Infrastructure for Software Engineers

Len Bass

Table of Contents

==================end of TOC=====================

Preface

The widespread adoption of the cloud has fundamentally expanded the skill set required by software engineers. Historically, most software engineers needed to know several programming languages and understand the elements of a three tiered architecture. State management was important but not predominant. Items such as networks, distribution, and security were the realm of specialists. Deployments were done by operations staff and incidents were handled through a well-defined escalation procedure.

Today, a software engineer is expected to be responsible for their portion of the system from “cradle to grave”. This, together with the architecture of the cloud, requires an engineer to understand state management in a more fundamental fashion, requires them to understand IP addresses, load balancers, scaling, and ports, among other topics. They also must understand the deployment process and operations in terms of 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 organizations’ 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 either early in their career or are juniors/seniors in Computer Science or Software Engineering. It provides the context within which modern engineers work and the skills necessary to operate within that context. 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 done in a classroom setting or in discussion groups to enable the students to gain a better understanding of the theory and its implications.

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. 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 more quickly come up to speed and become productive in an organizational setting.

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.

========================end of preface==================================

Reader's Guide

We have structured this book into four distinct portions. This draft contains the material for Part 1.

Part 1 introduces the platform that is used in most modern computing environments.

> Chapter 1 discusses virtualization. It introduces the concepts of virtual machines and containers. It also discusses how virtual machines are provisioned.
>
> Chapter 2 discusses networks. IP addresses, ports, and DNS servers. It also describes the protocols (IP and TCP) that are commonly used to communicate over networks.
>
> Chapter 3 discusses the cloud. It covers the allocation of VMs, load balancing across instances of the same VMs, autoscaling, failure in the cloud, and coordination in a distributed system. State management is also discussed here since state management is fundamental to the use of containers.
>
> Chapter 4 discusses containers. How they are managed, moving them, repositories, clusters, serverless architectures, and automation.

Part 2 discusses management of the platform described in Part 1. This involves security mechanisms, patching and incident handling.

Part 3 discusses how application developers use the platform. Microservice architectures and the deployment pipeline are the two major topics.

Part 4 discusses managing applications. Configuration parameters, management of credentials, logging, and monitoring are discussed in this part.

========================end of readers guide===========================

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 your assignment is to 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. Not too bad. 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 you service, our Docker repository is called organization_docker_hub." Huh? If these instructions are bewildering then Part 1 is for you. Part 1 describes the cloud as a platform for development.

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 in this environment.

A cloud data center has tens of thousands of computers communicating over a network and accessible externally via the internet. Any application in the cloud ideally acquires the resources (CPU, network, disk) that it needs only for as long as they are needed. Furthermore, these resources are either acquired automatically and dynamically by the application or reserved in advance by the application developer. The resources are available as virtual machines or containers residing on physical machines. Thus, in Chapter 1 we introduce virtualization. Virtual machines virtualize the hardware. More recently containers have been used to virtualize the operating system. We introduce containers in Chapter 1 and discuss them in more detail in Chapter 4..

The physical and virtual machines in the cloud communicate over networks – both internal and external. Communication over networks is in terms of messages. Two concerns immediately arise when messages are the communication mechanism – how are messages 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 application within that machine. Furthermore, networks have structure and sub-structure. Chapter 2 is a discussion of networks.

With virtualization and networks as a background, Chapter 3 discusses the cloud. The cloud allocates resources, we will see how. When tens of thousands are computers are involved, failure of some of them is inevitable. The possibility of failure permeates the design of applications designed to run on the web. In Chapter 3, we introduce the possibility of failure. This possibility will be elaborated in future chapters. Virtual machines get overloaded and this overloading is managed by having multiple instances of a virtual machine. These instances can be allocated dynamically. Load balancers and autoscaling mange the creation and deletion of multiple instances. Load balancers and multiple clients introduce the problem of state management which we discuss. Finally, time and the passage of time is a problem in a distributed system. Luckily, we now have open source solutions to provide 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 4 we return to containers. Containers are managed by different mechanisms than virtual machines although there is a relation. Containers can be stored in a library, can be scaled similarly to virtual machines, and are used in "serverless architecture". All of these topics are discussed in this chapter.

============================end of introduction to part 1=========================

Chapter 1 – Virtualization

Almost all computing these days is done on virtual machines. Even embedded systems such as automobile 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 and a virtual machine image
- How virtual machines are provisioned
- The distinction between virtualizing hardware and virtualizing operating systems.

Isolation and sharing

In the 1960s, the computing community was frustrated by the problem of sharing resources on one physical machine among several independent applications. The inability to share resources meant that only one application could be run at a time. Computers of that period cost millions of dollars and most applications only used a fraction of the available resources, this was clearly inefficient.

Several mechanisms emerged to deal with sharing. The goal of these mechanisms was to isolate one application from another while still sharing resources. Isolation allows developers to write programs as if they were the only ones using the computer, while sharing resources allows multiple applications to run on the computer at the same time. Since the applications are sharing one physical computer with a fixed set of resources, the isolation cannot be complete. Suppose, for example, one application controls all of the CPU, then the other applications are locked out. For most purposes, however, these two mechanisms, with some modification, have been sufficient.

=============================SIDEBAR===========================

The terminology we will be using throughout this book differentiates the terms *application, client* and *service*.

A service is a coherent collection of functionality. Some services are exposed to end users and others are not.

A client for a service is another service that is a user of that service. A client service has its own clients.

An application provides a collection of services to an end user. The services that are end user facing are clients for other services which are, in turn, clients for still more services.

If you shop on Amazon, for example, the interface you see may have over 100 services embedded in it. Each one of these services is a client for services not available to an end user.

Implicit in these definitions is a distinction between an end user and a client. An end user is a human, possibly an administrator, and a client is a piece of software.

=====================END OF SIDEBAR==========================

We now describe the resources that are shared. Then we will describe the isolation and sharing mechanisms. There are four such resources:

1. The Central Processing Unit (CPU). Modern computers have multiple CPUs (and each CPU can have multiple processing cores)
2. Memory. The physical computer has a fixed amount of physical memory.
3. Disks Disks are used for the storage of instructions and data. A physical computer has an attached disk with a fixed among of storage capacity.
4. A network connection. Historically, networks did not appear until the 1970s but we can ignore that detail. Every non trivial physical computer has a network connection through which all messages pass. These include both outbound messages and inbound messages.

Now we turn to the isolation and sharing mechanisms

1. Processor sharing is achieved through the scheduling mechanism. The scheduler decides which thread is to get an available processor and this thread maintains control until that processor is rescheduled. No other thread can gain control of this processor without going through the

scheduler. Rescheduling occurs either when the thread yields control of the processor or an interrupt occurs.

2. Memory sharing and isolation is achieved through partitioning memory. Figure 1.1 shows a simplified view of the instruction cycle on a physical machine, which ignores pre-fetching, caching, speculative execution, and a number of mechanisms that improve performance while allowing us to pretend that this simple model is what really happens. The next instruction to be executed is fetched from memory, the instruction is decoded and executed. The next instruction to be executed is either a result of the execution of the current instruction or the next sequential instruction in memory. The execution of an instruction may reference memory locations.

   The hardware can be modified to support memory isolation. The operating system assigns each partition a tag. The execution of an instruction fetched from a memory location inside a partition is conditioned by the tag of that partition. Each memory access as a result of the fetched instruction, whether to fetch the next instruction or for the retrieval or storage of other memory locations, is checked to see that the tag associated with the executing instruction matches the tag of the memory location. If the tag of the instruction does not match the tag of the memory location being accessed, then an error interrupt is generated.

3. Disk isolation is achieved by partitioning the disk among the active applications. All accesses to the disk go through a disk controller that is responsible for, among other things, verifying the legality of the access.
4. Network isolation is achieved through the identification of messages. The physical computer controls its own network access and messages from within that computer must go through that controller. Each inbound message includes an IP address that identifies the destination of that message to the physical computer network access. We will go into detail about IP addresses in the next chapter.

This chapter is concerned with virtual machines – which are made possible by the type of sharing we have just discussed. There is another usage of the term virtual machine – the Java Virtual Machine (JVM) – which is not included in this book. From the perspective of this book a JVM is a specialized service executes Java byte code.

Now that we have seen how the resource usage of an application can be isolated from the resource usage of another application, we turn to a discussion of Virtual Machines (VMs).

Virtual Machine

Figure 1.2 shows several VMs residing on a physical computer. The physical computer is called the “host machine” and the VMs are called “guest machines”. Figure 1.2 also shows a hypervisor which is an operating system for the virtual machines.

To put this in the context of our previous discussion of isolation, each VM can be viewed as an application to the hypervisor. The hypervisor and associated hardware enforces the isolation between the VMs.

One issue with virtual machines is the overhead associated with virtualization. That is, how much slower would an application run on a virtual machine than on the native processor. This turns out to be a complicated question since it depends on the particular application and the virtualization technology used. Virtualization technology is improving all of the time but overheads of around 2% have been reported by Microsoft on their Hyper-V virtualization engine.

From the perspective of software inside a VM, it looks just like the software is executing inside of a bare machine. The VM has a CPU, memory, I/O devices, and an internet connection. In Chapter 2, we will see that each VM also has its own IP address.

A VM is booted just as a bare machine is booted. That is, there are built in instructions that read a boot program from a disk, either a removable disk, a hard drive internal to the computer or a drive connected through a network. In the case of a physical computer, the connection to the boot program is made during the powering up process. In the case of the VM, the connection to the boot program is established when the VM is created.

The hypervisor has two main functions: 1) managing the code running in the VM, and 2) managing the VMs themselves. To elaborate:

1. Code inside the VM that does not involve communicating outside the VM is executed as described above through the normal instruction cycle. Code that accesses a virtualized disk or network are 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 interrupt that occurs asynchronously. This interrupt is initially fielded by the hypervisor. Since multiple VMs are operating on a single host machine and each VM may have 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 one 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. 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. They must be created and deleted, among other things. Managing VMs is a function of the hypervisor. The hypervisor does not decide on its own to create or destroy VMs. It acts on instructions from a user or, more frequently, from a cloud superstructure. We will explore how this works in Chapter 3, 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 deleting VMs, the hypervisor monitors them. Health checks and resource usage are a portion of the monitoring. The hypervisor is also in the defensive perimeter of the VMs with respect to attacks. Performance of the hypervisor is also a concern since all external interactions of the VMs go through the hypervisor.

All of these concerns mean that the hypervisor is a complicated piece of software.

VM images

Just as a physical computer without software just sits there consuming power, a virtual machine without software is not very useful. A physical computer gets software loaded into it through accessing directly a built in hard disk, an external disk or the network. The same is true of a virtual machine except that instead of direct access, it accesses its potential sources of software through the hypervisor.

The first thing to realize is that the software to be loaded into the computer (real or virtual) exists as a collection of bits on some media. These bits are read either by a bootstrap program or by the hypervisor and loaded into the memory of the computer. Control is then transferred to a known location within the computer where the computer begins its instruction cycle based on the instruction situated at this known location.

Let us approach the question of where these bits come to be in a backwards fashion. That is, suppose we have an existing computer (again real or virtual) that has been loaded with a collection of software. That is, the bits of our computer contain instructions and data that make up the software. Then we can write these bits to a media just as we write any other set of data. If it is a virtual machine then this writing is done by the hypervisor. This enables us to save the contents of a computer and use that content to boot another computer. The collection of bits that came from writing the contents of an existing VM is called a VM image.

So now the question of how these VM images came to be becomes a question of how the creator of an image bootstrapped software in a computer from pre-existing images. For the creators of new hardware, the bootstrapping is done by hand coding machine instructions. For everyone else, there are libraries of common VM images that you can download.

Thus, the process of creating a new virtual machine with, for example, the latest version of Ubuntu consists of finding the Ubuntu download web site, choosing the version you wish to download, copying this version to a known location on your host machine or on the internet, and pointing the hypervisor to this known location when it creates a new virtual machine.

This same process is used not only for operating systems but also for software that you are creating. You can create a virtual machine image as a portion of your development process and then use this image when you create production copies of your software. Copies based on the same VM image are called *instances* of that VM. As we will see when we discuss the cloud, a portion of the startup process for a VM in the cloud is to specify where the VM image for the new VM is to be found.

The key to this whole process is the realization that a set of bits can be a set of instructions or a set of data depending on interpretation. That is, the instruction cycle retrieves a set of bits that it interprets as an instruction but that same set of bits from the same location can be data if you wish to write it out. Thus, a VM image can be read or written as data or can be interpreted by a computer as a set of instruction and associated data.

We summarize this process in Figure 1.3 which shows a disk file being generated from one VM and subsequently loaded by the hypervisor into another VM.

Provisioning

We have discussed loading a VM image but not really discussed how a VM is loaded with a variety of software. Assume that you have loaded the kernel of an operating system into your VM. For specificity we will use Ubuntu as our example but there are many different operating systems and variants of these operating system. Ubuntu is one variant of Linux which is itself a variant of Unix. Ubuntu follows a six month release cycle but in between releases are some number of maintenance releases. An Ubuntu release is numbered XX.YY.ZZ where XX is the year of the release, YY is the month of the release, and ZZ is the number of this maintenance release.

Loading a kernel will give you a command line interpreter (CLI) that allows you to issue Linux commands. Now assume you wish to build a VM with the LAMP stack (Linux, Apache httpd server, MySQL, and PhP). Given the kernel, the next step is to load Apache. One technique is to go to the Apache web site, find a binary for your operating system and download it using a command such as wget. You do not yet have a browser on your OS so you will need to find the appropriate web site using an external browser but then you can issue the wget command accessing this web site. There are several problems with this approach:

- The release you are downloading may not be compatible with your version of the OS.
- If you are updating a prior release of the Apache Server, the old version should be removed from your system.
- The Apache Server may depend on other software which you have not yet downloaded.

Enter a package management system. Your operating system will come with a package management system. For Ubuntu, it is called apt-get. This package management system accesses a repository in which compatible versions of the Apache Server are maintained. Issuing the command “apt-get install apache2” will download the latest version of Apache 2 that is compatible with your version of the operating system. It will also download any software on which the Apache Server depends. Issuing the command “apt-get update” will download the latest version of any of the packages managed by apt-get including the Apache Server. *Caution* – issuing the commands as we have included them will not work. There are several subtleties that you must worry about but we defer these to an exercise.

Just as there are multiple variants of Linux, there are multiple different package managers. You, or your organization, will choose one as the one you use. Just as moving between two different variants of Linux should only be done with awareness of why you are doing it, moving between two different package managers should only be done with awareness.

The next improvement in provisioning a virtual machine is to automate the provisioning. This could be done by scripting the apt-get commands into a file and executing that file or it could be done, more flexibly, with a specialized provisioning tool. Vagrant is an example of a provisioning tool. It takes a Vagrantfile as input and provisions a VM according to the instructions it finds in the Vagrantfile. Figure 1.4 shows Vagrant provisioning a VM.

The virtues of using a provisioning tool such as Vagrant are

- Vagrantfiles can be versioned controlled so they can be shared and retrieved, if necessary.
- Every member of the team can provision their systems with the same software. This will pay benefits when integrating.
- The scripting language used in Vagrantfiles is specific to VMs and is more flexible than a script written in the OS command language.

Containers

VMs solve the problem of sharing resources and maintaining isolation that was posed in the 1960s. However, shipping VMs around the network is time consuming. Suppose you have an 8 GB(yte) VM. You wish to move this from one location on the network to another. Assuming your network is 1 GB(it) per second this will take 64 seconds. In practice, a 1 GB network operates at around 35% efficiency. Thus, transferring an 8GB VM will take around 3 minutes. Now there are efficiencies that can reduce this time but the result is still on the order of minutes.

Containers are a mechanism to maintain the advantages of virtualization without paying such a performance penalty.

Re-examining Figure 1.2, we see that a VM executes on virtualized hardware under the control of the hypervisor. In Figure 1.5 we see several containers operating under the control of a container runtime 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 share the same underlying hardware set, all containers share the same operating system.

This gives us one source of performance improvement. As long as the target machine has a Docker Engine running on it, there is no need to transfer the operating systems. Since modern operating systems are on the order of 1 GB(yte), this saves a substantial amount of time.

The second source of performance improvement is the use of layers in containers. To understand this, we describe how a container image is constructed.

We demonstrate the construction of a container image in layers using the LAMP stack. LAMP is Linux, Apache, MySQL, and PHP. LAMP is a widely used stack for constructing web applications.

The process begins with you creating a container image containing Linux. This image can be downloaded from a library using the container management system. Once you have created the Linux 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 Linux. 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, 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 of the elements of the stack so the time it takes is the time to move the total stack. Now, 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 only moves PHP. This saves the movement of the rest of the stack. Since changing a software component happens much more frequently than its initial 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 milliseconds.

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. By version controlling the specification file, each member of your team can create an identical container image and modify the specification file as needed.

Figure 1.5 shows a container image with the LAMP stack where the various pieces are separated to indicate they were created as layers and can be loaded independently.

One final point is that containers can be embedded inside of VMs. Thus, when you deploy a container image, it is into a VM hosted on a physical machine. Figure 1.6 shows this hierarchy of physical machine, VM, container.

=========================SIDEBAR=========================

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 Chapter 4 we will discuss container repositories. In fact, a container repository 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.

==================END SIDEBAR=======================

Summary

Hardware virtualization allows the creation of several virtual machines sharing the same physical machine. It does this while enforcing isolation of memory, I/O, network and CPU. 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.

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

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.

Exercises

Remember when you perform these exercises that the internet is your friend. Begin an exercise by searching for “tutorial [subject]”. Someone will likely have written a description of how to use a particular tool or perform a particular operation. If you get an error message when following the tutorial instruction, someone else has likely gotten it and written an explanation. You can find the explanation by pasting the error message into your search engine.

1. Create a virtual manage on your host machine
2. Load an Ubuntu kernel onto the virtual machine.
3. Load LAMP into your virtual machine using apt-get. What are the errors that you made?
4. Load LAMP into your virtual machine using Vagrant. How long did it take to set up a working Vagrantfile?
5. Create a Docker container image with the LAMP stack by using a Dockerfile.

Discussion Questions:

1. Where do you find the Ubuntu kernel? Who provides the funding for the creation of the various releases of the kernel?
2. Enumerate different package managers. What are the differences between them?
3. Two VMs hosted on the same physical computer are isolated but it is still possible for one VM to impact the other VM. How can this happen?
4. Enumerate different provisioning tools. What would cause you to choose one over another?
5. How does the container management system know that only one layer has been changed so that it only needs to transport one layer?

===============================end of chapter 1==================================

Chapter 2 – 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 protocols. You will also be familiar with ports and TCP/IP. The domain name system (DNS) is how you find an IP address given a URL. Other networking topics discussed are subnets and firewalls as well as tunneling through firewalls.

We begin with IP (Internet Protocol) addresses.

IP addresses

Every device on the internet, both real and virtual, is assigned an IP address. This address enables messages to be sent to that device. It also allows for the device to be physically located – with some restrictions.

IPV4 was the first widely used version of the Internet Protocol addressing scheme. It uses 32 bits to represent an address. This is typically formatted as XXX.YYY.ZZZ.QQQ. Web sites such as whatismyip.com will report the IP address of the computer you use to access it. 32 bits allows somewhat over 4 billion addresses. This is not enough for the modern world and all of the available IPV4 addresses have been assigned.

Because of the foreseen exhaustion of IPV4 addresses, a different IP addressing scheme, IPV6, was approved in 1996. 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. Both addressing schemes use similar routing strategies and these strategies are based on how the addresses are assigned.

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.

ICANN allocates numbers through a hierarchy down to your local ISP (Internet Service Provider). Your ISP can be one of the major telephony carriers such as AT&T or Verizon, an industry such as IBM, a university such as Carnegie-Mellon University or a smaller independent provider. Each ISP has a block of numbers that it can allocate to devices connected to it.

When you connect a physical device to a network, you are connecting it to an ISP. The ISP to which it is connected assigns it an IP address. When you create a new VM, it is also assigned an IP address by the hypervisor. Depending on the attributes of the VM IP address, the hypervisor may generate the IP address or interact with the ISP to acquire an address.

Two attributes of the IP address deserve special attention.

- Static or Dynamic

  Your ISP can give your device an IP number for permanent use or with every boot. Giving a permanent number allows those devices that send messages to your device to save the IP number and not have to rediscover it. This is useful for routers, servers and other portions of the hardware infrastructure but is not common for devices running applications. Those devices are given an IP address with each boot. Having dynamic IP addresses allows your ISP to more efficiently use the IP addresses that it has to allocate.

  DHCP (Dynamic Host Communication Protocol) is the protocol most commonly used to manage dynamic IP addresses. When you boot your device, it broadcasts a message to its local network and that message is understood by the DHCP server on the local network. Other messages follow to establish your device's presence on the local network.

  This process of "discovery" whereby a message is broadcast to a local network and interpreted in such a way as to allow further specific communication is used in multiple fashions in the network and we will see further uses of it.

- Public or Private

  The IP number your device is assigned can be kept private to your network or can be known publically throughout the internet. Keeping the IP number private to a local network allows the same IP number to be used in multiple different networks. This is one technique that is used to make the IPV4 addresses extend beyond their 32 bit limit. It also allows organizations to separate their internal devices from the broader internet and place firewalls intended to ensure security and privacy at their perimeters. We will return to firewalls later in this chapter.

  A division of ICANN has designated the following IP numbers 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 only accessible from within its local network.

One other set of IP numbers are 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 reserved address most commonly used. When your local network router sees this address as a destination, it will send the message back to the originating device. 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.

When a VM is created, it can be created with a public (persistent) IP or with a private IP. If it is created with a private IP, the hypervisor manages the state of that IP assignment. That is, if the VM is paused or otherwise interrupted, the IP address may be removed and re-assigned.

When a container is created, it also can be created with a public or private IP address. In this case, it is the container runtime that manages the state of the IP assignment, with the assistance of the hypervisor.

Message Delivery

Messages are actually delivered from one device with an IP address to another device through a network of computers called “routers”. We do not need to 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 rats chewing on the wires. This results in incorrect messages being delivered or messages being dropped. We will see compensation mechanisms 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 has to recognize that it may already have received 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.

If the destination of the message is a VM, then the message is routed through the hypervisor to the appropriate VM. If the destination is a container, then the message is routed through hypervisor to the container runtime in the appropriate VM and then to the container.

IP Protocol

The IP address is used to target messages to devices but interpreting the messages requires that there be agreement between the sender and recipient about the formatting and meaning of the various bits in the message. This is the purpose of the IP protocol. It specifies that a message consists of two portions: the header and the payload. The header specifies such things as the sender and the recipient and provides instructions to the router that pass the message along. The format of the payload is a matter for the sender and recipient to agree on.

As you might expect, the headers for IPV4 and IPV6 differ. 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 data). The minimum value for the header is 20 bytes. The maximum value is 60 bytes.
- Identification. This is a field used for the identification of this message. We mentioned that messages might be delivered through multiple simultaneous routes. What this means is that 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 that require real-time data streaming and therefore make use of the DSCP field. An example is Voice over IP (VoIP), which is used for interactive data voice exchange.
- Total Length. The minimum size is 20 bytes (header without data) and the maximum is 65,535 bytes.
- Fragmentation information. The maximum packet length in IPv4 is 65.535 bytes. This is not enough for some messages so IPV4 has a number of fields that allow for fragmenting a message into pieces and then recovering the original message from the, possibly reordered, fragments.
- 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 data portion of the IP datagram. 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 for error-checking of the header. When a packet 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 packet. Errors in the data field must be handled by the encapsulated protocol. When a packet arrives at a router, the router decreases the TTL field. Consequently, the router must calculate a new checksum.
- Source address. This field is the 32 bit IPv4 address of the sender of the packet. Note that this address may be changed in transit by a network address translation device.
- Destination address. This field is the 32 bit IPv4 address of the receiver of the packet. As with the source address, this may be changed in transit by a network address translation device.

Some observations

- Note the limited size of the fields. This makes sense since IPV4 was defined in an era where communications over networks were slow and, hence, smaller packets were important. These size limitations, however, are why IPV4 became outmoded and is being replaced by IPV6.

- A second observation is the Time to Live field. This is intended to prevent messages from circulating forever in the network. For the Internet Protocols both V4 and V6, this is the number of routers a message goes through. DNS records have a field called Time to Live which has totally different meaning and use. We will revisit TTL when we see DNS but the overloading of the name is confusing.
- Note also the allowance for error. The 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 similar to the Differentiated Services Code of IPV4.
- Flow label. IVP6 does not support fragmented data payloads and this label is used to tell routers to keep packets with the same label on one path so that they will not get out of order.
- Payload length. The default payload limit is 64KB although packets of up to 1 Gbyte are allowed. The default payload maximum is the same as the maximum of IPV4.
- Next Header. This specifies where in the packet the payload begins.
- Hop Limit. As in IPV4
- Source address. The 128 bit address of the device that generated the packet. As with IVP4 this may be modified many times during the life of a packet.
- Destination address. The 128 bit address of the intended destination. As with IPV4 this may be modified during the life of the packet.

Some differences between IPV4 and IPV6 are:

- The size of the address space. 32 bits versus 128 bits.
- The IPV4 concept of reserved addresses (local host) is called a “unique local address” in IPV6.
- There is no IPV6 concept of private addresses that can be multiply assigned behind gateways or proxies as there is in IPV4.
- 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 was standardized in 1996 and as of the end of 2017, less than 25% of internet traffic was IPV6. Because of the difference in address size, all of the routers and transmission computer have to be modified to be IPV6 compatible. In addition, the use of private IP addresses extended the life of IPV4 by decades.

The payload of the IP Protocol whether V4 or V6 will likely include other protocols. We will see this with TCP which is embedded into the payload of the IP Protocol. We will also see further nesting when we discuss tunneling. The ultimate content payload may be nested two or three layers deep into the IP Protocol payload.

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

DNS

IP addresses are not suitable for most end use applications. First, they are hard to remember, especially IPV6. Secondly, and more importantly, they are not persistent. We have seen how IP addresses for a device may change from boot to boot. 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 applications. This is the purpose of domains and the Domain Name System (DNS). You can think of the DNS as a table of names vs IP addresses. We show a simplified view of DNS in Figure 2.3. Names in this case are Uniform Resource Locators (URLs). Your application or browser sends a URL to the DNS and gets back an IP address. This IP address is then used to send messages to the application 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 2.3. 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 replicated and distributed computers scattered around the world.

URL Structure

URLs are the names of the various servers that you may wish to access. URLs have a structure with the various elements separated by ".”s. Thus, [insert publisher's URL. For purposes of this discussion, we assume this is www.myppublisher.com] is a web site where you could order this book. It ends in .com. There is a portion of the DNS that knows all of the .com domains. This knowledge is stored in the .com DNS servers. So to find the IP address, you ask the .com DNS server for the address of "mypublisher".

If you are paying attention, you realize that we are jumping the gun. How do we know the address of a ".com" DNS server? This gets us back to the Internet Assigned Numbers Authority (IANA). IANA is a division of ICANN. IANA maintains a list of "root servers" and their IP addresses. There are currently 13 such servers. These root servers maintain the IP address of the servers for the Top Level Domains (TLDs). .com is a TLD, as is .edu, .org, and others. The list of root servers is published by IANA and is built into many applications, including your browser.

Thus, the process for finding mypulisher.com is to ask the root server for the address of the .com DNS server and then ask the .com DNS server for the address of the mypublisher DNS server and then ask that server for the address of www. This yields the address of www.mypublisher.com.

Time to Live

If your browser accessed the root server every time you made a web request, the root server would get overloaded. So each server in the DNS hierarchy has an associated Time to Live (TTL). This is different from the TTL in the IP Protocols. A 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 browser can cache the IP address of a TLD and assume that it is valid for 24 hours. If the address of the TLD changes, it would take 24 hours for that change to propagate throughout the internet.

Different servers in the DNS hierarchy may set the TTLs to different values, typically smaller as you get closer to the actual server that you are trying to access. For reasons that we will see, it is advantageous to set the TTL for some servers to be a small value such as 5 minutes or less.

Overload and failure

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

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

Computers also get overloaded. One technique for managing overloading is to use DNS based load balancing. Rather than the DNS server returning a single IP in response to a URL request, it returns a list. The list consists of a number of distinct computers that are running the application you are trying to reach with the URL. Then you would try the first one on the list, if that doesn't respond, you would try the second and so forth. The DNS server could balance the requests by rotating the list. 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 replicates of the application.

Now you know how to get a message to a computer but it has still not gotten to the application you wish. This is the role of ports and TCP.

Ports

Figure 2.xx shows an old fashioned telephone switchboard. A call would come in over one number, the operator would ask for the extension and then 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 the phone and begin speaking.

This is exactly what happens with ports in computer systems. The IP address gets the message to the desired computer but it is the port number that gets it to the appropriate application. The application will be listening on a particular port 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 applications. There are many of these but some of the most common are:

- 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)

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.

TCP

Just as IP addresses required an IP Protocol to enable interpretation of messages, ports have several protocols that enable interpretation at the port level. The most common is the Transmission Control Protocol (TCP) which provides reliable, ordered, and error-checked delivery of a message. Before going into details of the TCP header, we will discuss the reliable, ordered, and error-checking aspects.

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

A sender sends a packet with a sequence number and then waits for an ACK. If the ACK does not arrive in a timely fashion, the packet is resent. The recipient will discard the second sending of a packet if it has already received the first sending. Thus, sequence numbers are both the means for reliability and for ordering of TCP packets.

The header also includes a checksum that is used for error detection. If the checksum indicates that an error occurred in transmission, no action is taken but, also, no ACK is sent. Thus, the sender sends the packet a second time.

There are optimization techniques that may reduce the number of ACKs required so not necessarily every packet generates an ACK.

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 application that is receiving the packet has a port on which it is listening, the application that is sending the packet is also listening on a particular port. Thus, both the sender and the recipient ports are necessary.

As with IIP addresses, the port numbers in a TCP message may be changed while the message is en route to its destination.

Subnets

A subnet is a logical division of the TCP/IP address space. IP addresses are on the same subnet if, somehow, they are logically connected. The connection may be that the devices with those IP addresses communicate frequently, it may be that the two devices are close together in some physical sense, or it may be for the convenience of the system administrator.

The main reason for creating a subnet is performance of the network. If your local network begins to suffer from performance because of excessive traffic then you should define subnets within your local network. Subnets allow routers to be more efficient and, hence, reduce the latency of messages within a subnet. You can still send messages from one subnet to the broader internet but these messages will go through a more complicated routing process.

A subnet is created within an address space by making the first portion of the IP address be the same for all of the devices on the subnet. The final portion of the IP address will identify which device on the subnet a message is sent to.

For example, suppose as a system administrator you are given a block of IP addresses that begin with 71.229.83. You can assign the last block of this IP beginning to the various computers under my control. Each of those computers will begin with the same IP address but they will be assigned 71.229.83.1, 71.229.83.2, etc.

Associated with each subnet is a subnet mask. This is a binary number that is logically ANDed with an IP address to get the subnet prefix. In our example, it 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.

Structuring your network

Subnets are a structuring mechanism based on IP addresses and its motivation is primarily performance of the network. Other reasons why your organization structures your network are protection and logical simplification. The structuring mechanisms in this case are computers that are in the path of messages to and possibly from the organization's network. These specialized gatekeeper computers have a variety of names – firewall, gateway, proxy server – but they all are intended to monitor and control network traffic into and out of the local network.

In theory, your network can consist of a variety of devices, each with their own IP addresses and all equally accessible from the internet. In practice, your organization wishes to restrict access 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 gateway. The basic screening gatekeeper is called a firewall. Simple firewalls operate only on the IP address of the sender and the port of the recipient.

The sending device's IP address may be on a blacklist in which case the message would be blocked by the firewall and not reach its destination. The message may be intended for a port that is closed to outside traffic. Again, the message would be dropped. Firewalls are configured using a set of rules. The rules are not necessary static. For example, your device may be put on a temporary blacklist because you have been sending too many messages through the firewall. In this case, you may be identified as being a portion of a distributed denial of service attack and be blocked for some period of time.

More sophisticated firewalls examine the content of the payload. They are looking for known types of malware. The examination of the content is complicated 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 two portions: a public portion where any external message will reach, and private portion (the intranet) which only legitimate messages should reach. This is shown in Figure 2.xx. The public portion is where, for example, an organization's external facing web server might be located. When you access the organization's web page, it is served by the external facing web server. This type of structure is called a demilitarized zone (DMZ) and is defined by firewalls before and after the external facing servers.

Firewalls may also be used within the intranet both for security reasons and for organizational reasons. Some information within an organization is restricted to authorized recipients. Placing these recipients behind a firewall means that traffic to these recipients can be monitored both to prevent others from seeing the information they generate and for forensic purposes in case a problem occurs.

A firewall also can be used to restrict problems. 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.

Organizations may also wish to keep their individual devices private from the broader internet. That is the purpose of a proxy server. You are sending a message to the open internet. You may have a private IP address or your organization wishes to keep your specific IP address from your recipient. Your message is sent out via a proxy server which replaces the source IP address in the message with its own IP address. Then when a response comes in, the proxy server forwards it on to you. Recall that when we discussed IP protocols we said that the source of destination may be modified while a message is in transit. This is one example of this modification.

Tunneling is a technique to allow packets through the firewall that otherwise would be rejected.

Tunneling

You have probably heard of and used a VPN (Virtual Private Network). 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 packets 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 2.xx. 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 on a device on the internet, possibly a mobile device.
- You access the VPN server through the internet and provide it with your credentials.
- These credentials are validated either by the VPN server itself or by the VPN server accessing the intranet to validate your credentials.
- Once your credentials have been validated, the VPN protocol is initiated between software on your device and software within the organization. This protocol uses the payload portion of the

TCP protocol to embed your messages which are then allowed through the firewall. The decoding software can be inside the DMZ or inside of the intranet.

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 a subsequent chapter. For now, just be aware that if you are setting up a VPN network, security is an overriding concern.

Summary

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

IP addresses are unwieldy and subject to change and so URLs are usually used to identify remote resources. The DNS is used to convert URLs into IPs. DNS has an internal structure that allows local organizations more control on the left most elements of a URL. The DNS also can implement a form of load balancing to send requests to different devices running the same application software.

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

Subnets are used to segment the IP addresses 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 publically available servers. A DMZ divides the world into the open internet, a controlled area open to anyone and a private area.

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

Exercises

1. Create a virtual machine as in Chapter 1 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 another.

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. TCPdump is one.

Discussion

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 proxy server to the internet. The proxy server replaces the source IP address with its own address. This means that the response goes to the proxy server, not to you. How does the proxy server know to forward the message to you as opposed to some other device in your intranet?

===========================end of Chapter 2==

Chapter 3 – The Cloud

Virtualization and networking are twentieth 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 allocates VMs both in response to explicit requests and in response to changing load,
- why you need to be concerned about availability for your services
- how messages are allocated to instances of VMs,
- how VMs can coordinate to share data, and
- how the choice of where to keep state affects computations,

We begin with a structural description.

Structure

The first thing to realize is that 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 the data center is the power it consumes. Figure 3.1 shows a data center – rows and rows of computers mounted in racks with high speed switches in between. In Chapter 2 we said that message latency is on the order of nanoseconds when the source and destination are “close”. Close equates to “in the same rack”. On the other hand, sending a packet from Europe to California takes around 150 ms. These numbers will affect how replication occurs and how up to date the replication can be but for now, let us discuss logically how the cloud works.

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 and each region has availability zones. .Each availability zone has data centers with their own power supply and physical location. The intent is that if one data center fails, losing power for example, the other data centers are unaffected. We will return to this point when we discuss availability and business continuity in Chapter xx.

All access to the cloud is over the internet. There are two main gateways into a cloud – a management gateway and a message gateway. Figure 3.2 shows these two gateways. The message gateway is as we discussed in Chapter 2 so 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 VM based on an existing VM image and it returns a public IP address. The management gateway is managing tens of thousands of physical computers. Each physical computer has a hypervisor that manages the VMs on it. So, the management gateway will ask each hypervisor whether it can manage an additional VM. It will select one of the positive responses and ask that hypervisor to, in fact, create an additional VM and return its IP address to the management gateway. The management gateway then returns that IP address to you.

The management gateway not only returns an IP address for an allocated VM, it also returns a URL. We discussed the structure of a URL in Chapter 2 and pointed out that the leftmost fields in a URL are controlled by the local organization. The URL returned after allocating a VM reflects the fact that the IP address has been added to the cloud DNS system.

Several points about this description:

- This is logically what occurs, not physically. Actually the interactions between the management gateway and the individual hypervisors are more complicated than we have just described. For our purposes, however, this logical description is sufficient.
- The VM image used to create the VM can be any VM image. It could be a simple service or it could be a portion of a bootstrap process to create a complex system.
- The hypervisor does not generate the IP address, it asks an IP address manager within the cloud for an available IP address. The IP address manager is one of the many services provided by the cloud unseen by the service you are developing. We will see others.

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

Access to the management gateway is through messages over the internet but these messages can be generated by a browser or by services. One type of service is a deployment tool and we will return to this when we discuss deployment.

You can ask the management gateway to allocate a VM but you cannot ask it to allocate a container. Containers, as we saw in Chapter 2, reside on VMs but the allocation of a container proceeds differently from the allocation of a VM. We will discuss container management in Chapter 4.

The management gateway also allows you to allocate a Virtual Private Cloud (VPC). A VPC is, as its name suggests, a collection of VMs acting as a cloud but logically separate from other VPCs in the cloud. A VPC is associated with your account and has its own access rules. Suppose, for example, that you or your organization wish to have separate clouds for the application group to do development and the financial group to manage billing and payroll. Then each group could have its own VPC. Access is controlled from one VPC to another but the cloud provider bills each VPC independently. Thus, your management knows how much of the cloud bill is being spent on application development versus financial management. VPCs are not only useful for organizational purposes but since the access rules are different they provide another level of security to the cloud.

Failure in the cloud

When a data center contains almost 100,000 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 disc drives, an average of around 5 computers and 17 discs will fail every day. Google reports similar statistics. In addition to computer and disk failures, switches can fail, the data center can overheat causing all of the computers to fail, etc. Thus, although your cloud provider will have relatively few total outages, the physical computer your VM is residing on my fail. What this means to you is that, 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. We will return to discussing availability when we discuss microservices.

In addition to outright failure, services running on public cloud may exhibit what is called “long tail latency”. What this means is that although the average latency for a request to a cloud based service may be within tolerable limits, a percentage of these requests – possibly as many as 5% - will have much greater latency than the average. 5 to 10 times the average. These long tail requests are a result of congestion somewhere in the path of the service request. The congestion could come from a number of sources – 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 will return to this topic when we discuss monitoring and microservicess.

The Load Balancing Function

Now we turn to managing the overloading of an individual VM instance. This 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 the DNS to distribute requests to a list of URLs rather than a single URL. Using DNS is both cumbersome and limited. Having a dedicated load balancer is a more natural solution. We titled this section "Load Balancing Function" because load balancers can be standalone systems or bundled with other functions such as proxy. A load balancer must be very efficient since it sits in the path of every message from a client to a service and so even when it is packaged with other functions, it tends to be logically isolated. We divide our discussion into two main portions: how load balancers work and state management. Then we discuss health management.

How Load Balancers work

The problem a load balancer solves is the following: There is a single instance of a service running on a VM. There are also too many requests arriving at this instance for it to provide acceptable latency. The solution is to have multiple instances hosting the service and distribute the requests among them. The distribution mechanism is a separate VM called a "load balancer". Figure 3.xx shows a load balancer distributing requests between two instances.

As a side note, what are values for "too many requests" and "reasonable response time"? These two questions lead to what is called "autoscaling" which we will discuss later in this chapter. For now, we are concerned with how a load balancer works.

In Figure 3.xx we show two clients generating requests. Each request is sent to a load balancer. The load balancer is managing two instances of the same service. Each instance was loaded by the same virtual 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".

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}.

The source field of the message is unchanged and so the service instance, whether 1 or 2, responds directly to the client that sent the message.

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”. Other algorithms for distributing the messages exist.
- The client sends its message to the load balancer whose IP it may have gotten from a DNS server. The client does not know, or need to know, how many instances of the service exist.
- Multiple clients may coexist. Each client will send its messages to the load balancer which does not care which client a message came from. The load balancer distributes the messages as they arrive. There is a concept called “sticky” that we ignore for the moment.
- Load balancers may get overloaded. In this case, the solution is to load balance the load balancer. That is, a message goes through a hierarchy of load balancers before arriving at the service instance.
- The load balancer is told which instances belong to it. That is, it has an interface which accepts the IP addresses of the instances that it is balancing. These instances may be static and given on initialization or they may be added dynamically such as from the autoscaler – see the section on autoscaling.

Failure detection and management

In the message modification scheme we described above messages are returned directly to the source IP in the message header and the return messages bypass the load balancer. This means the load balancer has no means of knowing whether a message was processed. More specifically, the load balancer does not know whether the instance is alive and processing or has failed.

A different scheme that guarantees processing of the message is called “Message queuing”. In this scheme, the message queue software, which may be built into the load balancer, maintains a queue of requests. When an instance is ready to process the next request it retrieves that request from a queue maintained by the message queue software. When it sends a response, it asks the message queue software to delete the request from the queue. If the instance does not respond to the message queue software within a reasonable time frame, the message is sent to another instance. If in fact, the request is processed by the first instance, just slowly, then the request may be processed twice. You must allow for double processing in your service, if you are using message queue software. With this scheme, the message is guaranteed to be processed by an instance.

Health checks are a means for the load balancer to determine whether the instance is performing properly. The load balancer will check the health of the instances assigned to it periodically. 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, the instance has an overloaded queue. It may not respond to the load balancer’s health check but once the queue has been reduced, 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.

State Management

We now turn to a topic that is under your control as a developer of a service and, as such, properly belongs in Part 3 of this book. On the other hand, we will shortly discuss topics that depend on knowing the distinction between stateful and stateless and so we discuss state here, rather than deferring it.

Management of state is important and not obvious when dealing with multiple instances of the same service. The key issue is where state is kept – in the service, in the client of the service, or in a location external to the service. We use a service that counts how many times it is called as an example. We will discuss three variants of this service.

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

```
int countv1()
 {
        int i = 0; //declare i and initialize it to 0.
        i = i + 1;  //add 1 to the last value of i
   return i;
  }
```

Variant 2: The service does not remember 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: Save the count externally to both the client and the service.

```
int countv3()
 {
  int a;
  a = dbase_get (“count”);  //retrieve current value
```

```
    a = a + 1;  //add 1 to the last value of a
    dbase_write("count", a);  //save current value
    return a;
   }
```

Now we examine these three variants in two different cases, each with two clients.

Case 1: Two clients, one instance of the service.

In variant 1 the service counts the number of calls it receives, regardless of which client calls it. Only one count is maintained.

In variant 2 each client counts the number of calls it makes. There are two counts, one for each client.

In variant 3, the database contains the number of calls the service receives, regardless of which client calls it. Only one count is maintained.

Case 2: Two clients, two instances of the service.

In variant 1 each instance of the service counts the number of calls it receives. There are two counts, one for each instance.

In variant 2, each client counts the number of calls it makes, regardless of which instance receives the call. There are two counts, one for each client.

In variant 3, the data base records the number of calls made, regardless of which client made them and which instance received them. There is one count.

From this example, it should be clear that how state is managed is important and will affect the results a service computes. We saw three different kinds of state:

1. State maintained in the service. The service is "stateful"
2. State maintained in the client. The service is "stateless".
3. Persistent state maintained in the database. The service is "stateless".

Common practice is to write services to be stateless. Stateful services lose their state if they fail and recovering that state is difficult. Also, as we will see in the next section, new instances may be created of a service and keeping the service stateless allows messages to be distributed to the new instance without regard for the history of the message.

"Sticky" messages contain an instruction to the load balancer to send the message to the same instance that handled the last message from this client. This type of message is used when the service is stateful and helps maintain consistency between the state of the client and the state of the instance providing its service. Sticky messages should not be used except in 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 and cause other instances to be generated. These new instances do not have the correct state to act on a sticky message.

A solution exists to manage small amounts of state shared among all instances of a service. We discuss this solution in the next section.

Distributed Coordination

Leslie Lamport described a distributed system such as the cloud as follows: “A distributed system is one in which the failure of a computer you didn’t even know existed can render you own computer unusable”.

We will use an example the problem of creating a lock across distributed computers. That is, there is some critical resource that is being accessed by services from two distinct VMs residing on two distinct physical computers. For our example, we assume this critical resource is a data item – think of it as your bank account. If we allow both services to independently access this data item, then there is the possibility of a race condition. Two independent deposits into your account overwrite one another and you are only given credit for one deposit. The standard solution to this problem is to lock the data item. A service cannot access your bank account until it gets a lock. So service 1 is granted a lock on your bank account and can make its deposit unhindered until it yields the lock. Then service 2, which has been waiting for the lock to become available, can lock the bank account and make the second deposit.

Two problems exist with this scheme: 1) the 2 phase commit protocol traditionally used to acquire a lock requires multiple messages across the network, each taking time, and 2): Service 1 may fail after it has acquired the lock preventing Service 2 from proceeding.

The solution to these problems involves complicated distributed coordination algorithms. Leslie Lamport, quoted above, developed the first such algorithm called Paxos. Paxos, and other distributed coordination algorithms rely on a consensus mechanism to resolve conflicts between various clients in the case of failure.

When clients wish to share state, they store this state in an in memory repository that implements a distributed coordination mechanism. Let us assume the Paxos algorithm is used for consensus. Paxos relies on the existence of a master that keeps the authorative state for the clients. A client queries a server to read this state. Multiple servers exist, one of which is the master. When a client wishes to write to the in memory repository, it passes this request on to its server which passes it on to the master. The master approves the request, choosing one request over another in the case of conflicts. The other servers are informed of the master server's decision so that reads will continue to be consistent.

The complication in this process comes when the master fails. In this case, a new master must be chosen and the state made consistent across all of the servers. This portion of the Paxos algorithm is extremely difficult to understand and to implement correctly.

As a service developer you, fortunately, do not need to understand or implement Paxos since it has been packaged into a variety of coordination packages. These coordination packages allow different instances of the same service to share a limited amount of state with very small latencies – on the order of microseconds. Packages such as Memcached, Zookeeper, Consul, etcd are all open source packages that implement Paxos and provide various types of interfaces to manage the lock problem.

We now revisit the distributed locking problem using Zookeeper as our sample distributed coordination package. Zookeeper maintains data in a hierarchal set of nodes. For the purposes of this example, the database system will create a node called "your account lock". When this node is created it has no children. Nodes in Zookeeper are either persistent or ephemeral. The "your account lock" is persistent. Now you create a child of that lock. The node you create is ephemeral which means that Zookeeper will clean it up for you and you do not need to worry about stale nodes or extra nodes. You also set a "watcher" for the node "your account lock". If the state of this node changes you are notified through a call back. We will see how this works in a moment. Now you ask Zookeeper to give you a list of the children of "your account lock". The children represent requests for a lock on your account. Zookeeper returns an ordered list and if your node is the first item on the list you own the lock. If it is not, your service can do other things. When the children on the list change, you are notified by the watcher and you check the list again to see if you own the lock. Once you own it, you perform your modification to the account and delete the node you created.

Because the Zookeeper server you are dealing with is "close" to your VM, your interactions with it are fast. Because Zookeeper only manages a limited amount of data, the servers keep it in memory so it is fast. Interactions with Zookeeper take on the order of milliseconds.

In the section on state management above, we assumed that shared state was kept in an external database. External databases involves disk accesses that are many times slower than memory accesses and, consequently, external databases should be used only for data that is intended to persist across multiple sessions. Distributed coordination systems give us a mechanism to manage shared state with minimal latency. As long as the shared state is small – less than 1MB – and is not required to persist beyond single invocations of an application, a distributed coordination system is used to save that shared state. One caveat is that the data in a distributed coordination system is assumed to be owned by instances. This is distinct from data in an external database that is not necessarily owned by any active instance. A consequence of this ownership concept is that at least one owning instance must be currently active for the distributed coordination system to maintain the shared state. As soon as all of the owning instances are no longer active, the shared state will be deleted.

Now we turn to the automatic creation and deletion of instances.

Autoscaling

Autoscaling is a service that cloud providers may offer that automatically creates new instances. Returning to Figure 3.xx, suppose the two clients generate more requests than can be handled by the two instances that exist. Autoscaling creates a third instance, based on the same virtual image as was loaded into the first two images. The new instance is registered with the load balancer so that subsequent requests are distributed among three instances rather than two.

Since the clients do not know how many instances exist or which instance is serving their requests, the clients can remain ignorant of autoscaling activities. Furthermore, if the existing instances are too many for the current number of requests, an instance can be shut down and removed from the load balancer, 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 most coordinate. When the autoscaler creates a new instance, it must inform the load balancer of the existence of that instance. Figure 3.xx shows a load balancer and an autoscaler. In this figure the autoscaler is specific to the load balancer but in practice an autoscaler can manage multiple load balancers.

Notice in Figure 3.xx that the autoscaler is monitoring the instances. The instances are unaware of the metrics that the autoscaler is monitoring. I.e., they are passive in the autoscaling process. This means that the autoscaler can only monitor the metrics that are collected by the cloud infrastructure. This is typically CPU utilization and I/O requests to the network. You set up a collection of rules for the autoscaler that govern its behavior. The information you provide to the autoscaler includes:

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

The autoscaler does not create or delete instances on instantaneous values of the CPU utilization or bandwidth. First of all these values shows spikes and valleys and are only meaningful when measured over some interval. Secondly, creating a new VM takes time – on the order of minutes. The VM must be loaded, connected to the network, and initialized before it can be utilized 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 new VMs when utilization rises, you can also set rules to provide a minimum or maximum number of VMs or to create VMs based on time. During a normal week, for example, load may be heavier during work hours and knowing this, you can allocate more VMs prior to the beginning of a work day and delete some after the work day is over. On the other hand, some services may be more heavily used outside of work hours. These allocations should be based on historical data about the pattern of usage of your services.

When the autoscaler decides to delete an instance, it can not just delete the instance. The instance may be in the process of servicing a request. Instead, the instance is notified that it should terminate its activities and shut down, after which it can be deleted. This process is called “draining” the instance. The autoscaler also notifies the load balancer not to send additional requests to the instance being terminated. You as a service developer are responsible for providing an interface to receive instructions to terminate and drain the instance.

Summary

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

Because of the sheer number of 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 distribution where as many as 5% of your requests will take 5 to 10 times longer than the average request. Thus, you as a service developer 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 have multiple instances of VMs containing your service. These multiple instances are managed by a load balancer. The load balancer receives requests from clients and distributes them to the various 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. Stateless services allow for easier recovery from failure and easier addition of new instances.

Limited amounts state can be shared among instances by using a distributed coordination service .Distributed coordination services are complicated to implement but are available from several different open source libraries.

The cloud infrastructure will automatically scale your instances by creating new instances when demand grows and deleting 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.

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 State Management and verify if 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 master?
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?

Discussion

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 fields 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?

========================end of chapter 3 =============================

Chapter 4 - Container Management

In chapter 1 we introduced the concept of container as a virtualization of an operating system. In Chapter 3 we discussed the cloud in terms of allocation of virtual machines. Although, as we said in Chapter 1, a container executes inside of a virtual machine, the allocation mechanisms for containers are somewhat different than the allocation mechanisms for virtual machines. This situation may change as the use of containers evolves but currently the mechanisms for the management of containers are distinct from the mechanisms for the management of VMs. This chapter deals with these container mechanisms. One comment about terminology. In Chapter 1 we made a distinction between containers – executing entities – and container images – collections of bits that could be executed. Although this distinction remains, the terminology used in practice blurs this distinction. The term “container” has become used both for the image and the executing image.

After finishing this chapter and performing the exercises you will

- Know why containers can be moved from one environment to another
- Know what a container repository is and how to store container images in them
- Know what a cluster is and what the orchestration of containers within a cluster is
- Know what a serverless architecture is
- Have a vision about total automation of containers

Container mobility

In Chapter 1, we introduced the concept of a container runtime with which the container interacts. Multiple different container runtime providers exist – most notably Docker, Kupernetes, and Mesos. As long as different container runtimes provide the same interface for the container, it will execute on any of these different container runtimes. The interface provided by container runtimes has been standardized by the Open Container Initiative. Thus a container created by one vendor’s package, say Docker, can be executed on a container runtime provided by another vendor, say Kupernetes.

What this means in practice is 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 non-trivial. If you embody all of the resource specifications as configuration parameters the movement of your container into production is simplified. We return to the topics of deployment and configuration parameters in later chapters.

Container repositories

Once you develop a container you will, typically, place its image into a repository. Container repositories such as Docker Hub provide features similar to the version control system Git. That is, you retrieve a container image through a “pull” command and you store a container image through a “push” command.

Furthermore, the images are version controlled. That is, you can retrieve the latest image pushed or a prior image. This provides the same benefits as version controlling source code. You can easily back up to a prior image, you can differentiate images being used to fix bugs in production from ones being used to develop new features.

Finally, repositories allow you to share your container images with others either locally on your team or globally as broadly as you might wish. Libraries exist with standard systems packaged as container images. Since the format of the images is standardized, you do not need to be concerned about what tool is used to generate the image. You do need to be concerned that you are not introducing malware into your system and we discuss security issues associated with downloading software from the internet in Chapter Security.

Clusters of containers

One method for visualizing a load balancer is to think of it as a manager of a collection of identical nodes. Clients send requests to a load balancer to get serviced by one of its nodes without regard for which node provides the service. Keep this analogy in mind when we discuss clusters.

A cluster consists of a master node and worker nodes. The master is the analog of the load balancer. It is the point of entry for requests for worker node service from clients. It keeps track of the health of the workers and schedules requests to the workers.

A node, both master and worker, is a VM. Recall that containers have to be hosted inside a computer, either physical or virtual. Nodes, however, are instantiated with the container runtime. Thus, in order to execute a container on a worker node, only the image need be loaded into the node. Loading container images is much faster than loading a VM image.VM images take on the order of minutes to load, container images take on the order or milliseconds.

Once the nodes are loaded, the master schedules requests to the various containers as in the load balancer analogy.

Kupernetes has one more element in its hierarchy – pods. Pods are collections of containers. The hierarchy is: worker nodes contain pods and pods contain containers. See Figure 4.xx. The containers in a pod share an IP address and port space. They communicate using standard inter process mechanisms such as IPC (Inter process communication). They have the same lifetime, i.e. the containers in pods are allocated and deallocated together. An example is placing a content manager and a web service that consumes that content into the same pod. Also placed in this pod is a volume used by the content manager to save content and the web service to retrieve that content.

Our picture so far has clusters being static. That is, a cluster is created, VMs are allocated to that cluster, containers (and pods) loaded into the VMs and the structure of the cluster remains unchained. In fact, there are two types of activities that make clusters much more dynamic – orchestration and scaling.

Cluster orchestration

In software engineering, the term orchestration is used for the coordinated management of a collection of activities. This definition applies to containers as well. Suppose, for example, you wished to build a two step machine learning pipeline. The first step of this pipeline is to standardize the data for the training of the model and the second step is to actually train the model. If you have one container for each purpose, you want to schedule them so that the standardization container runs first and its output is fed to the training container which is run second. This is an example of a workflow and the orchestration of this workflow must be specified.

You do not want to have to specify the sequence of activities manually each time you wish to train a model on new data but rather to automate the specification. One technique you could use is to write a single purpose script to run the pipeline. Another technique is to use a general purpose tool to perform the coordination. The general purpose tool understands containers and how to connect and sequence them and so using that tool could save you a lot of time. Of course, there is a learning curve for the tool and so the time you save has to be balanced against the time it takes you to learn the tool.

Learning a general purpose tool probably does not pay if all you wish to do is specify one workflow but, more likely, you will have many such orchestration specifications that you must accomplish. In that case, learning the tool is a one time cost that is amortized over all the uses you make of the tool.

You have just read the case for container orchestration tools such as Kubernetes. These tools will manage the orchestration for a cluster of containers. This orchestration performs the following functions, among others:

- Scheduling requests among the containers (as in the load balancer analogy)
- Scheduling the allocation and deallocation of containers
    - In terms of usages such as in the pipeline example above
    - In terms of load such as with autoscaling VMs
- Monitoring the health of the containers and disabling and deallocating unhealthy containers

An additional function that a container orchestration tool will perform is to reallocate the containers across VMs. Suppose, for example, a physical machine on which your VMs and containers are hosted is about to be taken out of service for maintenance. Your cloud provider will give you warning of this event and the orchestration tool understands this warning and allocates VMs and containers on a different physical host and switches the load to this new host. In the case of an unplanned outage of the physical machine, the orchestration tool will recognize the outage and allocate new VMs and containers on a distinct host and perform these actions without serious disruption to the clients.

The type of reallocation we have just discussed assumes that the containers and the services bundled in these containers are stateless. See Section 3.xx for a discussion of stateful and stateless.

Serverless architecture

Recall that allocating a VM involves locating a physical machine with additional capacity and loading a VM image into that physical machine. In other words, the physical computer constitute a pool from which to allocate. Suppose now that instead of allocating VMs into physical machines, you wish to allocate containers into container runtimes. That is, there is a pool of container runtimes available into which containers are allocated

Since load times for containers are on the order of milliseconds, the response time can be very fast as long as the container and the container runtime are readily available. Now carry this one step further. When you allocate a VM, it persists until it either fails or your pause or deallocate it. This is because VM allocation and loading is relatively time consuming – on the order of minutes. When you allocate a container into a container runtime, since the allocation is so fast, there is no necessity to persist the container. It can be recreated with every request.

This, in a nutshell, is what is called *serverless architecture*. In fact, there are servers and container runtimes but since they are allocated dynamically with each request, the servers and container runtimes are embodied in the infrastructure. You, as a developer, are not responsible for allocating or deallocating them.

A consequence of the dynamic allocation in response to individual requests is that the containers cannot maintain any state. That is, the containers must be stateless. When we discussed dynamic coordination, we introduced the idea of having rapid retrieval and storage of small amounts of state shared across multiple clients. These small amounts of state cannot be stored in a container that is used in a serverless architecture since containers managed by serverless architectures must be stateless.

Automating everything

Now we speculate about how the container infrastructure might evolve. We identify the problems that must be solved in order to totally automate the containerization of a procedure writing in your favorite language and discuss what solutions to these problems might entail. We emphasize that this section is speculation on our part.

Suppose you developed a service on your laptop using your favorite programming language and you developed it as a procedure. That is, it has an interface that is called from other procedures and it calls dependent procedures through their interface. Now you would like to push a button and your procedure is “containerized” and deployed into the cloud. Furthermore, the cloud infrastructure can create multiple instances of your new container, orchestrate them as we have discussed, and move them around to better utilize the cloud resources. The instantiation of a new instance takes on the order of milliseconds. We begin with a discussion of how interfaces can be managed.

Interfaces

The service as you wrote it is invoked through a simple procedure call. Containers are invoked through messages. So the first step in realizing the vision we just described is to put a wrapper around your procedure that translates messages into your interface. The messages come through https with parameters, let us say encoded as JSON. A portion of the wrapper must understand https and translate the https with JSON into the parameters you expect.

There is more to the wrapper than that, however. Presumably you wrote your procedure to be called synchronously, to process that call, and to return the values you calculated. Messages arrive asynchronously. Thus, the wrapper also contains a queuing mechanism that places incoming message on the queue and, when the message has been processed, takes it off the queue and presents the next message to your procedure. The queuing mechanism and the calculations are done as separate threads since messages can arrive at any point while the calculations can be done.

Errors can occur with the formatting and receipt of the https and these error messages must be generated and sent to the client.

The values your procedure calculates are send through a message are encoded into JSON for response message. So the wrapper does not only deal with incoming messages but also with the response message.

The interactions with the client of your containerized procedure are not the only interactions involving your procedure. Your procedure invokes dependent procedures, some of which are packaged with your procedure and some of which are their own containers. The ones that are packaged with your procedure can be invoked as you wrote them but the ones that are packaged as their own containers must be invoked using https. So a portion of the process of preparing your procedure involves preparing https messages for the containerized dependent procedures. Assuming you wrote the invocations of these other procedures to be synchronous, the response messages from the invoked procedures must be dealt with synchronously. Error responses will result in your procedure returning an error response to your client.

Containerization

Once the interfaces are resolved, then containerizing your procedure is the next step. The automation must take care of three things:

1. Creating the container. As you will recall, creating a container manually has two steps – loading the correct software and building the layers. As a result of defining interfaces, the automation knows the boundaries of the software to be loaded into the container – it is your procedure plus any dependent procedures included as code or binaries. You have identified which dependent procedures are themselves containers so what remains should be included in the container

generated from your procedure. Your procedure and dependent code must be compiled so that only binaries are loaded into the constructed container.

Layers are composed of binaries so that those portions of your procedure and dependent procedures that are separate compilation units will each be a layer.

2. Arranging the configuration parameters. Configurations for your container will be handled in the same fashion as configurations for your procedure. We discuss techniques for managing configuration parameters in section xx. The automation does not change this.

3. Setting up the network connections. From the portion which defines the interfaces, the automation knows which other containers your container can communicate with and it sets up network connections to those containers.

Orchestration

Orchestration involves the sequencing and interactions among multiple containers. Examination of your container does not provide enough context to deduce the sequencing and interactions. The automation of making a container from your procedure becomes a sub-process of the containerization of a collection of procedures. Using business process models as a description of the necessary orchestration actions and automating the conversion of the execution of the business process model provides a method to automating a much larger execution context than that provided by containers.

Summary

Containers are a packing mechanism that virtualize operating systems. A container can be moved from one environment to another as long as a compatible container runtime is available and the interface to container runtimes has been standardized.

Containers can be stored in repositories. The repositories identify various versions of the container. Storing a container 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 cluster facilities scaling and orchestration. The cluster manager schedules requests to individual containers and is responsible for scaling both up and down of the container which it manages.

Serverless architecture allows for containers to be rapidly instantiated andemoves the responsibility for allocation and deallocation to the cloud provider infrastructure.

One goal of container automation is to simplify the movement of a procedure from native code to a container that can be managed at runtime. The elements of this automation are changing some interfaces to use https rather than direct calls, creating containers from the code, and automatically converting orchestration instructions into an executable form.

Exercises

1. Store the container you created in exercise xx in chapter 1 into a repository. Have a colleague/classmate retrieve it and execute it.
2. Create a docker swarm with two instances of the container you used in exercise 1.
3. Use Kubernetes to set up a simple pipeline such as described in the section on container orchestration.

Discussion

1. What is the relation between a cluster and a sub net.
2. How are orchestration rules specified in Kubernetes?
3. What portions of the deployment vision as described above have been realized and what portions remain to be realized?
4. 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?
5. Identify the locations in this chapter where the word “container” was used when it should have been “container image”.
6. Read the first paragraph in the Introduction to Part 1 again. How many of the concepts do you now understand?

=======================end pf Part 1========================