Carnegie Mellon

institute for SOFTWARE RESEARCH

The DevOps Ecosystem

Len Bass

Topics AM

- Introductions
- What is DevOps
- Virtualization
- Networks
- The Cloud
- Security and SSH
- Basic tools
- Lunch

Introductions

- Len Bass - instructor (lenbass@cmu.edu)
  - 50+ years in computer industry
  - 15 years at Univ of Rhode Island, 25 years at SEI, 3 years at NICTA
- Author of books on software architecture, user interface software, DevOps, and deployment and operations

You

- What is your background?
- What would like to get from the workshop?

Carnegie Mellon

institute for SOFTWARE RESEARCH

What is DevOps?

Overview

- **Velocity of releases**
- Causes of delay in releases
- Definition of DevOps and categorization of DevOps processes

Velocity of new releases is important

- Traditionally organizations deployed a new release quarterly or monthly.
- In the modern world, this is too slow.
- Internet companies deploy multiple times a day

© Len Bass 2018

Release schedule statistics

- Etsy releases 90 times a day
- Facebook releases 2 times a day
- Amazon had a new release to production every 11.6 seconds in May of 2011

© Len Bass 2018

Overview

- Velocity of releases
- **Causes of delay in releases**
- Definition of DevOps and categorization of DevOps processes

Why have a new releases?

- An event triggers a requirement for a new release
  - It could be an idea for a new feature on an existing system
  - It could be a merger between two organizations
  - It could be a problem with a system in production
  - it could be competitive pressure
  - It could be a desire for more efficiency
- In any case, the event causes a requirement for creating a new system or modifying an existing system or systems.
- Modifying an existing system is, by far, the most common response.

© Len Bass 2018

Over the wall development

Board has idea

Developers implement

Operators place in production

Time

© Len Bass 2018

11

The triggering event causes requirements to be generated

- Requirements could be expressed in a variety of fashions
  - Documents
  - User stories
  - Intuition of the developers
- Requirements are divided and assigned to development teams

Developers organization

- There are a variety of different development teams
- Each team has its requirements
- Developers on each team work on requirements, in some fashion
- At some point a team feels they have satisfied the requirements they have been assigned
- Their code is complete – their job is done.

© Len Bass 2018

What is wrong?

- Code Complete ≠ Code in Production
- Between the completion of the code and the placing of the code into production is a step called: Deployment
- Deploying completed code can be very time consuming because of concern about errors that could occur.

What is the work flow for a multi-team effort?

- You develop and test your code in isolation
- Your code is integrated with code developed by other teams to see if an executable can be constructed.
- The built system is tested for correctness
- The built system is tested for performance and other qualities (staging)
- The built system is placed into production

Errors can occur

- Technology inconsistencies
- Incorrect use of various modules
- Incorrect parameter settings
- etc

© Len Bass 2018

Time is passing

- Every error must either be corrected or prevented.
- Correcting errors takes time
- Preventing errors can be done through some combination of
  - Process
  - Architecture
  - Tooling
  - Coordination among teams.
- Coordination takes time.

How much time?

- Historically, releases are scheduled for once a quarter or once or twice a year to give time to coordinate and adequately test.
- This means there may be months delay before a new concept or feature is added to a system.
- This delay has become more and more unacceptable.

- Velocity translates into time to market!!

© Len Bass 2018

Overview

- Velocity of releases
- Causes of delay in releases
- **Definition of DevOps and categorization of DevOps processes**

What is DevOps?

DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality.

- DevOps practices involve developers and operators' processes, architectures, and tools.
- DevOps is also a movement – like agile.

DevOps processes can be divided into three categories

1. Reduce errors during deployment
2. Reduce time to deploy
3. Reduce time to resolve discovered errors

© Len Bass 2018

Summary

- DevOps is motivated by a desire to reduce the time to market for changes to a system
- Historically releases were scheduled events – taking months to prepare
- DevOps is defined as a collection of processes whose goal is to reduce the time between commit and normal production
- These processes either
  - Prevent errors
  - Speed up the detection and correction of errors

Questions?

© Len Bass 2018

Virtualization

What is a Virtual Machine?

What is the difference between a VM and a VM image?

How does a VM image become a VM?

Where does a VM image come from?

What is the piece of software that turns VM image into VM called?

What is the difference between a type 1 hypervisor and type 2 hypervisor?

© Len Bass 2018

Networking

Bring up browser in virtual machine

Bring up browser on host

In both browsers, enter “what is my ip”

What is an IP address?

What entities on network get an IP address?

Why does your host browser and the VM browser show the same IP address?

Where do IP addresses come from?

What is the difference between public and private IP addresses?

What is a port?

How do IP addresses and ports collaborate to get messages to the correct application on the correct VM?

What is DNS?

Finding www.mse.isr.cmu.edu

- Begin with “root server”. There are ~13 root servers with known IP addresses. These are built into the router. (https://www.iana.org/domains/root/servers)
- Access root server to get IP address of the .edu DNS
- The .edu DNS has the IP of the .cmu.edu DNS and so forth.
- Eventually you get to a DNS server that is under local control
- This allows MSE to change the IP of the various local DNSs without changing anything up the hierarchy.

Time to Live

- Clients do not access a DNS server for every request. It would generate too much internet traffic.
- Associated with each DNS entry is a Time To Live (TTL).
- This is also called a “Refresh Interval” in the DNS resource record called Start of Authority (SOA).
- The client or the ISP caches the IP addresses associated with DNS entries and these entries are valid for the TTL.
- Leftmost DNS servers have TTL on order of minutes.

© Len Bass 2018

Carnegie Mellon

institute for SOFTWARE RESEARCH

DevOps: Engineering for Deployment and Operations

The Cloud 1

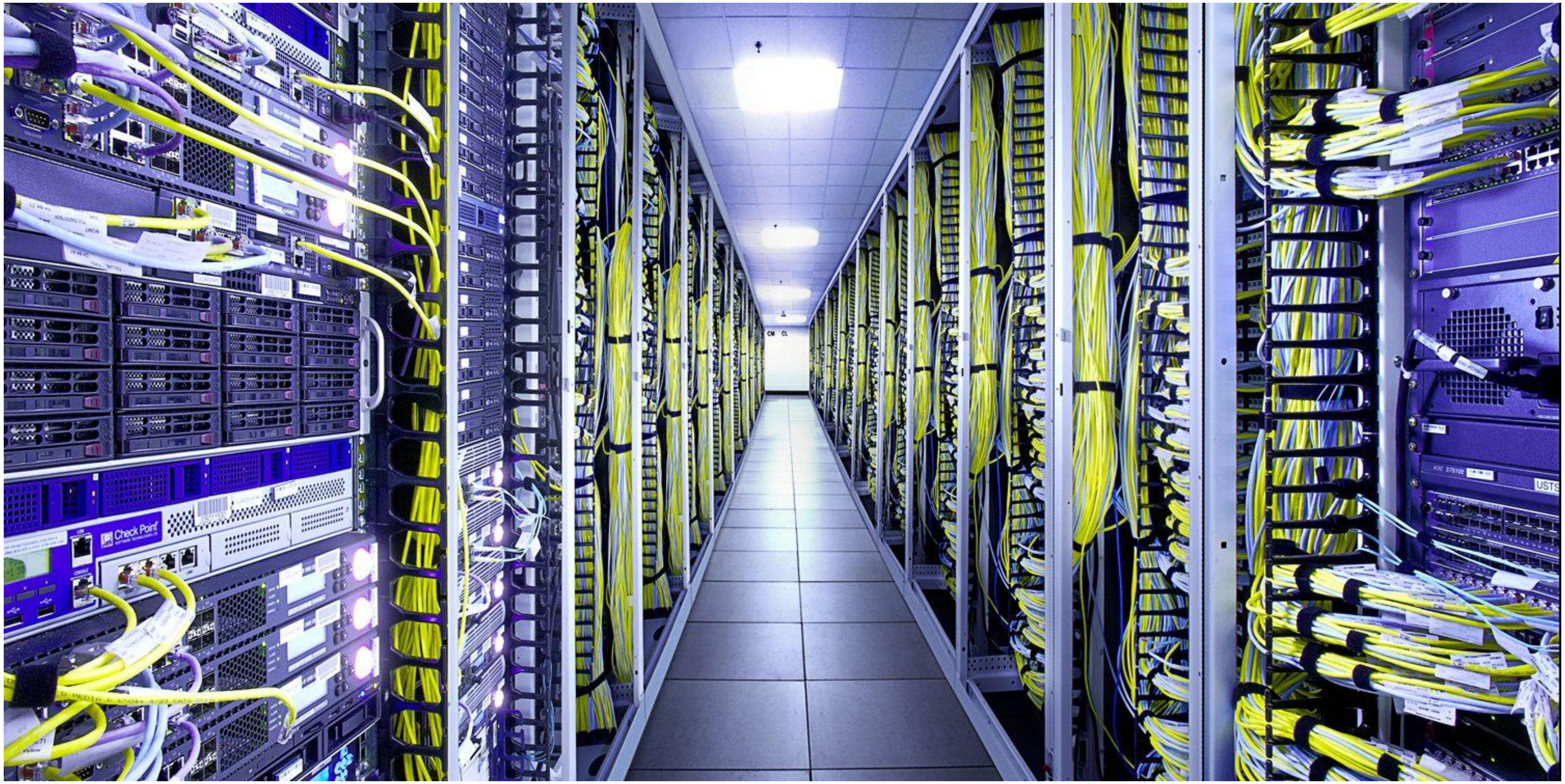
Overview

- **Structure**
- Failure in the Cloud
- Scaling Service Capacity

Data centers in the cloud

- A cloud provider (AWS, Google, Microsoft) maintains data centers around the world.
- Each data center has ~100,000 computers.
- Limited by power and cooling considerations.

Data Center



Organization of data centers

- Each data center has independent power supply, independent fire control, independent security, etc
- Data centers are collected into availability zones and availability zones are collected into geographic regions.
- AWS currently has 20 regions each with several availability zones.

Allocating a virtual machine in AWS - 1

- A user wishes to allocate a virtual machine in AWS
  - The user specifies
    - A region
    - Availability zone
    - Image to load into virtual machine
    - …

Allocating a virtual machine in AWS - 2

- AWS management software
  - Finds a server in that region and availability zone with spare capacity
  - Allocates a virtual machine in that server
  - Assigns IP address to that virtual machine (public)
  - Loads image into that virtual machine
- VM can then send and receive messages

Overview

- Structure
- **Failure in the Cloud**
- Scaling Service Capacity

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Leslie Lamport

Failures in the cloud

- Cloud failures large and small
- The Long Tail
- Techniques for dealing with the long tail

© Len Bass 2018

Sometimes the whole cloud fails

Selected Cloud Outages - 2018

- Feb 15, Google Cloud Datastore
- March 2, AWS Eastern region networking problems
- April 6, Microsoft Office 365. Users locked out of their accounts
- May 31, AWS Eastern Region hardware failure caused by “power event”
- June 17, Microsoft Azure. Heatwave in Europe and malfunctioning air conditioning
- July 17, Google. Load balancer issue affected services globally
- July 16, Amazon Prime. Not AWS but Amazon sales
- Sept 5. Microsoft Office 365. Botched update to Azure authentication

And sometimes just a part of it fails …

A year in the life of a Google datacenter

- Typical first year for a new cluster:
  - ~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
  - ~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
  - ~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
  - ~5 racks go wonky (40-80 machines see 50% packetloss)
  - ~8 network maintenances (4 might cause ~30-minute random connectivity losses)
  - ~12 router reloads (takes out DNS and external vips for a couple minutes)
  - ~3 router failures (have to immediately pull traffic for an hour)
  - ~dozens of minor 30-second blips for dns
  - ~1000 individual machine failures
  - ~thousands of hard drive failures

  slow disks, bad memory, misconfigured machines, flaky machines, dead horses, …

Amazon failure statistics

- In a data center with ~64,000 servers with 2 disks each

  ~5 servers and ~17 disks fail every day.

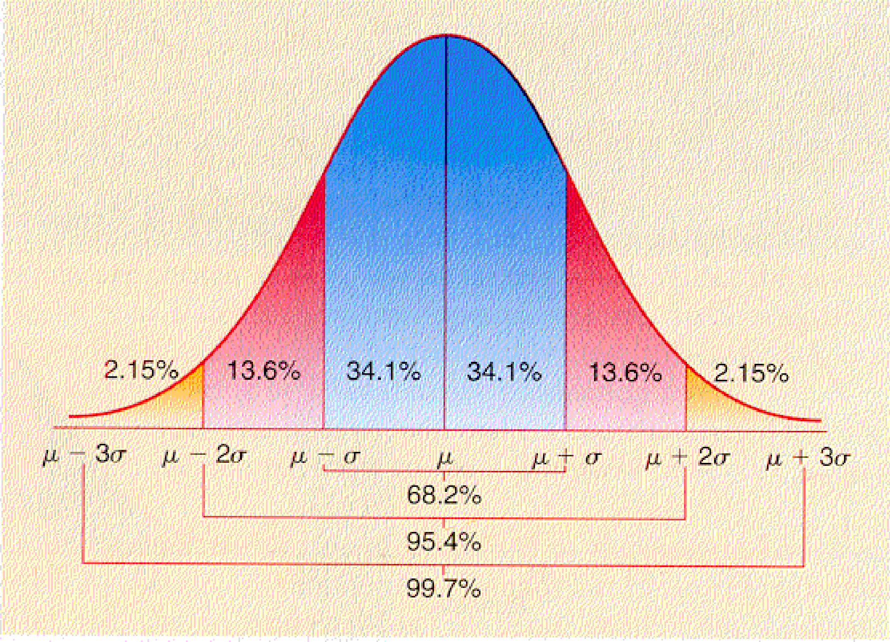
Failures in the cloud

- Cloud failures large and small
- The Long Tail
- Techniques for dealing with the long tail

© Len Bass 2018

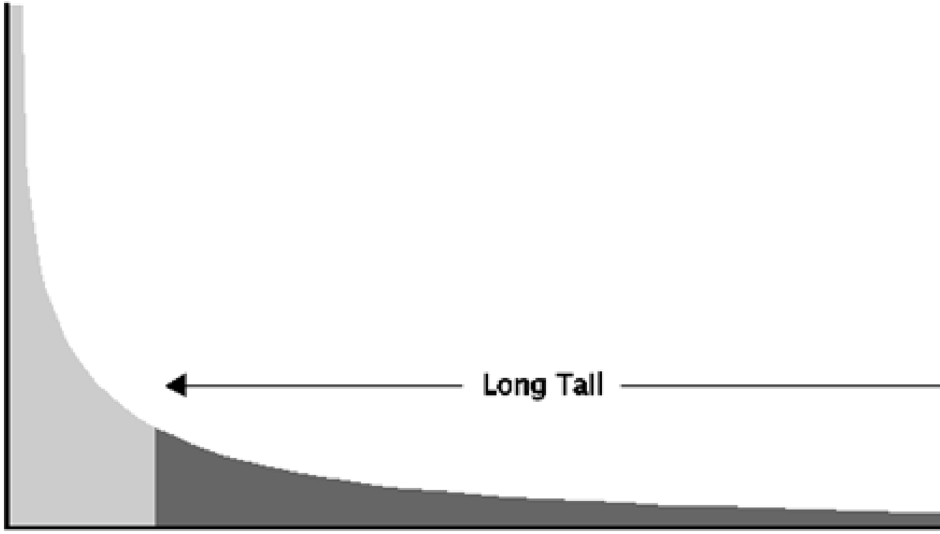
Short digression into probability

- A distribution describes the probability than any given reading will have a particular value.
- Many phenomenon in nature are “normally distributed”.
- Most values will cluster around the mean with progressively smaller numbers of values going toward the edges.
- In a normal distribution the mean is equal to the median



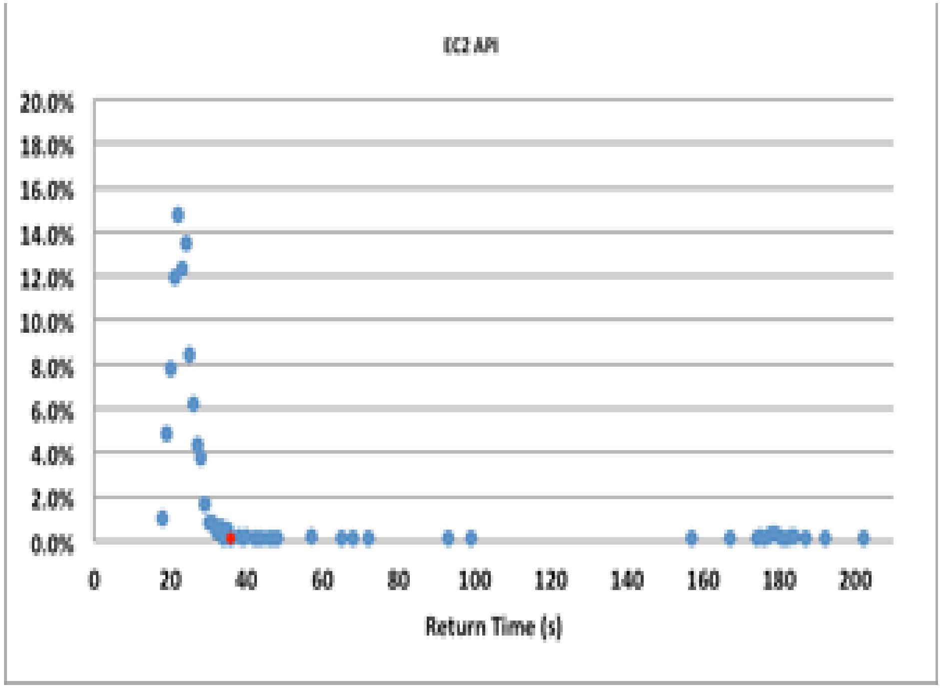
Long Tail

- In a long tail distribution, there are some values far from the mean.
- These values are sufficient to influence the mean.
- The mean and the median are dramatically different in a long tail distribution.



What does this mean?

- If there is a partial failure of the cloud some activities will take a long time to complete and exhibit a long tail.
- The figure shows distribution of 1000 AWS "launch instance" calls.
- 4.5% of calls were "long tail"



Failures in the cloud

- Cloud failures large and small
- The Long Tail
- Techniques for dealing with the long tail

© Len Bass 2018

What can you do to prevent long tail problems?

- Recognize that failure has occurred
- Recover from failure
- Mask failure

© Len Bass 2018

Recognizing failure

- In a distributed system, the only communication between distinct computers is through messages.
- A request (message) to another machine goes unanswered.
- The failure is recognized by the requestor making the request asynchronously and setting a timeout.

© Len Bass 2018

Health Check

- Health check. Requires one machine to act as a monitor for other machines.
  - Machine sends a message to a monitor periodically saying “I am alive”
  - Ping/echo. Monitor sends a message to the machine being monitored and expects a reply within specified period.

Recovering from failure

- Retry
- Instantiate another instance of failed service
- Graceful degradation
- Set “circuit breaker” to prevent trying to use service again.
- Netflix Open Source Hystrix library provides some useful functions

© Len Bass 2018

Masking failure

- “Hedged” request. Suppose you wish to launch 10 instances. Issue 11 requests. Terminate the request that has not completed when 10 are completed.
- “Alternative” request. In the above scenario, issue 10 requests. When 8 requests have completed issue 2 more. Cancel the last 2 to respond.
- Using these techniques reduces the time of the longest of the 1000 launch instance requests from 202 sec to 51 sec.

© Len Bass 2018

Overview

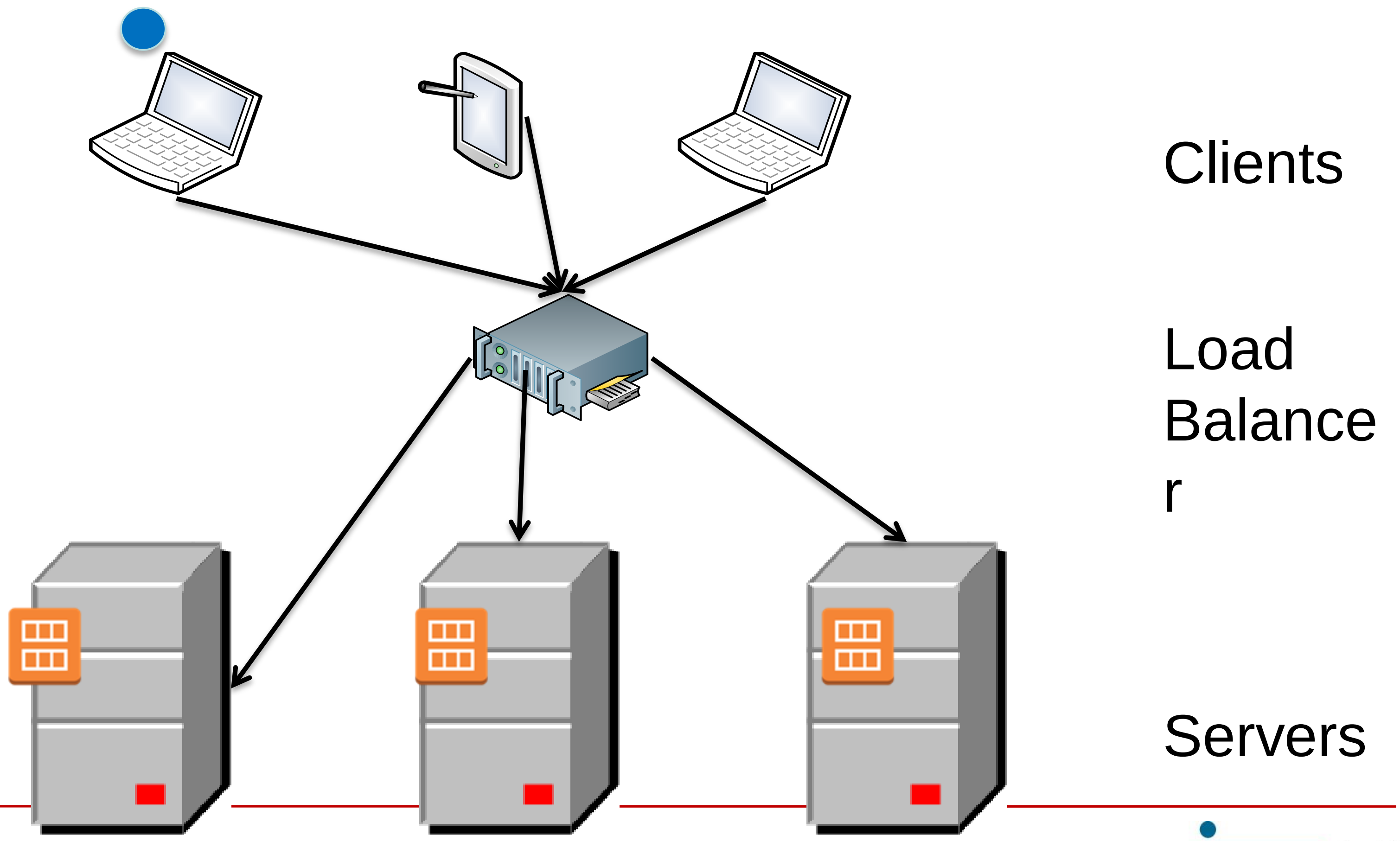
- Structure
- Failure in the Cloud
- **Scaling Service Capacity**

© Len Bass 2018

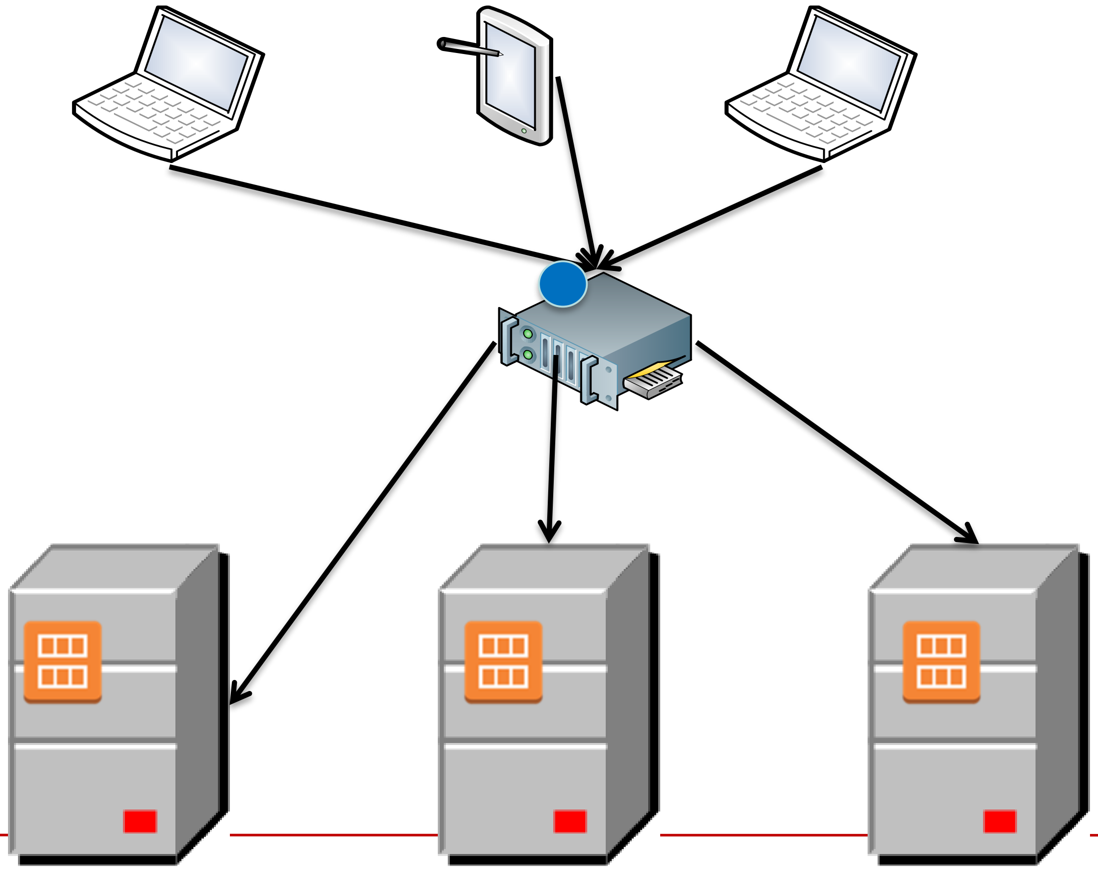
Load Balancer

- One server may not suffice for all of the requests for a given service.
  - Have multiple servers supplying the same service.
  - Use “load balancer” to distribute requests.
- Server is registered with load balancer upon initialization
- Load balancer monitors health of servers and knows which ones are healthy.
- Load balancer IP is returned from DNS server when client requests URL of service.

Message sequence – client makes a request

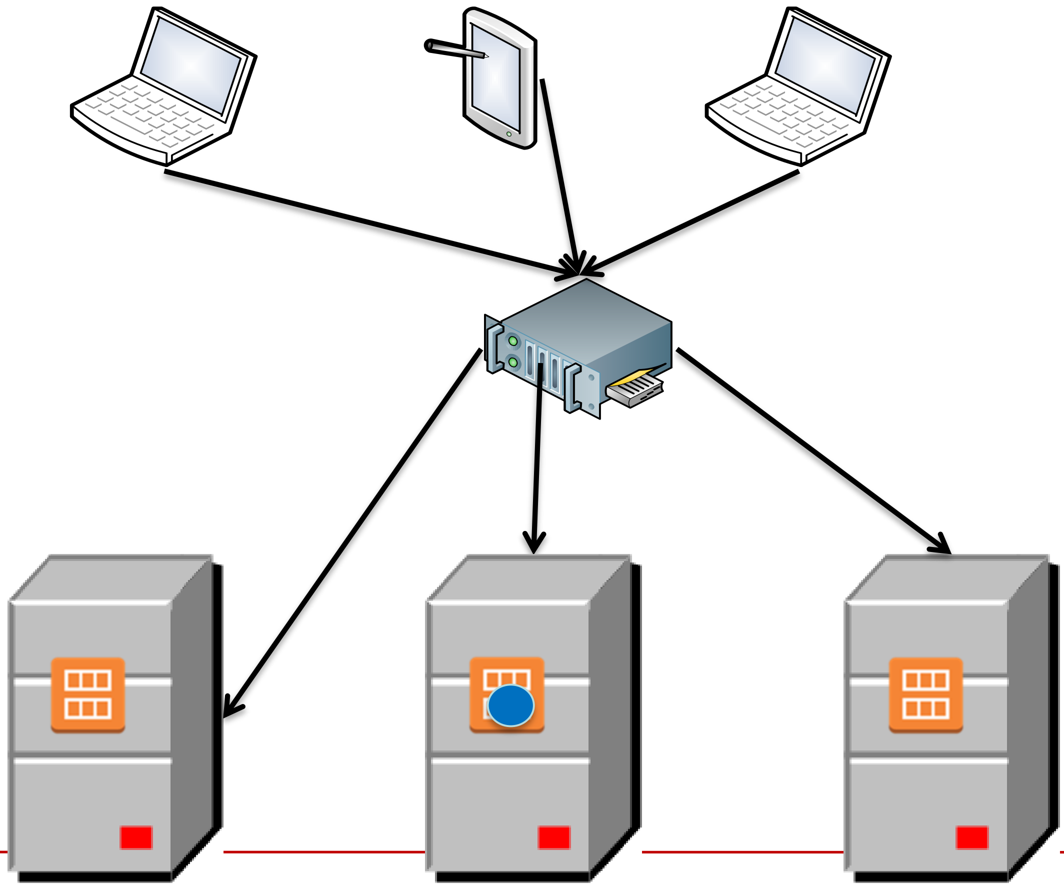


Message sequence- request arrives at load balancer



isr institute for SOFTWARE RESEARCH

Message sequence – request is send to one server



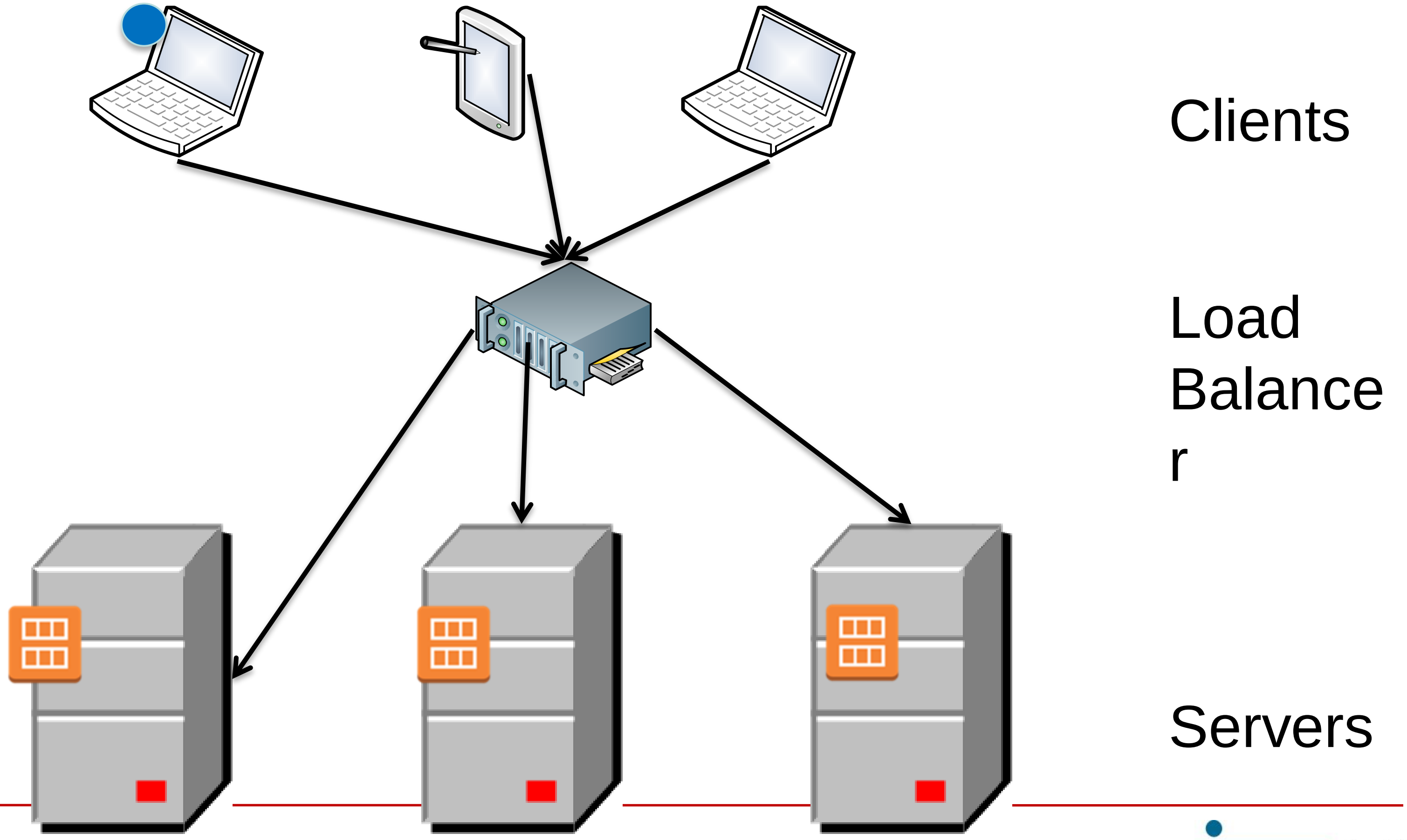
Clients

Load Balance r

Servers

© Len Bass 2018

Message sequence – reply goes directly back to sender



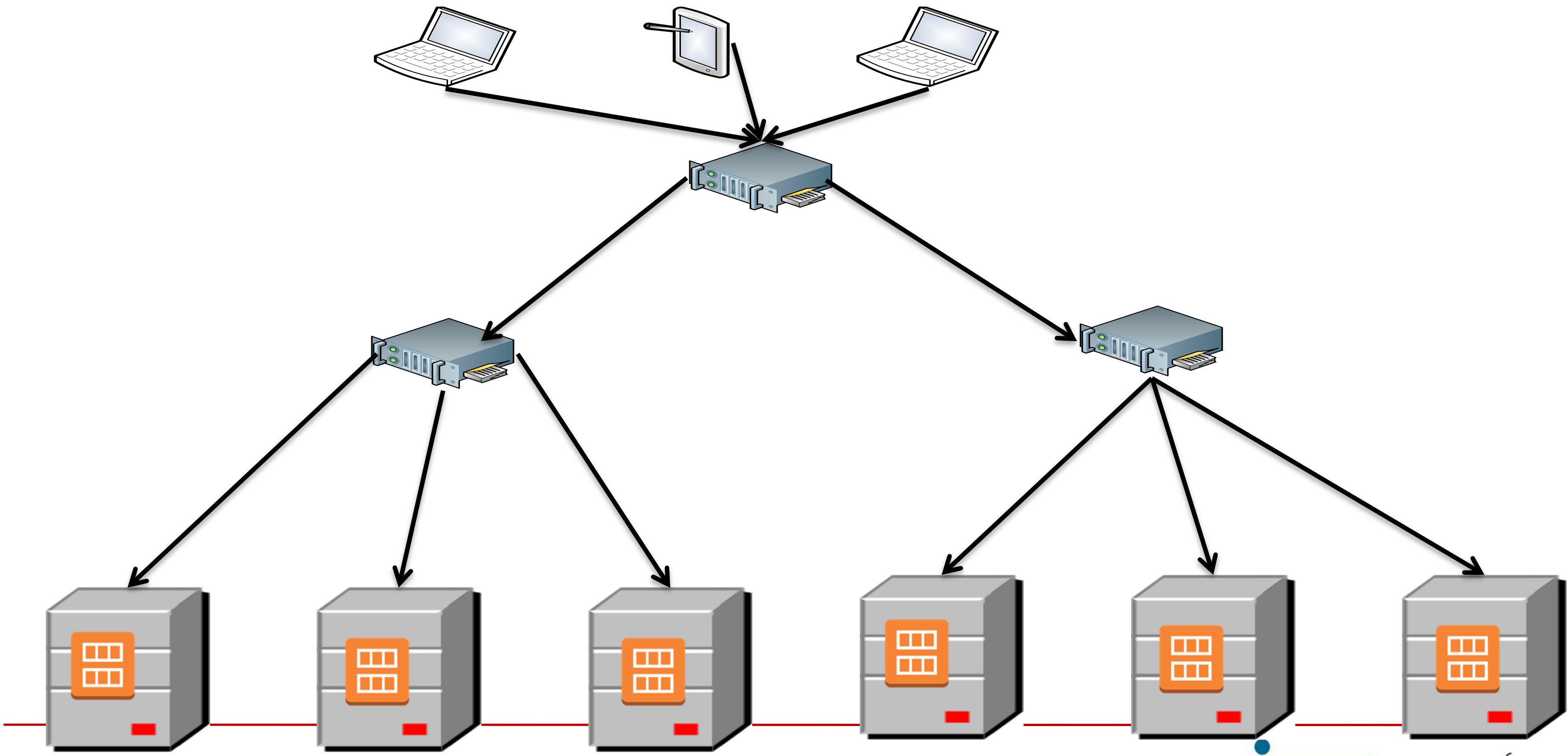
Note IP manipulation

- Server always sends message back to what it thinks is the sender.
- Load balancer changes destination IP but not the source. Then reply goes directly back to client
- Load balancer (now acting as a proxy) can change origin as well. In this case, reply goes back to load balancer which must change destination (of reply) back to original client.

Routing algorithms

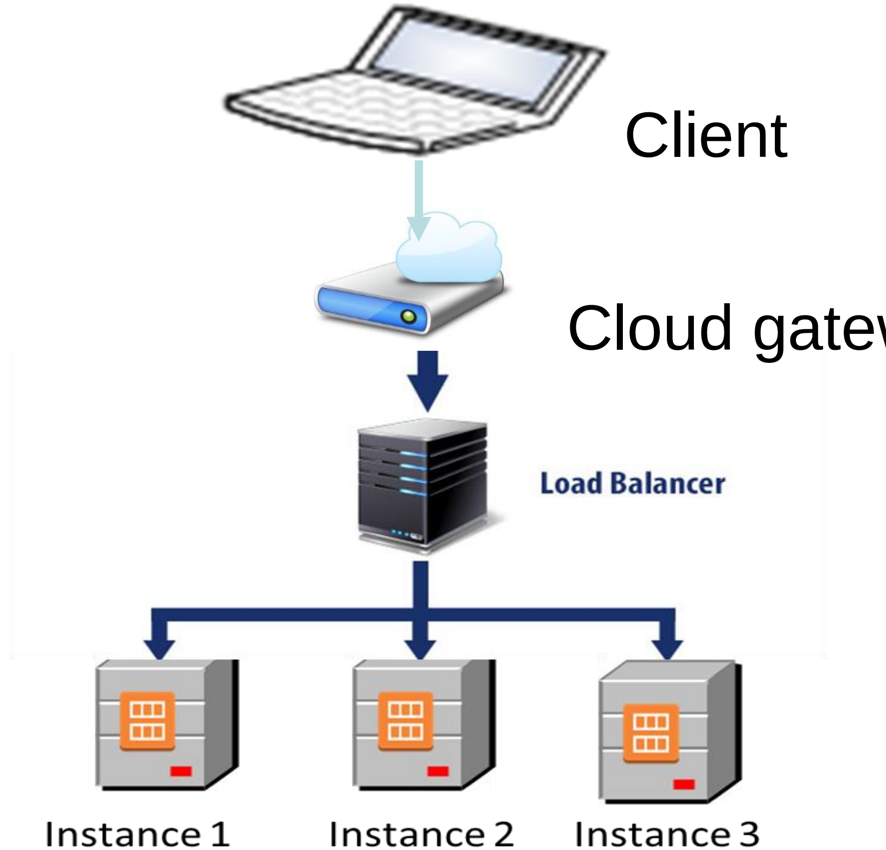
- Load balancers use variety of algorithms to choose instance for message
  - Round robin. Rotate requests evenly
  - Weighted round robin. Rotate requests according to some weighting.
  - Hashing – IP address of source to determine instance. Means that a request from a particular client always sent to same instance as long as it is still in service.
- Note that these algorithms do not require knowledge of an instance's load.

Suppose Load Balancer Becomes Overloaded – Load Balance the Load Balancers



Combining pictures

- IP payload address is modified multiple times before message gets to server
- Return message from instance to client may go through the gateway (proxy) or may go directly back to the client.



Summary

- The cloud consists of regions with availability zones
- A new VM is placed in an availability zone in a region
- Instances in the cloud can fail.
  - Must be detected
  - Long tail is a cloud phenomena that developers must be aware of.
- Load balancers distribute requests among identical servers.

Carnegie Mellon

institute for SOFTWARE RESEARCH

DevOps: Engineering for Deployment and Operations

The cloud 2

Overview

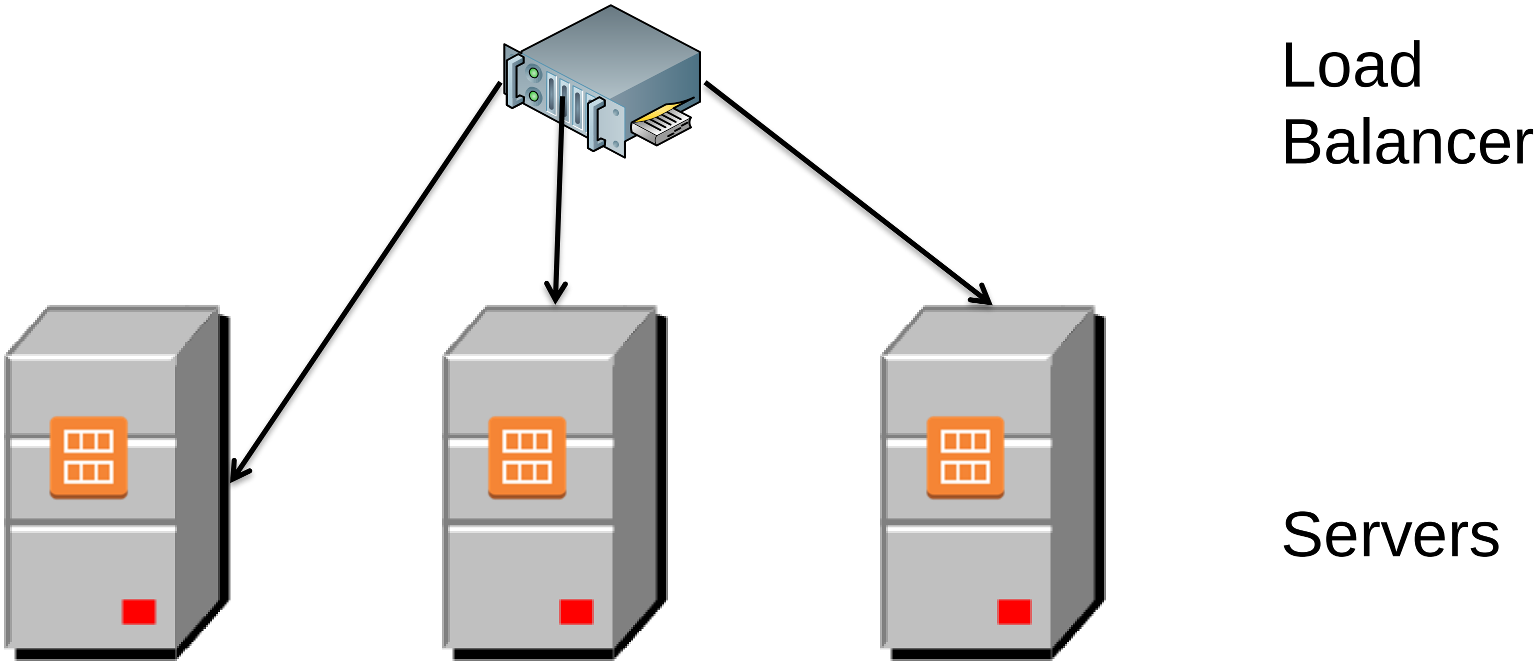
- **Autoscaling**
- Distributed Coordination

Suppose servers become overloaded

- As load grows, existing resources may not be sufficient.
- Autoscaling is a mechanism for creating new instances of a server.
- Set up a collection of rules that determine
  - Under what conditions are new servers added
  - Under what conditions are servers deleted

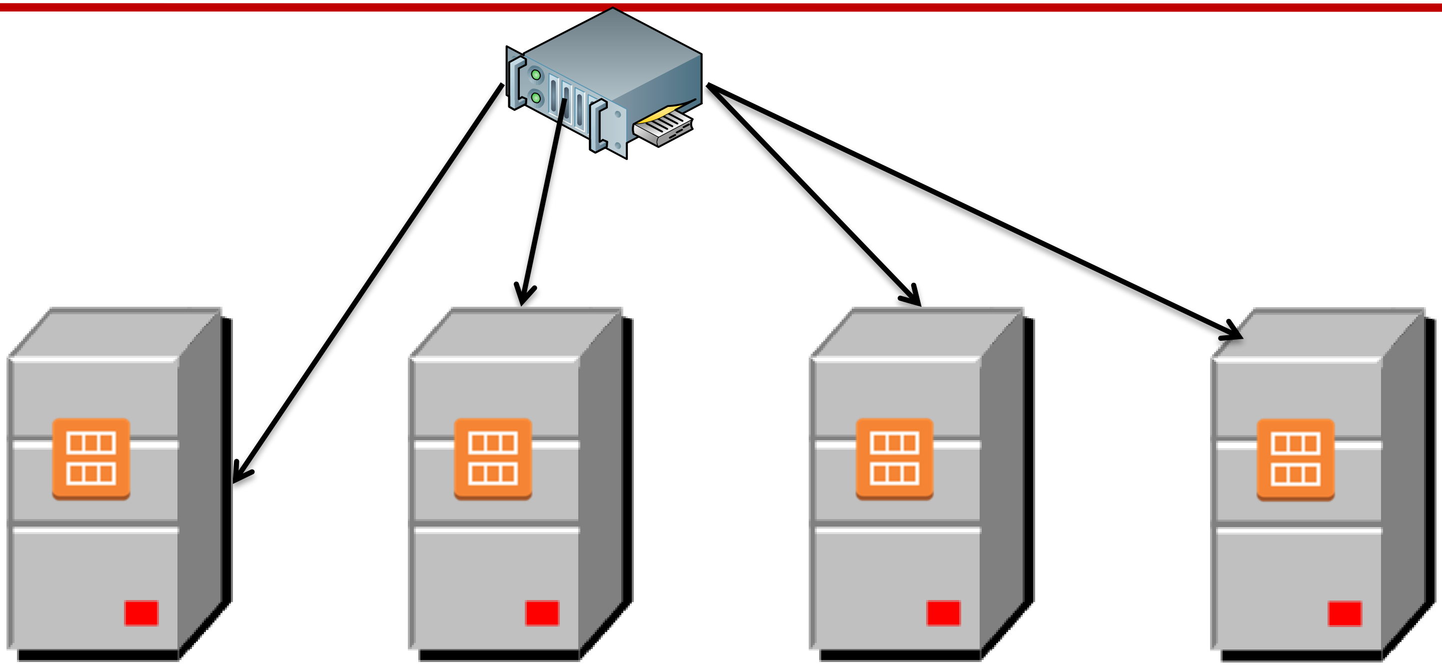
© Len Bass 2018

First there were three servers



© Len Bass 2018

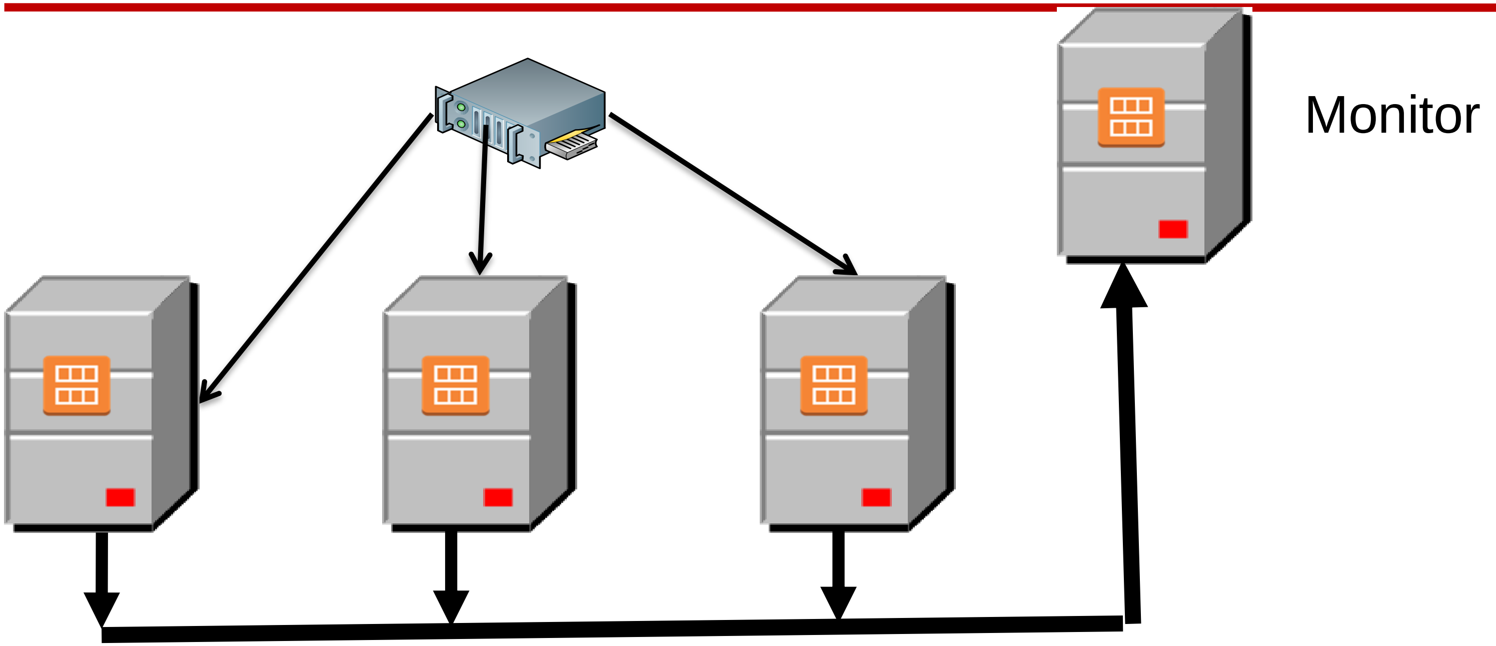
Now there are four



Issues:

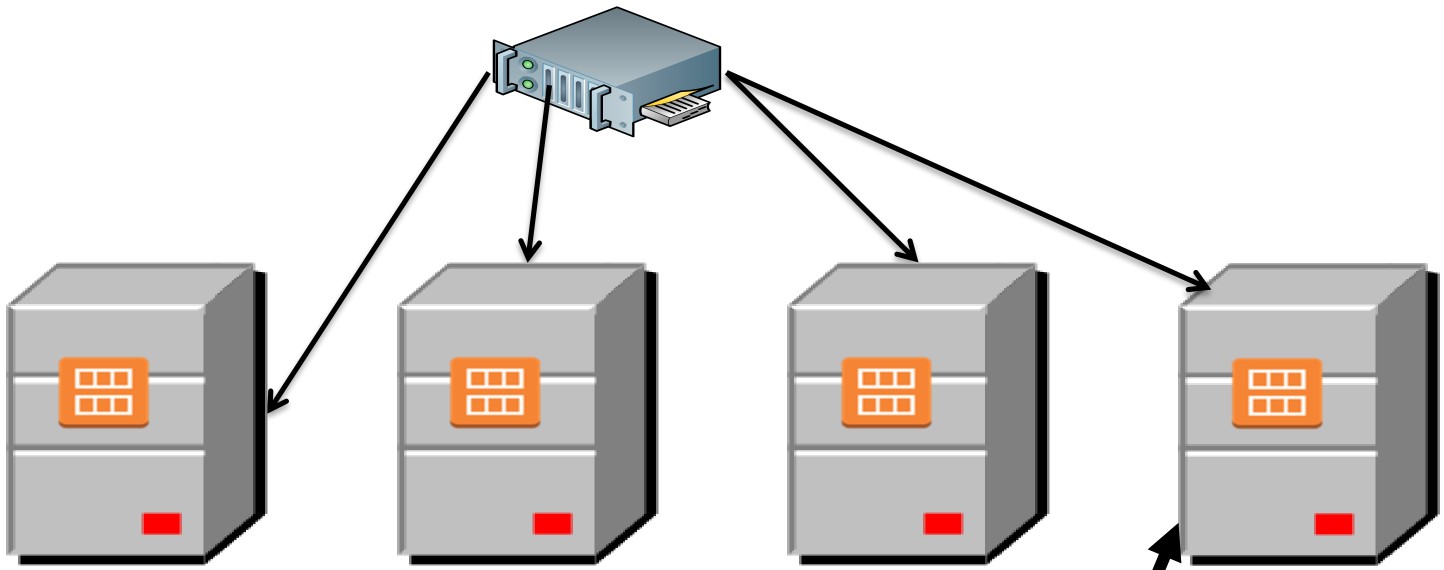
- What makes the decision to add a new server?
- How does the new server get loaded with software?
- How does the load balancer know about the new server?

Making the decision

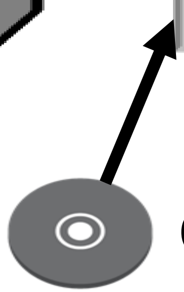


Each server reports its CPU usage to a monitor
Monitor has a collection of rules to determine whether to add new server. E.g. one server is over 80% utilization for 15 minutes

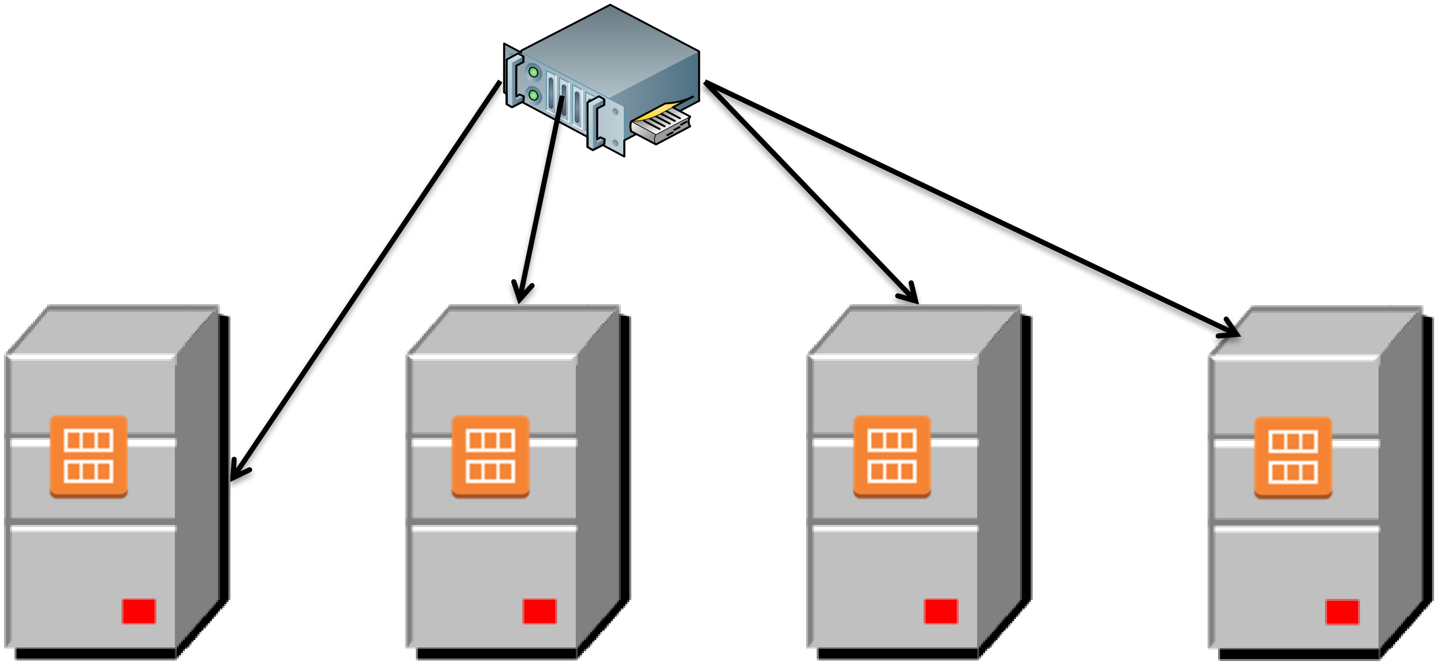
Loading the new server with software



When a new instance is created, it is created based on a launch configuration, security groups, etc


Configuration specifications

Making the load balancer new server aware



The new server is registered with a load balancer by the monitor.

Overview

- Autoscaling
- **Distributed Coordination**
  - Atomicity
  - Locking
  - Consensus algorithms

Atomicity

- Some operations must be done atomically (or appear to be atomic).
- Race conditions result from non atomic operations.
  - Data base updates may result in inconsistent data if not atomic
  - Resources (e.g. memory) may be overwritten if not atomic
  - Interrupt handlers must be atomic
  - etc

How do I achieve (or appear to achieve) atomicity?

- Time stamps?
  - Many protocols involve putting a time stamp on messages for error detection and ordering purposes.
  - Time stamps are often used to identify log messages used for debugging problems.
- Couldn’t time stamps be used to order activities?
- Problem is that clocks drift and so clock time may be different on different computers.

Synchronizing clocks over networks

- Suppose two different computers are connected via a network. How do they synchronize their clocks?
- If one computer sends its time reading to another, it takes time for the message to arrive.
- NTP (Network Time Protocol) can be used to synchronize time on a collection of computers.
  - Accurate to around 1 millisecond in local area networks
  - Accurate to around 10 milliseconds over public internet
  - Congestion can cause errors of 100 milliseconds or more.

© Len Bass 2018

Suppose NTP is insufficiently accurate

- Financial industry is spending 100s of millions of dollars to reduce latency between Chicago and New York by 3 milliseconds.
  - Well within error range of NTP
- GPS time is accurate within
  - 14 nanoseconds (theoretically)
  - 100 nanoseconds (mostly)
- Timestamp messages with GPS time
  - Used by electric companies to measure phase angle
- Atomic clocks
  - Used by Google to coordinate time across all of their distributed systems.

How about using locks?

- One technique to synchronize is to lock critical resources.
- Can lead to deadlock – two processes waiting for each other to release critical resources
  - Process one gets a lock on row 1 of a data base
  - Process two gets a lock on row 2.
  - Process one waits for process 2 to release its lock on row 2
  - Process two waits for process 1 to release its lock on row 1
  - No progress.

More problems with locks

- Locks are logical structures maintained in software or in persistent storage.
- Getting a lock across distributed systems is not an atomic operation.
  - It is possible that while requesting a lock another process can acquire the lock. This can go on for a long time (it is called livelock if there is no possibility of ever acquiring a lock)
- Suppose the virtual machine holding the lock fails. Then the owner of the lock can never release it.

© Len Bass 2018

Locks in distributed systems

- Utilizing locks in a distributed system has a more fundamental problem.
- It takes time to send a message (create a lock, for example) from one computer to another.

© Len Bass 2018

How much time do operations in distributed systems take

- Main memory reference 100 ns
- Send 1K bytes over 1 Gbps network 0.01 ms
- Read 4K randomly from SSD. 15 ms
- Read 1 MB sequentially from memory 0.25 ms
- Round trip within same datacenter 0.5 ms
- Read 1 MB sequentially from SSD 1 ms
  (4X memory)
- Disk seek
  10 ms
  (20x datacenter roundtrip)
- Read 1 MB sequentially from disk 20 ms
  (80x memory, 20X SSD)
- Send packet CA->Netherlands->CA 150 ms

© Len Bass 2018

* dean-keynote-ladis2009_scalable_distributed_google_system

Implications of latency numbers

- State stored in persistent storage (disk or SSD) will take longer to fetch than state stored in memory.
- State stored in a different datacenter will take longer to access than state stored locally, especially across continents.
- Persistent store is typically replicated both for performance (latency) reasons and for availability (failure) reasons.
- => keeping data consistent across different occurrences is important but difficult.

Is there a solution?

- The general problem is that you want to manage synchronization of data across a distributed set of servers where up to half of the servers can fail.
- Paxos is a family of algorithms that use consensus to manage concurrency. Complicated and difficult to implement.
- An example of the implementation difficulty
  - Choose one server as the master that keeps the "authoritative" state.
  - Now that server fails. Need to
    - Find a new master
    - Make sure it is up to data with the authoritative state.

Luckily

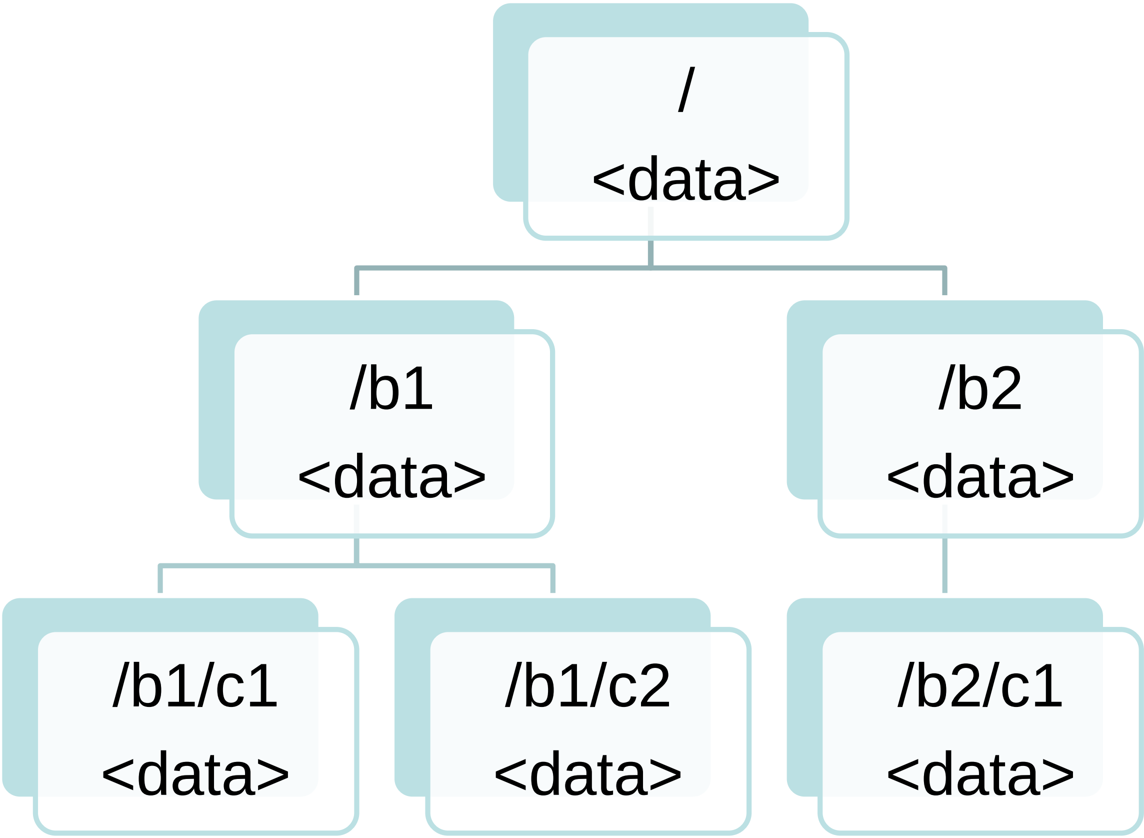
- Several open source systems are now available that
  - Implement Paxos or an alternative consensus algorithm
  - Are reasonably easy to use.
- Sample systems
  - Zookeeper
  - Consul
  - Etcd
  - memcached

© Len Bass 2018

Zookeeper as an example

- Zookeeper provides a guaranteed consistent (mostly) data structure for every instance of a distributed application.
  - Definition of "mostly" is within eventual consistency lag (but this is small).
  - Zookeeper keeps data in memory. This is why it is fast.
- Zookeeper deals with managing failure as well as consistency.
  - Done using consensus algorithm.

Zookeeper znode structure



Zookeeper API

- All calls return atomic views of state – either succeed or fail. No partial state returned. Writes also are atomic. Either succeed or fail. If they fail, no side effects.

Function	Type
create	write
delete	write
Exists	read
Get children	Read
Get data	Read
Set data	write
+ others	

© Len Bass 2018

Example – Lock creation

- Locks have names.
- Client N
  - Create Lock1
    - If success then client owns lock.
    - If failure, then it is placed on waitlist for lock 1 and control is returned to Client N
  - When client finishes, it deletes Lock1
  - If there is a waitlist, then those clients are informed about Lock 1 deletion and they try again to create Lock 1
- If Client N fails, Zookeeper will delete Lock 1 and waitlist clients are informed of deletion

© Len Bass 2018

Other use cases

- Leader election
- Groups membership
- Synchronization
- Configuration

© Len Bass 2018

Summary

- Autoscaling creates new instances based on utilization of existing instances
- Synchronization of data across a distributed system id complicated
- It is difficult because of
  - Timing delays across distributed systems
  - Possible failures of nodes of various types
- Open source systems implement complicated consensus algorithms in a fashion that is relatively easy to use.

© Len Bass 2018

Carnegie Mellon

institute for SOFTWARE RESEARCH

Infrastructure Security

Overview

- **What is security?**
- Cryptography
- Public Key Infrastructure and Certificates
- Key Exchange
- Transport Level Security (TLS)
- Secure Shell (SSH)

Basic security definition

- Short (but memorable form) - CIA
  - Confidentiality
  - Integrity
  - Availability

More precise security definition

- **Authentication**: assurance that communicating entity is the one claimed
- **Access Control**: prevention of the unauthorized use of a resource
- **Data Confidentiality** protection of data from unauthorized disclosure
- **Data Integrity** assurance that data received is as sent by an authorized entity
- **Non-Repudiation** protection against denial by one of the parties in a communication
- **Availability** – resource accessible/usable

© Len Bass 2018

Overview

- What is security?
- **Cryptography**
- Public Key Infrastructure and Certificates
- Key Exchange
- Transport Level Security (TLS)
- Secure Shell (SSH)

© Len Bass 2018

Cryptography

- Three forms of encryption
  - Symmetric
  - Asymmetric
  - One way - hash
- NIST (US National Institute for Science and Technology) certifies algorithms and implementations for encryption.

Data

- One more concept – data
  - At rest – on disk or in memory
  - In transit - on the network

Symmetric encryption

- Use same key for encrypting and decrypting
- 4000x faster than asymmetric encryption
- Suitable for data at rest
- Diffie-Hellman (discussed shortly) is an algorithm for establishing a symmetric key

© Len Bass 2018

Weaknesses of Symmetric encryption

- If attacker discovers key, then has access to all encrypted data
- No authentication with symmetric encryption
- NIST approved algorithm is AES with key lengths of >128 bits

Asymmetric Encryption

- Also known as public/private key encryption
- Uses different keys for encryption and decryption
- Based on difficulty of factoring product of two large primes (NP difficult)
- NIST approved algorithms: DSA, RSA, ECDSA >1024 bits

© Len Bass 2018

Hashing

- A hash is a one way encryption based on a public algorithm with no key
- Not possible (very difficult) to decrypt
- Used to verify integrity of data
  - Passwords: save hash of password but not password. When user enters password, compare to hash to verify.
  - Downloads: publish hash of software available for download. Compare hash of downloaded software. Verifies that software has not been modified.
- NIST approved algorithm is SHA-3.

Overview

- What is security?
- Cryptography
- **Public Key Infrastructure and Certificates**
- Key Exchange
- Transport Level Security (TLS)
- Secure Shell (SSH)

Public Key Infrastructure (PKI)

Message → Encoder → Gibberish → Decoder → Message

- PKI is based on public and private keys
- Public key is known to everyone
- Private key is known only to you
- Messages encrypted with public key can be decrypted by private key (and vice versa)
- Message sent *to* you is encrypted with your public key. Only you can read it
- Message sent *by* you is encrypted with your private key. Decrypting it with your public key guarantees that you sent it

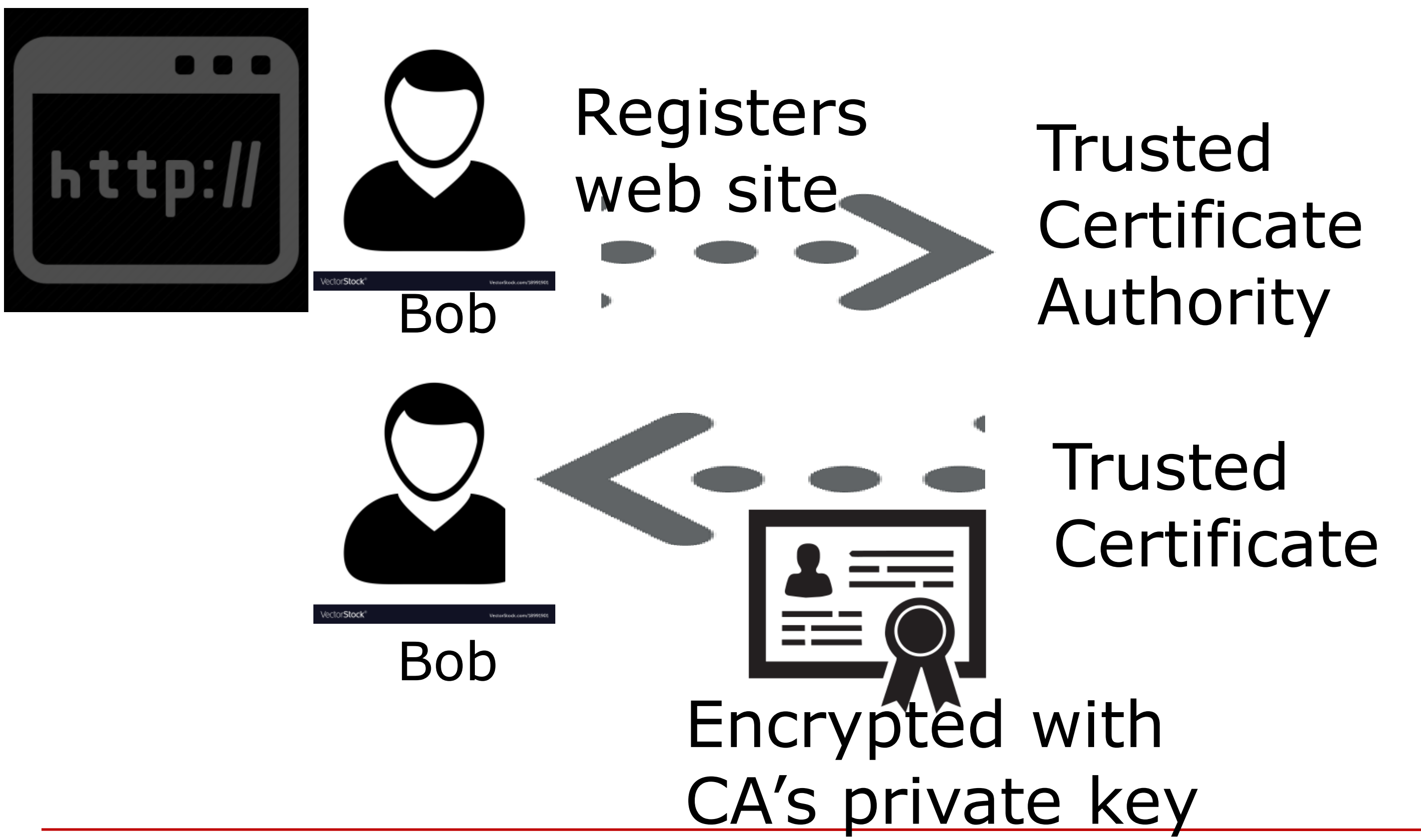
© Len Bass 2018

Certificates

- Certificates are used to establish that a web site is what it claims to be.
- Certificates are based on PKI
- Three important elements of a certificate
  - URL of web site that has been certified
  - Signature of a trusted certificate authority
  - Certificates are encrypted

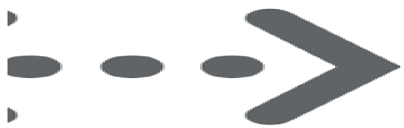
© Len Bass 2018

Registering with Certificate Authority (CA)



Accessing Web Site

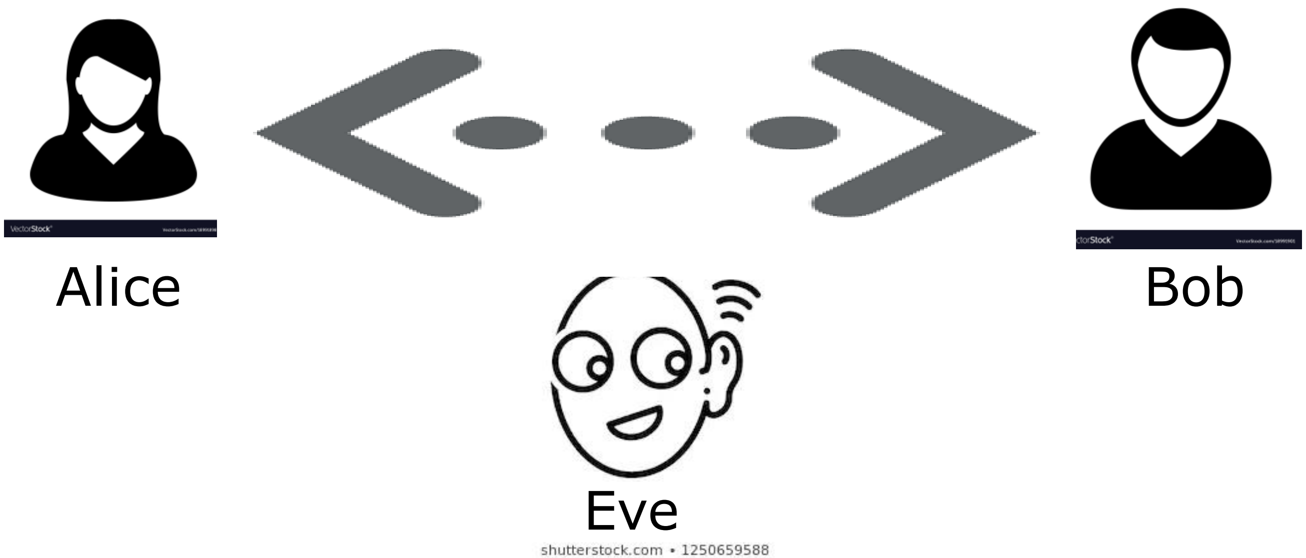
Alice



Accesses Bob's web site

Alice decrypts using CA's public key and knows she is talking to Bob's web site

Key exchange

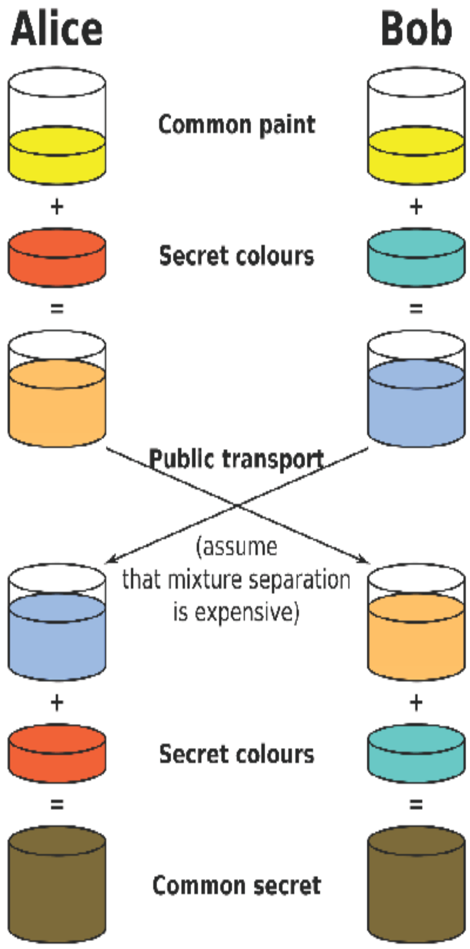


Alice and Bob wish to communicate securely even though Eve is eavesdropping.

Diffie-Hellman algorithm allows this.

Diffie-Hellman (intuitively)

- Alice and Bob agree on a common color
- Each chooses a secret color
- Each mixes their secret color with the common color
- Each sends their mixture to the other
- Each now adds their secret color
- Alice and Bob end up with the same color but decoding it is difficult
- This color is the shared key for symmetric encryption



Overview

- What is security?
- Cryptography
- Public Key Infrastructure and Certificates
- Key Exchange
- **Transport Level Security (TLS)**
- Secure Shell (SSH)

Man in the middle attack

- You are in the airport scanning for an available ISP
- You find “freewifi” and get an IP address from them.
- “freewifi” may be an attacker
- “freewifi” can modify messages to spoof you and steal your credentials

© Len Bass 2018

TLS

- TLS (Transport Layer Security) is the basis for https
- Builds on Diffie-Hellman and Public Key Infrastructure
- Thwarts man-in-the-middle attacks

© Len Bass 2018

TLS protocol - 1

- You wish to access a web site using https
- Handshake to determine which version of the protocol to use
- Web site sends certificate to you to demonstrate authenticity. Certificate is encrypted using Certificate Authority private key.
- Your OS has been pre-loaded with public keys of major CAs so you can decrypt certificate.

TLS protocol – 2

- Diffie-Hellman is used to establish secure key
- Information being exchanged is encrypted/decrypted using this key
- Once session is terminated, key is discarded.
- If you reconnect to the same web site the protocol is used to establish a different secure key

Thwarting man in the middle

- Man in the middle may see all messages but
  - Credential is encrypted so it cannot be modified
  - Diffie-Hellman protects against eavesdropper (the man in the middle)
  - Your communication with web site is encrypted using key unknown to man in the middle

Overview

- What is security?
- Cryptography
- Public Key Infrastructure and Certificates
- Key Exchange
- Transport Level Security (TLS)
- **Secure Shell (SSH)**

© Len Bass 2018

SSH

- Secure Shell (SSH) is a standard protocol and supporting software that enables the control of one computer remotely from another
- Uses public/private key but SSH is unrelated to PKI and TLS
- SSH has a concept of “known addresses” that allows logging into remote computer without a password.
- SSH is used by tools to provision and manage collections of computers.

© Len Bass 2018

TLS vs SSH

- TLS allows communication between two arbitrary parties using PKI
- SSH allows communication where one party knows the IP address it wishes to communicate with
- Could TLS be used instead of SSH? Yes, but:
  - They have different historical roots and SSH is very embedded in practice
  - Using TLS would require having certificates for many more machines.

© Len Bass 2018

Summary

- Security is CIA + authentication + authorization
- Three different encryption styles.
- PKI is based on Asymmetric encryption and is the basis for certificates to validate web sites.
- Diffie-Hellman supports secure communication even when there is an eavesdropper
- TLS builds on Diffie-Hellman to thwart man in the middle attacks
- SSH allows one computer to control another. Used heavily in operations

© Len Bass 2018

Questions?

Basic Tools

Overview

- **Version Control**
- Configuration Management Tools
- Provisioning

Version Control System (VCS)

- Maintains textual information
- Shared among team members
- Centralized or distributed
- Three functions
    1. A check-out/check-in process
    2. A branch/merge process
    3. Tagging or labeling versions

© Len Bass 2018

Centralized version control system

- A centralized VCS has a central repository.
- Users must connect to that central repository to check in or check out files. I.e. internet connection is required.
- System can maintain knowledge of who is working on which files.
  - Allows informing team members if another member checks out a file
  - Allows locking of files or locking of check in.
- Subversion is common centralized VCS.

© Len Bass 2018

Distributed VCs

- A distributed VCS also has a central repository but interactions are different from centralized VCS
- User gets a copy of the repository for local machine
- Check in/check out of a file is from local copy.
- Does not require internet connection to check out
- System has no knowledge of which team member is working on which file.
- Git is common distributed VCS.

Check in/Check out

- Check out results in a local copy
- User can modify local copy.
- Check in copies it back to repository – local or central.
- New version gets new number.
- VCS can perform style or other checks on check-in

Branch process

- Repository is structured as a tree with branches
- Files exist on a branch
- New branches can be created with a duplicate set of files. Allows for different tasks on same file. E.g. adding features and performing bug fixes. These actions might be done on different branches.
- Each branch has a purpose and the files on one branch are isolated from the files on a different branch.

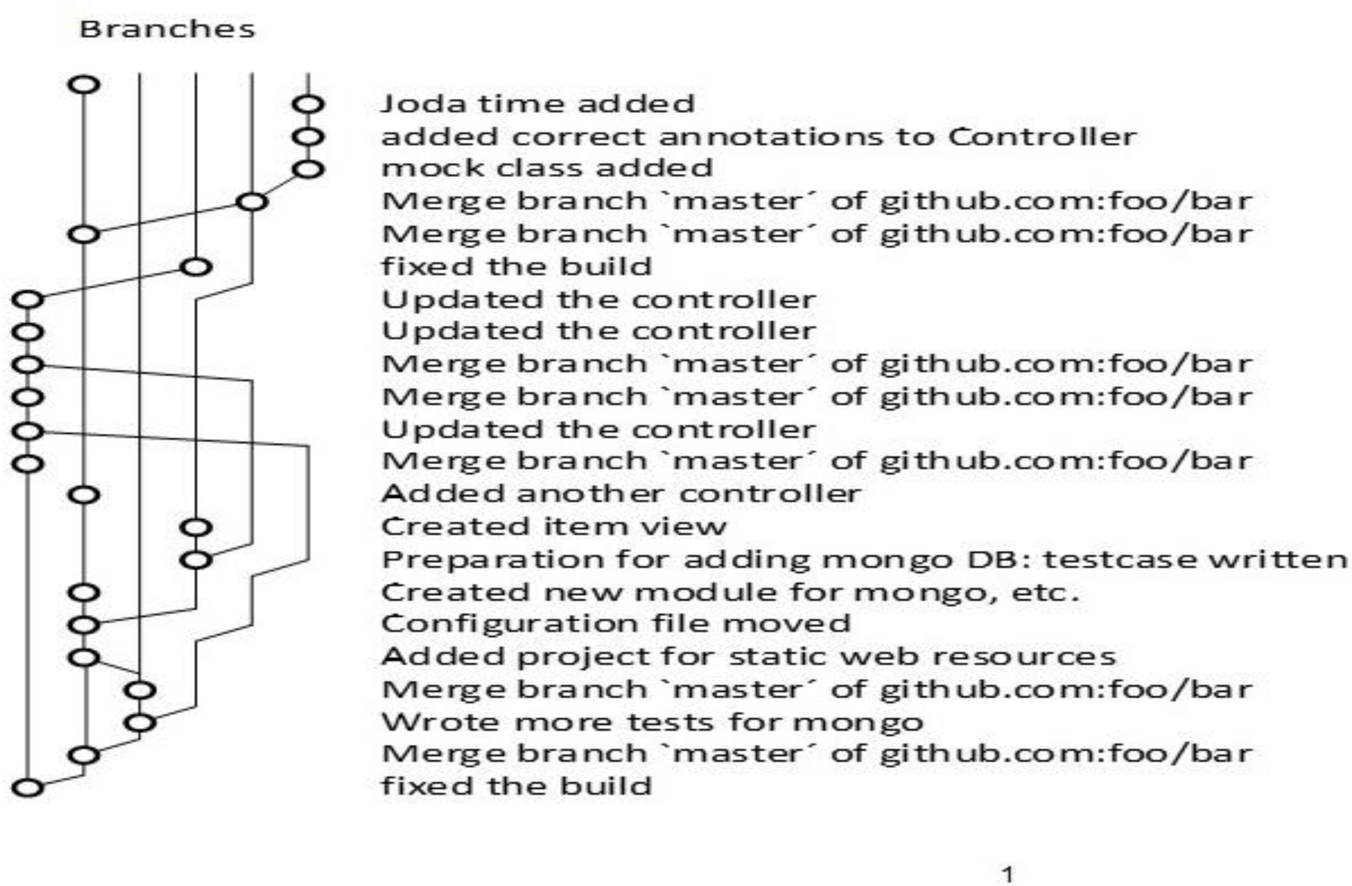
© Len Bass 2018

Merge process

- Two branches can be merged.
- Differences in two versions of the same file must be resolved.
- Best practice is to merge frequently since reduces number of differences that must be resolved.

© Len Bass 2018

Sample branch structure



© Len Bass 2018

Best practices for version control

- Use a descriptive commit message
- Make each commit a logical unit
- Avoid indiscriminate commits
- Incorporate others' changes frequently
- Share your changes frequently
- Coordinate with your co-workers
- Remember that the tools are line-based
- Don't commit generated files

© Len Bass 2018

Overview

- Version Control
- **Configuration Management Tools**
- Provisioning

Configuration Management (CM) Systems

- The purpose of a configuration management system is to maintain consistency across a set of machines.
- Common tools are:
  - Chef
  - Puppet
  - Ansible

Actions of a CM tool

- A CM tool resides on a server
- Controls collection of other machines – virtual or physical
  - Identified by IP addresses and grouped by functions. E.g. Web server, Development platform, DB server
  - Controls them through SSH (or TLS) from CM server to clients
- Extensible through plug-ins

Specification

- Specification of actions is done through scripting language
- E.g. “make sure all web servers have latest patch for nginx 15.8”
- Specification should be version controlled.

© Len Bass 2018

Overview

- Version Control
- Configuration Management Tools
- **Provisioning**

Manual Provisioning

- Given a virtual machine or a container, you can provision (add software) manually by using package manager, e.g. apt-get in Ubuntu.
- Problems with manual provisioning are
  - Errors in specification
  - Inconsistencies among team members
  - Repeatability
  - No history

Consistency is important

- Consistency on platforms reduces error possibilities.
  - Different team members should develop on the same platform
  - Development platform should be consistent with the production platform
  - Updates should be applied to all machines in a fleet automatically (configuration management tools)
  - Replicas across data centers should have same software installed down to version and patch numbers.

© Len Bass 2018

Provisioning tools

- A provisioning tool such as Vagrant or Cloud Formation (for AWS) provides a scripting basis for establishing environment
- Allows one command provisioning. E.g. Vagrant up
- Maintains consistency among team members
- Can be version controlled

Summary

- Version control systems enable controlled sharing and modification
- Configuration management systems maintain consistency of software and versions across classes of machines
- Provisioning tools simplify the creation of an environment

Questions?

Agenda PM

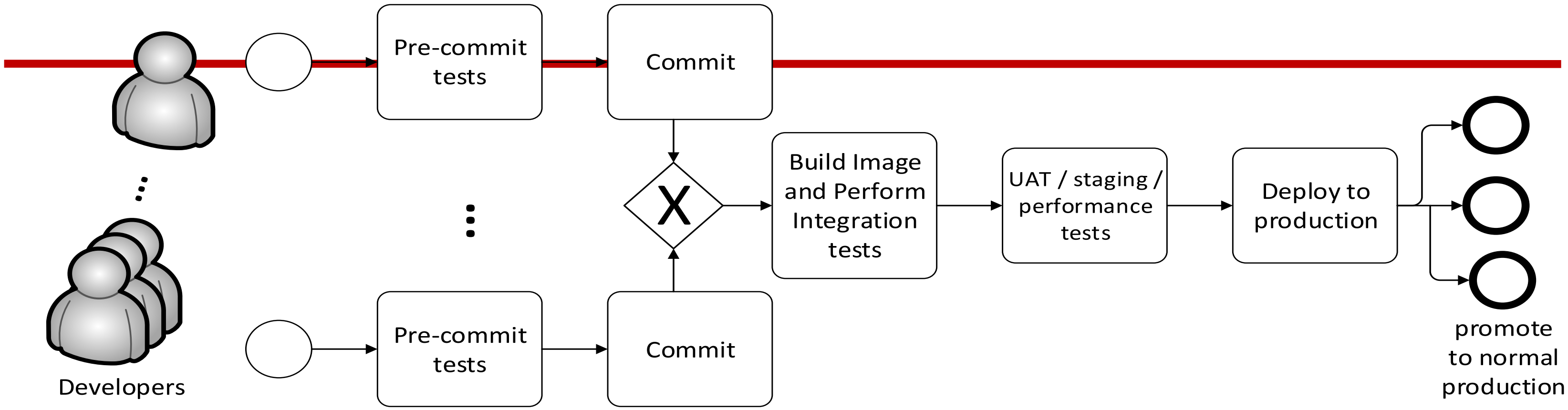
1:00-1:30	Introduction to deployment pipeline
1:30-2:30	Vagrant Assignment
2:30-3:00	Containers
3:00-3:15	Break
3:15-3:45	Docker assignment
3:45-4:45	Microservice architecture
4:45-5:00	Wrap up

Carnegie Mellon

institute for SOFTWARE RESEARCH

Deployment Pipeline

Deployment pipeline



- Developer creates and tests code on local machine.
- Checks code into a version control system
- Continuous integration server (CI) builds the system and runs a series of integration tests.
- After passing the tests, the system is promoted to a staging environment where it undergoes more tests including performance, security, and user acceptance tests.
- After passing those tests, the system is promoted to provisional production where it undergoes even more tests.
- The system is finally promoted to normal production but the tests do not necessarily stop.

Goals during deployment pipeline

- Team members can work on different versions of the system concurrently
- Code developed by one team member does not overwrite code developed by others
- Code produced by one team can be integrated with code produced by other teams
- Code is the same during different stages
- Different stages serve different testing purposes
- Different stages are isolated from each other
- A deployment can be easily rolled back if there is a problem.

Desirable qualities of deployment pipeline

- Traceability
- Testability
- Tooling
- Cycle time

Traceability

When any code gets into production, it must be reconstructable

- All components that go into the executable image can be identified
- All tests that were run can be identified
- All scripts used during the pipeline process can be identified.

When a problem occurs in the system when it goes into production, all of the elements that contributed to the system can be traced.

Testing

- Every stage (except production) has an associated test harness.
- Tests should be automated.
- Types of tests are
  - Sunny day tests
  - Negative tests
  - Regression tests
  - Static analysis

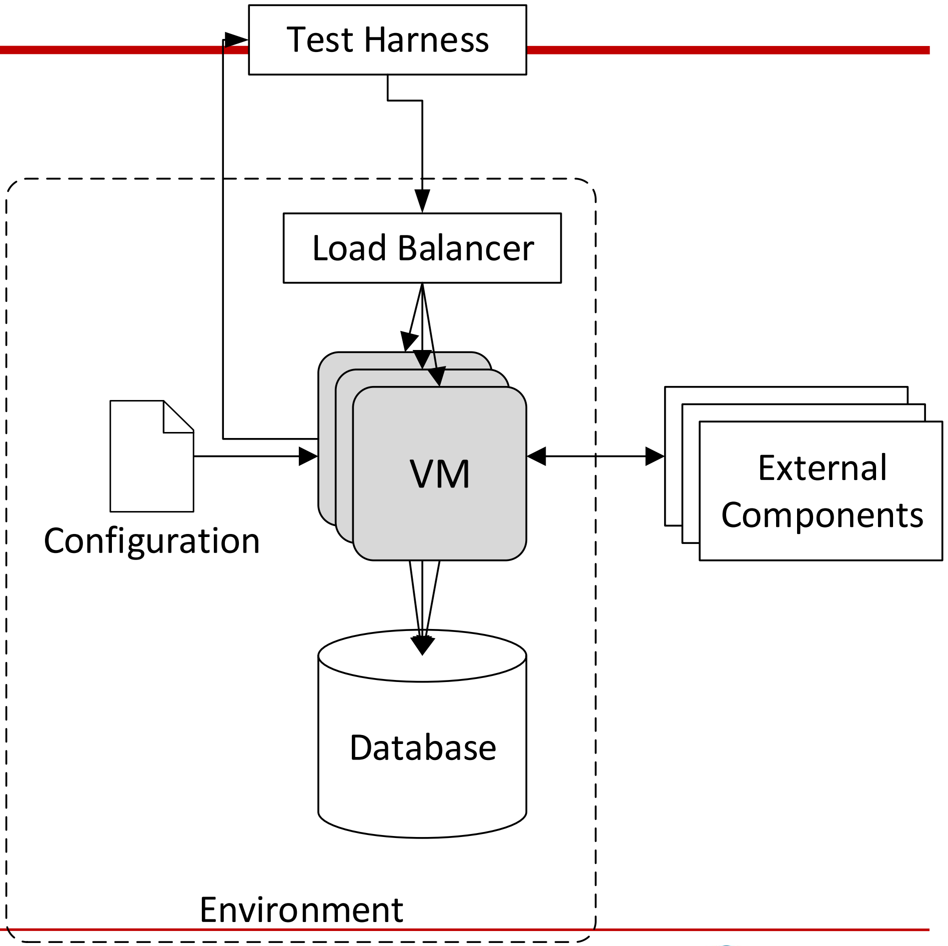
Database test data

- Every stage has a database
- Minimal during development
- More substantial during build
- Realistic during staging
- After each test, database must be reconstituted to ensure repeatability
- Personally Identifiable Information (PII) must be obscured.

© Len Bass 2018

Test Harness

A test harness generates inputs and compares outputs to verify the correctness of the code.



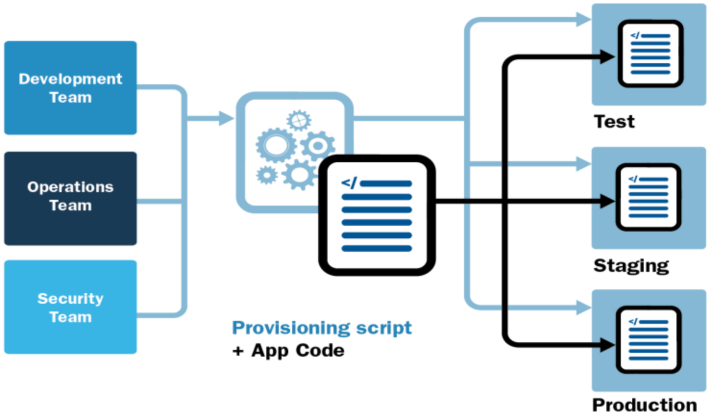
Tooling

- Continuous Integration tool – builds image and runs test – e.g. Jenkins
- Deployment tool – places images into production. E.g. Spinnaker, Ansible, Chef
- These tools must execute on a platform
- Vagrant is a tool to help provision the platform

© Len Bass 2018

Scripts for pipeline tools

Scripts for pipeline tools are an example of infrastructure as code (IaC). They should be version controlled and tested just like application code.

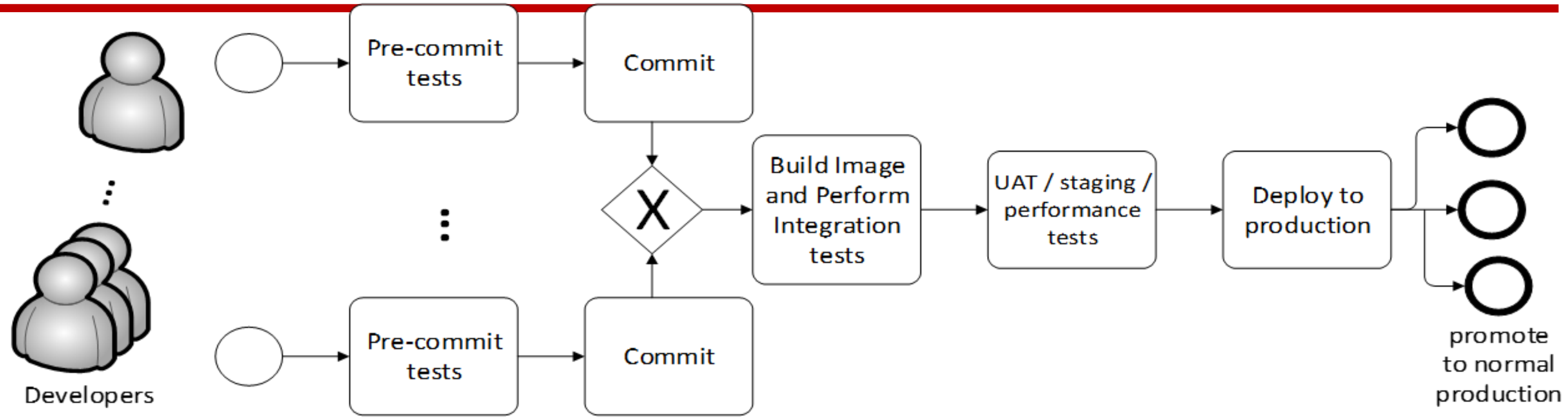


Cycle time

- Cycle time is the time between the commit and the placing into provisional production.
- Systems should move through the pipeline quickly
- The time depends on
  - How large are the components that are constructed in the build stage
  - How long does it take to run tests on the system

© Len Bass 2018

Different environments in pipeline



- Development environment
- Build (also called Integration) environment
- Staging environment
- Production environment
- Different organizations may have additional environments defined

Sample pipeline

- A sample pipeline without staging environment can be found at

https://github.com/cmudevops/class-materials/blob/master/deployment%20workflow.pdf

Summary

- A deployment pipeline is a series of stages where each stage has
  - A specific testing purpose
  - An isolated environment
- Each stage in the pipeline is controlled by some set of tools.
  - Scripts for these tools should be version controlled and tested.
- The pipeline has a set of desirable properties that can be used to measure it

Assignment

- Use Vagrant to create two ubuntu virtual machines in VirtualBox where one can ping the other.

© Len Bass 2015

institute for SOFTWARE RESEARCH
© Len Bass 2018

Relevant web sites

- https://docs.vagrantup.com/v2/getting-started/index.html
- http://www.sitepoint.com/getting-started-vagrant-windows/

institute for SOFTWARE RESEARCH

Vagrant instructions

https://github.com/cmudevops/workshop-instructions

© Len Bass 2018

Discussion of Assignment

- Compare doing assignment with Vagrant vs how you would do it manually.

© Len Bass 2018

Carnegie Mellon

institute for SOFTWARE RESEARCH

Containers

Overview

- **Container definition**
- Layers and deployment
- Container repositories
- Clusters and orchestration
- Serverless Architecture

© Len Bass 2018

Goal

- Want to package executable such that it
  - Is isolated from other executables
  - Is lightweight in terms of loading and transferring
- Containers provide such a packaging
- Containers require run time engine which provides many services

Containers

Process
- Isolate address space
- No isolation for files or networks
- Lightweight

Container
- Isolate address space
- isolate files and networks
- Lightweight

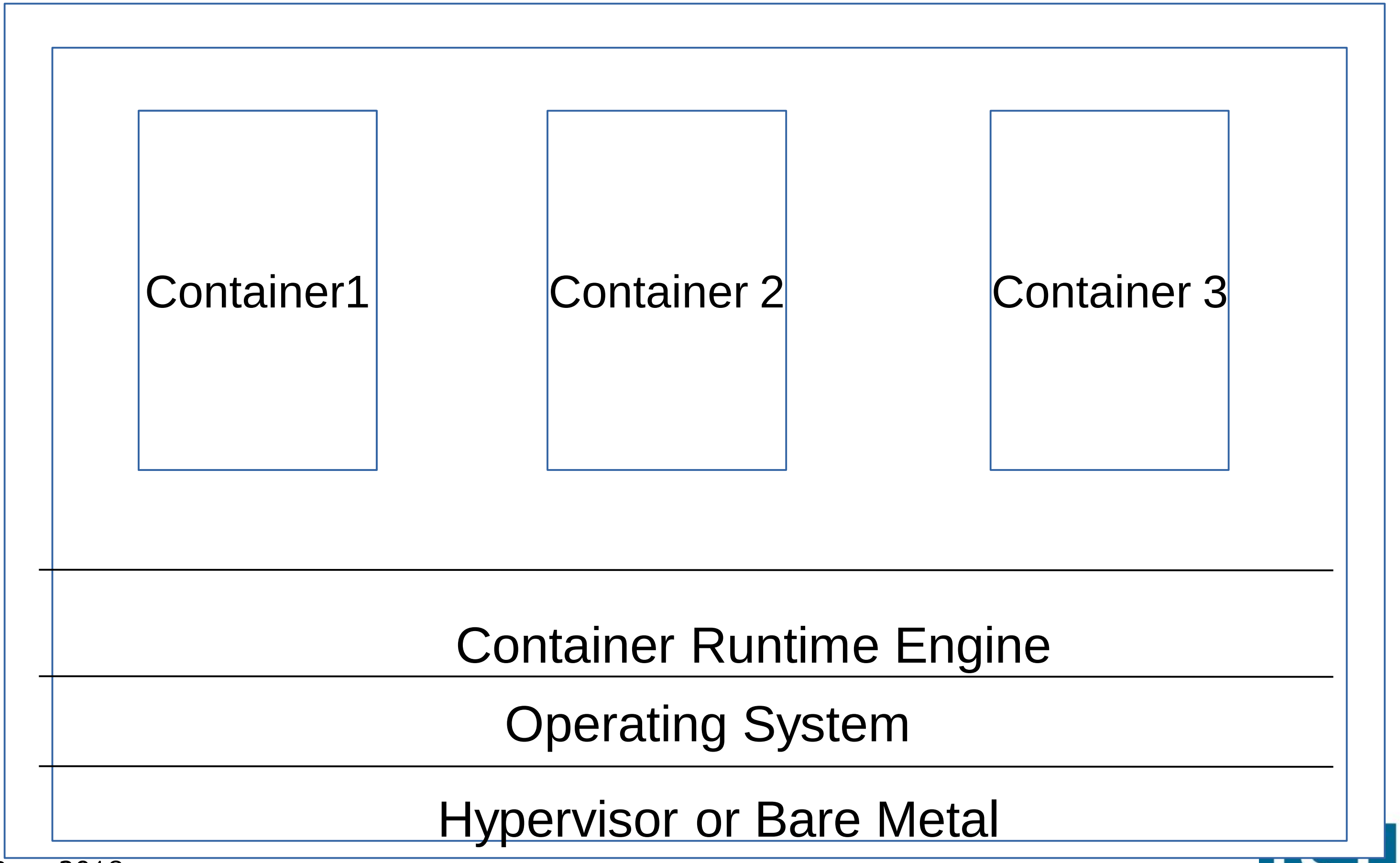
Virtual Machine
- Isolate address space
- isolate files and networks
- Heavyweight

© Len Bass 2018

Container images

- As with VMs,
  - A container image is a set of bits on a disk.
  - A container is an executing entity
- Terminology is not always followed.
  - “Container” is used to refer both to images and executing entities

Container hierarchy



© Len Bass 2018

VMs vs containers

- VMs are virtualized hardware.
  - Each VM can run its own OS.
  - All OSs and apps use same instruction set
  - VM manager is hypervisor
  - Each VM has its own IP address
- Containers are virtualized operating system.
  - Each container can use Its own binaries and libraries
  - All apps use same OS
  - Container manager is, for example, “Docker Engine”
  - Each container has its own IP address
  - Containers have limitations in network and file capabilities

Persistence

- VMs, depending on the Virtual Machine Manager, will generally persist until user takes explicit action or VM fails.
- Some container runtimes, e.g. Lambda, may not persist their containers.
  - Container will not be deleted in the middle of processing a message
  - State should not be stored in container if it will be deleted
  - IP addresses should not be shared if container will be deleted

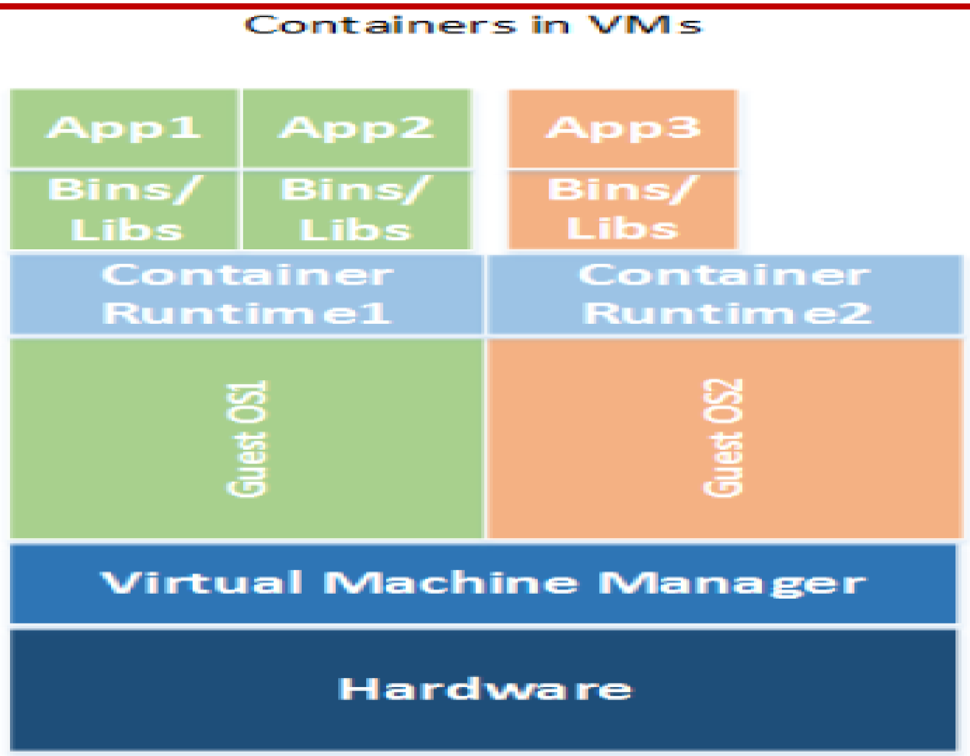
© Len Bass 2018

Portability

- Containers are portable across platforms as long as the container runtime supports the same format.
- The Open Container Initiative has standardized the format of containers. Consequently, runtimes from one vendor will support containers constructed from another vendor
- This allows a container to be moved from one environment to another.
  - Same software is moved
  - Network connections must be established for each environment

© Len Bass 2018

VMs and containers can be combined



Overview

- Container definition
- **Layers and deployment**
- Container repositories
- Clusters and orchestration
- Serverless Architecture

Layers

- A container image is structured in terms of “layers”.
- Process for building image
  - Start with base image
  - Load software desired
  - Commit base image+software to form new image
  - New image can then be base for more software
- Image is what is transferred

© Len Bass 2018

Exploiting layers

- When an image is updated, only update new layers
- Unchanged layers do not need to be updated
- Consequently, less software is transferred and an update is faster.

Speeding up loading across a network

- Loading a VM across a network takes minutes
- Suppose your cloud provider kept your container images local to the physical machine where they are run.
- Then when you invoke the container, it will load in milliseconds, not minutes.

Trade offs - 1

- Virtual machine gives you all the freedom you have with bare metal
  - Choice of operating system
  - Total control over networking arrangement and file structures
  - Time consuming to load over network

© Len Bass 2018

Trade offs - 2

- Container is constrained in terms of operating systems available
  - Linux, Windows, and OSX
  - Provides limited networking options
  - Provides limited file structuring options
  - Fast to load over network

© Len Bass 2018

Overview

- Container definition
- Layers and deployment
- **Container repositories**
- Clusters and orchestration
- Serverless Architecture

© Len Bass 2018

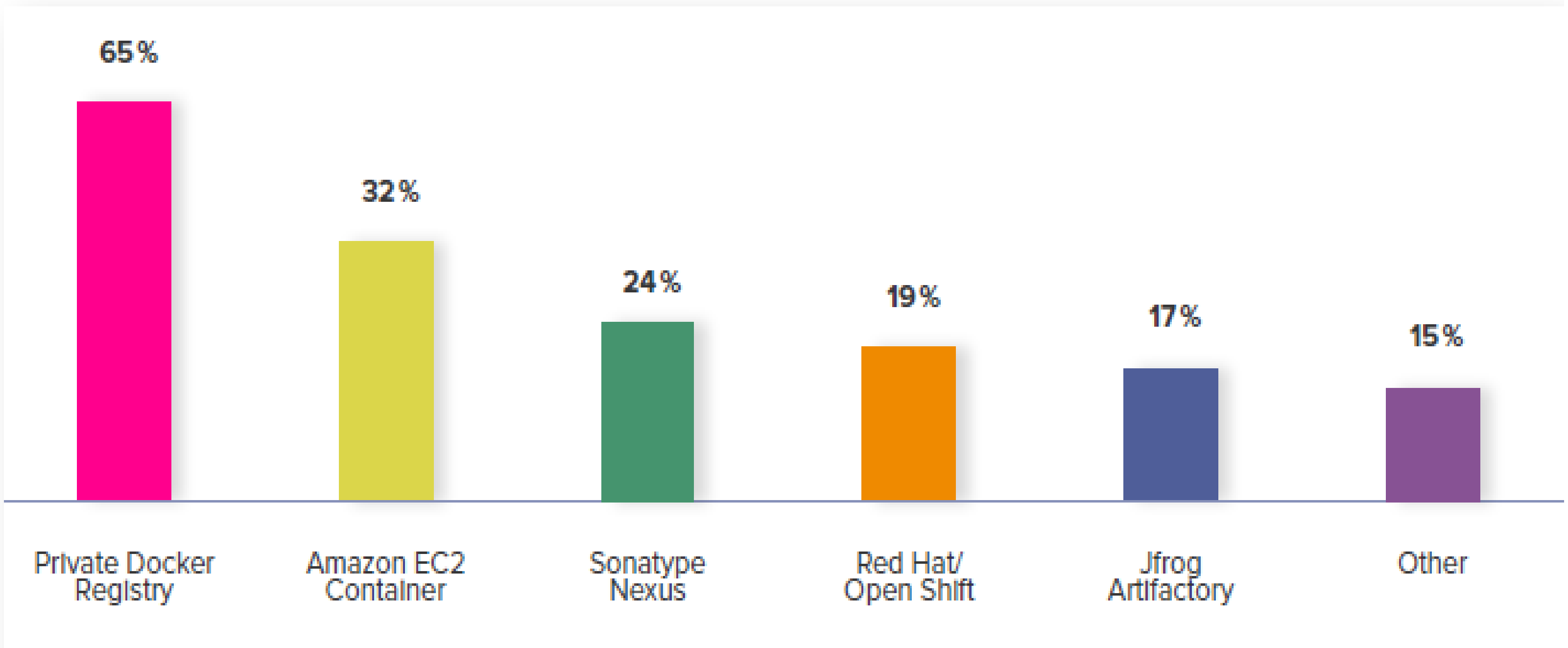
Container repositories

- Container images are typically stored in repositories
- Similar to version control systems
  - Accessible with permissions
  - Push/pull interface
  - Images can be tagged with version numbers
- Docker Hub is a publicly available repository

Integrating with development workflow

- Multiple team members may wish to share images
- Images can be in production, under development or under test
- Private container repository allows images to be stored and shared.
  - Any image can be “pulled” to any host
  - Tagging as “latest” allows updates to be propagated. Pull <image name>:latest gets the last image checked into repository with that name.

Private container registries used by organizations



DevOps Community Survey 2018 - Sonatype

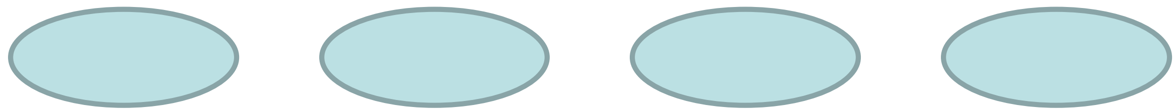
© Len Bass 2018

Overview

- Container definition
- Layers and deployment
- Container repositories
- **Clusters and orchestration**
- Serverless Architecture

Allocation of images to hosts

images



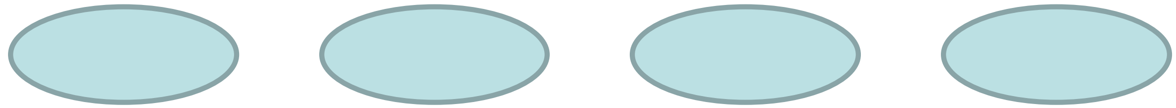
To run an image, the image and the host must be specified

hosts

With basic Docker this allocation must be done manually

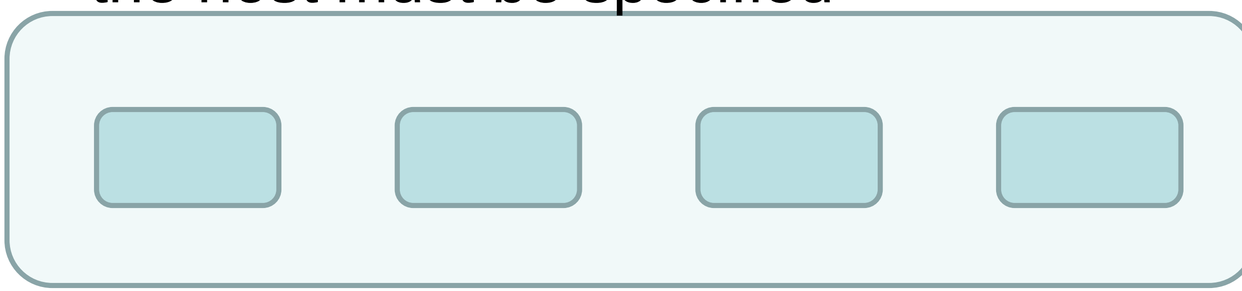
Container orchestrator

image



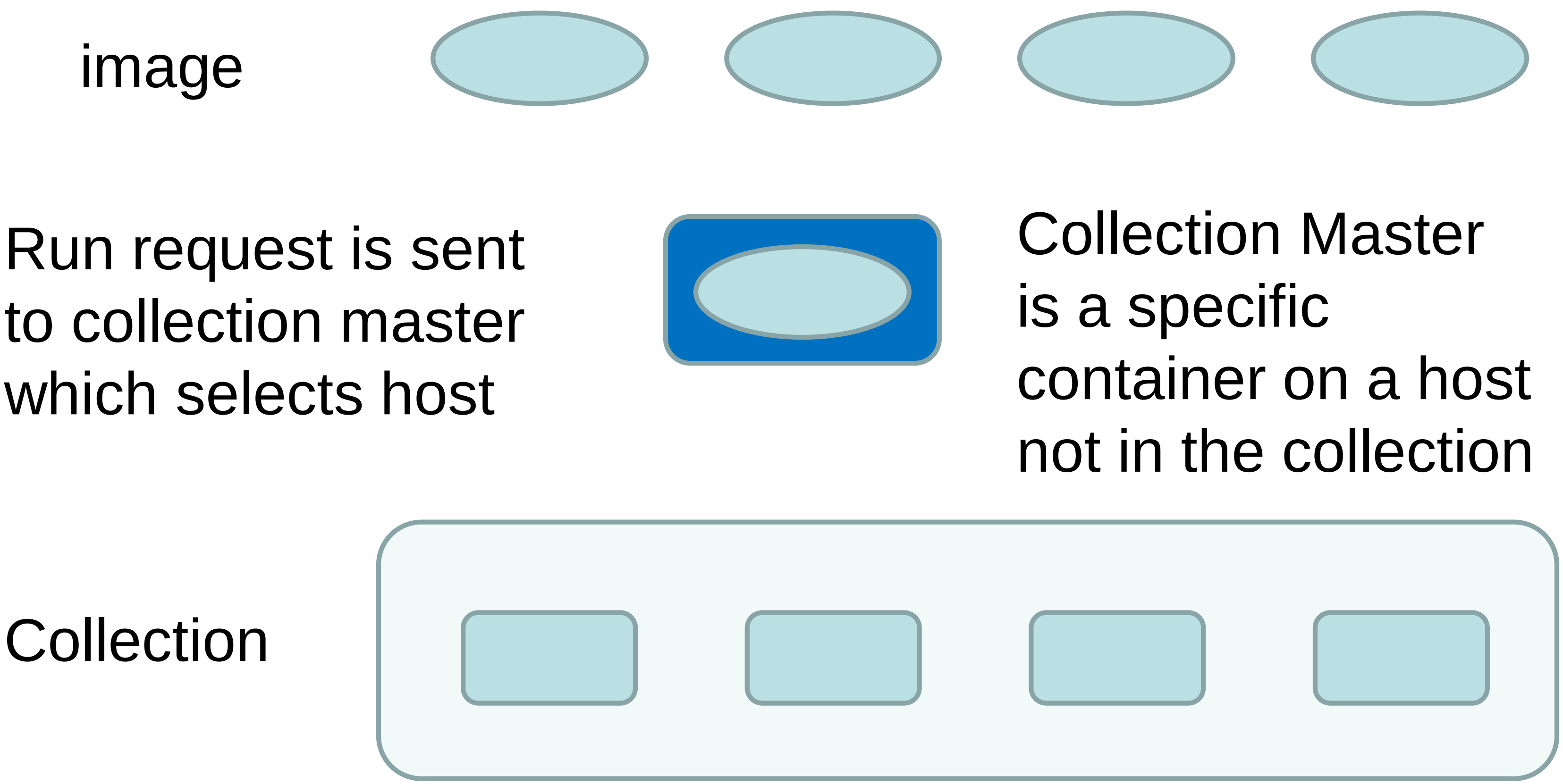
To run an image, the image but not the host must be specified

Orchestrator encapsulates hosts into a collection



A collection looks like a single host from the point of view of allocation but actually consists of multiple hosts

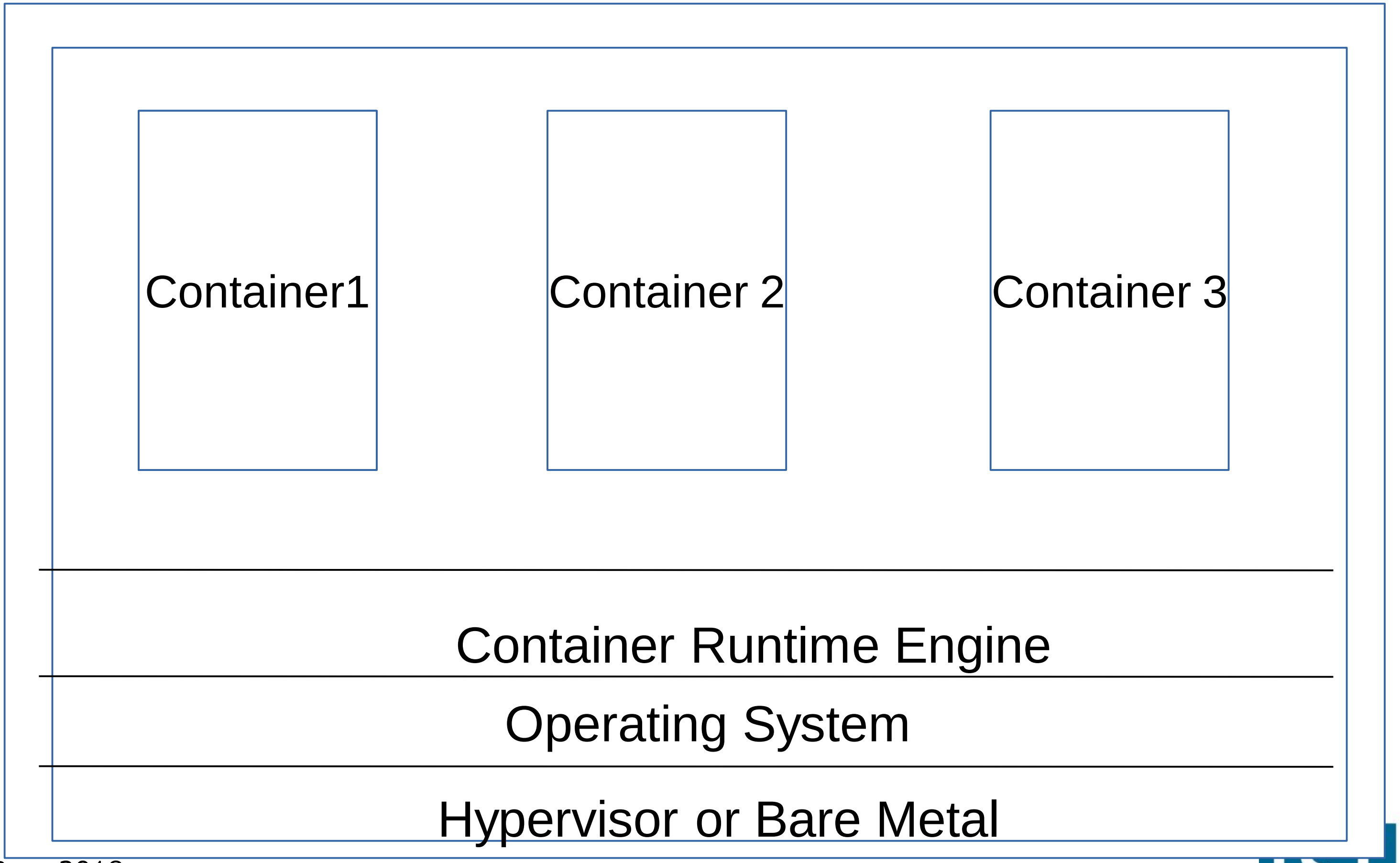
Collection Master



Grouping containers

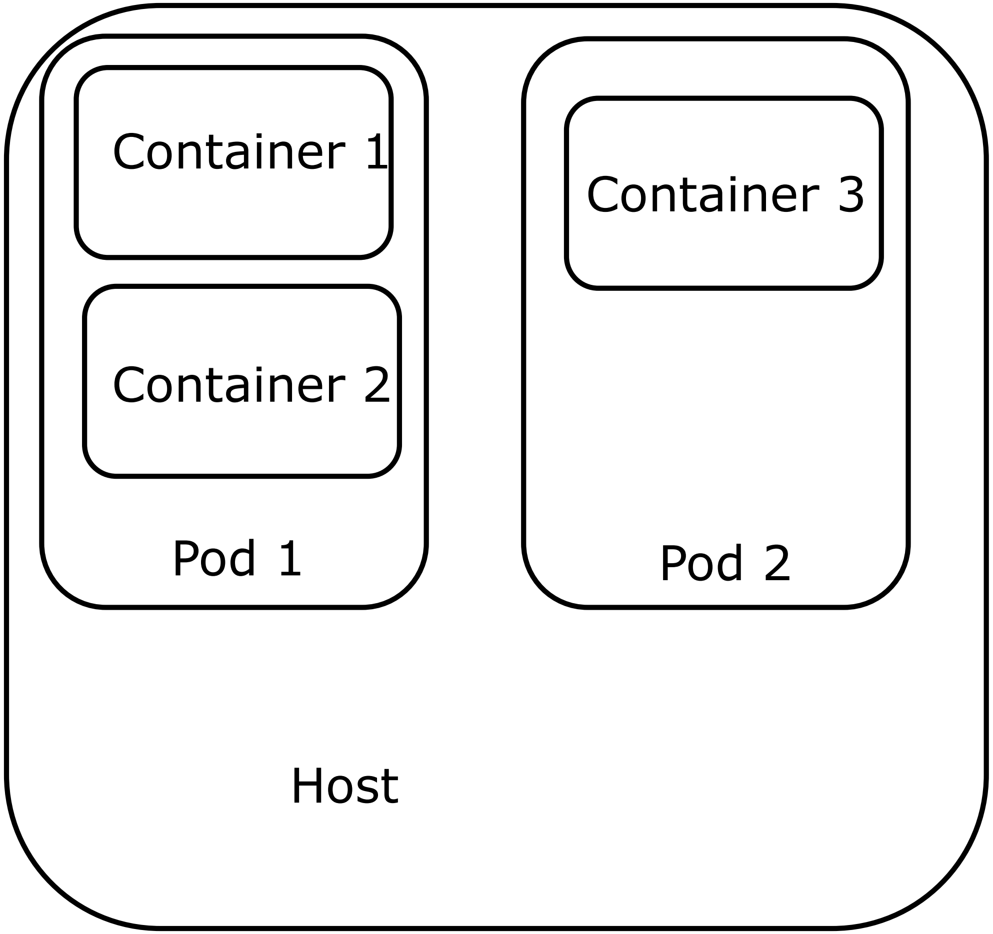
- Up to this point, containers are arbitrarily assigned to available host without regard for communication among containers.
- Suppose two containers communicate frequently. E.g. app container and logging containers. Then you would like them to be allocated into same host to reduce communication time.
- This is rationale for “pods”.
- A pod is a group of containers treated as a single unit for allocation.

Container hierarchy (again)



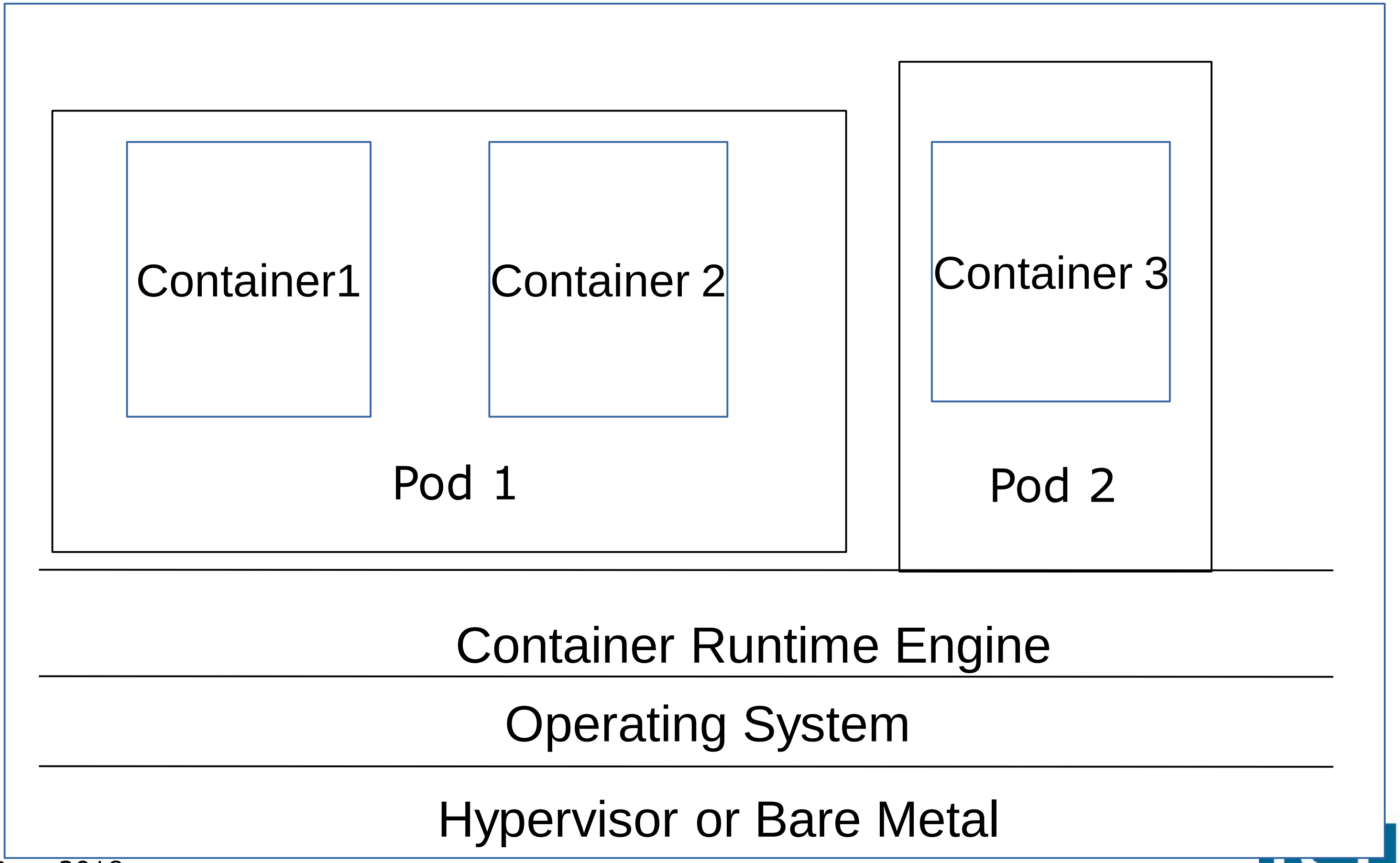
© Len Bass 2018

Adding pods to hierarchy



© Len Bass 2018

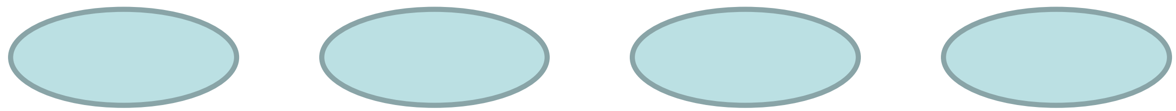
Revisiting container hierarchy



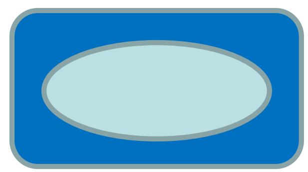
© Len Bass 2018

Collection Master revisited with pods

Images collected into pods

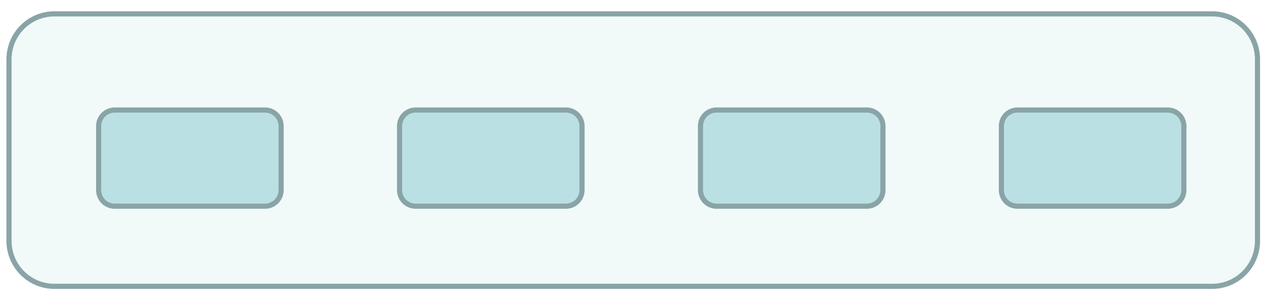


Run request is sent to collection master which selects host



Collection Master is a specific container on a host not in the collection

Collection



© Len Bass 2018

Scaling collections

- Having an instance in a collection be automatically replicated depending on workload is accomplished by utilizing autoscaling facilities of orchestrator
- Kubernetes is a container scaling and orchestration engine that supports pods.

Overview

- Container repositories
- Clusters and orchestration
- **Serverless Architecture**

Serverless

- Cloud providers such as AWS maintain pool of partially loaded containers that only require application specific layer. The term “serverless” is used to describe these partially loaded containers.
- Load in milli secs.
- For AWS, only one request per instance. In AWS, serverless hosts are called Lambda
- Impacts architecture of application.

© Len Bass 2018

Summary

- A container is a lightweight virtual machine
- Docker allows building images in layers and deployment of a new version just requires deploying layers that have changed.
- Containers can be clustered and can be deployed and scaled by cluster.
- Serverless architecture takes advantage of fast load time of container images but with restrictions on usage.

Docker Assignment

Instructions can be found at

https://github.com/cmudevops/workshop-instructions

Carnegie Mellon

institute for SOFTWARE RESEARCH

Microservice Architecture

DevOps processes can be divided into three categories

1. Reduce errors during deployment
2. Reduce time to deploy
3. Reduce time to resolve discovered errors

- Microservice architecture helps with all three categories

© Len Bass 2018

Overview

- **Microservice architecture definition**
- Containers and microservices
- Reducing errors during deployment
- Reducing time to deploy
- Speeding up incident handling

© Len Bass 2018

Definition

- A microservice architecture is
  - A collection of independently deployable processes
  - Packaged as services
  - Communicating only via messages

~2002 Amazon instituted the following design rules - 1

- All teams will henceforth expose their data and functionality through service interfaces.
- Teams [services] must communicate with each other through these interfaces.
- There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.

© Len Bass 2018

Amazon design rules - 2

- It doesn’t matter what technology they[services] use.
- All service interfaces, without exception, must be designed from the ground up to be externalizable.

- Amazon is providing the specifications for the “Microservice Architecture”.

In Addition

- Amazon has a “two pizza” rule.
- No team should be larger than can be fed with two pizzas (~7 members).
- Each (micro) service is the responsibility
  of one team
- This means that microservices are
  small and intra team bandwidth
  is high
- Large systems are made up of many microservices.
- There may be as many as 140 in a typical Amazon page.

Services can have multiple instances

- The elasticity of the cloud will adjust the number of instances of each service to reflect the workload.
- Requests are routed through a load balancer for each service
- This leads to
  - Lots of load balancers
  - Overhead for each request.

Digression into Service Oriented Architecture (SOA)

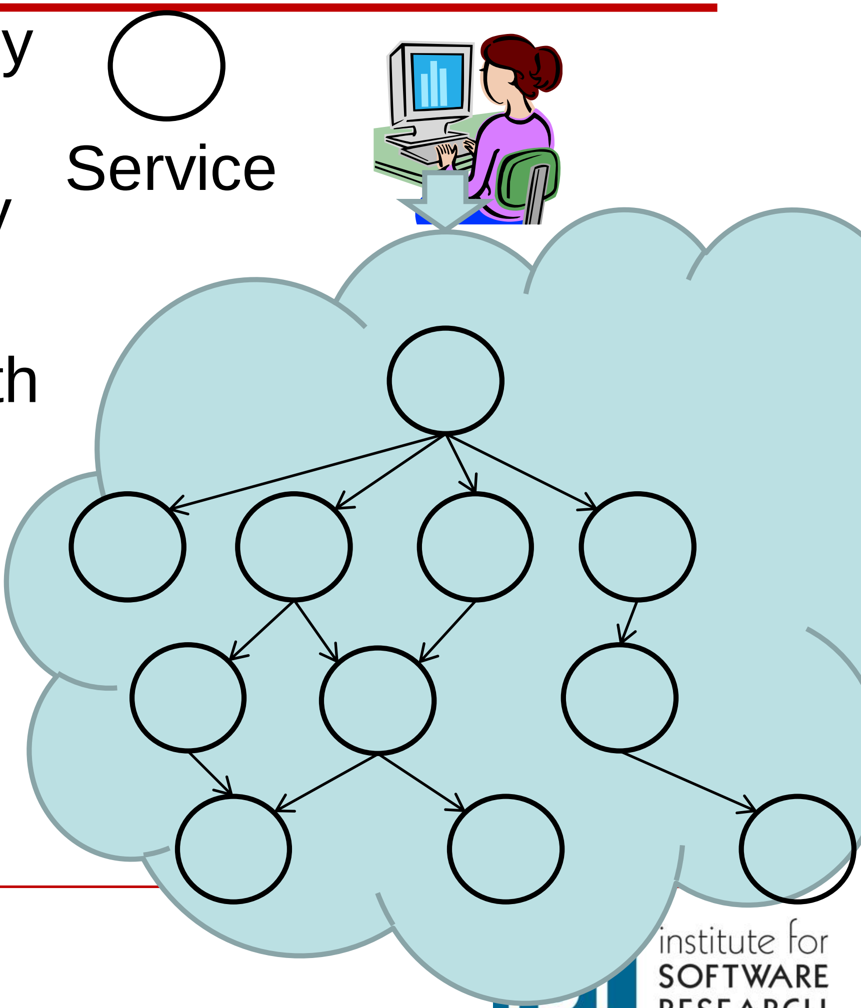
- The definition of microservice architecture sounds a lot like SOA.
- What is the difference?
- Amazon did not use the term “microservice architecture” when they introduced their rules. They said “this is SOA done right”

SOA typically has but microservice architecture does not

- Enterprise service bus
- Elaborate protocols for sending messages to services (WDSL*)
- Each service may be under the control of different organization
- Brokers
- etc

Micro service architecture

- Each user request is satisfied by some sequence of services.
- Most services are not externally available.
- Each service communicates with other services through service interfaces.
- Service depth may
  - Shallow (large fan out)
  - Deep (small fan out, more dependent services)



Quality attributes for microservice architectures

- Availability (+)
- Modifiability (+/-)
- Performance (-)
- Reusability (-)
- Scalabilty (_+)

Overview

- Microservice architecture definition
- **Containers and microservices**
- Reducing errors during deployment
- Reducing time to deploy
- Speeding up incident handling

© Len Bass 2018

Microservices and Containers

- Although microservices and containers were developed independently, they are a natural fit and are evolving together.
- A microservice will use a number of dependent services. Common ones are:
  - Metrics
  - Logging
  - Tracing
  - Messaging
    - gRPC
    - Protocol buffers
  - Discovery
  - Registration
  - Configuration management
  - Dashboards
  - Alerts

Packaging microservices

- Each dependent service will be packaged in its own container.
- Containers can be grouped in pods
- Allows for deploying and scaling together

Overview

- Microservice architecture definition
- Containers and microservices
- **Reducing errors during deployment**
- Reducing time to deploy
- Speeding up incident handling

© Len Bass 2018

Microservice architecture reduces errors during deployment

- One common source of errors during integration is inconsistency in technology choices
- E. g. Team A uses version 2.1 of a library, team B uses version 2.2. Two versions are incompatible.
- With microservice architecture,
  - each team makes their own technology choices.
  - Services communicate only via messages
  - No requirement for teams to use the same version of a library. Teams can even use different languages.
- Fewer errors

© Len Bass 2018

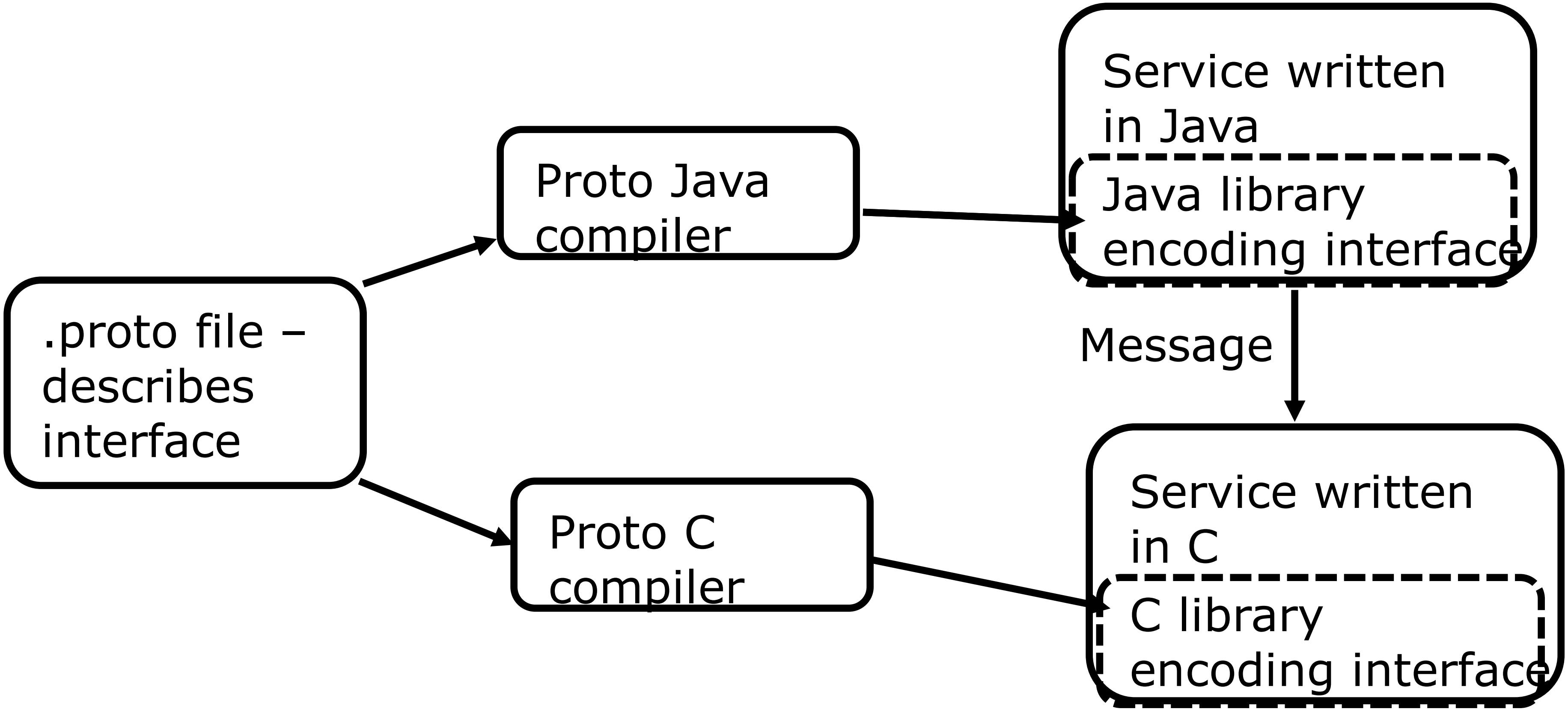
Protocol Buffers - 1

- Schema defines data types
- Binary format
- A protocol buffer specification is used to specify an interface
- Language specific compilers used for each side of an interface
- Allows different languages to communicate across a message based interface

Protocol Buffers – 2

- Service A written in Java calls Service B written in C
- Interface specification written as .proto file
- Java protocol buffer compiler produces Java procedure interface for Service A
- C protocol buffer compiler produces procedure interface for Service B
- Service A code calls Java procedure interface which sends binary data received by Service B procedure (written in C)

Protocol Buffers – framework



Also

- No requirement for teams to coordinate to choose language, libraries, or dependent technologies
- Fewer meetings, shortens time to deployment.
- Collection of .proto files documents the interfaces of the architecture

© Len Bass 2018

Overview

- Microservice architecture definition
- Containers and microservices
- Reducing errors during deployment
- **Reducing time to deploy**
- Speeding up incident handling

Continuous Deployment

- Continuous deployment is process whereby after commit, software is tested and placed in production without human intervention
- Continuous development goes through the same steps except a human has to sign off on the actual deployment.
- In either case, time to deployment is shortened.

© Len Bass 2018

Designing for Deployment

- It is possible that different versions of a single microservice may simultaneously be in service
- It is possible that new features may be supported in some microservices but not in others.
- Must design to allow for these possibilities.

© Len Bass 2018

institute for SOFTWARE RESEARCH

Situation

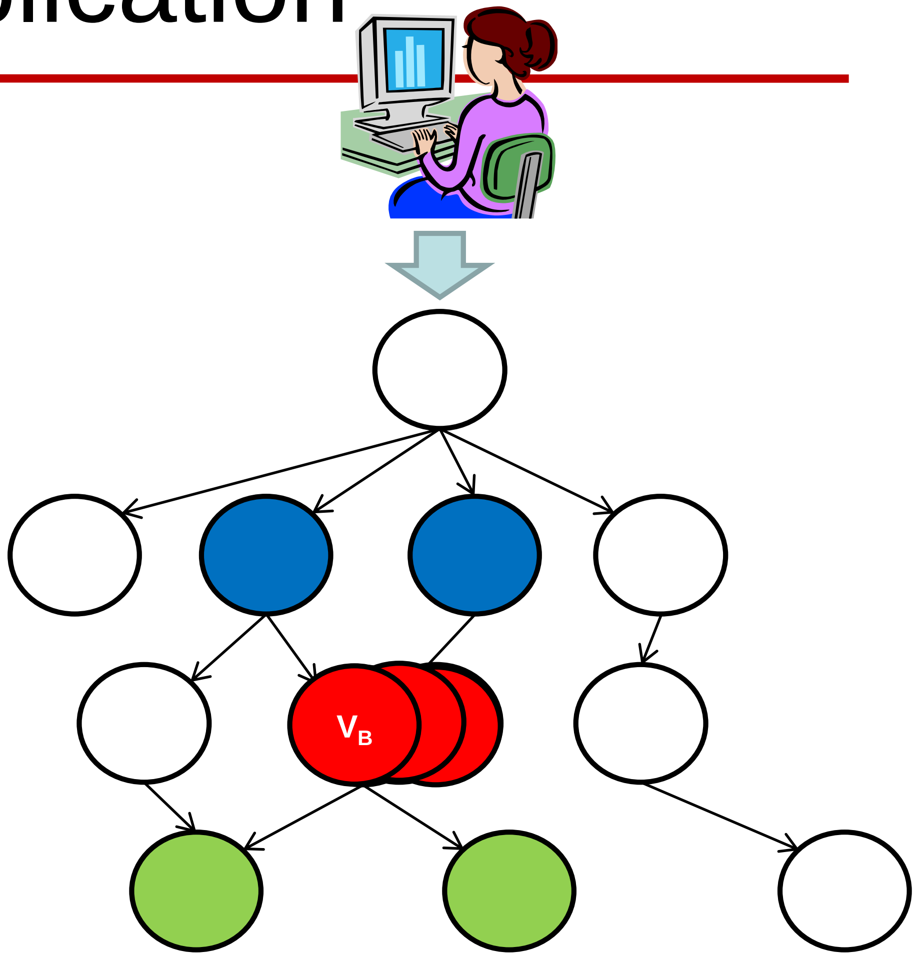
- Your application is executing
  - Multiple independent deployment units
  - Some of these deployment units may have multiple instances serving requests
- You have a new version of one of the deployment units to be placed into production
- An image of the new version is on the staging server or in a container repository

© Len Bass 2018

Deploying a new version of an application

Multiple instances of a service are executing

- Red is service being replaced with new version
- Blue are clients
- Green are dependent services



Staging/container repository

Deployment goal and constraints

- Goal of a deployment is to move from current state (N instances of version A of a service) to a new state (N instances of version B of a service)
- Constraints:
  - Any development team can deploy their service at any time. I.e. New version of a service can be deployed either before or after a new version of a client. (no synchronization among development teams)
  - It takes time to replace one instance of version A with an instance of version B (order of minutes for VMs)
  - Service to clients must be maintained while the new version is being deployed.

Deployment strategies

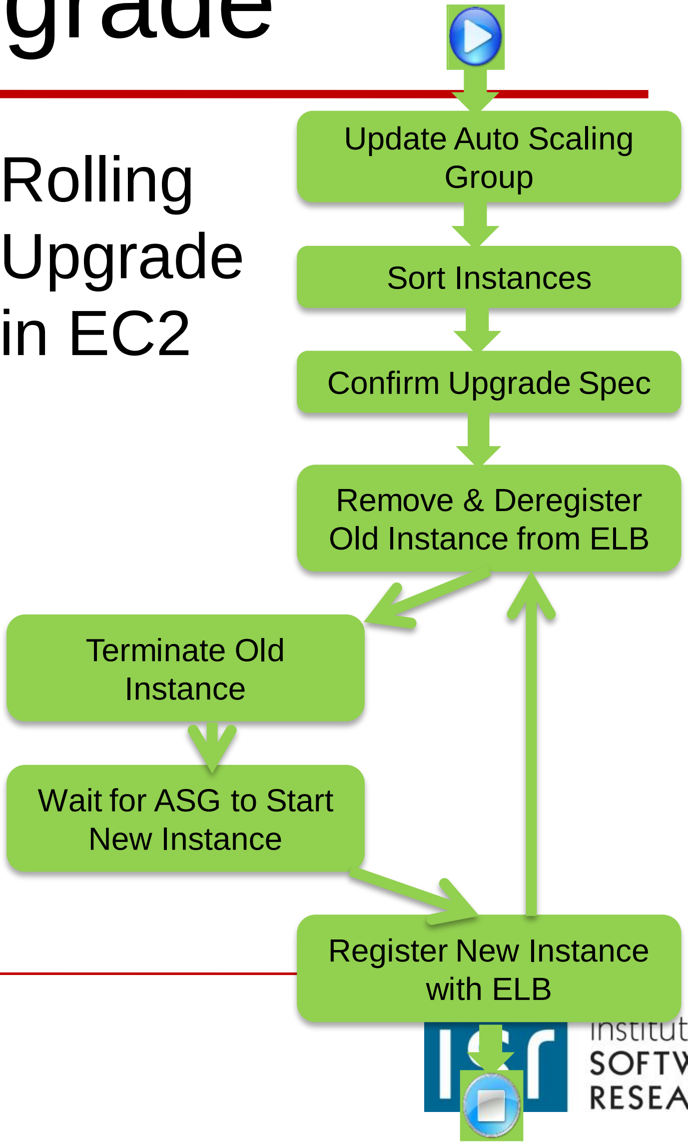
- Two basic all of nothing strategies
  - Blue/Green (also called Red/Black) – leave N instances with version A as they are, allocate and provision N instances with version B and then switch to version B and release instances with version A.
  - Rolling Upgrade – allocate one instance, provision it with version B, release one version A instance. Repeat N times.
- Partial strategies are canary testing and A/B testing.

© Len Bass 2018

Trade offs – Blue/Green and Rolling Upgrade

- Blue/Green
  - Only one version available to the client at any particular time.
  - Requires 2N instances (additional costs)
- Rolling Upgrade
  - Multiple versions are available for service at the same time
  - Requires N+1 instances.
- Rolling upgrade is widely used.

Rolling Upgrade in EC2



Problems to be dealt with

- Temporal inconsistency
- Interface mismatch

© Len Bass 2018

Temporal inconsistency example

- Shopping cart example
  - Suppose your organization changes its discount strategy from discount per item to discount per shopping cart.
  - Version A' of your service does discounts per item
  - Version A'' does discounts per shopping cart.
- Client C's first call goes to version A' and its second call goes to version A''.
- Results in inconsistent discounts being calculated.
- Caused by update occurring between call 1 and call 2.

Temporal inconsistency

- Can occur with either Blue/Green or rolling upgrade
- Prevented by using feature toggles.
- Feature toggle puts new code under control of if statement keyed to toggle.

  If toggle then

  new code

  else

  old code

Preventing Temporal Inconsistency

- Write new code for Service A'' under control of a feature toggle
- Install N instances of Service A'' using either Rolling Upgrade or Blue/Green
- When a new instance is installed begin sending requests to it
  - No temporal inconsistency, as the new code is toggled off.
- When all instances of Service A are running Service A'', activate the new code using the feature toggle.

Feature toggle manager

- There will be many different feature toggles
  - One for each feature
- A feature toggle manager maintains a catalog of feature toggles
  - Current toggles vs instance version id
  - Current toggles vs module version
  - Status of each toggle
  - Activate/de-activate feature
  - Remove toggle (will place removal on backlog of appropriate development team).

© Len Bass 2018

Activating feature

- The feature toggle manager changes the value of the feature toggle.
- A coordination mechanism such as Zookeeper or Consul could be used to synchronize the activation.

Interface mismatch

- Suppose version A’’ has a different interface from version A’
- Then if Service C calls version A’’ with an Interface designed for version A’ an interface mismatch occurs.
- Recall that Service A can be upgraded either before or after Service C.
- Interface mismatch is prevented by making interfaces backward and forward compatible.

Overview

- Microservice architecture definition
- Containers and microservices
- Reducing errors during deployment
- Reducing time to deploy
- **Speeding up incident handling**

© Len Bass 2018

You build it, you run it

"There is another lesson here: Giving developers operational responsibilities has greatly enhanced the quality of the services, both from a customer and a technology point of view. The traditional model is that you take your software to the wall that separates development and operations and throw it over and then forget about it. Not at Amazon. You build it, you run it. This brings developers into contact with the day-to-day operation of their software. It also brings them into day-to-day contact with the customer. This customer feedback loop is essential for improving the quality of the service."

-*Wener Vogels*

https://queue.acm.org/detail.cfm?id=1142065

Scenario

- It is 3:00AM and your pager goes off.
- There is a problem with your service!
- You get out of bed and log onto the production environment and look at the services dashboard.
- One instance of your service has high latency
- You drill down and discover the problem is a slow disk
- You move temporary files for your service to another disk and place the message “replace disk” on the operators queue.

Information needs

- Metrics collected by infrastructure
- Logs from instance with relevant information
- Central repository for logs
- Dashboard that displays metrics
- Alerting system
  - Monitoring latency of instances
  - Rule: if high latency then alarm

© Len Bass 2018

Logs

- A log is an append only data structure
- Written by each software system.
- Located in a fixed directory within the operating system
- Enumerates events from within software system
  - Entry/exit
  - Troubleshooting
  - DB modifications
  - …

© Len Bass 2018

Logs on Entry/Exit

- Protocol Buffer compilers automatically generate procedures that are called on entry/exit to a service
- These procedures can be made to call logging service with parameters and identification information.
- Logs on entry/exit can be made without additional developer activity

Metrics

- Metrics are measures of activity over some period of time
- Collected automatically by infrastructure over externally visible activities of VM
  - CPU
  - I/O
  - etc

© Len Bass 2018

Repository

- Logs and metrics are placed in repository
- Repository generates alarms based on rules
- Provides central location for examination when problem occurs
- Displays information in dashboard that allows for drilling down to understand source of particular readings.

© Len Bass 2018

Summary

- Time to market is driver for processes to
  - Reduce errors during deployment
  - Reduce time to deployment
  - Respond to incidents more quickly
- Microservice architecture helps all three goals
- Continuous deployment/development requires designing to avoid various kinds of inconsistencies
- Developers carry pagers as a means to speed up incident handling

© Len Bass 2018

Questions/feedback

All information in this workshop can be found in:

Deployment and Operations for Software Engineers

by

Len Bass and John Klein