage and create tools has emerged in multiple organizations. Google and Netflix create their own tools and other organizations have a unit to manage their tool suite.

9.6.2 New and changed responsibilities

New units automatically imply new roles, but many existing roles are assuming new responsibilities with the adoption of DevOps practices.

Traditionally, the operations group was responsible for resource acquisition and management. A development team would create a system and the operations team would acquire resources to operate that system. The Cloud has changed that. Developers now have responsibility for allocating resources and for reviewing and acting upon the monitoring information collected from operating the system.

Microsoft and Alibaba Cloud have announced the "Open System Model" which defines responsibilities for developers and operators in terms of platform-based tools

- Developer builds systems in containers
- System operator packages and deploys containers
- Infrastructure operator tunes as necessary at run time.

One model for incident handling is "You built it, you run it". This model, introduced by Amazon, makes developers the first responders in the case of an incident.

9.7 Metrics

A process improvement effort must be able to measure whether the changes it introduces have a positive impact. Choosing the correct metrics is important since emphasizing the wrong metrics will lead to incorrect behavior. Also, having too many metrics makes it difficult to optimize any single one. DORA (DevOps Research and Assessment) proposes the following four metrics that have become widely adopted.

1. Lead time for changes. Lead time for changes is the length of time between committing a code change to the trunk branch and when it is in a deployable state. For example, when code passes all necessary pre-release tests.
2. Change failure rate. The change failure rate is the percentage of code changes that require hot fixes or other remediation after production. This does not measure failures caught by testing and fixed before code is deployed.
3. Deployment frequency. Deployment frequency is the number of deployments into production per unit time. For example, Amazon reports thousands of deployments per day.
4. Mean time to recovery. Mean time to recovery (MTTR) measures how long it takes to recover from a partial service interruption or total failure.

Notice how these metrics line up with the definition of DevOps we presented in the introduction. DevOps is a set of practices that aims to shorten the systems development life cycle (lead time metric) and provide continuous delivery (deployment frequency metric) with high software quality (change failure rate and mean time to recovery metrics).

9.8 Summary

DevOps Is motivated by a desire to reduce the deployment time. This is the time taken between code completion and placing that code into production.

DevOps has an impact on the software development life cycle. Most stages of the life cycle have tools to support the activities of that stage.

DevOps differs from agile in its focus. Agile focuses on getting to code complete. DevOps focusses on getting completed code into production

DevOps emphasizes automation and introduces incentives that reflect both code production and operation.

New roles and organizational units can be created because of DevOps practices. Existing roles may also have responsibilities added.

DORA metrics are widely used to measure the results of adopting DevOps practices.

9.9 Exercises

1. Determine how your organization manages its tool suite. Which roles perform which functions on a given tool?
2. How much time was spent on coordination meetings during your last project? What errors occurred despite the agreements reached at these meetings?

9.10 Discussion Questions

1. What would your reaction, as a developer, be if you were asked to be a first responder when an incident occurs? Would you be willing to take on this role if it meant being awakened at 3:00AM because of an incident?
2. In your organization, how are developers incentivized? How are operators incentivized? Is there a tension between those two sets of incentives?
3. Currently, for every 100 developers there are around 10 operators and one person devoted to security. Will the movement of responsibilities to developers and the increased emphasis on security change these numbers?

Chapter 10 Basic DevOps Tools

When you finish this chapter, you will know

- The meaning and utility of Infrastructure as Code
- A variety of different categories of tools
    - Issue trackers
    - Version control systems
    - Provisioning
    - Configuration management
- The problem of vendor lock in
- How to manage configuration parameters

10.1 Coming to Terms

Branch – the duplication of an object under version control so that development can proceed independently.

Environment variables– a variable in the operating system

Idempotent – executing a solution multiple times always achieves the same results.

Merge – joining two branches in a version control system. Inconsistencies in the two branches must be resolved.

Resource file– a file containing configuration parameters with a name and location known to the service for which the parameters are intended.

Script – a list of commands to be executed by a specific tool. Different tools will have scripts in different languages.

Trunk in a VCS – the branch of a project in a version control system from which development proceeds.

10.2 Infrastructure as Code

As we said in Chapter 9 What is DevOps? automation is a key portion of DevOps practices. Automation is manifested as a combination of tools and scripts for those tools. These tools and scripts manipulate both the code being developed and the infrastructure that supports the creation and operation of an executable from that code. The creation of scripts to manage the infrastructure is called *Infrastructure as Code* (IaC).

The name IaC is intended to convey that the scripts should be treated as code. They should be tested, version controlled, and shared just as code is tested, version controlled, and shared. The scripts used to control DevOps tools are primarily declarative as we described in Chapter 8 DevOps Preliminaries.

One source of errors for deployed systems is inconsistency among the environments used by developers. If Developer 1 uses one version of an operating system and Developer 2 uses another version, then errors may appear when integrating the output of the two developers. This is equally true of other environments. If the test environment is different from the development environment or the production environment is different from the development and test environments, then errors due to inconsistency may occur.

Inconsistencies among environments go deeper than operating system versions. Different versions of libraries can be inconsistent in terms of their behavior or even their interfaces. Different choices for configuration parameters can introduce errors as well.

IaC is used to prevent inconsistencies from occurring. A script for provisioning an environment is developed and shared among all developers. The created environment will be identical for all developers. If the environment must be updated, the script is modified, checked into a version control system, and shared with all developers. Each developer then executes the new script and has a copy of the new environment. Because the script is version controlled, it is possible to recreate an older environment if needed to make a repair to an older version being used in production.

This example demonstrates what is mean by treating the scripts as code. The description for sharing programming language code among developers and retaining older versions for error repair is the same whether the subject is scripts or programming language code.

10.2.1 Best practices

The best practices for IaC are the same as with code.

- Scripts are checked into version control system. This allows both sharing with team members and retrieving past versions of the scripts while troubleshooting.
- Scripts are scanned for security issues. Just as code can be inspected automatically and manually to determine conformance to security practices, IaC can be examined both manually—through inspections—and automatically—through scanning tools—to determine conformance to security practices. An example is creating a VM in the cloud without specifying access parameters.
- Scripts are tested and reviewed for correctness, security, and compliance. Again, as with code, various review practices can be applied to scripts and they should be tested prior to being placed into production.

10.2.2 Costs of IaC

Writing and using scripts is not free. Some of the costs are:

- Scripts must be developed. The initial development takes time.
- Scripts must be tested and reviewed. Tests for correctness must be developed and reviewers will spend time performing the review.
- Scripts must be updated when tools or processes are modified. As with other forms of software, the first version of a product that is used is almost never the last version. Scripts must be updated, and the new versions tested.

10.2.3 Idempotence

Something is idempotent if applying it twice yields the same result. The term comes from mathematics. A simple example is the identity function. A DevOps example is that a deployment command always sets the target environment into the same configuration, regardless of the environment’s starting state.

If a IaC script is idempotent, then the current state of the infrastructure will not affect its execution. So, for example, it could be scheduled to execute at periodic intervals without concern of the state of the system when it is executed. Idempotence is achieved by either automatically configuring an existing target or by discarding the existing target and recreating a fresh environment.

Scripts should be written to be idempotent. Testing idempotence can be one of the test cases for a script.

10.2.4 Infrastructure drift

Suppose you've deployed five instances of a service via IaC with the default 128 MB of memory. When you examine them, you find that three of them are 256 MB

How did this happen? It might have happened because someone noticed instances had poor performance and increased the memory allocation. Of course, there might be other explanations. People do not always follow processes or rules.

Is this bad? Suppose poor performance is subsequently noticed on these instances. Is it one of the smaller memory instances or one of the larger memory instances? The responder must determine which one since it will affect the response. This is an example of infrastructure drift.

Infrastructure drift occurs when the state of your infrastructure is different than the state generated by your IaC. It will not occur if all infrastructure changes are performed through the IaC rather than done directly on the infrastructure.

Security

Infrastructure drift can also cause security problems. For example, someone is having an access problem and changes the IAM setting to make access easier. Depending on how it was changed, it may open a system to unauthorized usage.

Detecting infrastructure drift

Infrastructure drift is a common enough problem that tools have appeared to scan the existing infrastructure and compare it to the specification in the IaC. Several open-source tools can check for infrastructure drift. They include Snyk and Driftctl.

Now we turn to the types of basic DevOps tools. We categorize the tools into issue tracking, version control, provisioning, and configuration management. Our categorization is not rigid since a natural tendency of tool vendors is to expand their feature set to cover more of the life cycle. The categories are described in terms of the functions necessary for that category and we give examples of tools in each category. Specific tools, however, may have features from more than one category.

10.3 Issue Tracking

An issue tracker is, as its name says, a tool that keeps track of issues. An issue can be a bug, a desired feature, or an incident. An issue is entered into the tracking database and given an ID. That ID is then used to identify subsequent activities. Subsequent activities can be the generation of code, the escalation or resolution of an incident, or the fixing of the bug. This allows determining the current state of any issue and the examining of the set of issues to see patterns. Issue tracking IDs are the key for tying together activities ranging from code development to the steps in the deployment pipeline to incidents that occur during operations.

Issue trackers are not, *per se*, DevOps tools since they date from before DevOps became a movement. The first web-based issue tracker was Bugzilla developed in 1998. Common issue trackers are Issue Tracker, Jira, and Solar Winds.

10.4 Version Control

A version control system (VCS) keeps track of modifications to textual files. The textual files can be programming language code, scripts, or any other system specific file such as documentation. The VCS stores the files and their modifications in a repository. The repository is shared among team members and access control mechanisms are used to prevent unauthorized access.

As with issue trackers, version control systems predate the DevOps movement. The first version control system dates from the 1970s. Modifications to a file in the version control system are accompanied by a comment that indicates the reason for the modification and, ideally, is tagged by an issue ID.

10.4.1 Basic functionality

The files in the repository are organized in groups. Each group is given a name. Typically, these names are version numbers. Thus, version 1 or file A is distinct from version 2 of file A. Different versions of file A coexist simultaneously. Version 1 might be in production and version 2 incorporates new features. Since the two versions are kept distinct, they will not interfere with each other. If a problem occurs with version 1, it can be retrieved from the repository, modified, and placed back in the repository without affecting version 2.

Sticking with this simple example, two points are important.

1. Version 2 was likely created initially as a copy of version 1. As changes are made to version 2, the two versions will diverge.

2. Any problem that was fixed in version 1 will likely exist in version 2 since version 2 was created to be a copy of version 1. The changes in version 1 must be *merged* into version 2 so that the problem does not reoccur when version 2 goes into production.

A file may have multiple versions. Version 1.2 is a copy of version 1 made after version 2 has been created. Any version of file A can be retrieved by specifying both file A and the desired version.

Files with the same version number form a tree. Creating a copy of a tree or a portion of the tree is called branching. Each branch may contain multiple files. You can create a new branch by forking an existing brach and this new fork will contain all of the files in the branch from which it has been forked.

The *trunk* of the version control tree is, typically, the latest stable branch of the tree. It is not under active development but a file in the trunk may be modified to fix problems in production. If a new branch is created from the trunk to fix a problem, it should be merged back into the trunk once the changes have stabilized to keep the trunk as the latest stable branch.

To implement these basic concepts, the VCS commands include

- Check in/check out. Check out will copy a branch or a file from the repository to the local file system. Check in reverses the process. A file is not available to other team members until it is checked in.
- Branch/merge. A branch creates a new copy of the set of files being branched. Merge identifies the differences between the two branches being merged and gives the option of choosing one alternative to continue in the merged branch.
- Version labeling/tagging. We described the labelling of a new branch or version of a file as being an automatic process. It is also possible for the end user to explicitly label or tag the branch or file.

10.4.2 Centralized vs distributed VCS

VCSs can be either centralized or distributed. Both types of VCSs maintain a central repository. The distinction between them is the process for checking in or checking out a file.

- In a centralized VCS, the end user must be online to check a file in or out. The VCS keeps track of which user has checked out a file and can prevent other users from checking out the file until it has been checked in. Subversion is a widely used centralized VCS.

- In a distributed VCS, the end user makes a local copy of either the whole repository or a branch. The user must be online to make this local copy. Once the local copy has been made, however, check in and check out of individual files can be performed without an internet connection. This gives the user more flexibility in terms of internet connectivity but prevents the VCS from knowing which files are currently being modified. Git is a widely used decentralized VCS.

10.4.3 Branching strategies

Some common branching strategies are:

- Centralized workflow: Teams use only a single repository and commit directly to the main branch.
- Feature branching: Teams use a new branch for each feature and don’t commit directly to the main branch.
- Personal branching: Similar to feature branching, but rather than develop on a branch per feature, it’s per developer. Every user merges to the main branch when they complete their work.

Organizations must decide whether to use one repository for all their projects or project specific repositories. A single repository facilitates reuse since all code is available to all developers. It also allows any developer to fix a problem with any code developed by the organization. A single repository can get confusing when an organization has many projects. Developers must identify the branch for the specific changes they wish to make. Multiple repositories decouple one project from another. The developers on a project operate independently in terms of libraries, development processes, and merging branches.

10.4.4 Best practices

Some best practices for the use of a version control system are:

- Use descriptive comments when committing and identify the commit with an ID from an issue tracker.
- Do not commit incomplete code.

- Commit logical units. If you are simultaneously working on two different issues, then commit them separately. When an issue has been resolved, commit that code rather than waiting until all of the issues on which you are working are resolved.

10.4.5 Security

In order to avoid insider attacks, some organizations require two people to verify a check in. A security review of the code must precede any check in. An insider attack would require two people to collaborate on the attack. This is much less likely than a single individual mounting an attack.

A side effect of requiring a security review is that code quality will be improved, as it is with any review.

10.5 Provisioning and Configuration Management

A provisioning tool creates an environment loaded with specified software. A configuration management tool ensures that files – including files containing VM or container images – are consistent across a collection of nodes. There is enough similarity between these two problems that we will discuss both in this section.

Together, these tools solve the following problem: You wish to have a collection of nodes loaded with identical specified environments and configured with specified versions of software. You do not care about the current state of any of the nodes, but you would like its state after you run the tool to be the one specified. If this description sounds to you that the script for the configuration management or provisioning tool should be in a declarative language (see Chapter 8 DevOps Preliminaries), you are correct. If you think that these tools are idempotent (as discussed in Section 10.2.3) you are also correct.

A configuration management or provisioning tool will examine a target node and decide whether the existing environment and software versions are consistent with the specification. If they are, no action is required. If some software elements are not consistent, those elements are updated to conform to the specification. In some cases, the update can be done without affecting other elements of the environment. In other cases, the existing environment and software elements are deleted, and new ones are created according to the specification. This is idempotent since successive executions of the tool will leave the target in the same state – that specified in the input to the tool.

10.5.1 Provisioning tools

Provisioning tools either create or modify

- A node or container at the target
- An environment. We discuss environments in detail in Chapter 11 Deployment Pipeline.

Some considerations governing the use of these tools are:

- A provider is included in the specification either implicitly or explicitly. This provider can be a directory on your local host, an on-premises data center, or a cloud provider.
- The specification will include a target name. If the target already exists, then executing the specification will make this target consistent with the specification. If the target does not exist, executing the specification will create the target.
- The desired software is loaded from the provider locations specified. These locations can be from the local host, from the on-premises intranet, or from the internet. If the desired software is from the internet, it should be screened for security issues.
- You must have permission to access the target provider, an existing node, and the desired software.
- Once the software has been provisioned, a scanning tool can check the Software Bill of Materials f(SBOM) or items with known vulnerabilities. We discuss SBOMs in more detail in Chapter 11 Deployment Pipeline.

Some examples of tools and their specification mechanisms are:

- Docker. Docker creates and provisions containers. The specification for Docker is in a Dockerfile. You should have experience with Docker from Exercise 2.6.4. See https://docs.docker.com/get-started/02_our_app/.
- Vagrant. Vagrant creates and provisions development environments. It enables all members of a development team to have identical development environments. We discussed this in Section 10.2. See https://developer.hashicorp.com/vagrant/tutorials/getting-started

- Terraform. Terraform creates and manages infrastructure. For example, your environment might consist of three instances of your system each running on a host, managed by a load balancer, and connected to the internet. Terraform will create the instances, load the images, set up the network connections, and instantiate a load balancer. Terraform specifications are given in configuration files. See https://www.terraform.io/.
- AWS. AWS Is a cloud provider, and you allocate and provision resources within it. The set of services AWS provides is continually growing but one service is CloudFormation. This enables you to create a VM within AWS. You specify the instance type, the image to be loaded into the instance, the security group settings, etc. See https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html

10.5.2 Configuration Management Tools

A configuration management tool solves the following problem: You have a collection of nodes and wish them all to have identical software loaded on them. The collection might be of laptops for an organization. You want all the laptops to have identical configurations on them. The collection might be two data centers, a primary and a secondary. You want the secondary data center to have identical software to the primary.

This sounds somewhat like the problem provisioning tools solve but the assumption behind a configuration management tool is that there exists a master copy of the various files. The configuration management system becomes a copier. The files to be made consistent are copied to the specified targets. The target nodes are passive in this process. This could be accomplished by using a provisioning tool, but that provisioning tool would need to be executed on each of the target nodes.

The configuration management tool executes on one node, the files to be copied exist on a file master and a secure communication protocol is used for communication during the copying. The secure communication protocol can be either SSH or TLS using self-signed certificates.

The targets are specified as being in categories and the configuration management tools will maintain consistency across the targets in each category. A category might be “web servers” and those targets that are web servers will be kept consistent.

Common configuration management tools are Chef (https://www.chef.io/), Puppet (https://puppet.com/), and Ansible (https://www.ansible.com/),

10.6 Vendor Lock In

Reliance on a single vendor is not desirable either from a business or a technical perspective. From a business perspective, the vendor may go out of business or discontinue the tool on which you are dependent. The vendor may raise prices.

From a technical perspective, the services provided by a vendor may suffer outages and you have no fall back when that happens. This has occurred with all the major cloud providers.

Vendor lock in occurs when you are dependent on a single vendor for a tool or services and have no economical method to switch to a different vendor.

A desire to avoid vendor lock in leads to a choice of tools that enables you to change vendors easily. The tradeoff is that vendor specific tools may have features not available from other vendors.

For example, AWS uses CloudFormation as a provisioning tool. Your CloudFormation scripts will not work for other cloud vendors. But CloudFormation has built into it knowledge of some AWS features such as how the AWS security model is managed. Terraform is cloud vendor agnostic. It has plug ins that allow you to use the same script for different cloud vendors. But security models differ from cloud vendor to cloud vendor and it is difficult to capture these differences generically. Terraform allows you to change cloud vendors but at the cost of not having direct access to cloud vendor features.

There is no right answer to the problem of vendor lock in. Standards would help you achieve portability but strict adherence to standards prevents the vendors from offering specialized features until the standards are changed. Changing a standard is a years long process.

10.7 Configuration parameters.

A configuration parameter is a parameter intended to be specified by a system administrator. Since system administrators should not be asked for input every time a system is invoked, off-line mechanisms are used to specify configuration parameters. Despite the similarity in names, there is no connection between configuration parameters and configuration management tools.

Examples of configuration parameters include network information (e.g., where is your DNS server?), database-connection information, and logging levels. All these things are usually different in each environment (e.g., development, staging, and production). Other things such as user interface background color, localization information, and security levels can also be configuration parameters.

Methods for providing configuration parameters to a service include

1. *Resource file.* A resource file is a file with a name and location known to the service. The service will read the file at initialization and assign the configuration parameters. A wide variety of formats exist for resource files. They can be key-value pairs where the key is the name of the parameter. They can be stylized language specifications such as YAML (YAML Ain't Markup Language) where the service initialization invokes a parser for the language.

2. *Environment variables.* An environment variable is a variable in the operating system. It provides a value for all processes running under the control of that operating system. Environment variables are provided to the service by the operating system when the operating system initially transfers control to the service. You can think of them as parameters in a method call from the operating system to your service. For example, Syslog is a Unix environment variable that specifies where log messages are to be stored locally. Environment variables are usually set in a script interpreted by the shell before invocation of the service.

3. *Database.* A specialized database can be used to store configuration parameters. Indices to the database include the environment and the name of the configuration parameter. Different environments have different sources for test databases. The URL of the test database can be placed in the configuration-parameter database and retrieved during service initialization. Parameters can be grouped by index fields so that the security parameters, for example, can be easily retrieved. Furthermore, access to the database requires authorization. This simplifies enforcing least privilege (see Chapter 14 Secure Development).

4. *Specialized tools.* Specialized tools provide the ability to interactively specify configuration parameters.

Regardless of the method used to manage configuration parameters, the parameters should be version controlled and a history maintained. Scripts and resource files can be stored in a version control system whereas a database used for configuration parameters will have its own method for maintaining change history.

As a developer, when you have a choice between two alternatives, it is easy to defer that decision and make it a configuration parameter. This flexibility comes with a price: When you introduce a new configuration parameter it must be managed and must be set. You should provide a default value if the parameter is not set and include checks for consistency if there are several related parameters. We know of a system that had more than 10,000 configuration parameters, and system installation became a nightmare because of inconsistent or unset configuration parameters.

Configuration-parameter names and values are typically stored in cleartext. They are not hidden or encrypted, and they are visible to anyone who has access to the source code for your infrastructure. If you are using a version control system, the values will be saved forever. Although you are probably not concerned that the world might learn the address of your DNS server, you should be concerned that the world knows your database username and password. People are watching and waiting. You want to keep some parameter values secret. You may even want to keep some of these values secret from members of the development team. We will go into managing secrets in detail in Chapter 14 Secure Development.

10.8 Summary

Scripts that create and manage development environments and infrastructure reduce errors and speed up the processes they encode. Such scripts are referred to as Infrastructure as Code. They should be tested and version controlled and, in general, treated as if they were programming language code.

An issue tracker provides an identification and status information for new features, repairing problems, or creating IaC.

A version control system maintains distinct versions of files whether those files are code, scripts, or other textual information. It maintains a history of changes and keeps logically connected files on a branch for ease of understanding.

Provisioning tools create an environment for a VM or a container. They also create and manipulate infrastructure. Configuration management tools ensure that collections of hosts have identical software, down to the version, installed.

Depending on a single vendor for a tool or service leaves an organization vulnerable to outages of the service or discontinuation of the tool. Using tools that enable a choice of vendors alleviates this problem but at the cost of not being able to take advantage of specialized features of a particular tool or vendor.

Configuration parameters are used to vary environment specific settings. Secret configuration parameters such as credentials require a different treatment than values that are not sensitive.

10.9 Exercises

1. Repeat Exercise 7.13.2 using Vagrant.
2. Perform the Terraform Getting Started Exercise. https://learn.hashicorp.com/collections/terraform/aws-get-started?utm_source=WEBSITE&utm_medium=WEB_IO&utm_offer=ARTICLE_PAGE&utm_content=DOCS
3. Execute the Ansible app config.yml example. https://www.ansible.com/overview/how-ansible-works

10.10 Discussion Questions

1. Design a language that combines the features of Vagrant and Ansible.
2. For your last project, did you lock in any vendor? How much effort would it take to change vendors?
3. For your last project, did you place any sensitive information in a script or in a log?

Chapter 11 Deployment Pipeline

After completing this chapter, you will know

- The different environments in the deployment pipeline,
- The life cycle of a environment
- The types of testing and data used for testing in each stage.
- Blue/green and rolling-upgrade models of deployment
- How to achieve consistency when performing an upgrade
- Partial deployments–canary and A/B testing–and the reasons for partial deployments.
- What is in a Software Bill of Materials (SBOM) and why it is important.

11.1 Coming to Terms

A/B testing – comparing alternative versions of a system with users to determine which version is the best from a business perspective.

Artifact database – a repository where all elements that are included in a production service are stored.

Beta test – releasing a test version of a system to a limited number of selected users to gather feedback.

Canary test – an online beta test.

CWE – Common Weakness Enumeration. A list of software weakness types.

Continuous delivery – a process that takes code checked into a version control system, builds a system, tests the built system, and is leaves the result on a staging server. A human must deploy the system into production.

Continuous deployment – a process that takes code checked into a version control system, builds a system, tests the built system, and is automatically placed into production.

Cycle time – the time it takes for your code to move through the deployment pipeline.

Development environment. – an environment designed for developing a module.

Environment– a set of resources such as VMs, a database, network connections, and infrastructure services, that support development, integration, testing, or production. Each environment is isolated from other environments.

False negative – the failure to generate an alert even though there is a problem.

False positive – an alert that occurs even though there is no problem.

Flaky test – a test that is not repeatable. It may fail on one execution and pass on another.

IDE – integrated development environment. An environment intended to support developers.

Module – a collection of coherent code. A development unit.

Repeatability – the ability to repeat a deployment pipeline and get the same results each time.

Roll back—reverting to a prior release (undo for deployments).

Traceability – the ability to determine the source of the artifacts and tools that are included or that affect a service in production.

Software Bill of Materials (SBOM) –a list of components in a piece of software.

Static analysis – an analysis technique that examines source code and looks for problematic constructions.

Tee – a Unix command that sends input to two distinct locations.

Unit tests – tests designed to test a single module

User-acceptance testing (UAT) – a set of tests to determine whether a system is acceptable to users.

Version skew – inconsistencies between different versions of services in production.

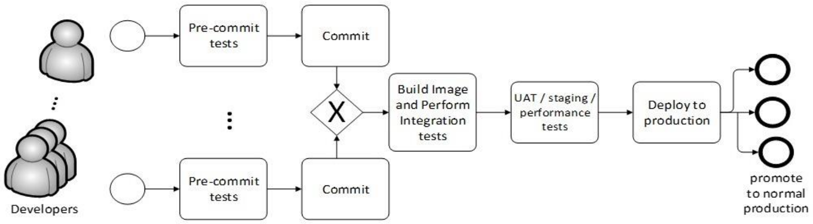
VM sprawl – losing track of all the VMs allocated for your systems.

11.2 Overview of a Deployment Pipeline

The deployment pipeline is the sequence of actions that move the code you write into production. Although some organizations use different stages, we present the pipeline as four stages: Development, integration, staging, and production. Each stage uses a separate environment that enforces isolation between the stages. Each environment has an explicit lifecycle that begins by allocating the necessary resources and ends with a tear-down phase where all the resources associated with that environment are released.

Figure 11.1 shows a deployment pipeline. Here we provide a short overview, and below we discuss each stage in detail.

Figure 11.1: A Deployment Pipeline


Developers
Pre-commit tests
Commit
Pre-commit tests
Commit
X
Build Image and Perform Integration tests
UAT / staging / performance tests
Deploy to production
promote to normal production

The pipeline begins at the development stage. In this stage, you are working on a single chunk of software called a *module.*[37] You develop your module and run *unit tests* on it in a *development environment*. After you complete your work on the module, you check it into a version control system, which triggers activity by the continuous integration server. The continuous integration server is an external service that operates in the *integration or build environment*. It compiles[38] your new or changed code, along with the code of other portions of your service and constructs an executable image for your service. This executable image is then tested for functional correctness. After the executable image passes your service's functional tests, it is promoted to the staging environment. The staging environment tests the quality of the service–performance under load, security vulnerabilities, and compliance. *User-acceptance testing* (UAT) may occur at this

[37] Here, we are using the term *module* in a general sense, as a unit of development work. Some programming languages have a construct called a module, which may or may not map to our use of the term.

[38] If you are working in an interpreted language such as Python or Javascript, there is no compilation step.

stage depending on the business context of your service. Passing the staging tests entitles your service to be deployed into production. Once in production, the service is monitored closely until there is some confidence about its quality. At that point, it is considered a normal part of the system and receives the same amount of attention as the other parts of the system.

You perform a different set of tests in each environment, expanding the testing scope from unit testing of a single module in the development environment, to functional testing of all the components that make up your service in the integration environment, ending with broad quality testing in the staging environment.

Sidebar: Effect of Virtualization on the Different Environments

Before the widespread use of virtualization technology such as we discussed in Chapter 2 Virtualization, the environments that we describe here were physical facilities. In most organizations, the development, integration, and staging environments were operated by different groups. The development environment might consist of a few desktop computers that the development team re-purposed as servers. The integration environment was operated by the test or quality-assurance team, and might consist of some racks, populated with previous-generation equipment from the data center. The staging environment was operated by the operations team and might have hardware like that used in production. A lot of time was spent trying to figure out why a test that passed in one environment failed in another environment. One benefit of virtualized environments is the ability to have *environment parity*, where environments differ only in scale and not in type of hardware or fundamental structure. The provisioning tools we discussed in Chapter 10 Basic DevOps Tools*, support environment parity by allowing every team to easily build a common environment and by allowing that common environment to mimic the production environment as much as possible.*

If the process through the pipeline into deployment is fully automated, i.e., there is no human intervention, then this process is called *continuous deployment*. If human intervention is required to place the service into production–as is required by some regulations or organizational policies–the process is called *continuous delivery*.

The pace of progress through the pipeline is called its *cycle time,* and many organizations will deploy to production multiple or dozens of times a day. Such rapid deployment is not possible if human intervention is required. Such rapid deployment is also not possible if one team must coordinate with other teams before placing their service in production. Later in this chapter, we will see techniques that allow teams to perform continuous deployment without consultation with other teams.

The cycle time is one of the qualities by which the pipeline can be measured. Another quality is *traceability*. When performing forensics on a problem with your service that occurs in production, all the elements that led to the service having a problem should be recoverable. That includes all the code and dependencies that are included in that service. It also includes the test cases that were run on that service and the tools that were used to produce the service. Errors in tools used in the deployment pipeline can cause problems in production. For example, we know of a case where a bug in a compiler caused a runtime error. Typically, traceability information is kept in an *artifact database*. This database will contain code version numbers, dependency version numbers, test version numbers, and tool version numbers.

A third important quality is *repeatability*. That is, if you perform the same action with the same artifacts, you should get the same result. This is not as trivial as it sounds. For example, suppose your build process fetches the latest version of a dependency. The next time you execute the build process, a new version of the dependency may have been released. For another example, suppose one test modifies some values in the database. If the original values are not restored, subsequent tests will not produce the same results.

Flaky tests are those tests that sometimes fail and sometimes pass. In addition to the causes we just mentioned, a flaky test can be caused by parallelism in the system. In one execution, process A may be invoked prior to process B. In another execution, the reverse may happen. If an error results from a specific execution sequence, a second execution of that test may generate a different sequence. Errors that result from flaky tests are difficult to debug. One technique is to examine the logs created by the different executions of the test to determine the sequence of execution.

A final important quality of a pipeline is security. 20-25% of security breaches are caused by insiders. Techniques to reduce the possibility of an insider attack on the pipeline are

- Limiting access to the pipeline tools,
- keeping a record of changes and the originator of the changes,
- Notifying team members that a pipeline tool has been updated or modified.

A deployment pipeline constitutes the last portion of the supply chain for a system. As such, its security is important. We will deal with the security elements of the various environments when we discuss them, but an overall security activity is to limit access to the tools used in the pipeline. 20-25% of security breaches are caused by insiders. Limiting access to the pipeline tools, keeping a record of changes and the originator of the changes, and notifying team members that a pipeline tool has been updated or modified are techniques to reduce the possibility of an insider attack on the pipeline.

The outline for the rest of this chapter follows the path that your code takes as it is checked in, built, staged, and deployed into production. We first discuss the concept of an environment, and then the stages in the deployment pipeline in more detail. We will describe the types of tests that occur in each stage of the pipeline and finish by describing the deployment process.

11.3 Environments

In the overview of the deployment pipeline, you can see that environments are an important concept. The purpose of each environment, except for production, is to provide a place to perform testing that is isolated from other development or testing activities that might be ongoing. This chapter will focus on the development, integration, and staging environments. The purpose of the production environment is to support end users. Although the production environment shares many characteristics with the other environments, there are important differences, including scale, availability, and security. We discuss some of these issues in Chapter 13 Post Production.

Every environment has a set of requirements and a lifecycle. We will first discuss the common requirements and lifecycle steps, and then discuss the specifics of each environment.

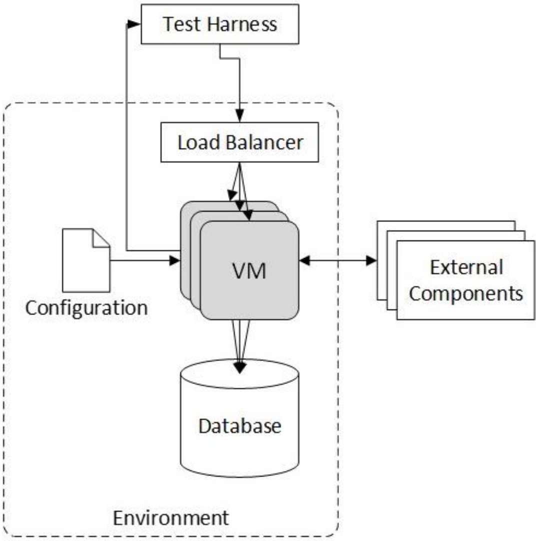
11.3.1 Requirements for an Environment

One requirement for an environment is that it is isolated from other environments. This includes address spaces, inputs, modifications to any database, and other dependencies on the processes being executed in the environment. A second key requirement is that to the extent possible, and consistent with the purpose of the environment, each environment should be realistic–it should reflect the production environment. A third key requirement is that the elements that go into promoting your code to the next environment should all be recorded in the artifact database to provide traceability and repeatability.

Figure 11.2 shows the elements of an environment. Each environment has

- a collection of virtual machines or containers
- infrastructure services (for example, a load balancer)
- a source of input
- a database
- configuration parameters
- external services

Figure 11.2: The Elements of an Environment



We now go into more detail on each of the elements.

- A Collection of Virtual Machines or Containers. These contain the modules, services, or system being executed in this environment. The collection of services in the environment grows as the system moves through the pipeline. In a development environment, a single module is being tested; in an integration environment, the service is being tested; and in a staging environment, the service plus other services is tested.
- Infrastructure Services. Your service depends on infrastructure services. In the development environment, you may need only a load balancer and logging, but as you move through the deployment pipeline, you will need other services such as registration and discovery. These may be provided by a PaaS, a service mesh, or in some other fashion. In any case, including these infrastructure services in the environment will allow you to uncover problems in interactions with these services earlier, and will simplify environment management by creating uniformity across all environments.
- Source of Input. One purpose of each environment is to perform tests. Runtime tests require a source of input. The input can come from a test harness or dynamic workload generator, from live users, or by replaying a

previously captured input from live users. Figure 11.2 includes a test harness. It has a source of input–the test driver–and the output from your service is sent back to the test driver. The test driver compares the actual output to the expected output and reports errors.

- Database. The contents of the database depend on the purpose of the environment, and we will discuss this in the sections on the specific environments. A key requirement across all environments, however, is that the database must be restored after each test. The tests must be repeatable and give the same results every time they are run. If that is not the case, it becomes very difficult to find errors in your code. Since any given test may modify data in the database, restoring the database after each test will ensure that each execution of each test begins from the same state.
- Configuration Parameters. As we said in Chapter 10 Basic DevOps Tools, a configuration parameter is a value that is bound at runtime, typically at initialization. Each environment will have a set of configuration parameters, for example, database connection string and location of external services. These configuration parameters will vary from environment to environment.

  We do not advocate placing credentials in a file such as is represented in Figure 11.2 since that may represent a security vulnerability. We discussed the options for the management of configuration parameters in Chapter 10 Basic DevOps Tools and will discuss the management of secrets in Chapter 12 Design Options.
- External Services Your service may also use external services. These can range from a service that broadcasts the weather to one that performs authorization. The treatment of these external services depends on which environment your service is in and whether the external service is read only or read/write. If your service is not in the production environment and the external service is read/write, it must be stubbed or mocked. Any writing to an external service from anything other than the production environment may affect the behavior of that service. In general, no action from anything other than the production environment should affect the production environment and writing to an external service may have that effect.

11.3.2 Lifecycle of an Environment

An important aspect of an environment is that it has a finite lifetime–it is created, used, and then cleaned up. Virtualization and IaC technologies allow you to automate the environment lifecycle. Automating environment lifecycle activities will make your development and test tasks repeatable by you or by a teammate.

Each environment in the deployment pipeline will have variations on the basic lifecycle, and we will discuss these when we go through the various environments. We briefly discuss these three steps now.

Create Step

Each lifecycle begins with a create step. This create step is triggered by some event and scripted, which allows the create step to be automated. Each environment will have a different triggering event, but the create step always has the following activities.

- *Create VMs or containers loaded with your software*. This creation should use a provisioning tool such as we described in Chapter 10 Basic DevOps Tools. Your script must specify the attributes for each VM, such as the size of the VM, the number of initial instances of the VM, the autoscaling rules, the security settings, and so forth. If you are using a container as your packaging mechanism, the scripts will be run by the container orchestration system.
- *Create a load balancer*. The load balancer is a separate VM or is allocated from a PaaS or service mesh. The create script will instantiate the load balancer and, since it knows the VMs to be balanced from the previous activity, it can register the VMs with the load balancer.
- *Create the test harness.* Your team or organization will probably standardize on one or more testing tools. The information necessary to create the testing tool and link it into the environment should be in the triggering script. Since the test cases should be version controlled, the test harness can find the latest version of the test cases by querying the version control system for the test cases.
- *Initialize the database for the environment.* The mechanism for doing this will vary depending on the environment, but the build script is responsible for creating the schema and loading the test data. The test data may be

loaded from a file or be synthesized algorithmically and inserted by a script, depending on the size of the data set.

- *Create environment-dependent configuration parameters*. These parameters will be items such as the URL of the database and the URLs of the external services being invoked by the system moving through the pipeline.

Every module that you work on is targeted to eventually be deployed into a production environment that contains specific versions of an operating system, libraries, dependent services, etc. Your development, integration, and staging environments for that module should use the same versions as the target production environment. Similarly, all your teammates who are working on modules that will be part of the same service should use environments with the same versions as that production environment. Different services may have different target production environments, but within a service, everyone should be using the same versions of the operating system and other dependent software. Incompatibilities among different versions of libraries or operating systems is a major source of errors when you combine your work with that of your teammates during the Integration stage of the deployment pipeline and will cause errors when you deploy to production.

Usage Step

Usage will vary depending on the purpose of the environment. We will discuss usage when we discuss the various environments.

Cleanup Step

The last step in the lifecycle of any environment is to clean up the resources used by the environment. This step is sometimes called “clean your bowl.” There are three portions of this step.

1. Save any persistent artifacts created by the environment. The development environment creates code that must be saved in a version control system. The integration environment creates an executable that must be saved for further tests or for placing into production.
2. Adjust any necessary data. The discovery system may have been set to point to an executable for various types of tests. It should be set to an

appropriate value. The testing may have created artificial users in persistent data. These artificial users should be removed.

3. Remove any resources created. It is easy to lose track of VMs, resulting in *VM sprawl*. This increases costs and has security implications. Lost VMs will not be patched when new vulnerabilities are discovered, and these orphaned VMs are a target for attackers to enter your system.

Scripting the cleanup step helps to reduce VM sprawl and keep only active VMs running in your cloud account. The scripting is generally not difficult: Some provisioning tools, such as Vagrant, use the same script to build and tear down the environment. Other tools, such as Terraform, allow you to destroy the entire environment with a single command. However, there may be cases where customization is needed. For example, you may need to copy test results from your test harness to an archive before tearing down the test harness.

As with the create step, the trigger for the cleanup step depends on the type of environment; but having a cleanup script makes the cleanup process automatic.

11.3.3 Tradeoffs in Environment Lifecycle Management

Using automated triggers for the build and teardown steps can cost time and money to create and destroy environments. We have seen organizations that triggered the building of a development environment every time code was checked out of the version control system, in preparation for the testing to be done before that code was checked back in. Then, the check in triggered the teardown of the development environment and the creation of an integration environment. A natural question therefore is, “Is the automatic creation and deletion of environments worth the effort?”

Recall that you create distinct environments to maintain isolation and keep activities in one phase of your development lifecycle from impacting the activities of others in that phase, and activities in other phases. The tradeoff then is between the errors prevented by isolating all activities into their own environment and the costs–both human and computing resources–of having multiple environments.

Just as the creation of version control systems was a response to the types of errors associated with file management, so the creation of environments is a reaction to the types of errors associated with interference of one set of activities with another. If everyone was perfect, neither type of system would be necessary. However, such perfection is not often found, and protecting developers from themselves or from teammates has proven to be useful. Furthermore, cloud computing makes it more affordable to create distinct environments for each developer and for each lifecycle phase, tipping the cost-benefit calculation in favor of automation.

11.3.4 Deployment Pipeline and Environment Variations

The deployment pipeline that we describe here is just one variation. Our example covers the common case of an organization using a microservice architecture for a system that the organization develops and operates. This encompasses organizations developing systems they operate for external customers in a software-as-a-service (SaaS) model and organizations developing systems that they operate for internal customers.

There are other development and operation scenarios. While most of the pipelines that we have seen have these same three stages before deployment to production (development, integration, and staging), the testing at each stage may vary. Here are a few examples:

- You test a complete service in the development stage (rather than just your own module), and then test the entire system (that service integrated with other services) in the integration stage.
- The quality testing that we describe in staging is performed in the integration step, and staging serves as a rehearsal for deploying to production, testing the deployment and rollback scripts and instructions.
- Your organization delivers a service such as a database or message queue that your customers use to build their systems. In this case, your integration stage might test “typical” customer use cases, and then your customer’s integration and staging will test your service in the context of the customer’s system.

In all cases, effective deployment pipelines exhibit some common characteristics: a set of well-defined stages that all software passes through, an isolated environment for each stage, and the use of automation wherever feasible.

Now we turn to the specific types of environments for our example pipeline, beginning with the development environment.

11.4 Development Environment

You use the development environment to create and test the module you are currently working on. This module may represent a new service, or it may be the maintenance of an existing service. In either case, you will interact with the version control system and an IDE (integrated Development Environment).

You should have a branch of the version control system where you keep the code you are developing. This branch may be newly created if you are developing a module from scratch, or you may have checked out code from an existing branch. Either one of these activities can be used as the trigger to execute the create step to build a separate development environment for your individual use.

11.4.1 Create

The create step creates and loads the VM or container with the software you need for your module, which includes operating system, libraries, and dependent modules. If you are creating a container, Kubernetes should be included in the test software configuration..

Your IDE should be set up to use this environment as the destination for its activities: When you compile your new or changed code into an executable form, the IDE should place the executable artifacts into the development environment so you can begin testing.

This brings us to the next step in the lifecycle of a development environment, the usage step.

11.4.2 Usage

You use the IDE to create or modify the code for the module being developed. The details of this depend on the IDE but you will need to test the code being developed. You test both the execution of the code you are writing and the quality of the code.

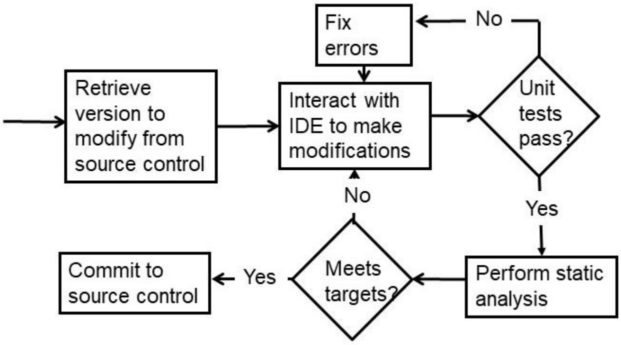
Two types of tests are used in the Development Environment:

- Run time tests. These tests are applied to an executing version of the module under development. Input is provided to the module and the output is compared to an expected output. This type of test requires a test harness.

- Static analysis. This type of testing is applied to the source code. Using techniques similar to those used in compilers, a static analyzer looks for problematic code constructions. The problems flagged by a static analyzer are those listed in the Common Weakness Enumeration (CWE).[39]

Figure 11.3 shows the workflow in the development environment.

Figure 11.3: Workflow in the Development Environment



Execution tests

You should have a collection of tests for your module. These tests include both sunny-day tests –exercising code paths with no exceptions or error conditions–and rainy-day tests exercising error conditions and exceptions.

Your tests will include new tests that you create before you write your code (if you are doing test-driven development) or as you write your code. If you are maintaining an existing module, you will also have regression tests that were created for previous versions of the module. Running regression tests ensures that you haven't broken existing functionality as you make changes.

Tests should be version controlled and saved in the version control repository.

After you have checked your tests into the version control system and the IDE has loaded your module into an executable form, the test step runs the tests against the module using the test harness. The tests are run by a specialized testing system

[39] https://cwe.mitre.org/data/definitions/1387.html

that reports errors to you. Look at Figure 11.2 again to see this portion of the workflow.

Static Analysis

In addition to testing the execution of your module, you can also check code quality. A static analyzer can process your code and detect certain types of errors. Because static analyzers will report errors that are not truly errors (false positives), you should not expect to get a clean bill of health from the static analyzer. This differs from the execution tests where you should expect all the tests to pass. Most static-analyzer tools can be configured to ignore certain errors or to include customized filtering of the analysis results. This tool configuration should be included in your environment configuration, so that all team members are running consistent tests.

This is also the time when a peer review of your code should occur. Ideally, every artifact should be reviewed, and the code you have generated is no exception. However, performing a complete peer review on every artifact is time consuming, both from your perspective and the perspective of the reviewers, so you must decide when and whether your code should be peer reviewed. The decision will depend on the importance of your module and your coding maturity.

Security

Secure coding techniques are outside of the scope of this book. Other books exist for this purpose. In the development environment some tools can be used to test the security of your module

- A static analyzer can look for known security weakness patterns.
- If you package your module as a container, a scanning tool can produce a bill of materials for your container and check it against known vulnerabilities. A bill of materials lists all dependencies included in your container. We discuss creating a Software Bill of Materials (SBOM) in Section 11.5.

11.4.3 Cleanup

You save the VM or container image that you created in the create step during cleanup. Saving the VM or container image from the development environment makes it available for use by future steps in the deployment pipeline, and it can be used to quickly run additional tests or to rerun existing tests. By saving the image, you reduce the time required for these activities.

You also check your module in. If needed, you merge your branch of the version control graph into another branch–typically the one from which it was created. The check-in action will trigger the integration step.

This step also records information in the artifact database, including the version number of the module being checked in and the version numbers of the tests. Also record the configuration parameters, the version of any tools used for testing or static analysis, and the version of the IDE and any plugins used.

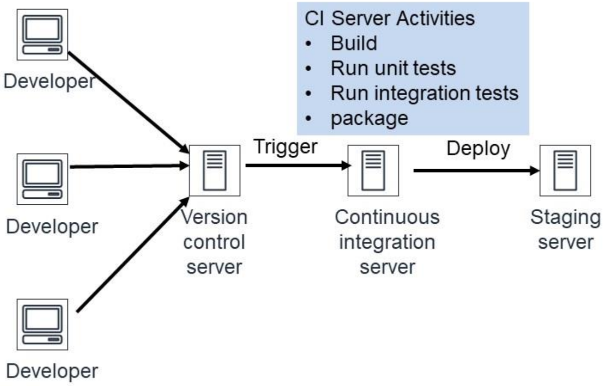
As we said above, all resources used in the development environment should be released when the environment is no longer needed.

Having a defined lifecycle for the development environment allows you to script the activities of the lifecycle stages using scripts created by you or by other members of your team. This eliminates much of the overhead from the development process and frees you to focus more of your energy and attention on creating the module.

11.5 Integration Environment

The integration environment builds an executable version of your complete service and tests its functionality. It is triggered when you check in your module. The activities during the integration stage are to get the latest copy of all the modules included in your service along with all the dependencies, compile all these modules, and build an executable. We described this process in more detail in Chapter 8 DevOps Preliminaries. The executable is then tested for functional correctness and, if it passes, it is promoted to the staging environment. Figure 11.4 shows an overview of the activities of the integration environment.

Figure 11.4: Overview of the Integration Environment Activities



We now discuss these activities in the context of the lifecycle for the integration environment.

11.5.1 Create

The create step is triggered by checking in code from the development environment. It creates virtual machines or containers for your service. The Continuous Integration (CI) server requires two virtual machines. One contains the CI server itself and the second one is the workspace for the CI server. The workspace is the target for the built service. This step also creates the rest of the environment–populating the test database, creating the test harness, creating a VM with a load balancer, setting up the configuration parameters for integration, and linking either to external services or to mocks of the external services.

11.5.2 Usage

This step is executed by a continuous integration (CI) server. It builds the executable for your service. This includes loading, compiling, and linking your source code, as well as the source code for all the other modules in your service. Any errors that prevent linking the compiled modules in your service– incorrect interface names, incorrect signatures for some languages, and other fundamentally syntactic types of errors–will trigger notification to you via email or via a web page from the CI server.

The usage step also includes testing. Your service Is tested for functional correctness: Are the outputs correct for each set of inputs? Qualities such as performance or security will be tested later, in the staging environment.

The build step created and populated the database that you will use for your integration testing. You want this data to be as close as possible to the variety of data that you will see in production. You might even use an extract from the production database if you don't include personally identifiable information (PII) or other restricted data. And, although you want a wide variety of data to test your service, you want to limit the size (or volume) of the test data set.

The volume of data is limited for two reasons. First, it will affect the time taken to run your tests. The data should be realistic enough to test all code paths including edge cases, but you don't want the tests to take too long to run. Testing is a time-consuming part of the integration stage, and the goal is to balance the time spent while still discovering as many errors as possible. The second reason to limit data volume is that the database should be refreshed after each test. Starting the database from a consistent state makes your testing repeatable. You do not want to have a test fail and then not be able to reproduce the error. Resetting the database contents between test runs can become a time-consuming process if the test data set is large. Both reasons argue for limiting the size of the test data set in this stage.

After loading the database, testing begins by running the unit tests for each module that is loaded from the version control system– the module you just checked in as well as other modules in the service. Although unit tests were run on the development environment, they are repeated here. There are two reasons for this repetition. First, the unit tests during the development-environment stage were run on a smaller database than is used in the integration environment. Errors caused by particular data values can be found and isolated more easily if the unit tests are run again than if they occur during integration testing. Second, the set of unit tests may have been extended from when the module was originally tested. Regression tests could have been added or other specialized tests could have been added to the original test suite. Because unit tests are fast, repeating the unit tests does not extend the test time extensively.

After the unit tests are run, your test harness runs the integration tests. These integration tests cover the whole service, as opposed to a single module. Integration tests may come from a quality-control group, from use cases for the service, from regression tests based on issues found in production, or from the development team.

11.5.3 Software Bill of Materials

A Software Bill of Materials (SBOM) is a formal, machine-readable inventory of software components and dependencies, information about those components, and their hierarchical relationships.

A typical SBOM for a service contains the following information:

- Author name
- Supplier name
- Component name
- Version string
- Component hash

An SBOM is a hierarchical structure that includes SBOMs for the elements included in your service. It is constructed during the build step for your service. Libraries are fetched and included in the built service and their SBOMs can be added to the hierarchical structure for your service.

An SBOM is used to perform security scanning both within the deployment pipeline and after the service is placed into production. Vulnerabilities in elements that are included in your built service can be detected by comparing the SBOM for your service with the Common Vulnerability and Exposures (CVE) list.[40] We discuss CVEs further in Chapter 14 Security Deployment.

A good analogy is with an automobile. An automobile is composed of parts from a variety of manufactures. Suppose you have purchased an automobile and some time later a defect is discovered in one of the parts. The automobile manufacturer then notifies owners of vehicles that contain that part.

In software terms, suppose a vulnerability is discovered in a version of a library component. How do you know whether any of your service included the version of the library component that has the vulnerability? Your system may be in production use.

[40] https://www.cve.org/

Scanners exist to examine the SBOM of services in production and determine whether any of them have known vulnerabilities.

11.5.4 Cleanup

After the service has passed its tests, the VM or container with that service is saved for use in future tests. This step records information in the artifact database, including the location of the service-specific package saved, the version numbers and source of all the included modules, and the version numbers of all the tests run. The version of the test tool and the continuous integration server are also recorded since defects in them may cause errors and because any errors not caught by the tests can be traced to these tools. Record the configuration parameters since the behavior of your service will depend on these parameters.

Finally, the resources used by the integration environment are released.

11.6 Staging Environment

The staging environment is where the whole system is tested for its qualities. The focus is usually performance under load, but testing will also cover security and other nonfunctional qualities. The staging environment should be as close to the production environment as practical. For systems with global scope such as Google or Amazon, replicating the production environment is not possible; however, your business scope may allow you to make this environment very close to production.

We begin with the create step of the lifecycle.

11.6.1 Create

The staging environment is triggered by the clean up step in the integration environment, after your service passed its integration tests. The integration environment will create all the VMs or containers for your service that exist in the production environment, and your service in the staging environment should not differ from that created in the integration environment. The configuration parameters created for the staging environment should be the same as the production environment, with a few exceptions:

- Use a separate staging-test database in place of the production database.
- Use credentials appropriate to the staging environment in place of production credentials.
- Mock any external services that your system writes to.

The database for the staging tests should be a copy of the production database, or a large extract, if the full production database is too large. When using production data, you must obscure any sensitive or restricted data. We discuss the obscuring of sensitive data in Chapter 14 Secure Development

As we discussed above, there is a tradeoff between test-data-set volume and the time to run tests and reset the database between tests. The number of tests run on the staging environment may be smaller than in the integration environment (although the test duration may be longer, especially for performance tests), and you can usually use a larger and more realistic test data set in the staging environment and still achieve complete testing within an acceptable duration.

11.6.2 Usage

The staging environment performs two main functions. It tests the executable and, if the tests are passed, it deploys the system to production. The testing involves several types of tests and input sources. We begin with load testing.

Load Testing

The purpose of load testing is to assure that your system delivers acceptable performance under load. Performance testing, especially for large-scale systems, requires significant knowledge and experience. Some software engineers specialize and devote their entire career to this field. Our discussion here barely breaks the surface of this subject.

Load testing comprises three activities: defining the load, applying the load, and measuring the system's throughput and/or latency.

1. In defining the load, you need to consider both the contents of the inputs and the timing of the inputs. The contents should reflect the distribution of values that you will see in production. It may be that your system receives mostly one type of request in production, or there are hotspots in your database (records that are accessed much more frequently than most other records). If so, the load in staging should mimic that. Similarly, if requests in production will come in bursts or the number of simultaneous users increases very slowly, the load should reflect that also.

   There are three possible sources for the load:

   a. *A load-testing tool.* A load-testing tool, such as Artillery, generates synthetic loads for systems. You provide the tool with a description of the input and its distribution. The tool generates synthetic loads in accordance with your description and measures the latency of the responses.

   b. *Playback.* Record input to the production version of the system and use the recording as a source of test input. This input is in the form of HTTP or HTTPS requests. Play these requests back on the new version of the system with the time interval that was used to record the requests. By recording the responses from the production version, you have a means of testing the responses of the new version. By timestamping requests and responses, you have an estimate of the latency and throughput. We discussed the problem with timestamps across different computers in Chapter 6 Measurement but in this case, you are measuring from the requestor’s computer, which is also the recipient computer. Thus, the latencies and throughputs you calculate will be consistent.

   c. *Tee the input to the production version.* “Tee” is a Unix command that takes one input stream and generates two output streams that are identical to the input stream. One branch of the tee will go to the production version and the other to the test version. Comparing the responses of the two versions allows you to test functional correctness and comparing the times of the responses will give you latency and throughput comparisons.

2. There are many approaches to applying the load. We mentioned load-testing tools, which combine load synthesis with load response. Another approach is to build a custom test harness. In any of these cases, you need to ensure that the software applying the load does not introduce any limits on the request rate during test execution – you want to be sure that you are measuring the performance of your system and not your test harness.

3. The measurement process requires careful design. One challenge is accurately handling long-tail latency, discussed in Chapter 4 The Cloud. This leads to running tests that execute so many requests that it is not practical to save the results of every request, and so the measurement framework must build latency histograms and calculate metrics on the fly. You will need a good understanding of statistics to build or choose a

measurement framework. Another measurement challenge is the heterogeneous performance delivered by the underlying hardware in the cloud. Some of the physical computers may be slow because of disk or network-hardware issues, or some may have newer processors that deliver better performance. Your performance testing should cover enough time and enough physical hardware to ensure that you are accurately predicting your system's performance in production. These are just two of many performance-measurement challenges.

Security Testing

The staging environment is also where security testing occurs. There are two types of security testing–runtime testing and static analysis.

Runtime Security Testing

Systems have vulnerabilities. A vulnerability is some portion of the system where an attacker can gain control of the system for a malicious purpose. Vulnerabilities are discovered and reported to both the vendors and to centralized vulnerability-collection centers. These centers subsequently make the information publicly available so that systems administrators and others can apply patches and remove the vulnerabilities. We discuss more about this in Chapter 14 Secure Development.

Security-testing tool vendors closely monitor vulnerability disclosures. OWASP (Open Web System Security Project) is one such tool vendor that focuses on vulnerabilities in web systems. Other security testing tools fall under the heading of pen (penetration)-testing tools. They test not only for web-facing vulnerabilities but also for vulnerabilities in other portions of the stack. These types of tools are used during the staging environment to perform runtime security testing on your system.

Static Analysis

We discussed static analyzers in Section 11.4. In that discussion, a static analyzer was applied to a single module. It is also possible to apply a static analyzer to the entire code base of your service. This use of static analysis happens in the Staging Environment.

Model Checking

Model checking is a technique that involves symbolically testing all possible paths of a system. You specify an error condition and the model checker will determine whether that condition can ever occur in your service. Model checkers suffer from state explosion caused by trying to symbolically execute systems that are too large.

Model checking has been applied on systems up to 10,000 lines of code and, thus far, have primarily been used for operating-system and device-driver functions.

Compliance testing

A third type of testing that occurs during the staging environment is testing for conformance to regulations and license provisions. This is also done using static analyzers. Regulations dictate management of specified data. All software, except for that in the public domain, comes with a license that specifies the terms under which you may use the software. Failure to comply with license terms may result in legal action against your organization, and possibly against your customers who use the noncomplying software.

Regulation compliance

Different types of regulations impose different requirements on how your system handles data. For example, HIPAA (Health Insurance Portability and Accountability Act) imposes a requirement that sensitive data be protected. A static analyzer can examine the code, see if sensitive data has been properly identified and highlight how it is protected. An analyst can then examine the highlighted code and determine whether the HIPAA security requirement is being met.

Other domains have their own regulatory requirements and specialized static analyzers can help determine whether the requirements are being met.

License compliance

Each type of open-source license allows you to do different things. The GNU General Public License (GPL), for example, restricts your commercial use of the software. Some licenses do not allow you to modify the software.

The static analyzer will look at all the source code included in your system and check the system’s license against a set of rules established, probably, by the legal department of your organization.

You should be aware of the existence of different types of licenses and the obligations that you and your organization are assuming by using code protected by these licenses.

User acceptance testing

The user-facing portion of your service must be tested to be sure it is acceptable to users. This may involve having actual users execute the system – a form of canary testing.

It may also involve testing how the user interface is displayed on different devices. Web pages use HTML to specify their display. The HTML is interpreted by each device to generate a display appropriate to the screen size and resolution. Hundreds of different display types are available ranging from smartphones to tablets to laptops to desktops. Ensuring that the displays are readable and usable is done during UAT testing. Common problems are inaccessible links and buttons and poor color contrast.

Deployment to Production

Once your system has passed the tests that occur in the staging environment, the executable– whether packaged as a VM or a container–should be placed on a server for deployment into production. It may be deployed automatically, or a human may be required to authorize the deployment depending on your domain and your organization's policies.

As before, artifact information is recorded in the artifact database. This information includes the URL of the testing database, the version of any tools used in this stage, and the settings of configuration parameters.

11.6.3 Clean up

The final step in the staging environment, as with all the other environments, is to release the resources used.

11.7 Deployment

Placing a new version of a service into production is more complicated than just installing an instance of the service and directing messages to it. We will cover the following concerns in this section.

- Compatibility of the new version with other services.
- Managing the deployment when multiple instances of your service are executing.
- Managing an error with your new version.
- Testing the new version.

We being by discussing feature toggles. They may play a role in version skew (discussed in Section 11.7.2) and rollback (discussed in Section 11.7.5)

Feature toggles

Feature toggles are a mechanism to allow different versions of the same service to be simultaneously active. Feature toggles are "if" statements that make the new code for a service version conditional so that the new code executes only when the feature toggle is on. If the feature toggle is off, then the old code of the microservice is executed.

Because you don't want to stop your application when deploying a new version of a microservice, you need to differentiate between "installing" a new version of a microservice and "activating" that new version. The new version is installed with the feature toggled *off* and when it is time to activate the feature, it is toggled *on*. The value of the feature toggle, whether toggled on or off, is maintained by a distributed coordination system so that all the instances of the microservice are toggled on–or off–at the same time.

Because feature toggles clutter up the code and make it difficult to understand, a feature toggle should be removed from the code as soon as the new feature is stable in production.

We continue by examining two options for deploying a new version of your service. For concreteness, you are replacing the old version of your service (Service A) with the new version of your service (Service A'). The staging environment has placed an image of Service A' on a staging server.

11.7.1 All-or-Nothing Strategies

We assume that all the instances of Service A are to be upgraded to Service A'. This is all or nothing in terms of the upgrade. We discuss partial upgrades in Section 11.7.3

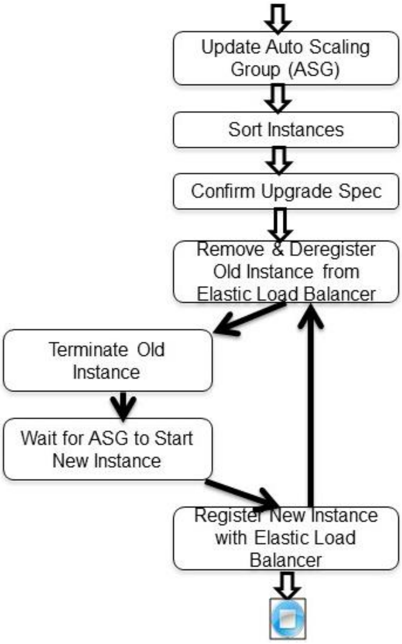
The general situation is that there are N instances of Service A, and you wish to replace them with N instances of Service A', leaving no instances of Service A. You wish to do this with no reduction in quality of service to the clients of Service A, so there must always be N instances of your service running. This is an aspect of continuous deployment.

There are two different approaches to the all-or-nothing deployment strategy:

1. Blue/green. Blue/green is also called red/black, or you could choose your colors. This option allocates N new instances and populates each with Service A’. After the N instances of Service A’ are installed, the discovery service is changed to point to Service A’ and the N instances of Service A are drained and deleted.
2. Rolling upgrade. Rolling upgrade replaces the instances of Service A with instances of Service A’ one at a time. (In practice, you can replace more than one at a time, but only a small fraction is replaced in any single step.) The steps of the Rolling Upgrade are
    - Allocate a new instance.
    - Install Service A’.
    - Begin to direct requests to Service A’.
    - Drain Service A from one instance and then destroy that instance.
    - Repeat above steps until all instances have been replaced.

Figure 11.5 shows a rolling-upgrade process for a system using Amazon’s EC2 cloud services.

Figure 11.5: Rolling Upgrade in AWS


Update Auto Scaling Group (ASG)
Sort Instances
Confirm Upgrade Spec
Remove & Deregister Old Instance from Elastic Load Balancer
Terminate Old Instance
Wait for ASG to Start New Instance
Register New Instance with Elastic Load Balancer

Tradeoffs between Blue/Green and Rolling Upgrade

There are two tradeoffs between the two approaches.

1. *Financial.* The peak resource utilization for a blue/green approach is 2N instances, whereas peak utilization for a rolling upgrade is N+1 instances. Before cloud computing, an organization had to purchase physical computers to perform the upgrade. Most of the time, there was no upgrade in progress, and these additional computers went unused. This made the financial tradeoff clear, and rolling upgrade was the standard approach. Now that computing resources are rented rather than purchased, the financial tradeoff is less compelling; but rolling upgrade is still widely used.

2. *Responding to errors.* Suppose you detect an error in Service A’ when you deploy it. Despite all the testing you did in the development, integration, and staging environments, when your service is deployed to production,

there may still be latent errors. If you are using blue/green deployment, by the time you discover an error in Service A', all the instances of Service A may have been deleted and rolling back to Service A could take some time. In contrast, a rolling upgrade may allow you to discover an error in Service A' while instances of Service A are still available.

Regardless of the strategies used, the deployer must have credentials both to access the image on the staging server and to deploy the image onto the target.

11.7.2 Version Skew

We will use an example to clarify this discussion. Assume that Service A is a shopping-cart service that allows clients to add items to a basket and calculates the total price including discounts. You are updating the service to shift from calculating the discount item-by-item to calculating the discount based on the complete purchase. This change requires changes to Services A, its clients (C), and its dependent services (D).

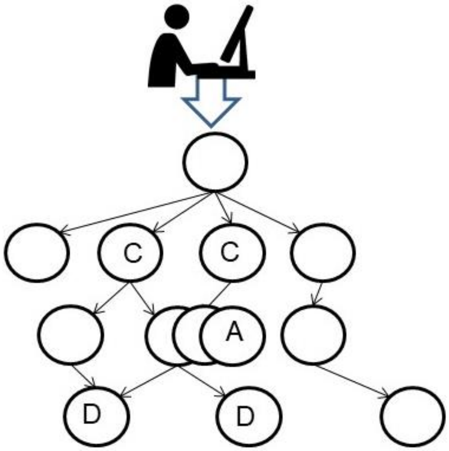
Recall that continuous deployment is the automatic placing of code into production without human intervention. It takes time for your code to go through the deployment pipeline.

Since it takes time for Service A' to go through the deployment pipeline, you have no idea of the state of your system when the new code is placed into production. Two consequences flow from the lack of human intervention in the pipeline.

1. It is not possible for you or your team to coordinate with other teams about the order of submission. Service A' may go into service before or after its clients and dependent services have been updated to reflect the changes that Service A' introduces.
2. It is not possible for you to know how many instances of Service A are executing and their current state.

Figure 11.6 shows multiple instances of your service, Service A, located somewhere in the service graph. Your service has clients (C) and its dependent service (D). Multiple instances of Service A are simultaneously executing.

Figure 11.6: End User Accessing Your System



When you perform the upgrade from Service A to Service A’, you must ensure that consistency of data is maintained. From Figure 11.6 you can see one potential source of inconsistency. Because you can update Service A at any time (continuous deployment), it may be that you update Service A before the team owning Service C makes their updates. Then Service C assumes version A when, in fact, version A’ is executing. Conversely, suppose Service C is updated before you update Service A and assumes Service A’ when, in fact, Service A is still executing.

Version skew is the term given to inconsistent versions of services simultaneously being in production. There are two types of version skew, which we call *temporal inconsistency* and *interface mismatch*.

The next sections discuss these two types of version skew.

Temporal Inconsistency

A request by client C to your service may be served by an instance running Service A. The return to client C will be a response that is based on Service A. Service C then makes a second call to your service using the response data, which may be served by an instance running Service A’. In the shopping cart example, the result is a cart in which some items are discounted twice–once as a single item by the Service A instance, and again when the Service A’ instance discounted the entire cart.

We label this as temporal inconsistency because the inconsistency derives from the timing of the requests with respect to the service version changes.

We discuss how to maintain temporal consistency after we discuss the other type of version skew,

Interface mismatch

The way we presented the example above, the interface in Service A and Service A' are identical. Suppose, however, that Service A' had a different interface. Then if Service C calls Service A' with an interface designed for Service A, an interface mismatch would occur. Interface mismatch can occur whenever the update from Service A to Service A' involves changing the interface of Service A. Avoiding interface mismatch requires that any client of Service A be able to call Service A' and get a correct response, regardless of whether the request is serviced by Service A or Service A'. Service A must handle the cases where Client C assumes that Service A has been upgraded but it has not, and conversely, the cases where Client C does not assume Service A has been upgraded but it has, and the upgrade involved changing an interface. In either of these cases, an error will result.

Avoiding version skew

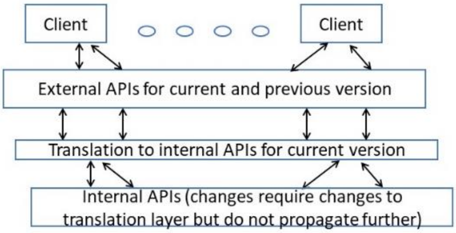
Two techniques exist for avoiding version skew – tagging messages with version numbers and using feature toggles.

1. Version skew can be avoided by tagging messages with the expected version of the recipient. In our example, this means every message from a client is tagged with either A or A' depending on the desired version of the service.

   It then becomes the responsibility of the recipient to respond appropriately. If Service A' received a message tagged with A, it should interpret it as conforming to the interface of Service A. Supporting an outdated interface is called maintaining *backward compatibility*. If Service A receives a message tagged with A', it should respond with an error message indicating that it does not recognize calls to A'. This is forward compatibility. The client then must be able to handle such an error message. The client can try again hoping to be routed to an instance of Service A', fall back to a different method of achieving its goal, or report failure to its invoker.

Managing backward compatibility requires a translation layer that translates from the external interfaces into the current internal interfaces. See Figure11.7

Figure 11.7: Multiple external interfaces translated to current internal interfaces.


Client
Client
External APIs for current and previous version
Translation to internal APIs for current version
Internal APIs (changes require changes to translation layer but do not propagate further)

Maintaining backward compatibility becomes awkward when multiple versions of an interface must be supported. It takes time for developers to continue to support old interfaces as well as being aggravating for the developers. It is a business decision as to how many old versions of the interface should be supported. Factors such as the criticality of the service and whether there are external clients of the service enter into the decision.

When we discuss protocol buffers in Chapter 12 Design Options , we will see an approach that supports interface version tagging.

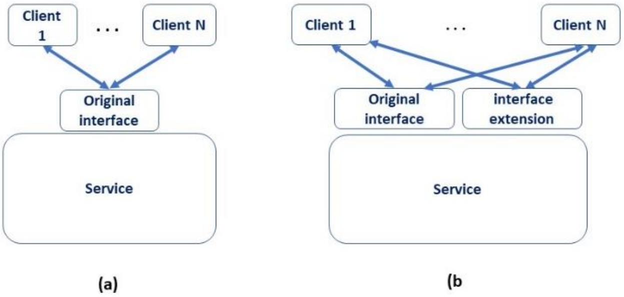
2. Using feature toggles to avoid version skew involves making a distinction between *installing* Service A' and *activating* Service A'. The following steps will avoid version skew among different versions of Service A:

   - Write new code for Service A' under control of a feature toggle.
   - Install N instances of Service A' using either the Blue/Green or the Rolling Upgrade approach. When a new instance is installed, begin sending requests to it without introducing any version skew, as the new code is toggled off.
   - When all instances of Service A are running Service A', activate the new code using the feature toggle. Use a distributed coordination service to ensure that all instances are turned on simultaneously.

Extending an interface

Tagging messages or using feature toggles are two approaches to interface mismatch. Another approach is to extend an interface rather than modifying existing fields. The fields in an interface extension have different names than the fields in the original interface. Then, if the fields in the calling message are named, there Is no ambiguity about the meaning of a call. Figure 11.8a shows an original interface and 11.8b shows an extended interface.

Figure 11.8: Interface Extension. (a) shows original interface and (b) shows an extended interface



Database Schema Evolution

As the features delivered by your service evolve, the structure of the data that your service uses will also have to evolve. There may be simple cases where that data is maintained in a flat file or as a blob in an object store, but you will typically use some type of database. We begin with a couple of general comments.

First, schema evolution is a well-known challenge, and there are entire textbooks devoted to only this topic. We are going to identify some key issues but realize that this is not a new problem. Second, schema evolution is an issue for any data store

– relational database, NoSQL database, or even a flat file. The schema defines what the data means in the context of your service and system. Relational databases and some NoSQL databases enforce *schema on write*, which means that you cannot add data that does not match the schema syntax. Other NoSQL databases and flat files use *schema on read*, which allows you to write anything and have the code in the reader's software create meaning from the data. In both cases, all writers and all readers must agree on the schema, or the syntax and meaning of the data, for the data store to be useful. Finally, in practice, schema evolution approaches are ad hoc: An approach that works for one schema and set of use cases may be completely inappropriate for the same schema with slightly different use cases.

One approach to schema evolution is to treat a schema as you treat an extended interface. The schema can be extended, but existing data remain valid. Any version of a service will access data using field identifiers that it knows are supported, or in the case of a NoSQL database without field identifiers, for example, that can interpret the data correctly.

In some cases, you can use tools that convert data from one schema to another while leaving the database online. This automatic conversion requires you to write specific translation routines to derive the new elements of the schema from the existing form.

11.7.3 Partial Deployments

Sometimes you do not want to change all instances of a service. Partial-deployment approaches are used for purposes such as quality control (canary testing) or marketing tests (A/B testing). The following sections discuss each of these in more detail.

Canary Testing

Before rolling out a new release, it is prudent to test it in the production environment, but with a limited set of users. This is the function that *beta testing* used to serve; it is now done with *canary testing*. Canary testing is named after the practice from the 19th century of bringing canaries into coal mines. The coal-mining process releases gases that are both explosive and poisonous. Because canaries are more sensitive to these gases than humans, coal miners brought canaries into the mines and observed them for signs of reaction to the gases. The canaries acted as early warning devices for the miners.

In modern software use, canary testing means to designate a set of testers who will use the new release. Sometimes, these testers are so-called power users or preview-stream users from outside your organization who are more likely to exercise code paths and edge cases that typical users may use less frequently. Another approach is to use testers from within the organization that is developing the software. For example, Google employees almost never use the release that external users would be using, but instead act as testers for upcoming releases.

In both cases, the testers get access to the canaries through DNS settings or through discovery-service configuration. After testing is complete, either the system is returned to its original version, or the new version is rolled out to all users.

A/B Testing

A/B testing is used by marketers by performing an experiment with real users to determine which of several alternatives yields the best business results. A small but meaningful number of users receives a different treatment from the remainder of the users. The difference can be minor, such as a change to the font size or form layout, or more significant. The different categories of users are compared based on a business metric where the business metric is organizational specific.

For example, eBay tested whether allowing credit card purchases drives up participation in auctions. Another example is a bank offering different promotions to open new accounts. Probably the most famous story is Google testing 41 different shades of blue to decide which shade to use to report search results.[41]

The implementation of A/B testing is the same as the implementation of canary testing. Discovery services are set to send requests to different versions and the different versions are monitored to see which one provides the best response from a business perspective. Feature toggles can also be used to control which version users see.

Note that canary testing incurs minimal additional costs. The system being tested is on a path to production. On the other hand, A/B testing requires the implementation of alternatives. For significant changes, some work will be wasted.

[41] https://sdsclub.com/shades-of-blue-experiment-and-what-it-means-to-a-data-scientist/

11.7.4 Rollback

Not every new version works correctly. Use in production may uncover functional or quality issues that require a version to be replaced. As we will see in Chapter 13 Post Production, your service has service-level objectives (SLOs). Once it goes into production, you should monitor these SLOs to verify that they are being met, and if you find that they are not being met, you may wish to replace the release.

Two options exist for replacing a release: roll back and roll forward.

1. *Roll back* means replacing the current version with an earlier version. This may involve discontinuing the deployment of the new release and redeploying a previous release that is known to meet your quality goals. It could also be accomplished by turning off the feature toggle used to activate the new release.
2. *Roll forward* means fixing the problem and generating a new version. This generally requires you to debug the problem and then be able to test and deploy the new version quickly.

11.8 Summary

The deployment pipeline begins when you check out your code from a version control system and ends when your system has been deployed for users to send it requests. In between, a series of tools and automated tests integrates the newly committed code, tests the integrated service for functionality, and tests the system for performance under load, for security, and for compliance to regulations and licenses..

Each stage in the deployment pipeline consists of an environment established to support isolation of the stage and perform the actions appropriate to that stage. The creation of the environment is triggered by some explicit action and ends with the release of all the resources used by that environment.

Code is developed in the development environment for a single module where it is subject to unit tests. The code is committed to a version control system that triggers the integration environment.

The integration environment builds an executable version of your service. The latest version of all modules in your service are used in the build. Tests in the integration environment are functional and include the unit tests from the various modules as well as integration tests designed for the whole service. When the various tests are passed, the built service is deployed to the staging environment.

The staging environment tests for various qualities of the total system. These include performance testing, security testing, compliance, and, possibly, user testing.

A system that passes all staging environment tests is deployed to production, using either a blue/green model or a rolling upgrade. In some cases, partial deployments are used for quality control or to test market response to a proposed change or offering.

11.9 Exercises

1. Write a script to create a development environment. Develop a Java module within that environment that tests whether an input value is a prime. Use Git to manage the versions of the script.

2. Write a script to create an integration environment for a system that prints out the first N primes. The system should have two instances of LAMP and use HAproxy as a load balancer. Use Junit as your test harness.

3. Use a CI server such as Jenkins to build a .jar package for the prime-number system.

4. Use Artillery to test your environment in Exercise 1 with 10 and 100 simulated users.

5. Deploy a new version of the prime-number generator in AWS using the Opsworks Rolling Deployment mechanism.

6. Roll back the version deployed in Exercise 5.

11.10 Discussion Questions

1. Describe how a buffer overflow enables an attacker to gain control of your system.

2. Write down all the information recorded in the artifact database for the integration step.

3. Determine all the licenses used in the .jar package created in Exercise 3. What restrictions are imposed by these licenses?

Chapter 12 Design Options

Before reading this chapter, you should review Section 5.5 Service Mesh. When you complete this chapter, you will be familiar with

- Two architectural styles for distributed systems and their discovery mechanisms
- Communication possibilities for distributed systems.
- How microservices relate to team size and team interactions
- The quality attribute characteristics of microservices

12.1 Coming to Terms

Enterprise Service Bus (ESB) – a piece of middleware used with Service Oriented Architecture that provides discovery and formatting services.

Marshalling – collecting information and preparing it for sending over the network.

Monolith – an architectural style in places all functionality in one deployable unit.

MTTD – Mean Time to Discovery. How long does it take to determine failure of a service?

MTTR – Mean Time to Repair. How long does it take to repair a failed service?

System of systems – a collection of independent systems integrated together to form one coherent system

Unmarshalling – Decoding a message into its constituent parts.

12.2 Introduction

An architectural style provides a framework for a detailed design of a service. It provides component types and the types of connections between them. For example, you are probably familiar with a client-server style. It consists of a server providing services simultaneously to multiple distributed clients. The most common example is a web server providing information to multiple simultaneous users of a website. This definition tells you that there are two component styles in a client-server. It also tells you that there are multiple clients for a single server. If you delve further into client-server, you will find that communication is initiated by a client using a discovery service to determine the location of the server. Furthermore, the client sends requests to the server and the server responds.

In this chapter, we will discuss two other architectural styles for distributed systems: Service Oriented Architecture (SOA) and microservice architecture. We will discuss the component types and the patterns of communication for these styles. Communication between different components of a distributed system depends on agreement on the form and content of the communication. You saw this when we discussed IP and TCP in Chapter 3 Networking. What that chapter left vague was the contents of the final payload. In this chapter we will discuss two common communication styles—Remote Procedure Call and Representational State Transfer—and how they structure the data that is passed.

You may run into the term “monolithic architectural style”. This style places all its functionality in one deployable unit. An independent database might be deployed as well. Usually, the functionality is connected to a framework such as RAFT (Reusable Automation Framework for Testing). The problem with this style is that all the modules in the system become coupled over time and with the addition of new functionality. This makes the system difficult to modify. Modern practice is to use many decoupled components (microservices) in place of the single component. You will run into a system that is monolithic if it is a legacy system that has not yet been decomposed into microsystems.

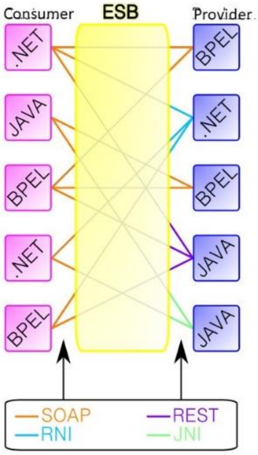
12.3 Service Oriented Architecture

A service-oriented architecture (SOA) consists of a collection of distributed components that provide and/or consume services. In addition, an SOA contains an Enterprise Service Bus (ESB). This is an infrastructure component that implements discovery for consumers. Providers must register with the ESB so that the consumers can communicate with them.

Typically, providers and consumers are standalone entities and are deployed independently. Components have interfaces that describe the services they request from other components and the services they provide. Communication among the services is typically performed by using the ESB which is responsible for translating among representations. Services can be implemented heterogeneously, using whatever languages and technologies are most appropriate.

Figure 12.1 shows an ESB with a collection of providers and consumers. All communication goes through the ESB and the ESB is responsible for discovery among the services. Notice that any consumer service can communicate with any provider service.

Figure 12.1: An ESB[42]



SOAs are used to integrate various standalone systems. Sometimes an SOA system is called a *System of Systems*. Suppose your organization is a bank that has just acquired another bank. You now have two copies of loan management software, fraud detection software, and account management software. Each of these systems has their own user interface and process assumptions. You can use SOA to manage the integration by creating a uniform database for accounts and loans. Attach the databases to the ESB and attach the other systems to the ESB. Now use the ESB to translate from the original bank specific formats to and from the new uniform database format.

For another example, suppose your organization is a health care provider. Your stakeholders include patients, insurance companies, laboratories, doctors, and nurses. Each of these stakeholders has been using a dedicated system for their interactions with your organization. This has resulted in inconsistencies between the information seen by each stakeholder. An SOA can be used to integrate the data managed by these systems into one coherent database and the ESB will

[42] Silver Spoon, CC BY-SA 3.0 <https://creativecommons.org/licenses/by-sa/3.0>, via Wikimedia Commons

translate from the stakeholder specific system into the more general system. This can be done without changing the user interface each stakeholder sees.

12.4 Microservice Architecture

Around 2002 Amazon promulgated the following rules for their developers. Although the term *microservices* came later, the core concepts trace back to these rules[43]:

- All teams will henceforth expose their data and functionality through service interfaces.
- Teams [software] must communicate with each other through these interfaces.
- There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no backdoors whatsoever. The only communication allowed is via service interface calls over the network.
- It doesn't matter what technology they [services] use.
- All service interfaces, without exception, must be designed from the ground up to be externalizable, ready to be exposed to developers outside of your organization.

Notice the linkage between teams and services in these rules. We will return to this in a moment but first we discuss the key elements of these rules.

- *The basic packaging unit is a service*. Services are independently deployable. One implication of independent deployment is that the presence of a service and its current location must be dynamically discovered. We have seen three mechanisms for discovery- DNS, ESB, and service mesh. Microservices have evolved into using service meshes for discovery. We will elaborate on microservices, containers, and service meshes below.
- *Microservices communicate only via network messages*. That is, using the protocols that are covered in the system layer of Figure 3.1. Network communication is an inherent portion of a microservice architecture. It is not a coincidence that microservice architectures date from around 2002.

[43] https://gist.github.com/chitchcock/1281611

This is when the cloud platform became mature enough to support fast network communication. Data-interchange protocols are therefore another important element of microservice architectures, and we discuss them shortly.

- *It doesn't matter what technology they use*. One common cause of integration errors is version incompatibility. Suppose your team is using Version 2.12 of a library and my team is using Version 2.13. There is no guarantee that these two versions of the library are compatible. More fundamentally, suppose your team wishes to use Java and my team wishes to use Scala. This is another implication of services being independently deployable – if both teams provide network-accessible interfaces, we do not need to agree on a development language. Technology independence actually follows from the first two points, but it is an important enabler for modern development practices.

The term "microservice architecture" is widely used, and we shall follow that usage. To be technically correct, a microservice architecture is an architectural style that provides some constraints on the architecture but does not have enough detail to be an actual architecture. An architecture, for example, defines interfaces for services and microservice architecture does not, by itself, define concrete interfaces.

Although not inherent in the definition, a microservice is intended to perform only a single function. You can visualize it as a packaging mechanism for a method. Although this visualization is not strictly accurate, it helps when you think about what to place in a microservice. A system designed around a microservice architecture will consist of many small microservices that coordinate to provide the system. In fact, Amazon's home page directly uses upwards of 140 services, and these call an even larger number of downstream services. Netflix has over 800 microservices.

Sidebar: Microservices as an Evolution of Earlier Ideas

Microservice architectures are related to service-oriented architectures (SOAs). Both use independent services that communicate via messages. However, the goals of the two approaches are different. Microservice architectures are used primarily for systems within a single organization, and the division of responsibility across the microservices within a system (i.e., the *architecture*) is controlled by that organization. On the other hand, SOA systems or systems are composed of either services that are developed and evolved by separate organizations or of services that put a modern wrapper around very mature technology such as a system for a mainframe computer. SOA systems must integrate these existing, unchangeable services, and so SOA includes elements such as ESBs, brokers, and elaborate protocols for integration and interoperability.

As we noted above, Amazon is usually credited for introducing these practices, and many developers rushed to copy them. However, many of the concepts in Amazon's rules arose much earlier in the history of software engineering. In 1972, David Parnas published a paper titled "On the criteria to be used in decomposing systems into modules,"[44] arguing that a module (a work assignment for an individual or team) should encapsulate a set of design decisions behind an interface. He called this *information hiding*, where information could include data structures and algorithms, implementation language, and external dependencies. This approach would allow teams to work independently of each other and would allow modules to be easily replaced as the system evolves.

Given the similarities between Parnas's approach and Amazon's rules, why did it take 40 years for this to become a widespread practice? One answer might be a lack of support from tools and technology. Although Parnas's concepts evolved into object-oriented design, the tools and technology to support that approach did not scale to larger and distributed systems. There was no way to enforce the rule "interaction only through published interfaces," so backdoors were used when it was inconvenient to use the interface. Also, there was no efficient way to package modules, so modules shared dependent libraries that prevented independent module deployment.

The environment in 2002 was more conducive to broad adoption than when Parnas introduced his concept of information hiding. Cloud computing brought VMs and fast networks, which allowed independent deployment and enforced access to services only through published interfaces. Adoption was accelerated as this architecture approach created a use case for containers, which were introduced in the late 1970s and had been a set of experimental features without a compelling system.

[44] https://www.win.tue.nl/~wstomv/edu/2ip30/references/criteria_for_modularization.pdf

Containers have become the standard technology for microservice architectures, and the two have become mutually supporting and mutually dependent.

12.5 Microservices and Teams

Amazon also, famously, has a “two-pizza” rule. Every team can be fed with two pizzas. Of course, this depends on the appetites of the team members, but in practice this rule limits team size to roughly seven people.

In addition, each microservice is *owned* by a single team. This concept of ownership is pervasive in discussions about microservices. It means that the team is responsible for the microservice throughout its full lifecycle, from initial development, test, and integration through bug fixes, enhancements, and ongoing production support. “You build it, you run it” is a capsule description of the responsibilities of a team with respect to a microservice.

A single team may own multiple microservices, but no microservice has multiple owners. A consequence of the team size and the ownership practice is that microservices are small. “Small” is a vague term, but 5K-10K lines of code is a common size for a microservice. Each microservice is owned by a single team and teams can make their own technology choices, which means that coordination between teams can be limited to the responsibilities implemented in each microservice and the service interfaces. Some organizations such as Netflix have a separate team whose responsibility is coordinating requirements among teams. In any case, limited coordination leads teams to treat other teams as they would treat outside entities, and this leads to defensive programming. That is, every service should verify that the parameters it has been sent make both syntactic and logical sense. Every team should be prepared for invocations that do not conform to their current specification, and these invocations should be treated gently. Gently in this case means an error return that says “I do not understand your invocation,” as opposed to returning an uninformative error message or, even worse, just failing silently.

In addition, microservices evolve. As a microservice supports more features, it may grow beyond the owning team’s capacity. In this case, the microservice is split. One portion of the split microservice remains with the original owning team and the other portion is assigned to another, possibly new, team. The portion not with the original team must be understandable by the new team and may or may not conform to the conventions of the new team. Thus, all the standard software engineering problems with maintainability do not disappear with the adoption of microservice architectures.

12.6 Microservice Qualities

As with any other architectural style, a microservice architecture favors certain quality attributes over others. The quality attributes are the *-ilities* of a system. The four that are most relevant to microservice architectures are performance, availability, security, and modifiability, but there are many others. In this section, we analyze the microservice architectural style from the perspective of these quality attributes as well as reusability and scalability. We order the quality attributes alphabetically in avoid accusations of favoritism and begin by considering availability.

12.6.1 Availability

As we discussed in Section 1.4.2, availability is the property that a microservice is running and ready to carry out its task when you need it. It is typically measured in terms of percentage of uptime over the course of some period, with scheduled downtime excluded from this measurement. For example, “four 9s” (99.99% uptime) translates to 52.56 minutes unscheduled downtime over a year. Downtime can be decomposed into MTTD (mean time to discovery) and MTTR (mean time to repair). MTTD is how long it takes to discover that a service failed and MTTR is how long it takes to repair or recover a service after a failure.

Recall that in Section 4.5 Scaling Service Capacity , we said that services are often deployed as multiple instances behind a load balancer, and we should distinguish between the failure of an instance of a microservice and the failure of the microservice itself. If there is only one instance of a microservice, the two cases are identical, but they have different detection and recovery mechanisms. You may discuss availability percentage, MTTD, and MTTR for either case, but the failure of a microservice is usually more consequential than the failure of an instance of that microservice.

In practice, it can be difficult to distinguish an instance that is exhibiting poor performance from an instance that has failed. The main mechanism for detecting failure in a distributed system is timeout. That is, an instance fails to respond to a message or fails to send an “I am alive” message within a specified time. Failure to respond or failure to send a health message may be due to a failure of the underlying hardware, a software crash, or overloading of the instance. Here, we consider the last case to be a failure (from the perspective of the client).

As we said, the fundamental failure-detection mechanism in a distributed system is timeout. To maintain availability, the timeout must be recognized by an entity that can take action to repair the service. Your browser, for example, may detect a message timeout when communicating with a web server but its action is limited to retrying the message, sending the message to another server, or reporting failure to you, the user. Your browser cannot repair the failed web server.

In a microservice architecture, an instance failure is detected by the load balancer. The load balancer may be running in an independent VM, or it may be a portion of the container orchestrator. A healthy instance sends an “I am alive” health message to the load balancer periodically, typically every 90 seconds. If the load balancer does not get a health message in an appropriate time, it puts the instance on an “unhealthy” list and stops sending messages to that instance. If it eventually does get a message from the instance, the load balancer removes it from the unhealthy list. The load balancer should log all these interactions with its instances. We discussed logging in Chapter 6 Measurement.

Recovery of an instance from failure is accomplished by creating another instance of the microservice. If the microservice is stateless, then creating a new instance is all that is required. The auto scaler can be configured to ensure that a minimum number of instances are always active to mitigate the time required to create a new instance. If the microservice is stateful, then the state that the microservice has retained must be recovered. A distributed coordination service such as Zookeeper or etcd can be used for this purpose.

As an aside, use of a distributed coordination system is, in essence, sharing state among microservices. It thus violates Amazon’s original prohibition about shared memory. Pragmatically, sharing state is necessary for purposes such as distributed locks and, in the case we are discussing, recovering state when an instance fails.

If the instance failed while processing a message from a client, there will be no response sent to that message. The client will retry the message and the load balancer will route the message to another instance.

The failed instance may have partially processed the client’s message, or it may be overloaded, and it slowly completes all message processing even though it has not sent a timely “I am alive” message to the load balancer. In this case, when the client resends the message, it will be processed again. When developing a microservice, you should be aware of the possibility that a message may arrive twice. Two approaches are possible to manage multiple responses from the same request.

1 Design the microservice interface to be *idempotent* (processing a message twice will produce the same result as processing it once). In some cases, it is not possible for a microservice to be idempotent. Again, a distributed coordination service such as Zookeeper or etcd can be used to share information about the state of each request among the instances of a microservice.

2 Tag each request with a unique identifier. This identifier is included in the response and this allows the requestor to detect multiple responses to the same request.

Now suppose that the microservice itself has failed. That is, all the instances of the service have failed. This could be the result of a network outage, a coding error in your service, or a hardware failure in your load balancer. In this case, the failure is recognized by the client. There are three actions it might take.

1. It could report the failure to the discovery service so that other services will not attempt to use the failed service.
2. It could set the “circuit breaker” in the circuit breaker pattern so that it does not attempt to call the service again.
3. It could attempt an alternate method to achieve its function. This alternate method might be degraded in some fashion (e.g., use a default value instead of the personalized value that the failed service was supposed to calculate) but will allow the client of the failed service to return a response to its clients.

In any of these cases, it should record the failure in the log for monitoring purposes.

12.6.2 Modifiability

Availability can be addressed by you as the developer of a single microservice. In contrast, modifiability is a property of a system, which allows microservices within the system to be conveniently and easily changed. Although you can contribute to this as the developer of a single microservice, achieving modifiability generally requires coordination among multiple microservice owners or the work of a software architect.

Agile practices call for early and frequent delivery, and any long-lived system must respond to changes in user needs, environment, and technology. Much of the work of software engineers involves making changes to existing systems, and systems with high modifiability make this easier by limiting the number of microservices that are impacted by a change. The traditional measures of modifiability are *coupling* and *cohesion*. You want to have high cohesion within a microservice, and low coupling between any two microservices. This is achieved when each microservice provides a well-defined function and that function has minimal overlap with functions provided by other microservices. Overlap of function will require modification of multiple microservices whenever one of them is changed. Decisions about how to assign functionality to microservices and how one microservice depends on other microservices are broader than the scope of a single developer, and as noted above, require the coordination across development teams or involve the work of an architect to make these system-wide decisions.

As systems grow and as a microservice is used by more than one system, the proliferation of services within a microservice architecture adds another concern. When a change is to be made, identifying all the microservices affected by that change becomes more difficult. Your organization must have a process for allocating changes to microservices. This process will involve maintaining a catalog of microservices and their functions, but it will also involve people with an overall view of the system and knowledge of the interactions among the microservices. These people will allocate functions or modifications to individual microservices but also coordinate interface specifications and assumptions.

12.6.3 Performance

As we discussed in Section 1.4.1, performance refers to the ability of the service to process requests within a client's time constraints. There are two fundamental measures of the performance of a microservice–latency and throughput. Latency measures how long it takes to respond to a request and throughput measures how many requests are processed in a given amount of time. For a single microservice instance, these metrics are directly related by

$$\text{throughput} = 1 / \text{latency}$$

However, most microservices comprise multiple instances running behind a load balancer, and the relationship between latency and throughput becomes more complicated. Generally, reducing instance latency will increase overall throughput.

Also, throughput can be increased in many cases, independent of latency, by adding more instances. However, there may be cases where the overall throughput or latency is limited by the performance of another dependent microservice.

Both latency and throughput can be measured directly by the microservice by using an internal clock. We said earlier that the clock time may vary between two distinct physical devices. Since the container hosting a single microservice instance runs on a single physical device, the measurements based on the device's clock are consistent.

In addition to the time taken by the microservice to process a request and send a response, there are two additional metrics that are important. All requests pass over a network, and the time for the message to go over the network from the sending client to the microservice is of interest. Since this metric uses clocks on two physical devices (the client and the microservice instance), you must be able to bound the clock-synchronization accuracy to assess the quality of this measurement. The second metric applies to queues. A message might spend time in a queue prior to being processed by a microservice. You should measure queue dwelling time as one component to the microservice latency. This is an indication of whether the microservice is overloaded

In addition, the time spent marshalling and unmarshalling messages contributes to both overall latency and throughput of a microservice. Finally, there will be time spent in overhead functions such as discovering the IP address of dependent microservices, sending health checks, and so on. All these times can be directly measured and can be logged for monitoring purposes.

Two questions come up at this point:

1. What should the latency and throughput be?
2. How do I improve the values if they are not being met?

We discuss setting alerting thresholds in Chapter 13 Post Production. For our purposes here, setting an appropriate target latency value depends on how your microservice will be used and its role in systems. For example, users have expectations about the latency of responding to one of their requests. Some requests seem trivial to a user and so should have low latency. Other requests seem complicated, and users are willing to take non-instantaneous responses. If your microservice is in the direct path of responding to a user request, you need to determine whether the user expects instantaneous response. If that is the case, research on human perception finds that users perceive a visible response with a latency of less than approximately 200 milliseconds as instantaneous. To meet that target, you need to understand how many other microservices are in the response path, and you need to allocate a latency budget for your service such that the sum of the budgets in the user response path satisfy the overall target. You can also use the historical latency measurements of your microservice, or similar microservices, to determine your latency budget.

If you are not meeting your time budget, the first step in improving the latency of your microservice is to determine where time is being spent. Taking the effort to optimize a portion of the microservice that is contributing only 10% of the time being taken will not produce a large improvement in latency. Focus on the places where substantial portions of your budget are being spent. Remember that processors and disks can completely or partially fail, impacting software performance. Run your microservice on a different physical device to ensure a problem is in your code, not in the hardware.

Microservices depend crucially on network traffic since a network message will take longer than a memory reference. Techniques for reducing the latency of messages include

- Caching. If the results of a message are stored in a local cache then the message does not have to be sent. Values such as the location of a dependent service can be cached since they will not change frequently.
- Sending messages in binary rather than in text form. Using textual representations of some data items requires more bits to be sent than using a binary representation.
- Collocating services that have high rates of communication between them. We discussed pods in Chapter 5 Container Orchestration.

12.6.4 Reusability

Code reuse has long been a Holy Grail for software engineering. But like many other absolutes, the desire to reuse code must be tempered. The virtues of reuse are that the code does not need to be redeveloped, thus saving time during development, and there is a single point of correctness, thus saving time during repair.

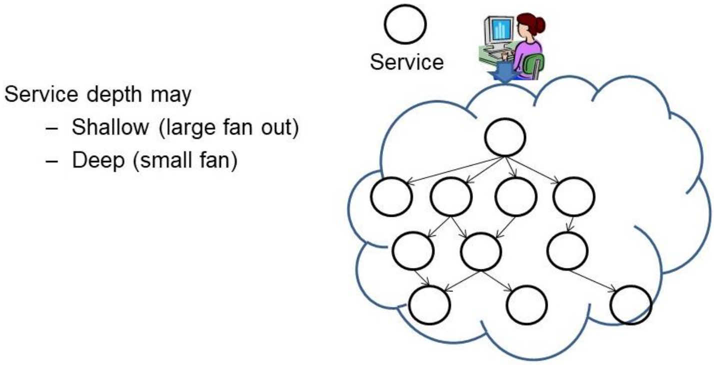
We distinguish between coarse-grained reuse that is encompassed in the architecture and fine-grained reuse of sections of code. Each microservice is owned by a single team, so for that team to reuse code from a different team (fine-grained reuse), they must discover the code, verify that it is suitable for their anticipated use and, if not, adapt it. All of this takes time. Depending on the size of the code to be reused, it may be faster and more expedient to develop the code independently.

Coarse-grained reuse involves reusing an entire microservice. One impediment to reuse is dependency compatibility among the constituent elements of the system. For example, if my service depends on one version of a data-access library and your service depends on an earlier version of that library, creation of a sytem that wants to reuse both of our services will require care in packaging and deployment to satisfy both of our dependencies. In a microservice- architecture using containers to package and deploy individual microservices, your dependencies are isolated and satisfied within each of your containers, which minimizes dependency management as a reuse concern.

Another concern with coarse-grained reuse is the amount of functionality allocated to each microservice. When designing a system using a microservice architecture, one key decision is whether to have large or small fan out. Fan out measures the number of dependent child services that a microservice makes direct requests to, as shown in Figure 12.2. In a design with large fan out, each microservice has many children and a request chain is usually short. Because this reduces the amount of message passing, this approach favors performance (lower latency). On the other hand, in a design with small fan out, each microservice has a small number of children and a request chain may be long, with higher latency. This design allows for allocating functions to the microservices in the chain in a way that increases the reusability of each service. Thus, the tradeoff is between performance and reusability.

The single-point-of-correctness argument for reuse makes sense when the computations are complicated and mission critical, such as how much money is owed on a particular account. For mission-critical computations, there should be a comprehensive set of test cases. If a team chooses to replicate mission-critical computations, their version is subject to the same test suite as if they had reused the mission-critical computation from a different team. The responsibility for correctness of mission-critical computations lies in the test cases, not in any individual implementation.

Figure 12.2: Service Fan Out



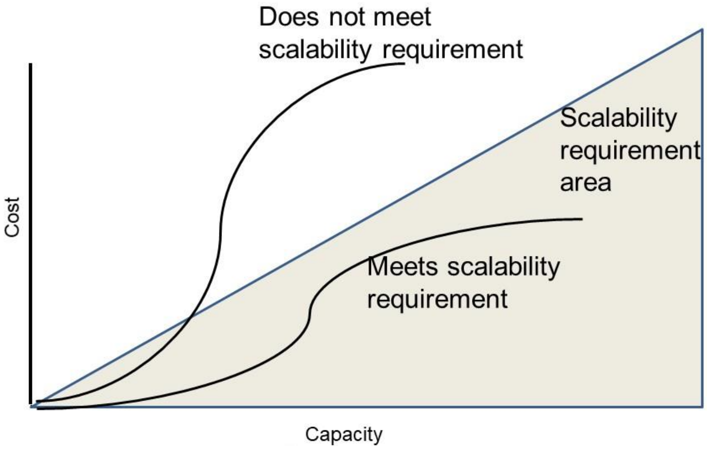
Service depth may
- Shallow (large fan out)
- Deep (small fan)

12.6.5 Scalability

Scalability is the property that a microservice can add resources to serve more requests. It is usually measured as the cost of the additional resources (processor, memory, storage, etc.) needed to achieve increased capacity. A service is “scalable” if the cost is a linear or sublinear function of capacity. An example of linear scaling would be a service for which doubling your request processing capacity requires doubling the number of VMs or containers you have allocated for a service, which should at most have doubled the cost. The shaded area in Figure 12.3 represents a region where the growth in capacity meets the cost requirement for scalability. If as you scale your service, the cost grows and stays in the shaded area, your service meets its scalability requirement. Sample curves from two different systems are shown. The upper curve goes out of the shaded area and is from a design that does not meet its scalability requirement. The lower curve stays within the shaded area and is from a design that does meet its scalability requirement. Many systems have a scalability limit: at some capacity point. For example, if your service relies on a dependent service and that dependent service has a fixed capacity, your service cannot scale beyond the capacity of the dependent microservice.

In a microservice architecture, scalability is achieved by *scaling out*. Additional resources come from adding VMs or containers, not from replacing VMs with larger and more powerful ones as would be done if we were *scaling up*. As we discussed previously, adding more instances for a service is easy if the service is stateless and more difficult if the service is stateful.

Figure 12.3: A Service Meets its Scalability Requirement if its Cost Stays in the Shaded Area as Capacity is Added



12.6.6 Security

As we discussed in Chapter 7 Infrastructure Security, security is best remembered by the acronym CIA–confidentiality, integrity, and availability. Confidentiality means that information can be seen only by authorized users. Integrity means that information can be modified only by authorized users, and availability means that the service is available to authorized users.

Security is a complicated subject with many subtleties. We discuss secure development practices in more detail in Chapter 14 Secure Development. For now, we enumerate some practices that you, as a microservice developer, should use.

- Use HTTPS instead of HTTP. HTTPS encrypts traffic sent over the internet to and from web servers using Transport Layer Security (TLS). Since that traffic is open to eavesdropping, it should be encrypted to maintain confidentiality and integrity. Recall that a orchestrtor can generate certificates. Consequently, TLS can be used without reference to external Certificate Authorities.

- Use authorization tokens such as Kerberos or SAML. These tokens as granted by an authorization service and verify that the client has appropriate privileges to access your microservice.
- Apply patches promptly. Most complicated software has security vulnerabilities, and these are constantly being discovered and patches released by the vendors or open-source projects. Malicious actors seek to exploit newly uncovered vulnerabilities, so you should apply these patches promptly to protect your services. In Chapter 14 Secure Development, we discuss the CVE (Common Vulnerability Enumeration) knowledge base that lists known vulnerabilities in a variety of software packages.
- Delete unused resources promptly. It is easy to lose track of virtual machines in the cloud. Aside from incurring operating costs for these resources, when you lose track of a VM it does not get patched and its vulnerabilities remain. Gaining access to a forgotten and unpatched VM can provide a malicious actor with sensitive information or credentials that can be used to break into active VMs.
- Do not write security-sensitive code such as password managers yourself. There are many subtleties associated with security-sensitive code and there are certified packages that provide these services. We discuss Vault, a credential management system, in Section 14.7.3 Managing Credentials.
- Do not embed credentials into code or scripts. We discuss credential management in Section 14.7.3 Managing Credentials, but embedding credentials into scripts makes it difficult to modify the credentials and provides access to anyone who gains access to the script. Placing a script with credentials into Github, for example, happens frequently enough so that there are monitors that check submissions and warn when it appears that a check-in contains credentials.

12.7 Microservices in Context

We now can tie together some of the concepts we introduced earlier: containers, service mesh, and context dependent discovery.

12.7.1 Containers and Microservices

Although containers and microservices originated independently, they are a natural fit. Microservices communicate only with messages, and containers are accessible only through network interfaces that are intended for message-based communication. Furthermore, the container orchestration mechanisms that are evolving are spurring the adoption of microservices.

In Chapter 5 Container Orchestration, we described a pod as a collection of related containers that are deployed and scaled together. The services that make up a PaaS or service mesh are ideal candidates to be collected in a pod with the microservice you are developing. Placing the platform service resources near your microservice will reduce the latency of messages and resource usage. Placing the platform services in a pod means that the services in the PaaS or service mesh will have multiple instances, one for each pod. Unless memory resources are a concern in your environment, reducing latency tends to be a better decision.

Containers provide more limited resources than VMs. Microservices are small and single purpose, and typically require fewer resources than multifunction processes. Again, there is a natural fit between these technologies.

12.7.2 Service mesh context

Recall from Chapter 5 Container Orchestration that service meshes provide discovery services where the services that can be discovered are limited to those that satisfy some set of criteria.

- Data center locality. The service should be local to the requesting microservice. Local in this context means on the same host or in the same rack.
- Canary or A/B testing. A canary or A/B test designates a collection of microservices as belonging to the version being tested. When we discussed service mesh, we pointed out that a service mesh discovery service is loaded by the orchestrator. If the orchestrator is aware of which instances are participating in the tests, it can populate the relevant discovery services appropriately.
- Geographic locality. Services that interact with users can be specialized to language, to region, or other user visible attributes. Again, the orchestrator can populate the discovery service with specialized microservices.

Other services that might be included in a service mesh are:

- *gRPC and protocol buffers.* These are discussed below. Placing them in the service mesh makes these features easily usable from within a microservice.
- *Configuration.* Configuration parameters control the tailorable portions of a microservice. These parameters include settings for the cloud such as security settings and settings for accessing external services. They also include settings for internationalization, such as language and color usage. Having a single service in the service mesh that all microservices use to get their configuration parameters enforces uniformity across different microservices.
- *Distributed coordination service.* In Section 4.7.2 Distributed Coordination, we discussed the distributed coordination problem. A service mesh should include an infrastructure service to solve that problem. Any time you need to synchronize with another microservice or when two instances of your microservice need to share state, you should use either a distributed coordination service for small amounts of state or use persistent storage for larger amounts of state. Kubernetes uses etcd as its coordination service.
- *Logging.* A common logging service provides uniformity in the logs, easing troubleshooting and incident response. A log message should include identifying information such as source id, task id, timestamp, and log code. The logs from each service instance are sent over the network to a single log repository. Placing this service in the service mesh helps ensure that all the expected information is provided, and the logs are generated in a common format to create a single view of your distributed system.
- *Tracing.* Different circumstances call for different levels of insight into the action of the microservice. The tracing service allows for these different levels to be turned on or off at runtime.
- *Metrics.* The key metrics that determine whether the microservice is performing as it should be collected and used. This service-mesh service will receive the metric values and interact with the dashboard service that is responsible for displaying the metric.
- *Dashboards.* The dashboard creates concise displays of the metrics collected about the behavior of each microservice with respect to its SLOs. A common dashboard service across all microservices ensures that

the information is displayed in the same fashion regardless of its source and helps the developer or other people who monitor the microservices interpret the output. For example, the metric values can be coded as red, green, or orange to reflect acceptable, unacceptable, or borderline with respect to a predetermined value. The dashboard service can automatically provide more detail as a metric goes from green to orange to red.

- *Alerts.* An alert triggers a page.[45] Setting the values at which alerts trigger and managing alert routing allows the microservice to monitor itself and inform relevant personnel when an event needs immediate attention.

We discussed monitoring in more detail in Chapter 6 Measurement. From the point of view of a microservice, the connections to these monitoring functions are available in a PaaS or service mesh.

12.7.3 Protecting Against Failure

In Section4.4 Failure in the Cloud, we discussed two types of failures. First, the host for your instance may fail. We have discussed dealing with this type of failure above in our discussion on availability. Secondly, and the subject of this section, is the long-tail phenomenon. In Chapter 4 The Cloud we discussed how response latency in the cloud has a long-tail distribution. This latency distribution occurs for both high fan-out service and low fan-out services (see Figure 12.2 above). In the high fan-out case, since your service cannot respond until all your dependent requests complete, congestion or failure by any one of your direct dependencies will increase your service's latency. In the low fan-out case, the dependency chain is long, and you are impacted by any failure or congestion in the request chain. Knowing that a significant fraction of the requests your service makes will experience long-tail latency, we now discuss how you, as a microservice developer, can protect against it in some cases.

- H*edged requests*. You make more requests than you need and then cancel the requests (or ignore responses) after you receive sufficient responses. For example, suppose you wish to launch 10 instances of a microservice. You issue 12 requests and after 10 have completed, terminate the requests that have not responded yet.

[45] See https://en.wikipedia.org/wiki/Pager. Today, most alerts are sent as text messages or automated phone calls, with few organizations still using pager devices for alerts. However, the term lives on; for example, referring to being on-call for incident response as "pager duty."

- A*lternative request*. In the above scenario, you would issue 10 requests. When eight requests have completed you would issue two more, and after you receive a total of 10 responses, cancel the two requests that are still remaining.

Since both approaches issue more requests than is necessary, using these techniques increases overall system workload. In some cases, this increased workload may cause congestion that exacerbates the long-tail latency effect. The decision to apply these techniques, and the tuning of the number of excess requests versus the impact of increased workload, must be made carefully.

Because the requests that are canceled may actually complete, the effect of processing a request more than once must be considered. If the request does not change system state–for example the request only reads data–then multiple executions are harmless. If the request can be undone–for example, if you start an extra instance before the request is canceled and can stop that instance without impacting the system–then again, multiple executions are harmless. However, if the request changes system state or writes data to durable storage, the operation must be idempotent.

12.8 Communication styles

In a distributed system, services communicate via messages. Successful communication first requires agreement between the sender and the recipient on meaning of the action to be performed by the message receiver, for example *read* or *write a data store*. It then requires agreement on how to interpret the variable data contained in the message, for example the values to be written to the data store.

Within your service, you organize your code into functions and classes, and make method calls to transfer execution control among these chunks of code. Because all this happens within a single process space, it is straightforward. In a distributed system calling a function on another service is more complicated, as the required information must be packaged in a message sent over the network and sent over an interface.

In modern software systems, three communication styles have become prevalent for interfaces. They differ in the explicitness of the contract between the client service and the providing service. Remote Procedure Call (RPC) requires an explicit contract between both sides of the call. Querying an API requires the providing service to have an explicit statement of its interface and the client can subsequently query for specific data items. Representational State Transfer (REST) has no explicit statement of the interface. It is implicit between the client and the providing service. We discuss these styles in the order in which they were developed – RPC, REST, Querying an API since each was a reaction to the problems of its predecessors.

12.8.1 Remote Procedure Call

Remote Procedure Call (RPC) was first standardized by the Internet Engineering Task Force (IETF) in 1995 as RFC-1831, and later updated to RPC 2.0 by RFC-5531 in 2009. An RPC message contains four elements. The first three elements are an integer that identifies the program (service) on the remote system, an integer that identifies the version of the remote program, and an integer that identifies the remote procedure (i.e., function or method). The fourth element is an untyped data block that contains the request arguments or the response results.

In the code you write, you use names to identify classes and methods. RPC uses numbers. The program-identification number is assigned by the Internet Assigned Numbers Authority (IANA), which we introduced in Section 3.3 IP Addresses. Using numbers instead of names was driven by efficiency (in 1995, networks were slow and every extra byte that was transmitted slowed down a system) and centralizing and controlling the assignment of the numbers ensured that services developed by different organizations would not collide when deployed in the same system. You control the assignment of the version and procedure numbers within your service. In practice, these three numbers are wrapped by an access library that allows you to refer to the service and procedure by name.

The basic steps involved, for most RPC implementations, are as follows.

- Write a package definition file to define the software interface. A package definition file looks like a standard procedure declaration with parameters of specified types. It defines the structure of the untyped block in an RPC message. Both the client and the providing service use the same package definition file.
- Run the RPC compiler to produce the stub code. Compilers are specific to the language of two actors – the client and the providing service. That is, the client is written in one language and uses an RPC compiler for that language and the providing service uses a compiler specific to the

language in which it is written. These two languages are not constrained to be the same.

- Link together the client modules (program, stub, RPC run time system) to make the client module.
- Link the modules in the providing service (a standard main program, the server stub, the server routines themselves, and the RPC run time system).

Then your code (the client) can call a providing service on another node using the parameters specified in the package definition file.

A remote procedure call creates a strong contract between the client and the providing service. Both sides must agree on exactly which procedures (methods) will be implemented by the providing service, and on how those procedures will be numbered.

The RPC standard does not specify how the message is sent from the client to the providing service. One option for message transport is TCP (or TLS if we want to improve security), as discussed in Chapter 7 Infrastructure Security. Another common approach is a message-queue infrastructure service. Finally, some RPC implementations use an HTTP POST request to send the request and response. The RPC standard does include methods for authentication, and optimizations such as packing multiple procedure call requests or responses into a single message.

Generally, RPC requests can be stateful or stateless. A stateful request depends on previous requests made by the client to the service. This is just like the typical programming paradigm you use to develop your service; for example, you must first open a log file before you can write to it.

12.8.2 Representational State Transfer

RPC requires close coupling of a client to a providing service, which is possible within the boundaries of a single organization. As the Internet and the World Wide Web grew in the late 1990s, this level of coupling became impossible to achieve--it was not practical to have a custom client for every service on the web. Roy Fielding conceived an approach that matured in parallel with his work on the HTTP/1.1 protocol. His REST architecture style allows very loose coupling between clients and services.

The key elements of REST are

- The requests are stateless. That is, there is no assumption in the protocol that any information is retained from one request to the next. Every request must contain all the information necessary to act on that operation, which leads to requests that contain a lot of information. Alternatively, the client and providing service agree where the necessary state will be maintained. Stateless calls support scalability and availability, as we have discussed previously.
- The information exchanged is textual–services and methods are accessed by name. The web from its inception was designed to be heterogeneous, not only across different computer systems but across different binary representations of information. REST emerged later than RPC, and networks had improved to the point that squeezing out performance by using integer IDs was not necessary.
- REST restricts methods to PUT, GET, POST, and DELETE. These map into the data-management concept of CRUD [Create (and initialize), Read, Update, and Delete]. A REST request identifies the resource that the operation should be applied to, and like RPC, a request or response contains a data element with arguments or results. However, unlike RPC where the client and server must agree ahead of time on the internal structure of that element, REST requires that the element be self-describing by labeling it with an internet media data type (or MIME type) so that any client knows how to interpret the data.

Strictly speaking, the REST architecture style does not specify the mechanism for sending messages from the client to the service. However, as noted above, the HTTP/1.1 protocol evolved as the first implementation of REST. Practitioners have found no need to create other implementations, and so today REST and HTTP are essentially synonymous.

While RPC favors high performance and a programming style that allows distributed services to be wrapped by an access library and called just like local services, REST promotes interoperability and enables rapid and independent evolution of clients and servers. Both are used extensively to build microservice-based systems: RPC is applied to parts of the system where interactions between services are well understood or where performance is a priority, and REST is used in end-user-facing services and areas that are evolving faster.

12.8.3 Querying an API

The API query style is exemplified by GraphQL. GraphQL was developed by Facebook in 3012. It became an open-source project in 2015. Currently, GraphQL compilers exist in multiple languages. GraphQL was developed to support mobile applications with limited bandwidth.

Within GraphQL, the providing service has an explicit definition of its interface. The elements in the interface are strongly typed (see Section 8.5 for a discussion of strong typing versus weak typing). The providing service has a run time engine that takes as input a query in the GraphQL query language and returns a JSON response with the data requested in the query.

The client sends a query in the GraphQL query language. The query is embedded in a procedure-like call to an interface exposed by the GraphQL providing service. The process on the client side is much like the client-side RPC process. There is an explicit schema that defines what values the client wishes to see, a schema compiler specific to the language in which the client is written, and a runtime library that performs the actual transmission of messages.

The main benefit of GraphQL is that the structure of the requested data is defined by the client. In REST, the structure is defined by the providing service. This means only the data the client requests is sent. The providing service does not send large amounts of data that the client needs to parse.

The main drawback of GraphQL is that allowable queries must be pre-defined. With REST, the client can retrieve a data set and parse it as it chooses. Thus, *ad hoc* queries over a data set can be better supported with REST than with GraphQL.

12.9 Structuring Request and Response Data

Requests and responses in any communication style contain data elements that must be interpreted by both the client and the providing service. This data must be packaged into a message sent over the network. Since clients and servers may not share the same programming language or operating system, we need mechanisms to format and structure the data.

The process of converting data from the representations used in your programming language (e.g., objects, dictionaries, arrays, etc.) into a format that can be sent as part of a request to a remote microservice is called *marshaling*, and the process of taking data from a request or response and converting it back to your programming-language representation is called *unmarshaling*. As we will see, the protocol you use to represent your data in the request will influence the performance and resources needed for marshaling and unmarshaling.

Below, we discuss the three most common protocols for structuring data.

12.9.1 Extensible Markup Language (XML)

XML was standardized by the World Wide Web Consortium (W3C) in 1998. XML adds annotations to a textual document. The annotations, called *tags*, specify how to interpret the information in the document by breaking the information into chunks or fields and identifying the data type of each field. XML grew out of an earlier markup language called Standard Generalized Markup Language (SGML). SGML was becoming large and unwieldy, as it was being applied to many types of problems, including markup for typesetting and visual formatting, tagging for information retrieval, and interoperable data interchange. XML emerged as a simplification of SGML to encode documents on the World Wide Web to be readable by both humans and machines.[46]

XML is a meta-language: Out of the box, it does nothing except allow you to define a customized language to describe your data. Your customized language is defined by an *XML schema*, which is itself an XML document that specifies the tags you will use, the data type that should be used to interpret fields enclosed by each tag and the overall structure of your document. XML schema provide capabilities to specify a very rich information structure. For example, consider how to describe a postal address in the United States:

> *The address should contain a building number and street name, with an optional apartment number; OR, it can contain just a post office box number; BUT in all cases it should contain a city, state, and ZIP code; AND the ZIP code has exactly 5 or 9 digits.*

XML schema allow you to express these types of structures declaratively by enumerating all the variations allowable for each field. In an XML document, each field is delimited by a start tag and end tag, and fields can be nested. The webpage https://schema.org/PostalAddress shows an XML schema for a postal address, and

[46] SGML also evolved into HTML, which used tags to encode how information should be displayed by web browsers.

examples of XML documents representing addresses. The example highlights a common criticism of XML: Documents are verbose (too verbose to include here), and the amount of text required for the tags can be more than the data that you are annotating.

XML documents are used to structure data for many purposes. Here, we are focused on requests and response data in a distributed system, but other uses include representing images, business documents, and static configuration files (for example, MacOS property lists).

One strength of XML is that a document can be checked to validate that it conforms to a schema. This prevents faults caused by malformed documents and eliminates the need for some error checking by the code that reads and processes the document. The tradeoff is that parsing the document and validating it are relatively expensive in terms of processing and memory. A document must be read completely before it can be validated and may require multiple read passes to unmarshal. This, coupled with XML's verbosity, can result in unacceptable service performance.

12.9.2 JavaScript Object Notation (JSON)

JSON structures data as name/value pairs and array data types. The notation grew out of the JavaScript language and was first standardized in 2013, however JSON is independent of any programming language. Like XML, JSON is a textual representation, but unlike XML, JSON has no schema capability to define a valid document structure. Using a name/value representation instead of start and end tags, JSON documents can be parsed as they are read, and the reader is responsible for much of the error handling provided by XML schema.

JSON data types are derived from JavaScript data types and resemble those of any modern programming language. This makes marshaling and unmarshaling JSON much more efficient than XML. The notation's original use case was to send JavaScript objects between a browser and web server, for example, to allow stateful interactions by allowing the browser and web server to easily transfer session state.

The web page https://schema.org/PostalAddress also shows the postal address example expressed in JSON.

12.9.3 Protocol Buffers

A specific form of the syntax of a package definition file is Protocol Buffers. This originated at Google and is typically combined with a binary form of RPC (gRPC). It was used internally for several years before being released as open-source in 2008.

Protocol Buffers use data types that are close to programming-language data types, making marshaling and unmarshaling efficient. Protocol-buffer messages have a schema that defines a valid structure, and the schema can specify required and optional elements and nested elements.

The Protocol Buffers open-source project provides code generators to allow easy use of Protocol Buffers with many programming languages. You specify your message schema in a *.proto* file, which is then compiled by a language-specific protocol buffer compiler. The procedure generated by the compiler will be used by the sender to generate the code to marshal and by the recipient to unmarshal data. Furthermore, the compiler can also generate logging calls so that all data transferred through Protocol Buffers can be automatically logged. In addition, the compiler can generate code to tag the message with the version number of the .proto file as we discussed in Section 11.7.2 on managing version skew.

The sender and the recipient may be written in different languages. They each will use protocol buffer compilers specific to their language. Since protocol buffer compilers exist for many different languages, the sender and the recipient can each be written in a language of your choice. One of the advantages of microservices is the independence of technology choices such as language across teams. The use of Protocol Buffers does not change this independence.

The web page https://github.com/mgravell/protobuf-net/blob/master/src/protogen.site/wwwroot/protoc/google/type/postal_address.proto shows the postal address example expressed as a Protocol Buffer.

12.10 Summary

The monolithic architectural style exists in legacy systems.

Service Oriented Architecture is used in enterprises as a means for integrating previously stand-alone systems. An Enterprise System Bus is used to manage communication among these systems.

A microservice architecture consists of a collection of independently deployable services. Each service is owned by a single team.

As with other architectural styles, the microservice architectural style makes tradeoffs that favor some quality attributes over others. Favored attributes are modifiability, availability, compatibility of versions, and scalability. These are at the cost of performance and, possibly, reusability.

Your microservice relies on a variety of additional services that are provided to you as a platform as a service within the cloud. These additional services include discovery and registration, distributed coordination service, the gRPC library, and logging and monitoring services.

Packaging your microservice as a container allows your microservice to be managed by one of the container management systems that we discussed in Chapter 5 Container Orchestration. It also provides a much lighter weight footprint than packaging your microservice in a virtual machine. Containerization is fast becoming the packaging mechanism of choice.

You also need to be aware of the possibility of a long tail for your cloud requests and hedge against that possibility through issuing extra requests that are cancelled if they are not needed.

The communication between a client and a providing service can have various levels of explicit specification of the interface. RPC explicitly specifies the interface and is a contract between the client and the providing service. REST implicitly specifies the interface and the client uses POST, GET, PUT, and DELETE methods only. The providing service has an explicit interface in GraphQL but the client only specifies the data that it needs.

Messages can be formatted in XML, JSON, or by Protocol Buffers. These formatting options differ in verbosity and in strong versus weak typing.

12.11 Exercises

These exercises use an assignment that you have written for another class. Choose such an assignment.

1. Package your assignment as a microservice.
2. Define Protocol Buffers for your microservice.
3. Write an invoking routine that is packaged as its own microservice and invokes the microservice from Exercise 1 using Protocol Buffers.

4. Write a simple discovery service that stores key value pairs where the key is the name of a microservice and the value is its IP address.
5. Register the microservice from Exercise 1 with the discovery service.
6. Use the discovery service from the microservice that you wrote in Exercise 3 to invoke your microservice.

12.12 Discussion Questions

1. An RPC call is routed to a port on the receiving side. This port is dynamically assigned. How does the message get routed to the correct port?
2. Why does the fact that a microservice is independently deployable mean that its location must be discovered at runtime?
3. What problems can arise with a system designed using a microservice architecture and how would you solve these problems?

Chapter 13 Post Production

You have deployed your code and its in production. Your job with respect to your service is not over. This chapter deals with testing and improving your service after it is in production. First, you should review Chapter 6 Measurement. Then after reading this chapter, you will know about:

- testing of systems in production,
- setting thresholds for alerts.
- incident handling,

13.1 Coming to terms

Escalation—moving responsibility for a problem to a higher level

Five whys—a technique for finding a root cause that involves asking “why” at each stage.

Instrumentation– providing measurements four software.

SLO—Service Level Objective. A goal for a service with respect to one measurement.

SLA—Service Level Agreement. A contract between a team and a client that specifies a guaranteed level of service for one quality.

SLI—Service Level Indicator. A measurement that indicates a value of a quality of interest.

Ticket—a report of a problem.

Telemetry– automatically sending the data gathered by an instrument to a recording site.

13.2 Testing in Production

Even after your service goes into production, there are tests that you can perform. The main motivation for testing in the production environment is that there are some aspects of that environment that cannot be replicated in other environments. The most common aspect is scale: If your system runs at national or global scale, it isn’t feasible to replicate that environment. Related to this is workload. As your scale grows, it becomes harder to generate a workload that represents what your system will encounter in production. Finally, the hardware, networks, and dependent services used in the production environment may be unique and hard to replicate in other environments. It is also possible that security vulnerabilities can be discovered in your production system.

13.2.1 Chaos Engineering

Testing in production is particularly important for mission-critical systems that operate at global scale and require 24/7 availability. One testing approach is *chaos engineering*, which is defined as "the discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production."[47] An experiment (or test) introduces a failure into the system. There is a hypothesis about what should happen, which defines the passing conditions for the test. The actual response of the system to the failure determines whether the test passes or fails. Chaos engineering depends on having sufficient observability into your system to be able to detect (and then diagnose and fix) a system failure.

In earlier times, these failures could be introduced by unplugging a network cable or turning off a physical machine.[48] Introducing such failures into a virtualized environment is more complicated. The most well-known of the live testing suites is Netflix's Simian Army, now deprecated, although some of the ideas such as the Chaos Monkey come from Google and other contributors. The members of the Simian Army are called monkeys– imagine a troop of monkeys climbing through your data center flipping switches and pulling cables. The monkeys include

- *Chaos Monkey*–randomly shuts down virtual machines in production to ensure that small disruptions will not affect the overall service. As a developer of a mission-critical service that requires 24/7 availability, you must prepare your service for failure. Part of this is stateless operation; another part is for your clients to assume failure quickly in the event they detect that your service is not responding.

- *Latency Monkey*–simulates a degradation of network service and checks to make sure that upstream services react appropriately. This is similar to

[47] See https://principlesofchaos.org

[48] Some quality-assurance group managers were known to "accidentally" trip over a power cord and unplug a physical server during testing, to observe how the system responded.

the Chaos Monkey except that it operates on network and network latency rather than on VMs.

The decision to perform testing in the production environment is usually made at a high level in the organization, as approaches such as chaos engineering may impact multiple systems. Some tests will fail, which may have significant organizational impact. Before testing in the production environment, your organization must have sufficient confidence in its infrastructure and systems that few tests will fail and that the test failures will not have catastrophic consequences.

13.2.2 Environment Checking Tools

There are also a collection of tools whose purpose is to ensure that the environment in which your system is running is appropriate in a number of dimensions. These tools include

- *Conformity scanners*–detects instances that aren't coded to best practices and shuts them down, giving the service owner the opportunity to relaunch them properly. Best practices here include phenomena that are externally visible such as generating logs.
- *Security scanners*–searches out security weaknesses. A security scanner may terminate the offending instances. It also ensures that SSL and DRM (Digital Rights Management) certificates are not expired or close to expiration. A security scanner can also check whether any new vulnerabilities have been found for your production system. Your platform provider has mechanisms to support security. In AWS, for example, every VM should belong to a security group. This membership is detectable through querying of AWS features. Certificates are kept external to your service and thus they are externally visible and can be checked for expiration.
- *Doctor scanners*–performs health checks on each instance and monitors other external signs of process health such as CPU and memory usage.
- *Janitor scanner*–searches for unused resources and discards them. It becomes easy in the cloud to fail to deallocate resources. Finding unused resources helps to eliminate them. In addition to the financial cost of maintaining unused resources, there may also be security implications. One published security breach occurred because a VM went unpatched

and hence was vulnerable to known attacks. Once the VM was broken into, credentials were retrieved that could be used for active VMs.

13.3 Service-Level Thresholds

Organizations and users depend on systems, and when a system doesn't deliver the needed capability, an organization's business may suffer, or users may not receive the service that they paid for. To help mitigate this risk, organizations and users create Service-Level Agreements (SLAs) with the organization that develops and delivers the systems that they depend on. An SLA identifies a metric (e.g., request latency for a service) and a threshold (e.g., 99% of requests will receive a response within 300 milliseconds). A typical agreement will contain many of these metric/threshold requirements. If the client is outside your organization, these SLAs may be part of a legal contract, and your delivery organization may have to reimburse your clients if your systems are not available, are insecure, or perform poorly.

Because breaking an SLA is a major incident for a development-and-delivery organization, most organizations allocate the requirements to meet an SLA down to the services within the system.

You do not want to be surprised by the breaking of an SLA, so you set a tighter threshold for internal monitoring purposes. This is a Service Level Objective (SLO). A SLO gives you some breathing room. If you violate an SLO, you have some time to repair the problem before the SLA would be violated.

Some SOs are not directly observable. Availability, for example, is a common SLA and, hence, SLO. Availability, however, cannot be directly measured and so you define a surrogate. This is the Service Level Indicator (SLI). The SLI for availability is typically error rates for requests. Thresholds are set in terms of SLIs. The SLIs are monitored on some periodic basis. The period will depend on the maturity of your service (shorter when the service is newly installed), and the criticality of your service to the overall meeting of the SLA.

Your service may have more than one SLI, with each monitoring a different function path or capability of your service. The terms *instrumentation* and *telemetry* are frequently used to refer to the recording of metrics from executing software. Instrumentation means providing measurements–placing an instrument in your software. Telemetry means automatically sending the data gathered by an instrument to a recording site. For example, if you wear a smart watch such as a

Fitbit, you have instrumented yourself and it sends the data it observes (telemetry) to a recording site where you can subsequently examine it.

Some typical SLIs are

- *Latency and throughput*. Latency is the time between a message arriving at a service and the response being returned. Through is the number of requests processed per unit time. Recording time of arrival and time of response allows the calculation of both latency and throughput.
- *Request satisfaction rate.* Recording a request on arrival and whether it was satisfactorily served on response is an availability measure.
- *Traffic*. Number of requests arriving at your service per unit time.
- *Saturation*. The measure of utilization of the resources (CPU, network, memory) that your service relies on.

Note that these SLIs can be determined by recording information at message receipt and message response. Generating too many recordings will have a performance impact and generating too few will not give you the information you need to understand how well the service is performing.

Monitoring software, discussed in Chapter 6 Measurement, uses the data recorded by the services, among others, as basic input. SLIs are used by the monitoring software to determine whether to send an alert.

You should define a benchmark for “good” SLI values. This will provide the values for the thresholds that you set. You can use historical values if you are deploying a new version of a service. If the service has been dramatically modified or is new, then setting the benchmark will be a matter of trial and error. Making the benchmark too tight will result in false positives, making it too loose will result in false negatives. Appropriate settings will speed the process of incident response – our next topic

13.4 Incident Response

An Incident an event that could lead to loss of, or disruption to, an organization's operations, services or functions. In software terms, an incident is either a performance or availability problem and we elaborate on this below. It could also be a security problem, generated by a network Intrusion Detection System and dealt with by the security specialists within an organization. We do not discuss security incidents.

13.4.1 Life Cycle of an Incident

An incident is detected by a monitoring system based on the SLI threshold values you have established. The monitoring system then sends out a page[49] to a first responder. The monitoring system also enters the incident in an incident repository. The incident repository is frequently called a *ticketing system* since, historically, incidents caused paper tickets to be generated which were passed upwards in a process of escalation.

You will also see the term *escalation* in conjunction with incident handling. Escalation means that the ticket has been passed on to someone who has more knowledge of how to handle that particular type of incident. One of the purposes of the ticketing system is to keep track of who has current responsibility for dealing with the incident.

There are two goals when an incident occurs. The immediate goal is to fix the current problem. Get the system running normally again. The second goal is to determine the underlying cause of the incident so that it does not happen again. We give an example from the point of view of you being the first responder.

We begin near the end of the example. You are sleeping and your pager goes off. There is a problem with the system! You drag yourself from bed, log into your production environment, and look at your system's dashboard. You discover from your dashboard that one instance of a service has a very high response latency. Now you drill down into the information from that instance. isolate the problem to a slow hard-disk drive–it hasn't completely failed, but read operations are taking a long time to complete. You reconfigure that instance to move its temporary files to another disk and continue to monitor the dashboard. It appears that this has solved the problem. You add "replace disk" to the list of tasks for the operator and go back to sleep. In the morning, you can make a more permanent solution.

As you can see from this example, there is a tremendous amount of information at your disposal, as well as tools that allow you to analyze this information. We discussed the source of this information in Chapter 6 Measurement

Once the first responder determines the immediate cause of the incident, the next step is to determine an immediate remediation. That is, what can be done to solve

[49] See https://en.wikipedia.org/wiki/Pager. Today, most alerts are sent as text messages or automated phone calls, with few organizations still using pager devices for alerts. However, the term lives on; for example, referring to being on-call for incident response as "pager duty."

the immediate problem. Once this has been done, the incident is closed. The final step is done without the time pressure of having an active system with a problem and that is determining the root cause of the problem and ensuring that the problem does not reoccur.

One technique for determining a root cause is the *five whys*. This is a technique that involves repeated asking the question “why?”. Using the example above of a failed disk, the first question is “why did the disk fail?”. Assuming an answer is “the firmware failed”, the next question is “why did the firmware fail?”. And so forth. The sequence of questions ends when a process reason is discovered. “We check the firmware weekly”. Then the remediation is to check the firmware more frequently.

The incident, the immediate response, and the root cause are all recorded in the ticketing system and reported to the service owner. Whether the root cause is acted upon is a question of risk analysis.

Having all this information in a ticketing system allows determining common problems and solutions to those problems. Systemic problems can be found and fixed.

13.4.2 Ensuring Quality

There are three models for ensuring quality in production systems. The first two explicitly involve first responders. “You Build It, You run it” from Amazon and Site Reliability Engineering (SRE) from Google. Production Engineering from Meta (Facebook) is a hybrid of the Amazon and the Google models and does not explicitly involve first responders.

You build it, you run it

> *There is another lesson here: Giving developers operational responsibilities has greatly enhanced the quality of the services, both from a customer and a technology point of view. The traditional model is that you take your software to the wall that separates development and operations and throw it over and then forget about it. Not at Amazon. You build it, you run it. This brings developers into contact with the day-to-day operation of their software. It also brings them into day-to-day contact with the customer. This customer feedback loop is essential for improving the quality of the service.*

-Werner Vogels[50]

Werner Vogels was the Amazon CTO as it grew into the ubiquitous online platform that it is today. One of the processes at Amazon was making developers responsible for post-production operations. This includes having developers carry pagers and be the first responders in the event of an operational problem. The pager duty is rotated among team members so that no single person is always on call.

One assumption behind this philosophy is that a developer of a service is best equipped to locate problems with that service. As a developer, you are familiar with the internals of the service, can interpret the information collected, and your feedback as to modifications to deal with the root cause of a problem will be heeded by your fellow team members.

A drawback to the You build it, you run it philosophy is the assumption that because a service triggered an alert, the service has a problem. The problem can stem from upstream clients or downstream dependencies. The problem may be in the network. A developer of a service may or may not have a good understanding of the environment in which their service operates.

Site Reliability Engineer

Google, Netflix, and now many other organizations have a different philosophy for first responders. They have separate teams call Site Reliability Engineers (SREs). A system is assigned to a SRE team, and a member of that team is a first responder when an incident occurs. As with the you build it, you run it philosophy, pager duty is rotated among members of the team.

SREs have an overall understanding of a system and its environment. Thus, their knowledge is broader than that of a single service developer. It is, of course, not as deep with any individual service.

SREs oversee monitoring and SLIs. They also share a knowledge base of problems with the network, with systems, and with techniques for trouble shooting problems. To encourage development teams to listen to the advice given them by SRE teams, an SRE team has the option to refuse to support particular systems.

[50] https://queue.acm.org/detail.cfm?id=1142065

Production Engineering
Production Engineering is Meta’s (Facebook) philosophy for ensuring quality systems. A production engineer is responsible for reliability, scalability, performance, and security for production services. They are a separate organizational unit (like SREs) but are embedded into development teams (like you build it, you run it). Their skill set is similar to SREs in that they must have a broad understanding of the infrastructure and its components. Since they are embedded in development teams, they acquire a detailed knowledge of the internals of particular services.

T3.5 Summary

You may test your service in production using tools that kill processes or introduce latency into networks. In addition to these problem-inducing tools, other tools used after production will perform various janitorial services such as cleaning up unused resources.

An alert comes to you through monitoring software that records externally visible phenomena. When one of these phenomena exceeds a preset limit, an alert is generated. The limit should be high enough so that there are few false positives but also low enough so that are few false negatives. Choosing values for alerting is a difficult task.

An alert goes to a first responder who can be a member of a development team or an independent team. Once the alert occurs, the first responder examines logs to determine what caused the alert to be sent.

Quality is ensured either by having developers be a first responder or by having a separate team responsible for quality. Members of a separate team must have a good relationship with development teams so that their advice is heeded.

13.6 Exercises

1. Install Grafana and use it to display CloudWatch data from AWS (or the equivalent from your cloud provider).
2. Have an independent person cause a failure on a system you built by means unknown to you. Use the logs generated by your system to determine the cause of the failure.

13.7 Discussion Questions

1. Live testing affects instances in production. Discuss the pros and cons of performing live testing.

2. Do you think you have the skill set to be either an SRE or a Production Engineer? What skills are you lacking?

Chapter 14 Secure Development

When you finish this chapter, you will understand

- How to manage credentials both for services and for individuals
- Some techniques for developing secure services
- The reporting of vulnerabilities and the patch management process
- The software supply chain and some techniques to protect against introduction of malware into the supply chain.

14.1 Coming to terms

Access control list – a set of rules that allows or disallow access to resources

API management system -- controls usage of an API

Attack – the use of vulnerabilities by an adversary to achieve a technical impact

Capability – the ability of a user or system to perform specific functions on a resource.

COTS – commercial off the shelf software.

Credential – a means to verify identity or tools for authentication

CVE – Common Vulnerabilities and Exposure catalog

Ephemeral – short lived

False positive – an incorrect indication that a problem exists

False negative – an incorrect indication that no problem exists

GDPR – General Data Protection Regulation. A set of rules adopted by the European Union to protect data.

HIPAA – Health Insurance Portability and Accountability Act. A set of rules in the United States to protect health data.

Patch – a software fix to repair a vulnerability.

PCI – Payment Card Industry data security standard. Security conventions required to accept credit cards.

PII – personally identifiable information. Information that can be used on its own or combined with other information to distinguish a single person's identity

Supply chain – anything that impacts this evaluation, production, and distribution of your system

Token – a type of ephemeral credential.

Vulnerability – allows an attacker to gain access to a system or network.

Weakness – errors that can lead to software vulnerabilities.

Zero-day exploit -- an attack that exploits a vulnerability on the same day that the vulnerability becomes widely known

14.2 What to protect

Nearly every system contains some data or resource that you need to protect, to maintain the CIA properties of confidentiality, integrity, and availability. You will be concerned with the confidentiality of data that is identified for protection by legal regulations or contractual agreements, or where malicious access could damage your organization's reputation. In some systems, you need to protect resources, such as APIs, from malicious use to protect availability of the system for legitimate users. Your service may be used in a cyber-physical system, in which a malicious access to software resources that compromises the integrity of the data could result in physical harm to people or property.

There are several categories of data that must be protected. The first is user credentials for system access – user IDs and passwords. Despite recommendations, people reuse identities and passwords on multiple systems, and an attacker who recovers this information from your system will try to use those credentials to launch an attack against other systems. While your system may contain nothing of value to an attacker, your credentials may unlock other more lucrative systems. Consequently, regardless of the necessity for security for other data and resources of your service, any user ID or password must be protected.

A second category of data that must be protected is corporate sensitive data. Items such as business plans, sales data, and specifications for products could all be important to your organization's competitors. You should think broadly about what data is critical — for example, knowing what A/B tests your organization is running might provide valuable information to your competitors.[51] Even non-profit organizations have sensitive data that must be protected. Government organizations may categorize data as secret, top secret, or another classification that requires special handling.

Legal regulations define another category of data that must be protected. In the United States, the term *personally identifiable information* (PII) designates information that can be used on its own or combined with other information to distinguish a single person's identity. This includes information such as name, birth date, postal or email address, government-issued ID number such as social security number, financial account information, and includes secondary information such as your mother's family name or first pet's name. The determination of which data is PII is context dependent and can change over time as other linkable information becomes public. In practice, the focus on PII is being replaced by the European Union's (EU) General Data Protection Regulation (GDPR), which mandates how organizations acquire and handle a much broader category of data called *personal information*. The GDPR governs how personal information about residents of the EU is handled and includes imposition of fines for mishandling data. Because it applies to data about EU residents regardless of where the organization holding that data is based or operates, GDPR has had global impact. The GDPR defines personal information broadly – it is any information relating to a person, directly or indirectly, and includes photographs, social media posts, and location (geographic and IP address).

Both the United States and the EU distinguish health-related and biometric data as a subset of PII or personal information. In the United States, individual medical data is regulated separately by the Health Insurance Portability and Accountability Act (HIPAA), and the EU designates this as *sensitive personal data*. Access to this data may require explicit consent of the individual and creating auditable records of all accesses.

Another legal concern is that some countries designate certain data that cannot be transmitted or stored outside the country's border. If your organization uses a commercial cloud provider, complying with this mandate requires special attention to selection of cloud region for your computation and storage, and attention to how data is handled by the infrastructure services that your provider delivers.

[51] https://blog.jonlu.ca/posts/experiments-and-growth-hacking

Some critical data must be managed in accordance with contractual agreements rather than government regulation. An example of this is compliance with the Payment Card Industry Data Security Standard (PCI DSS, or simply PCI). Any vendor that accepts a credit card for payment or processes payments must comply with these standards as part of their agreement with the credit card issuers. PCI DSS covers storage and transmission of cardholder name, card number, security code, and expiration date to control credit card fraud.

Specific practices for protecting legally or contractually regulated data are beyond the scope of this book, as this crosses over from technical concerns to legal concerns, and nothing in this book should be construed as legal advice. Most commercial and government organizations have guidelines or policies for how to identify and handle such data, and your organization’s enterprise architecture may embody these policies as specific technical guidance.

Examples of data protection practices that many organizations adopt include

- *Avoid the collection of critical data (if possible) or minimize the amount the critical data acquired and retained.* For example, data can be anonymized or *de-identified* before storing so that it is not traceable to an individual.
- *Encrypt critical data when* at rest *and* in motion *(on disk and over the network).* Critical data includes credentials and corporate sensitive data as well as data protected by laws or regulations or contracts.
- *Use data models and schemas that separate critical information from other data.* This provides several benefits. Access control to the critical information can be separated. Business analytics can use only non-critical data, allowing the critical data to be deleted as soon as feasible. Finally, you may use different storage services for each type of data to comply with geographic storage restrictions.
- *Do not expose critical data in logs*. Log files are rarely encrypted and pass across many screens when there is a problem in the production environment. In some cases, log files may be shared with commercial off-the-shelf (COTS) software or cloud vendors if the problem may be related to their software or environment.

In addition to data, critical resources must also be protected. A list of techniques (some 200 items) for protecting critical data and resources is available in the U.S.

National Institute of Science and Technology (NIST) publication 800-53.[52] Critical resources are attacked for one of two reasons: to inhibit the availability of a service or data (the A in CIA) or as a means for gaining access to critical data (the C and I in CIA). Resources include both physical resources and virtual resources (abstractions built on real resources), such as[53]

- *CPU.* You can protect CPU utilization from misuse through configuring your operating system's process scheduling and by setting limits on physical CPU utilization by a container or VM.
- *Memory.* A recent study found that 70% of the vulnerabilities in Microsoft products were due to memory safety defects.[54] These include reading uninitialized memory and reading or writing to memory outside your service's allocated memory space. Actions that you can take in developing your service include using secure coding practices such as those of the Open Web System Security Project (OWASP).[55]
- *Disk space*. Disk capacity can be protected from misuse by setting physical limits on the size of virtual container and VM volumes and by setting limits on the virtual disk I/O bandwidth available to a container or VM (often referred to as *IOPS,* or inputs/output operations per second, by cloud service providers).
- *Network access.* Protecting network access when using a cloud service provider can be more challenging than protecting other resources. Network bandwidth limits may be implicit in the instance type you choose for your VM but these limits may be specified qualitatively (e.g., high bandwidth or medium bandwidth), and the exact limit quantity may change over time. Firewall rules can limit incoming connections to only those from IP address ranges where your clients are running and limit outgoing connections only to IP address ranges where services that you

[52] https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r4.pdf

[53] https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.performance/id_crit_resources.htm

[54] Mark Miller at BlueHat IL, 7 Feb. 2019. https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf

[55] https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide

depend on are running. However, this increases the need for coordination of deployment configuration among services.

Another type of resource that may need to be protected is your service's APIs. Access control (e.g., authentication and authorization, discussed below) provides some protection for APIs and other critical resources. The following list of practices for securing web systems is taken from Wikipedia.[56]

- *Sanitize inputs at the client side and server side*. *SQL injection* is a common type of vulnerability that embeds malicious SQL queries in API parameters. Other attacks involve providing input with excessive length to cause a buffer to overflow as part of a memory safety attack or providing massive amounts of random data to impact your service's performance and memory utilization. Sanitizing and verifying API input parameters before using them for any processing will help secure your service.
- *Encode request/response.* The HTTP/HTTPS protocol allows you to specify the character encodings that your API will accept and produce and allows you to more strictly validate your API inputs.
- *Use only current encryption and hashing algorithms.* NIST provides a list of currently validated algorithms and implementations. You should restrict your choice of encryption algorithms and algorithm configurations/options to the NIST list.
- *Do not allow HTTP/HTTPS requests to list a directory.* Allowing a malicious user to poke around in your service provides them with information to find vulnerabilities in your service. Related to this is the practice of hiding web server information: For example, the default error response on many web servers includes the server type, version, and other information that a malicious user can use to find vulnerabilities.
- *Do not store sensitive data inside cookies.* Cookies are stored on a client and, consequently, are outside of your control.

Services should be reviewed for security. There are different security review processes[57] for requirements, design, and development. There are also specialized security tests that can be used when the service is ready for test.

[56] https://en.wikipedia.org/wiki/Software_development_security
[57] https://www.owasp.org/index.php/Code_Review_Introduction

14.3 Software Supply Chain

Much of the software that is executed by your services, from the operating system to middleware and API frameworks, was developed by someone outside your organization as open-source or commercial off-the-shelf (COTS) software. Organizations should be concerned with the origin of all the parts that are assembled to produce a complete system. This is called the *software supply chain*, borrowing a term from physical manufacturing processes.

In addition to functional suitability of the software package, security concerns about the software supply chain include the maintenance of confidentiality, integrity, and availability. For example, software packages that include back-door access that bypasses authentication will not maintain confidentiality. Packages that have functional defects may affect the integrity of your system, and packages that are not actively maintained may affect the availability of your system. When acquiring COTS packages, you may be able to address these concerns as part of the contractual agreement with the vendor, although *the vendor's license* may not allow for negotiation or tailoring. When choosing to use open-source software, all the responsibility falls on the organization that decides to use the package.

Note that your software supply chain comprises the packages that are used during the execution of your system as well as the packages and tools that are used to develop, build, test, and deploy that system. These include your version control system, compiler/interpreter, test framework, deployment orchestrator, etc.

14.5.1 Selecting open-source packages.

Achieving a high-quality software supply chain begins by selecting high quality packages to include in your system and continues as you maintain those packages over the lifetime of your system. Some criteria for evaluating packages delivered by open- source projects are

- *Project maturity and development activity.* The use of the project in other systems, in production, demonstrates a reasonable level of maturity. Projects with a steady stream of commits from a broad community of contributors indicates that developers are interested in the project and that it is less likely to be abandoned. Finally, even stable, feature-complete packages should still show some activity to maintain compatibility as languages and dependencies evolve and to remediate security vulnerabilities.

- *Identified and engaged maintainer(s)*. An open-source project maintainer is the individual or small team that sets the project direction and priorities and accepts or rejects contributions to the project's codebase. Without oversight by an engaged maintainer, contributions can introduce vulnerabilities and malicious code. For example, in 2018, the maintainer of a popular node.js package lost interest, and this allowed malicious code to be committed.[58]
- *Repository used for the project.* The project should be housed in a modern repository with a defect-tracking system. Especially for very mature projects that may have started using a development infrastructure that is now obsolete, migration of the project to a modern development infrastructure indicates community commitment and openness. There should be build and test automation that allows a package user to assess test coverage and product quality.
- *Download-confirmation hash.* Open-source packages are replicated to mirror sites to make downloading them more efficient. An attacker may modify the package at one of the mirrors so that users download a compromised version of the package. The original developers publish a hash code of the package on a different location from the mirror so that a user downloading the package can compare the hash of the download to the published hash and verify that they are downloading an actual copy of the original package. There are many tools available to perform this validation.
- *Pedigree.* The pedigree of the core development team can indicate quality and the likely longevity of the project. Projects with commercial sponsorship may have more resources and be able to respond more quickly to fix defects or maintain compatibility with other parts of the package's ecosystem.

14.3.2 Securing the supply chain

After designing and developing your system, you may subject it to attacks as part of your security testing. There are two broad categories of attack testing. The first category, sometimes labelled *cooperative assessment* or *tabletop exercise*, is conducted by external experts in cooperation with the development team. This is a form of white-box testing, where the testers have full visibility into the system's

[58] https://gist.github.com/dominictarr/9fd9c1024c94592bc7268d36b8d83b3a

design and implementation. These exercises are scenario driven. Based on known vulnerabilities and weaknesses, the experts define an attack scenario, and the developers explain how the system will respond to the attack. Through this discussion, issues in the system's design and implementation are identified, and approaches to fix and mitigate are developed.

The second category of attack testing is called *adversarial assessment* or *penetration testing (pen test)*. The test team uses the same tools and methods as a malicious adversary to attack your system. The attack goes through the attack lifecycle phases noted above, although sometimes the test team receives non-public information about the target system to expedite the reconnaissance phase. There are several variations of this type of testing. In one variation, the target system's development and operations teams are aware that the test is being performed and are prepared to defend against the simulated attack. In this variation, the attackers may be labeled *red team* and the defenders labeled *blue team*, borrowing terminology from military exercises. This type of event often combines testing objectives with training objectives. In the other variation, the defenders are not notified of the simulated attack, and the event tests both the software and the operations processes to defend and recover. The duration of the test can vary: The attackers may have limited time for reconnaissance, weaponization, and delivery. This constrains the attack patterns than can be used – limited time available leads to the use of "off-the-shelf" attack patterns and technologies.

Security of supply chains has gotten a great deal of attention following some well publicized attacks on the supply chain.

We quote from two reports about hardening supplying chain.

1. The following material ix taken from a Cloud Native report "Software Supply Chain Best Practices"[59]

   This paper puts forth four key principles crucial to supply chain security:

   First, every step in a supply chain should be "trustworthy" as a result of a combination of cryptographic attestation and verification. No step in the

[59] https://github.com/cncf/tag-security/blob/main/supply-chain-security/supply-chain-security-paper/CNCF_SSCP_v1.pdf under the Creative Commons license.

supply chain should rely on assumptions about the trustworthiness of any previous steps or outputs — trust relationships must be explicitly defined.

Second, automation is critical to supply chain security. Automating as much of the software supply chain as possible can significantly reduce the possibility of human error and configuration drift.

Third, the build environments used in a supply chain should be clearly defined, with limited scope. The human and machine identities operating in those environments should be granted only the minimum permissions required to complete their assigned tasks.

Fourth, all entities operating in the supply chain environment must be required to mutually authenticate using hardened authentication mechanisms with regular key rotation.

NSA/CISA have released a Kubernetes Hardening Guidance[60], including the following hardening measures and mitigations:

- *Scan containers and Pods for vulnerabilities or misconfigurations.*
- *Run containers and Pods with the least privileges possible.*
- *Use network separation to control the amount of damage a compromise can cause.*
- *Use firewalls to limit unneeded network connectivity and encryption to protect confidentiality.*
- *Use strong authentication and authorization to limit user and administrator access, as well as to limit the attack surface.*
- *Use log auditing so that administrators can monitor activity and be alerted to potential malicious activity. Our blog post, "Chain"ging the Game – how runtime makes your supply chain even more secure, provides nice examples of runtime threat detection.*
- *Periodically review all Kubernetes settings and use vulnerability scans to help ensure risks are appropriately accounted for and security patches are applied.*

[60] https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/2716980/nsa-cisa-release-kubernetes-hardening-guidance/

A variety of scanners exist in the marketplace that will examine the bill of materials, The SBOM, of the containers in your system and report known vulnerabilities.

14.4 Weaknesses and Vulnerabilities

Software weaknesses are errors that can lead to software vulnerabilities. A software vulnerability allows an attacker to gain access to a system or network. Both weaknesses and vulnerabilities are listed in publicly available catalogues. The CWE[61] is the Common Weakness Enumeration catalog and vulnerabilities are listed in the CVE[62] (Common Vulnerabilities and Exposures) catalog.

A vulnerability is a specific weakness (or collection of weaknesses) in a particular software package that an attacker can use to gain access to a system. For example, vulnerability CVE-2018-9963 states

> *This vulnerability allows remote attackers to disclose sensitive information on vulnerable installations of Foxit Reader 9.0.1.1049. User interaction is required to exploit this vulnerability in that the target must visit a malicious page or open a malicious file. The specific flaw exists within the parsing of JPEG2000 images. The issue results from the lack of proper validation of user-supplied data, which can result in a read past the end of an allocated object. An attacker can leverage this in conjunction with other vulnerabilities to execute code in the context of the current process.*

Note that the vulnerability applies to a specific package and version. In this case, the weakness was lack of proper validation of user-supplied data. The vulnerability allows the attacker to execute code on the system.

The CVE list provides a common identifier for vulnerabilities but few technical details. Other catalogs, such as the National Vulnerability Database (NVD),[63] are indexed using the CVE identifiers and include impact scoring, which qualities (i.e., confidentiality, integrity, availability) are affected, and information about fixes, remediations, and workarounds.

An important term in the security vocabulary is *attack*. An attack is the use of vulnerabilities by an adversary to achieve a technical impact. An attack has

[61] https://cwe.mitre.org

[62] https://cve.mitre.org

[63] https://nvd.nist.gov

multiple lifecycle phases: reconnaissance, weaponize, deliver, exploit, control, execute, and maintain. Like a vulnerability, an attack is concrete and targets a particular system (or systems using particular software packages). Catalogs of attack patterns include the Common Attack Pattern Enumeration and Classification (CAPEC).[64] There are many catalogs of specific cyberattacks, maintained by cyber-security vendors and consultants. The term *zero-day exploit* refers to an attack that exploits a vulnerability on the same day that the vulnerability becomes widely known.

When selecting COTS and open-source packages, you want to ensure that the package does not contain any vulnerabilities, or if it does, that you include any patches or remediations in your initial configuration. You can search a catalog such as the CVE or use an open-source or commercial tool to scan your service for vulnerabilities (for example, if you are using GitHub for version control, it automatically scans your repository using the CVE and privately alerts you to any issues).

Note that many of the open-source scanning tools are dual use technology: The same tools are used by both developers, to build secure systems, and by adversaries, to attack those systems. Since vulnerability enumerations are linked to a package and version, you can identify vulnerabilities in COTS without access to source code. Vulnerability detection is objective and definitive: If your service uses a version of a package that has a reported vulnerability and you haven't otherwise remediated the vulnerability (e.g., through your configuration of the package), your service may be attacked. A vulnerability scan should not be treated like static analysis tools.: There are no false positive outputs here and there should be no filtering. Creating an SBOM, allows scanning for vulnerabilities at runtime. The scanner comares the SBOM to the CVE and notifies you of any matches.

Weaknesses, on the other hand, are potential problems. There are many commercial and open-source weakness scanners, with many reporting results using CWE identifiers. These include static analysis tools that operate on source code (usable only on open-source packages and your own service code), and tools that operate on binary executable files (usable on any software). A challenge when using these scanners is false positives – the output of a scan may identify hundreds or thousands of *possible* weaknesses, and it is impractical to analyze every warning. You will need to perform filtering or other post processing of the scan

[64] https://capec.mitre.org

results, and sometimes this filtering may result in true positive indications being suppressed.

One special aspect of the supply chain is the deployment pipeline we discussed in Chapter 11 Deployment Pipeline. Surveys[65] have found that more than 25% of cyber-crimes were committed by insiders–that is, from people working within your organization. For example, an insider could modify your continuous integration server so that malware is inserted into all your services, compromising all your systems. As we said, techniques to prevent this type of attack are to apply access control to restrict the modification of the deployment pipeline to a limited number of personnel and to notify all members of the development team when one of the elements of the pipeline has been changed.

14.5 Vulnerability Discovery and Patching

The process by which a vulnerability gets into the CVE and is then subsequently patched has the following steps:

1. Vulnerability is discovered, either by the developing organization or by an external individual.
2. Vulnerability is reported to CERT Coordination Center (CERT/CC). This starts the disclosure window (currently 45 days) before the vulnerability is publicly reported. The vendor or open-source maintainer is privately notified of the vulnerability, and the disclosure window gives them time to prepare a patch and incentive to react before the vulnerability information becomes public.
3. Vulnerability is publicly disclosed and listed in the CVE. This involves contacting the CVE maintainer (currently the MITRE Corporation) and receiving an identifier. This identifier is used to reference the vulnerability thereafter. The vulnerability is also entered into the National Vulnerability Database (NVD).
4. The vendor or open-source project issues a patch, referencing the CVE ID. The patch is publicized by the vendor, and a link to the patch is placed in the NVD.

[65] See https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=499782 and https://www.verizonenterprise.com/resources/reports/rp_DBIR_2018_Report_execsummary_en_xg.pdf

5. You are informed of the patch either through monitoring of the vendor’s mailing list or of the NVD, or through an automated patch management service.
6. You apply the patch.

Your challenge is to execute Step 5. There are many vulnerabilities being disclosed, each accompanied by a patch or other remediation. You use packages produced by multiple vendors. Each vendor has multiple products, and each product has multiple patches. If you monitor each vendor’s mailing lists, you will be receiving too much information every day to process efficiently. You should use an automated service that will filter this information for you based on the software that you are actually using. The SBOM allows you to know all the dependent pieces of software that are loaded on your behalf.

When you are informed of an available patch that is applicable to your software, you must decide whether to apply the patch. This will depend on the severity of the problem that the patch fixes and the disruption to your organization from applying the patch to all the instances using that software package. Suppose, for example, a patch is released for the version of the Ubuntu kernel that you are using for your production systems. You may have hundreds or even thousands of instances running this version of Ubuntu. Applying the patch is equivalent to releasing a new version of all your systems. Each system must be tested, the new version deployed, and old versions taken out of service.

One option is not to apply the patch and rely on the next normal build of your system to get to the latest patch level. A strong argument against delaying patch deployment is that well over 90% of successful attacks are made against unpatched vulnerabilities.[66] Some organizations rebuild all their systems daily or on a frequent periodic basis so that all their systems incorporate their patches without special action by developers.

Patch management is a serious problem for organizations. A best practice is that organizations have policies that specify the actions that you should take with respect to vulnerabilities and patches.

[66] https://www.whoa.com/data-breach-101-top-5-reasons-it-happens/

14.6 Authentication

To maintain access control over resources, your service must be able to prove that a user on the internet is who they say they are (authenticate users). We begin by discussing identification factors for users.

14.6.1 Identification factors

Three categories of identification factors are: 1) what you know, 2) what you have, and 3) what you are.

1. What you know. Items in this category are passwords, pin numbers, and keys.

2. What you have. Items in this category are a smartphone with known number or a smart card

3. What you are. Various forms of biometric identification (fingerprints, facial recognition) exist and the technology for recognizing biometric identifiers continues to improve.

Two factor authentication (TFA), where the user must provide two different factors, are becoming much more common.

14,6.2 LDAP

The Lightweight Directory Access Protocol (LDAP) is the standard method for managing credentials of members of an organization. LDAP is a protocol with different implementations. Active Directory is an implementation by Microsoft of the protocol.

Centralizing an authentication service makes sense when an organization has multiple systems that require authentication. Almost every organization fits this description. When a new individual joins the organization, they are given a login identifier and a method for authenticating themselves. Regardless of whether the login identifier address and authentication method are provided by the new member or generated by the organization they must be entered into the central repository. LDAP is the protocol used to make and retrieve entries into the repository.

Every system in the organization will register with the LDAP server to provide its authentication requirements. A user of one of these systems will provide their login credentials, the system will pass them to the LDAP server, and if the user has provided valid credentials, the LDAP service will return an acknowledgement that the user is known to the organization.

LDAP also plays a role when an individual leaves the organization. Their LDAP entry is deleted, and they can no longer use the systems of the organization. One security rule calls for rapid removal of a former member's authentication, but this depends on the circumstances of the individual leaving the organization.

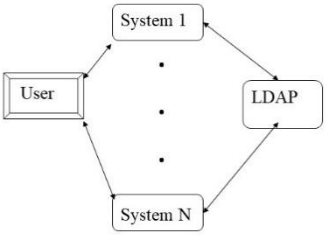
14.6.3 Single Sign on

Users do not like providing their credentials multiple times to a set of independent but related systems. Single Sign On (SSO) is a mechanism whereby providing credentials once is sufficient.

Figure 14.1 shows the protocol used to achieve single sign on. First, the systems to share a single sign on must be registered with the LDAP server. The user must be registered with LDAP as well. Then when a user provides their credentials to one of the systems, the credentials are forwarded to the LDAP service which validates them. LDAP returns a token to the system requesting the validation. The system returns the token to the user.

Tokens are stored on the user's local system. Logins to the other related systems are made using the token. The token is ephemeral. That is, it expires after a certain period. After expiration, the user must present their credentials again to use one of the systems. Two representations are widely used for tokens: Kerberos (a multi-headed dog that guards the gates of the Underworld) and SAML (Security Assertion Markup Language). Tokens are not only used in LDAP and SSO but also in OAuth and other authorization methods.

Figure 14.1: Single Sign On Protocol



14.7 Authorization

Authorization is the process of determining whether a user or system has the privileges to perform a specific function on a resource. The resource could be data, an API, a file, or a hardware resource.

The specific function will depend on the type of the resource. Data can be read, written, or deleted, files can be read, written, deleted, or executed. APIs and hardware resources can be accessed.

There are two ways of looking at authorization. One is from the perspective of the resource. From this perspective, authorization is granted to a list of users, roles, or holders of credentials. Access Control Lists (ACLs) are typically used to enumerate the entities entitled to perform functions on the resources.

The second perspective is from the point of view of the user or system. A user or system has the *capability* to perform specific functions on a resource. Access control lists are more widely used than capabilities.

Several principles control how authorization should be performed.

- Authorization policy should be governed by a set of rules established by a system owner. These rules should be independent of any implementation of the system. That is, the rules can be changed without requiring changes to the code and redeploying the system. These rules should be version controlled. The rationale for this principle is it simplifies administration.
- Least privilege. Users require different privileges to do their jobs. Any individual user should be granted the least privileges necessary for their job. The rationale for this principle is that it reduces the number of individuals who must be trusted with sensitive information.

14.7.1 Role Based Access Control

When an organization has more than several hundred members, it becomes too cumbersome to manage credentials individually. Role Based Access Control (RBAC) is a method for managing the credentials for groups of members.

Every member is assigned to one or more roles. Credentials are assigned to roles. For example, a developer is a role, a tester is a role, a bank manager is a role, a student is a role, and so on. A student may also be a teaching assistant and that would be an additional role assigned. The roles an individual performs are recorded in the LDAP server. When an individual logs in, the credentials they are granted will be based on their role. When an individual changes their role – a bank teller may be promoted – their new role is entered, and their old role deleted.

RBAC simplifies the administration of credentials but, as with everything, it has problems.

- Explosion of roles. Suppose in a bank with multiple branches, one branch gives managers the power to approve loans up to $10,000 and another branch gives managers power to approve loans up to $20,000. Are these two different roles? Managing the subtleties of the various roles becomes a problem.
- The same credentials are assigned to everyone is the same role. Suppose an individual leaves and they had database credentials. How are these credentials rescinded for that individual and not for the other individuals in that role? This is one use case for rotation of credentials. Rotation, in turn, is one use case for Vault, discussed in a subsequent section.

14.7.2 OAuth

OAuth is an open-standard authorization protocol or framework that provides applications the ability for "secure designated access." For example, you can tell Google that it's OK for Linkedin to access your contacts without having to give Linkedin your Google password. This minimizes risk: In the event Linkedin suffers a breach, your Google password remains safe.

OAuth doesn't share password data but instead uses authorization tokens to prove an identity between consumers and service providers. OAuth is an authentication protocol that allows you to approve one application interacting with another on your behalf without giving away your password.

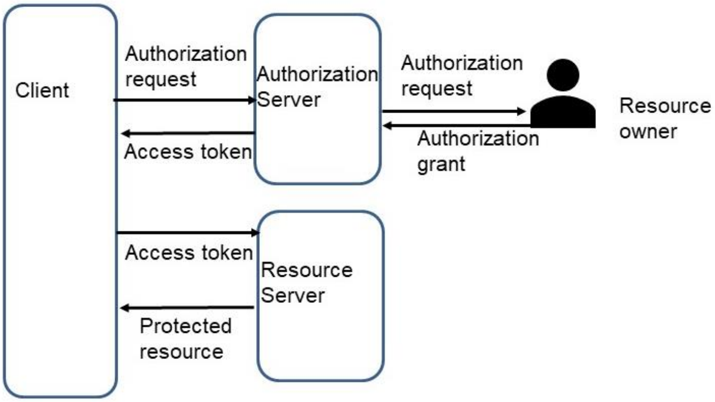
The OAuth protocol[67] is a standard that replaces authentication credentials with *access tokens* that are created, exchanged among, and used by four roles. The *resource owner* is the person (or service) that has the credentials to access the protected resource. The *resource server* hosts the protected resource and can accept and use tokens to grant access to the resource. In the scenario described above, your service would act as an *OAuth client* that accesses the protected resource with the authorization of the resource owner. Finally, an *authorization server* issues access tokens to the OAuth client, after authenticating the resource

[67] IETF RFC 7591

owner and obtaining authorization. The flows between these roles are shown in Figure 14.2.

Note that implicit in Figure 14.2 is an authentication of the client if the client is a user. OIDC (OpenID Connect) is an identity management layer intended to work with OAUTH to provide identity management for individuals. It could link to a LDAP server or other authentication mechanism. Be aware that the authentication must exist for users to be authorized.

Figure 14.2: OAuth Roles and Data Flows



In the OAuth flow in Figure 14.2, OAuth requires the client, the resource owner, and the resource server to coordinate with a central authorization server. Every API is treated as a separate protected resource. The registration protocol for OAuth is complicated but fundamentally, the API (protected resource) registers with an OAuth authorization server. The resource owner registers with OAuth and receives an identification token that specifies the privileges of that user. The client requests a token from the OAuth server that allows it to access that protected API. The client passes this token and the user token to the API and is granted access. Notice that three different entities are registered with the OAuth authorization server–the protected API being accessed, the client performing the access, and the resource owner. The protocol is quite general but one sequence to achieve access to an API is

1. The resource owner authenticates to the OAuth authorization server. This returns a single sign-on identification token that is saved so that the

resource owner does not have to log in for every access to the protected API.

2. The client gets an initial access token that can be used to register specific instances of the client. The client identifies itself to the OAuth authorization server. It specifies the access privileges available to any access through this instance.
3. The credentials of the user of the client and of the client are then sent to the protected API, which verifies them with the OAuth server.

OAuth servers may be embedded into *API management systems* since they control and monitor access to APIs. An API management system monitors usage of an API and, in addition to access control, will gather API usage metrics for analytic purposes.

Note that in OAUTH it is the responsibility of the client to maintain the access token. Each system or user may have different access tokens setting up a proliferation of access tokens. This is one use case for Vault.

14.7.3 Managing Credentials[68]

A credential is anything that can be used for authentication or authorization. Username/password, database credentials, tokens from Single Sign On are examples of different types of credentials. Vault is a common open-source tool used to manage credentials.

Credentials might be recorded in multiple locations and, potentially in clear text. Both of these factors may cause leakage of the credentials. Vault is a centralized solution. By making credential management a service used by other systems, the service can be responsible for encrypting the credentials and the individual systems delegate the responsibility for saving and retrieving the credentials to Vault.

Furthermore, Vault can provide access control and an audit trail of who has accessed which credentials. Vault can also be responsible for rotation of credentials. Suppose the database access credential is available to an individual no longer with the organization. By changing the database access credential in Vault, individuals or systems that need access to the database will retrieve up-to-date

[68] This material is adapted from https://learn.hashicorp.com/tutorials/vault/getting-started-intro-ui?in=vault/getting-started-ui

versions of the credential whereas the individual no longer with the organization will have no access.

Once there is a centralized solution to the problem of managing credentials, the credentials can be made ephemeral. We saw an ephemeral credential in Section 14.3.3 Single Sign on. Ephemeral credentials have expiration periods. If a system leaks a credential through exposing it in a log or a traceback, the compromised credential has a lifetime after which it is no longer valid.

Note that there are two actors associated with a credential. There is the resource owner and the client. The resource owner must be registered with Vault so that the client can access credentials for that resource.

14.7.4 Managing Credentials for Access to Services

Now we have the mechanisms for one solution to the authorization problem. SSO, OAuth, and Vault work together to do the management. The following steps will provide secure access for systems and individuals.

1. The resource server is registered with Vault. This is at the initiative of the resource owner. This allows Vault to manage the credentials for the resource server.
2. A system that acts as a client can register itself with Vault. An individual stores their SSO token in Vault. Vault can be linked to the SSO server to give Vault automatic access to the user's credentials. This step gives Vault identification and access information about individual clients.
3. A client uses OAuth to request access to the resource. The returned access token is stored by the resource into Vault.
4. The client retrieves the token from Vault. Vault verifies that the client's credentials allow them access to the resource and returns the current access credential for the resource.
5. The client uses the access credential to access the resource.

This process has a set up cost in terms of registering resources with Vault but the cost is one time.

14.8 DevSecOps

In recent years, DevSecOps has replaced DevOps as the abbreviation for the set of practices and tools that we have described in this book. Some will claim these are two separate disciplines, others that the Sec is mainly to emphasize that security is an inherent portion of DevOps. It is true that historically DevOps has been about the portions of operations that do not include security. Yet security has always been a portion of the description of the duties of the IT organization. We agree with the second view that DevSecOps serves to emphasize the security aspects of operations.

The impact of this relabeling, however, has been significant. Now security personnel are included in DevOps teams and multiple security testing and scanning tools have newly entered the market. To summarize the security aspects of DevSecOps, we see

- Authentication and authorization services including credential management
- Monitoring of individual microservices and consolidated logging.
- Secure communication between pods in Kubernetes (e.g. with TLS)
- Tools that examine the bill of materials for containers and compare them to known vulnerabilities.
- Securing the deployment pipeline
- Securing the orchestration platform
- Protecting Infrastructure as Code
- Building secure container registries

Most of these practices have matured dramatically since the Sec was added to DevOps.

14.9 Summary

You must identify which data and resources you wish to protect. Examples of such data are credentials, PII, and company proprietary information. Government regulations specify some data to protect, e.g., GDPR, HIPAA. Other processes are developed privately and are enforced through contracts such as PCI. Placing sensitive data in log files is a bad practice since log files are stored and transmitted in clear text and can be read by a great many individuals.

Authentication is the process of ensuring that you who you say you are. Organizations use centralized servers implementing LDAP both for authentication and for Single Sign On.

Authorization is performed for APIs, and OAuth requires the end user, the client, and the service API all to register with a central server.

Vault is a credential management system that provides credential rotation and centralizes credential management in one system.

Developers also must access and protect credentials to utilize services. Role-based access control is a technique for reducing the overhead associated with providing individuals with the credentials they need to do their jobs.

Your software supply chain comprises all the externally developed software that is used to develop, test, deploy, and operate your system and infrastructure, including packages that you build into your services. The supply chain must be vetted for security. Packages that are used can be scanned and the results compared to repositories of weaknesses or known vulnerabilities. One special concern in the supply chain is the integrity of the deployment pipeline. Insider attacks can compromise the pipeline and introduce malware into all your services.

The CVE provides a list of vulnerabilities, and the process for getting a vulnerability onto the CVE includes both keeping the vulnerability unannounced for a period to allow the vendor to react, and then making it public to encourage the vendor to react in a timely fashion.

14.10 Exercises

1. Install and use OAuth to protect an API for the microservice you used in the exercises in 13.7.

2. Integrate OAuth with Vault and repeat exercise 1.

14.11 Discussion Questions

1. Find the last several vulnerabilities for Jenkins. How many of them have been patched?

2. Find the details of one of the recent credit card breaches. The Security and Exchange Commission issues reports where you can find the details. Did the company that was broken into comply with PCI?

3. Generate a list of roles for your organization or university. What privileges should each role have?

4. How long after a vulnerability is discovered and published should you apply a patch?

15 Disaster Recovery

In Chapter 12 Design Options we talked about approaches to keep your service available when an instance or a network connection fails. The scope of the hardware and network failures that we considered in that chapter were limited. In this chapter, we will discuss how to handle catastrophic events.

When you finish this chapter,

1. you will know how to quantify restoration goals in terms of *recovery point objective (RPO)* and *recovery time objective (RTO)*.
2. You will see that not all systems have the same disaster recovery requirements,
3. You will understand approaches to restore your system at a geographically separated location to achieve the required RPO and RTO.

15.1 Coming to Terms

Big data – data too large to back up

Data shards—a piece of big data

Disaster—a flood, fire, earthquake, or other incident that puts a data center out of commission.

Failover—invoking the secondary data center.

Risk—the probability of an event occurring times the cost if it does occur.

RPO—Recovery Point Objective. Goal for limiting data loss in the event of a disaster.

RTO—Recovery Time Objective. Goal for getting a system back into operation after a disaster.

15.2 Disaster Recovery Plan

The types of events that we consider here, such as fire, flood, earthquake, or tornado, may impact the people, processes, and technology that support your business, and may impact your organization, your customers, or both. A comprehensive plan to keep your business operating in response to such an event must address many concerns: The U.S. National Institute of Standards and Technology (NIST) identifies eight different types of plans that should be considered,[69] including a plan for establishing communication with external customers and internal staff and a plan for protecting people and equipment at a particular facility.

The terminology can be confusing. We will use the term “business continuity” to refer to the complete set of concerns, and “disaster recovery” to refer to the information technology concerns. You may play a small role in developing some of the plans that focus on management concerns of overall business continuity, and you will likely play a larger role in plans that focus on the more technical concerns of disaster recovery.

All business continuity planning is a risk-mitigation activity. Risk can be quantified: The risk of an event is the probability of the event occurring multiplied by the loss incurred if the event does occur. Business continuity planning focuses on events that disrupt an essential facility or activity for an extended period. These can be localized disruptions such as a fire that destroys your headquarters building, or regional disruptions such as a hurricane or typhoon that devastates thousands of square miles. There are people in your organization or consultants who specialize in calculating event probabilities and costs. Executives must then decide how much your organization will spend to prepare for and respond to these events.

Disaster recovery is one part of overall business continuity. As we will see, a disaster recovery plan usually involves securely relocating systems and infrastructure services to an alternate location that is geographically separated from the location affected by the catastrophic event. A disaster recovery plan has multiple levels. Parts of the plan will address the overall recovery process, identifying the alternate location(s) and basic infrastructure services needed by all systems. There will also be specific procedures for restoring each system and each microservice.

Although the goal of a disaster recovery plan is to restore technology operation, the plan covers more than technology and will consider people and other organizations. For example, suppose a disaster occurs and the person who can unlock the encrypted backup is on vacation. Or suppose a disaster occurs and your customers have lost power and cannot access your systems regardless of

[69] See Section 2.2 of NIST Special Publication 800-34, *Contingency Planning Guide for Federal Information Systems*, https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-34r1.pdf

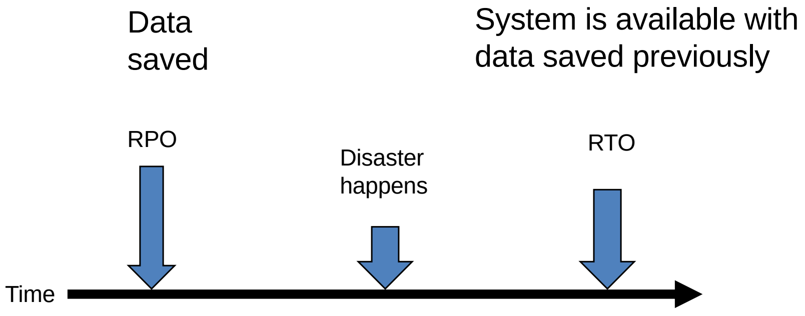
whether your systems are available or not. These are all outside of our scope. Here, we will focus on the cost factors, techniques, and considerations that go into the technical aspects of a disaster recovery plan.

Let’s start with how to specify your restoration goals, using the measures recovery point objective (RPO) and recovery time objective (RTO).

15.2.1 RPO and RTO

To understand RPO and RTO, visualize your system executing over time and saving information in a database. When a disaster occurs, there are two fundamental questions you must answer: How long until my system is serving users again, and how much data am I willing to lose in responding to the disaster? These two questions are formalized in the measures RPO and RTO. Figure 15.1 shows these two values graphically.

Figure 15.1: RPO and RTO Presented Graphically



RPO–Recovery point objective. What is the point in time for which I have a copy of the database? Data stored in the database between the last backup and the disaster will be lost. Assuming backups are periodic, the RPO is the time interval between backups.

RTO–Recovery time objective. What is the maximum amount of time that the system can be down before customers can access it again?

Your initial reaction is probably to specify “0” for both RTO and RPO–you want immediate restoration with no data lost. How much will that cost? Achieving very low RTO and RPO values can be quite expensive. What is it worth to you to achieve these goals? Think of your disaster recovery plan as an insurance plan.

How much are you willing to pay for insurance against something that may or may not happen?

The first step to quantify the cost is to prioritize your systems.

15.2.2 Prioritizing Systems

Depending on the scope and nature of your business, your organization may have hundreds if not thousands of different systems. These will range from mission-critical systems that process revenue-generating customer transactions to monthly reports of how much vacation time each employee has accrued. While every system is important to someone in your organization, not every system (and perhaps none of them) requires a “0” RPO and RTO. For example, if your organization is a financial institution such as a bank, allowing online access to customer accounts is mission critical, whereas your mail server is less critical and a print server may be even less critical. On the other hand, if your business is commercial printing, you might prioritize restoring the print server over other systems.

Grouping systems into restoration tiers, based on RPO and RTO, simplifies planning by allowing you to share recovery processes across systems within a tier. A common model divides all systems in your organization into four tiers. Tier 1 will have, nominally, a 15-minute RPO and RTO, Tier 2 will have two hours for each value, Tier 3 will have four hours, and Tier 4 will have 24 hours. Tier 1 systems are mission critical, Tier 2 are important support systems, Tier 3 are less important support systems, and Tier 4 is everything else.

The exact RPO and RTO values will depend on your organization’s business. They do not have to be equal. For Tier 1, the RPO and RTO targets will usually be different. If you are an online retailer, then your Tier 1 targets will have smaller values, and so forth. Setting an RPO target may depend on the system’s workload characteristics and its function: How many transactions will be lost, per minute of RPO? If you know that you have lost some data, can you recover it, for example, by examining the logs of upstream services and manually recreating the lost transactions, or by asking users to check for and re-enter missing transactions? You can see that developing the values for the different tiers and categorizing the systems requires a deep understanding of your organization’s business and strategy.

One final note is to be sure that you include some minimal software-development capability in your restoration prioritization and planning. Particularly for a smaller organization, where your software-development group is impacted by the same disaster that strikes your other systems, you may need the capability to troubleshoot and fix your systems during the restoration process, and your plans should reflect this need.

Once the systems have been prioritized and RPO and RTO values assigned for the different tiers, you can begin to develop technical strategies for system recovery. Before we go into strategies, however, we discuss data-center structure and geographical distribution.

15.3 Data Centers

Your systems execute on hardware inside a data center. Your organization might operate its own data centers, lease space in a service provider's data center, use a public cloud, or use some combination of these. In any case, a data center is a physical location that has roughly 100,000 computers along with essential services such as a backup power source (for some period), physical security and access controls, its own fire-suppression system, and its own air-conditioning system.

Now suppose a disaster occurs. For the purposes of discussion, we will assume it's a flood, but the same considerations apply to other disasters. Your organization has taken reasonable precautions to prevent such a disaster, such as not situating the data center in a flood plain where there is frequent flooding, but despite that prudence, an event occurs that floods the data center. The flood affects all the computers in the data center, as well as routers, network cables, and other hardware. There is NO software running in this data center after the flood, and it will take days or weeks to clean the facility, dry or replace the hardware, and restore normal operation. In the meantime, your organization's only way to restore operation of software systems is to run them in a secondary computing facility, either operated by your organization or rented from a provider. This secondary data center should be geographically located far enough away from your primary data center so that both data centers will not be affected by a single disaster.

In addition to computing resources, to restore a system to service you need two more things: You need all the software that the system comprises (including the software for any infrastructure services that it depends on), and you need valid data for the system (and possibly for some infrastructure services). This switch from your primary data center to the secondary one is called *failover*, and we discuss the process in detail later in this chapter.

A disaster recovery strategy has three parts: identify a secondary computing facility, provide all needed software, and provide data. Each part changes at a different rate. If your peak computing requirements change slowly, the selection of a secondary computing facility might be reviewed once or twice a year. Your software will change much more frequently. If you use a continuous delivery deployment pipeline, as described in Chapter 11 Deployment Pipeline, your software could change many times every day. Finally, the data for your system will change nearly constantly, as every user request is likely to change the data.

These three parts define what is needed to achieve a required RTO and RPO. To achieve an RTO, you must re-assemble all the pieces of software and data at the secondary location, which takes time; and to achieve an RPO, you need to make and retain snapshots of your software and data, and then securely move them to the secondary location.

Much of the thinking about disaster recovery approaches emerged before the rise of cloud computing. An organization either owned its data center facility and hardware, or leased space in a shared facility. Because computing hardware was a relatively expensive resource, disaster recovery approaches focused on minimizing that cost. Cloud computing has changed some of the cost considerations, but many of the other considerations, such as the time needed to securely transfer data to the secondary facility, have not changed.

There are four approaches commonly used for disaster recovery. Note that there *is not* a one-to-one correspondence between these approaches and the four system-priority tiers that we discussed above. We will begin with the approach that provides the longest RTO, and work toward shorter RTOs.

Cold secondary location: This location has space available, and basic power and cooling, but no computing hardware in place. Before software and data restoration can begin, computing hardware must be acquired and installed, which will take at least several days. This approach is the least expensive and may be satisfactory for some systems or services used by employees. For example, your software-development organization may use this approach to restore full capability.

Warm secondary location: A warm location has space, power, and cooling and has computing hardware and network connections in place. There may be some basic infrastructure services in place; for example, a management gateway as we discussed in Chapter 4 The Cloud. However, the needed system software and data is not loaded. In the case where your organization is not using cloud computing, a warm location may be an owned or leased data center that serves as the primary location for less-essential systems. When the disaster occurs, the less-essential systems are moved to other hardware and essential systems are restored using the warm computing hardware. Restoring the essential systems may take as little as a few hours. If your organization uses a public cloud, a warm secondary location would be to use computing resources in a different availability zone or region within your cloud provider's infrastructure, or to use a different cloud provider.

Hot secondary location: A hot location has all the features of a warm location, with the addition that the most current versions of software systems and infrastructure services are loaded and executing. However, a hot location will not have the most current system data. Organizations sometimes partition systems and data to improve system performance. For example, each geographic region (e.g., North America, Asia, Europe) uses an instance of a sales system that runs in a data center within that region, and only summary sales data are shared between regions. In case of a disaster affecting the data center in one region, one of the other regional data centers can serve as a hot secondary location. Restoring service at a hot location depends on only the time needed to securely restore system data.

Mirrored locations: Mirrored locations, as the name implies, have identical software and data at two or more data centers. In this configuration, the RTO and RPO can be truly "0." Before the use of cloud computing and distributed database technology, deploying mirrored locations was expensive for an organization and was used only for organizations that required 24/7 availability. Today's technology largely eliminates the cost barrier. Unlike the other three approaches to disaster recovery, this approach requires that the system be designed from the start to support mirrored data. This can impact performance, as represented in the PACELC[70] tradeoffs, and this approach generally increases system design complexity, as the system must handle (non-disaster) faults in communication

[70] See Daniel Abadi's work, summarized in https://en.wikipedia.org/wiki/PACELC_theorem.

between the mirrored sites. However, for many systems where low RTO and RPO are needed, cloud computing makes this approach a viable alternative.

How can we map the concept of “location” if your organization is using a commercial cloud? As we discussed in Chapter 4 The Cloud, most large cloud service providers divide their infrastructure into *regions*. A region is then divided into *availability zones*. An availability zone is a logical data center (which may, in fact, comprise several physical buildings at nearby locations), with separate power, cooling, and internet connections. The probability of two availability zones within a region failing simultaneously is very low. For the warm, hot, and mirrored approaches, your primary location would be one availability zone and your secondary location could be another availability zone within the same region. Cloud providers may automatically assign availability zones when allocating resources, but for disaster recovery purposes, you should explicitly select the availability zones for your software and data.

If your organization is not using a commercial cloud for your primary operating location, an availability zone in a cloud could provide a cold or warm secondary location. For a cold location, your organization would choose a cloud provider and establish the organizational and individual accounts that would be needed to restore operations in the cloud. This would incur little or no cost except for failover testing (as discussed below). At a warm location, you would take the additional step of copying your data and software to cloud storage, which would incur storage costs but no execution costs except for testing.

Having decided on an approach for a secondary location, you now must provide the software for your system and your infrastructure services and the data for your system. We’ll first consider the issues involved with providing the data, beginning with systems in Tiers 2-4 where the data is backed up periodically and not kept on site. After that, we will discuss restoring software at the secondary location.

15.4 Data Management

In this section we discuss the management of the data for the various tiers.

15.4.1 Strategies for Tier 2-4 Systems

When a disaster occurs, you must assume that all the data at the affected data center is lost. To have access to that data for recovery, you must have made a backup copy of that data. The frequency of the backups is determined by the RPO.

If we assume the notional RPO targets that we discussed above for each tier, then Tier 4 systems should be backed up once a day, Tier 3 systems backed up every 4 hours, and Tier 2 systems every two hours.

You next need to consider the storage media you will use for your backups and the location where the backup media will be stored. These will be influenced by your answers to two questions:

1. *Is online storage at your secondary location always available (i.e., the storage is operating and is accessible over the network from your primary data center)?* This should be the case if you choose the warm or hot approaches discussed above. In that case, the data could be replicated at that data center. Replication is a one-way copy from disk storage in one data center to disk storage in another data center. If the secondary data center location is not always available, your only option is to back up the data to tape.
2. *What is the volume of data that must be backed up for your system?* Moving large amounts of data over the internet is a slow process. Network-transfer rates over a wide area network (WAN) between data centers will vary based on how much bandwidth you purchase from your provider, but 150 megabits per second is a typical effective speed.[71] Recalling that there are 8 bits in a byte means that the effective speed can be measured in kilobytes per second. You now see the rationale for the old joke, "What is the fastest way to get a terabyte of data from New York City to San Francisco? Pick an airline that has a nonstop flight and send a tape."

Your media options are either tape or disk, and the choice will be dictated by the location of the secondary data center, the amount of data to be backed up, and the time required for the data to be replicated at the secondary data center. With these three values, you can make your backup media choice.

When you backup your laptop computer, you probably choose to perform *incremental backups*. An incremental backup of a file system copies only the files that have changed since the last backup, which takes much less time than copying the entire file system. While this approach works well for your laptop, the systems

[71] When transferring data from one region to another within a single cloud provider's infrastructure, the data stays within the cloud provider's network, and you may see better performance than the internet speeds referenced here.

that you develop often have a small number of very large files (such as a relational or NoSQL database), and these files are constantly changing. In this case, an incremental backup of the file system provides no advantage. However, databases usually create operation logs, which record the changes to the database files. These logs, which are much smaller than the complete database, can be backed up. Restoring the database after a disaster becomes a process of restoring the operation-log files and applying the operations that modify the database. Like any incremental approach, there will reach a point when the time needed to restore and apply the operation logs takes as much time as to perform a full database restore. You generally use a combination of occasional snapshot full backups with more frequent incremental operation log backups.

A couple of final notes regarding removable media (either tape or disk) for backups: First, after writing the backup copy, the media must be promptly removed from your location. Performing hourly backups but letting the media sit in a cabinet until the messenger comes at the end of the day will not deliver your desired RPO if the disaster occurs as the messenger is pulling into the parking lot. Second, because the media is going to leave the physical security of your location, all backups should be strongly encrypted. Misplaced or stolen backup media are a common cause of data breaches. Backup encryption introduces the need for key management as part of your disaster recovery planning.

15.4.2 Tier 1 Data Management

For systems with stringent RPOs, the situation is entirely different. The data management approaches we discussed above for Tier 2-4 systems rely on data backups that occur outside the scope of your system. For Tier 1 systems, your data management strategy will depend on the type of data and processing that your system performs, but it almost every case, the strategy you choose will affect the design of your system and services. We begin by discussing transactional data where the RPO requirement is that no transactions be lost.

To achieve this RPO, you must keep an up-to-date copy of the system data at the secondary data center location. The replication can be bidirectional (often called a *master-master* configuration), where either data set can be updated and the update is propagated to the other copy of the data set. The mirrored secondary location approach discussed above relies on bidirectional data replication, which requires special techniques to prevent inconsistency between the data at the two locations. This is handled within your database (either relational or NoSQL), and as we noted above in the discussion of mirrored secondary locations, there is some performance penalty.

The replication can be one way (often called a *master-follower* configuration) where one data set is updated and the other maintains a copy. This is simpler than bidirectional replication. For some systems, even though your RPO is "0" (no transactions are ever lost), your RTO may not be "0"–it may be acceptable to have a period where your system is not available, and no new transactions are processed while the system is restored. One-way replication can satisfy this requirement with less complexity than bidirectional replication.

Transactional data is just one class of system data. Within a Tier 1 system there may be other types of data.

- *Unreplicated data.* Some data, such as session data, may not be replicated. In this case, a user would be required to log in again in the event of a disaster. Whether this is acceptable for your system is a business decision not a technical one, but if some data can be lost, this would affect the technical solution.
- *Infrequently changed data.* Items such as the static portion of web pages, e-shopping data, videos, and pictures are changed infrequently if at all. Although your system will treat this as data, you can use your configuration management system (discussed in Chapter 10 Basic DevOps Tools. if it is software, use one of the approaches that we discuss below.

15.4.3 Big Data

The data-management strategies we discussed above all assume that all your data is at your primary location, and for Tier 2-4 systems, it is feasible to backup and restore the data set. Some systems will operate on so-called "big data." An informal definition of big data is any data set that is too large to be backed up. Big data is managed in complex, distributed database systems that split the data set into chunks called *shards*. Each shard is replicated to create several copies that are distributed across multiple locations, and distributed coordination mechanisms built into the database system keep all copies of a shard consistent[72] across all locations. The database system is configured so that no data will be lost

[72] The consistency level will depend on the database system you choose and how you configure it. See https://en.wikipedia.org/wiki/Consistency_model.

if there is a disaster that affects one of your locations and may be configured for resilience against multiple location failures.

If your system uses one of these distributed database systems, the database system should be deployed at a hot or mirrored secondary location, so that the data will be accessible when restoring your system at that location. If you use a cold or warm secondary location, you will need to add a new location to a distributed database. Not all database systems support adding a new location, and when supported, the time needed to bring the new location online can be very long (10s of hours) due to the large-scale data movements needed to relocate shards.

15.5 Software at the Secondary Location

The third piece of a disaster recovery plan is to have the appropriate software to run your system at the secondary data center. There are several strategies, and your choice here will depend on the RTO and the time required to install the software. The approaches range from rebuilding the system from source code, copying build packages and assembling them into images, or copying images. For any of these approaches, you should be able to reuse IaC scripts and your configuration management tools. Keep in mind that in the case of a disaster, you will be restoring multiple systems at the secondary site, which may impact the performance your configuration management tools. You should make observations and measurements during your failover testing and consider explicitly orchestrating the restoration process to manage the load on your configuration management tools.

Once again, the system tiers lead you to choose an appropriate strategy.

15.5.1 Tiers 2-4

Any of the strategies identified above are viable to restore the software for Tier 2-4 systems. Approaches that build less software (e.g., copying complete images to the secondary location) have lower recovery time, at the expense of copying more images to the secondary site and then storing those images at the secondary site. To ensure correct behavior and compatibility with the schema of the restored data, the restored software must be identical in version number to the software executing at the primary location. The frequency of software changes at your primary location becomes a factor to consider. If you are changing software hourly and it takes four hours to copy an image to the secondary location, you will not be able to ensure that the secondary location has the correct version if there is a disaster.

If you are using a warm secondary location, you may be able to use the available computing resources to build your system from source at the secondary location whenever your deployment pipeline (see Chapter 11 Deployment Pipeline) deploys to production at the primary location. You would then save that image at the secondary location.

15.5.2 Tier 1

Because your Tier 1 systems will almost always use a hot or mirrored secondary location, the secondary location will be part of your production environment. Every time your deployment pipeline pushes to production, the software at the secondary location will be updated.

A Tier 1 system running in a hot secondary location should execute silently–it should not allow any output to affect the behavior of the primary location or modify the backup database. If you are using an event-based system, then silent execution is handled by not sending events to the system at the secondary location. Once the failover occurs, events are directed to the secondary location and it can respond.

15.5.3 Other Data and Software

The discussion above has focused on the data and software of your system. You (or someone in your organization) also must consider *all* the data and software on which your system depends. This begins with direct dependencies, such as infrastructure services. Some infrastructure services have persistent data that changes frequently, such as your authentication and authorization service. The RPO for that data might be “0.”

Another important set of dependencies is your development toolchain and your deployment pipeline toolchain. If your recovery strategy is to build software from source code, the RTO must include time to restore your development toolchain.

A final set of dependencies that we note is license or entitlement keys. If your system, infrastructure services, or development tools use any commercially licensed software, the software in the secondary data center must have the appropriate license keys.

As you can see, disaster recovery planning is complicated. Large organizations may keep in-house expertise on staff, while smaller organizations rely on consultants to develop and implement plans. Both large and small organizations will probably have their disaster recovery plans audited by risk-management experts.

15.6 Failover

You now have implemented your disaster recovery plan–you have designated a secondary location, and both your data and your software are recoverable if there is a disaster. As we noted above, the switch from operating at your primary data center to a secondary when a disaster occurs is called *failover*.

The failover process has three activities: *trigger* the switch to the secondary location, *activate* the secondary location and restore data and software at the secondary location, and *resume operation* at the secondary location.

What happens in each of these phases, and how do you test that? Those are the topics for this section.

15.6.1 Manual Failover

The first case we consider is that the failover is triggered manually, with a human deciding and acting to switch operation to the secondary location. The decision to initiate failover has business consequences. For some systems, once the failover process begins, the system is unavailable at both the primary and secondary sites until restoration is completed. For these and other reasons, many organizations leave it to a person to assess the magnitude and impact of the event and to manually trigger failover to begin.

Activating the data center at the secondary location and restoring the software and data should be scripted so that it can be activated by a single command or button press. For systems failing over to a cold secondary location, there is a relatively long process to acquire and install hardware and activate network connections before even thinking about restoring software and data, while at a warm secondary location, the software can be immediately loaded and started, and the backup data moved online. If your system is using a hot or mirrored secondary location, the software is already executing and the data should be up to date.

To complete the activation of the secondary location, you must redirect user requests from the primary location to the secondary location. This usually is done by changing the discovery service configuration. Until this completes, user requests will be sent to your (now failed) primary data center. If possible, a message “temporarily out of service” should be posted on the former primary data center to inform users of the problem. Posting a message will not always be possible depending on the speed with which the disaster hits. A final caution, although it may be obvious: the script that activates the secondary data center should not be executed on the former primary data center.

Having completed activation and restoration, your system can resume operation at the secondary location.

15.6.2 Automatic Failover

As you will see in this section, automatic failover is complicated, and there is a small risk of triggering a failover unnecessarily. For those reasons, automatic failover is used for systems that have a *very* short RTO, where the time needed for human decision and action is too long. For these systems, you will be using a hot or mirrored secondary location, with software executing and data online at both the primary data center and the secondary location. A falsely triggered failover, where the primary data center continues to operate after the secondary location has been activated, may result in data inconsistency between the two locations. To automate the failover trigger, an infrastructure system that is monitoring the primary data center must detect a failure. Robust detection of failure in a distributed system is one of the eternal computer science problems–you will find research published on this topic going back to the 1980s. Recall our discussion in Chapter 12 Design Options about the difficulties of distinguishing a service providing slow responses from one that has failed.

Your failure detector should not generate false-positive responses – it should not decide that there has been a failure if the primary data center has not actually failed. A false positive might cause data inconsistency since there will be two copies of the database both operating as the master. If your RPO allows you to use a slow replication approach, the database at the secondary location may also lag the primary location. What are the causes of a false-positive decision?

- *The data center did not fail, just a VM in the primary data center.* Your system is executing on one or more VMs in the primary data center. You are monitoring the health of these VMs from the secondary data center and your monitor observes a lack of response from the VMs. This lack of response could be caused by a disaster in the primary data center, but it could also be caused by failure of the VMs on which your system is executing. In this latter case, it is better just to initiate another VM in the primary data center rather than fail over to the secondary data center.

- *Neither the VMs nor the primary data center failed but the VMs were slow to respond.* This could be caused by other problems such as an overload or slow response from a dependent service.
- *The network between the two data centers failed or was congested*. In this case, the health check will not succeed but failure of the primary data center is not the cause.

For these reasons, automatic failover should be used only when needed to satisfy a stringent RTO. The risk in allowing data inconsistency in the case of a false-positive failover trigger can be ameliorated by using a distributed coordination system such as we described in Section 4.7 Sharing Distributed Data. Database write operations can be buffered in a distributed coordination system until both copies of the database system acknowledge the operation. This will slow down the write operations, since the distributed coordination system will not be as responsive as one maintained strictly within a single data center, but it will prevent inconsistency of the databases.

To summarize this section, automatic failover should be used only in cases where the RTO is less than the human reaction time, and special care should be taken to avoid inconsistency between the database in the primary data center and the secondary data center.

15.6.3 Testing the Failover Process

Like all your software artifacts, you should test the script that orchestrates activation and restoration during failover. At a minimum, this testing should be done when there are changes to your production environment and when there are changes to your secondary location. You should also test this periodically as a training and practice drill for the people involved, and because there are probably unidentified dependencies on software and environment that may have changed and broken the script.

Unlike many of your other software artifacts, testing this script is difficult unless you allow all clients to actually failover to the secondary location, and accept the downtime and data loss allowed by your RTO and RPO. Since that is not usually allowed, you will have to address at least three challenges in testing this script.

1. *You do not want to interrupt service to your clients.* If your organization can afford scheduled down time, the testing might be accomplished then. Before performing any testing, be sure that the production database has been backed up in case it accidently gets corrupted and allocate enough test time to restore the production data if the test fails. If your organization cannot afford scheduled down time, one option is to test the failover using your staging environment. To the extent that your staging environment matches your production environment, you can generalize the test results from staging to production.
2. *The test fails and the production database becomes corrupted.* This can happen if the failure allows writing to the database at the secondary location and then that database synchronizes with the production database. You could configure the database at the secondary to be read-only for the duration of the test if your systems can run successfully with the database in this mode. Alternatively, if your database is small enough to be easily copied, you can use a separate copy of the backup database for the failover testing.
3. *Your clients receive messages from the secondary location.* While this is the desired behavior if there was a real disaster, these messages should not escape during a test.

15.7 Summary

Your organization should be able to resume operating after a disaster. Achieving this takes planning. The planning is driven by the recovery point objective (RPO) and recovery time objective (RTO). These values will vary for different systems and can be used to prioritize your systems into tiers. Tier 1 systems have the smallest values of RPO and RTO and take the most careful planning to restore.

You will require a data center at a secondary location and a means of backing up your production database and production software and restoring both to this secondary data center. Configuration management systems are useful in ensuring that the software is consistent between the two data centers and your database-management system, possibly utilizing a distributed coordination service, can be used to keep the backup database synchronized with the primary database.

The failover process should be performed by a script that is triggered either manually or automatically. Regardless of how the script is triggered, testing it can be difficult because of the possibility of interfering with production operations, corrupting the production database, or having clients receive messages during the testing process.

15.8 Exercises

1. Install MySQL to replicate across availability zones in a master-follower fashion. What is the latency of the synchronization?
2. Write a script to make the current follower be the master and the current master be the follower.
3. Change the DNS server so that requests are sent to the secondary data center.

15.9 Discussion Questions

1. Find out where your organization has its primary data center and determine the extent of recent disasters in that area that affected the data center.
2. There is an inconsistency between our definition of RPO and Figure 15.1. What is that inconsistency?

16 Thoughts on the Future

In this chapter we will make some predictions about the future of DevOps. We will discuss how to convince your organization to adopt DevOps practices, DevOps evolution, and keeping current with changes in the field

16.1 Coming to Terms

Machine learning– a branch of artificial intelligence and computer science which focuses on the use of historical data and algorithms to discover patterns in apparently unrelated current data.

Pilot project- mini-version of a DevOps project that tests the viability of executing DevOps at full scale

16.2 Transitioning your organization to DevOps

DevOps practices continue to be adopted by more organizations. This means that you may be asked to develop a plan to help your organization adopt DevOps. Some ideas to help you with your plan:

1. The four DORA metrics we discussed in 9ection 9.7 Metrics are important. These are lead time for changes, change failure rate, deployment frequency, and mean time to recovery. You will need to a) know the current values of these metrics and, b) devise targets you would like the organization to achieve.
2. You will need to develop cost estimates for the transition to DevOps practices. Cost estimates together with the improvements in the metrics will enable your management to determine whether they wish to invest in adopting DevOps.
3. To execute step 2) you should organize a pilot project. A pilot project is a limited time and effort project where a team uses DevOps practices. The participants in the project should be thought leaders for the organization who are enthusiastic about the effort. Having thought leaders involved in the project will help others in the organization accept the transition to whatever DevOps practices are to be adopted.

4. You will need a budget for the pilot project. Put together a pitch for your management that includes activities from other organizations in your field, potential benefits, potential pilot project, what you expect to learn from the pilot, and the time and cost for the pilot.

16.3 Evolution of DevOps

There are two trends that will shape the immediate future of DevOps. The evolution of markets, tools and vendors and the application of DevOps to domain specific problem areas.

16.3.1 Market, Tool, and Vendor Evolution

Currently there are an incredible number of DevOps tool vendors. They cannot all survive. Vendors will consolidate and tools will disappear. This brings us back to the problem of vendor lock in. If your organization is locked into a tool that drops out of the marketplace, it will have to proceed with an expensive retooling effort.

You should expect the growth of de facto standard tools and actual standards. This trend will take time to mature since standards efforts take years to produce a standard and de facto standards will require consolidation in the marketplace.

An associated trend is the growth of multi-vendor cloud environments. The multi-cloud environment market is estimated to reach around $30 billion by 2028. Your organization will likely wish to reduce the risk of depending on a single vendor by utilizing multiple cloud vendors.

Some tools are cloud provider agnostic. Kubernetes and Terraform are two examples. Cloud agnostic tools have the virtue of being usable on different cloud providers. They have the disadvantage of not being able to take advantage of cloud provider specific features.

A different trend to expect is the growth of end-to-end deployment pipeline tools. Currently, specialized tools exist for different portions of the DevOps process cycle we showed in Figure 9.2. Expect to see tool chains from a single vendor that are integrated and that cover all or much of the cycle.

Although no examples yet exist, once there are single vendor end-to-end tool chains, expect to see the emergence of tool language translators that translate from one vendor’s tool chain to another.

Successful open-source tools tend to have a longer lifetime than proprietary tools. These tools, typically, are supported by one or more companies that provide their employees to support the tool. These companies make revenue by selling support for the tool to organizations that use the tool. Many popular DevOps tools – Docker, Kubernetes, Jenkins, Vault, Istio, Ansible, Chef, Terraform – are open-source. Using these tools will help your organization avoid vendor lock in.

16.3.2 Domain Specific Problem Areas

Another trend that will gain prominence in the future is the growth of domain specific problem areas. These problem areas will generate new tools and new DevOps processes. Three such domain specific problem areas already exist.

1. DevOps for Machine Learning. Systems that utilize Machine Learning have two distinct pipelines. One is for the code than makes up the system and the other is for the data used for the predictions from the system. The data will need to be changed, tested, and refreshed as the system is used.
2. DevOps for government or other organizations that contract their software development to external organizations. For these types of organizations to incorporate DevOps practices and tools, they will need to make organization wide decisions and incorporate these decisions into the RFPs and contracts by which the sub-contractors are governed.
3. Database DevOps. Maintain version control and roll-back capability for the data in a database.

16.4 Keeping up

Every decade or so, the computer industry undergoes a radical transformation. The 1970s saw the introduction of networks, the 1980s the introduction of personal computers, the 1990s were when the world wide web was introduced, the 2000s saw the cloud, and the 2010s, DevOps. In the 2020s, quantum computing is a possible transformative technology.

Although you are probably not thinking long term at this point, it is likely that you will spend decades in the computer business. The implication is that you will need to keep up with emerging technologies. You do not need to become an expert in each new technology but rather just be conversant with the technology and put some thought into how that technology might impact your career.

One tool to determine emerging technologies and their maturity is the Gartner Hype Cycle. The hype cycle represents the maturity, adoption, and social system of specific technologies. Figure 16.1 shows the stages of the hype cycle.

1. Technology trigger. The technology has gotten sufficient recognition that Gartner judges it is worth tracking.

2. Peak of Inflated Expectations. Enough success stories involving the technology has its advocates making extravagant claims for it.

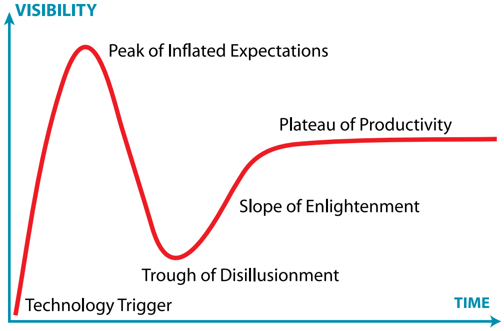
3. Trough of Disillusionment. A sufficient number of failures involving the technology lead to bad publicity and negative reactions to the technology.

4. Slope of Enlightenment. More instances of how the technology can benefit the enterprise start to crystallize and become more widely understood.

5. Plateau of Productivity. Mainstream adoption starts to take off. Criteria for assessing provider viability are more clearly defined. The technology's broad market applicability and relevance are clearly paying off.

Figure 16.1: Gartner Hype Cycle[73]

6.



The Gartner Hype Cycle is one means for discovering and evaluating new technology. There are also newsletters and blogs that you can subscribe to that provide updates and opinions on emerging technologies. You should not spend an excessive amount of time researching new technologies, after all you have a job that you must do. Once a week or every couple of weeks ought to be adequate unless a technology emerges that is of specific interest to you.

[73] By Jeremykemp at English Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=10547051

We hope you have enjoyed this book and wish you the best of luck with your career however it may evolve.

Authors

Len Bass is an award-winning author who has lectured widely around the world. His books on software architecture are standards. Len has over 50 years' experience in software development, 25 of those at the Software Engineering Institute of Carnegie Mellon. He also worked for three years at NICTA in Australia. He has worked on domains ranging from operating systems to database systems to user interface systems to automotive systems to financial systems to defense systems to the electric grid. He is currently an adjunct faculty member at Carnegie Mellon University, where he teaches a course in DevOps.

John Klein is a Principal Member of Technical Staff at the Carnegie Mellon University Software Engineering Institute (SEI), where he does research and consulting in scalable system architecture, working with commercial and government customers in domains that include analytics, scientific computing, financial services, and command and control. He has worked as an architect and manager in companies ranging from Fortune 100 to pre-IPO startups, in businesses that include telecommunications, video networking, and defense.