RX-M Cloud Native Consulting

Building Cloud Native Applications on Cloud Foundry

An in depth look at the microservices architecture pattern, containers and Cloud Foundry

3: Container Packaging

Objectives

- Define application container
- List several container technologies
- Explain the purpose of Docker
- Contrast Virtual Machines and containers
- Explore the relationship between
  - IaaS and Virtual Machines
  - PaaS and Containers
- Understand the nature of Docker Images

What is a container?

- Lightweight Linux environment
  - Now lightweight Windows environments as well
- Encapsulated and deployable
- Runnable
- A way to package applications for reliable deployment
  - Particularly popular for packaging microservices
- Made widely popular by Docker
  - Docker simplifies configuring, constructing and running containers
  - Docker Inc. provides a container platform including systems for publishing and sharing containers
- Container technology predates and enables Docker
  - Docker was the dominant mover in this space during 2013-2015
  - Competition is maturing
  - Standards exist and are evolving

```
$ time docker run ubuntu echo "hello world"
hello world

real        0m0.319s
user        0m0.005s
sys         0m0.013s
```

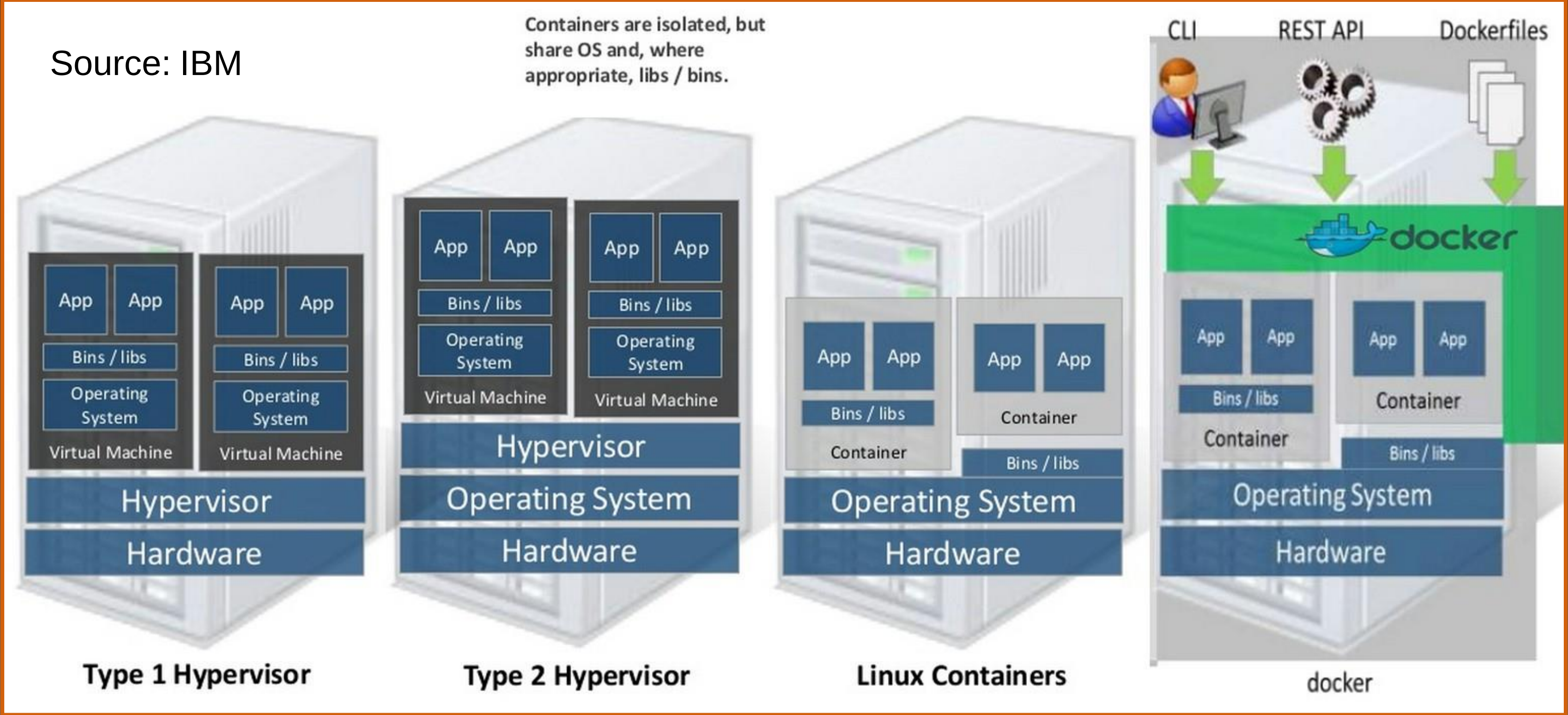
Disk usage: less than 100 kB
Memory usage: less than 1.5 MB

VMs and Containers

- VM
  - A virtual machine for operating systems
- Container
  - A virtual operating system for applications
- These are not mutually exclusive and many environments become optimal when properly combining both

Containers supply only the executables and library interfaces necessary to mimic the application dependent aspects of a Linux distribution (Ubuntu, RHEL, SUSE, etc.), leveraging the fact that all Linux systems use the same underlying kernel

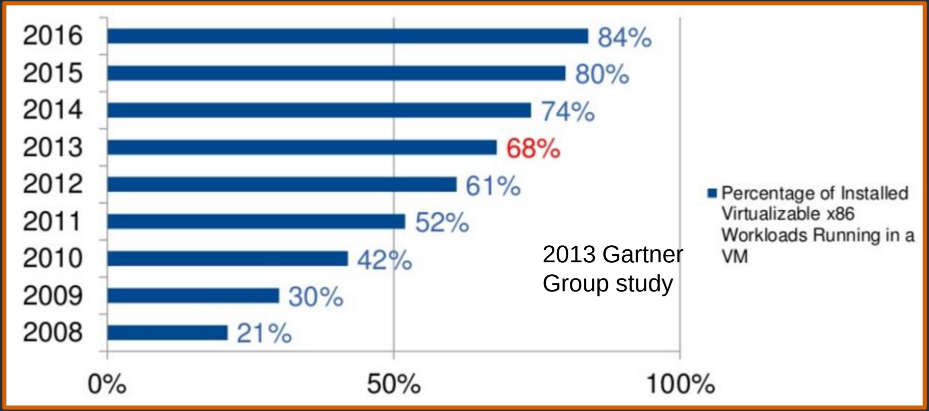
Source: IBM

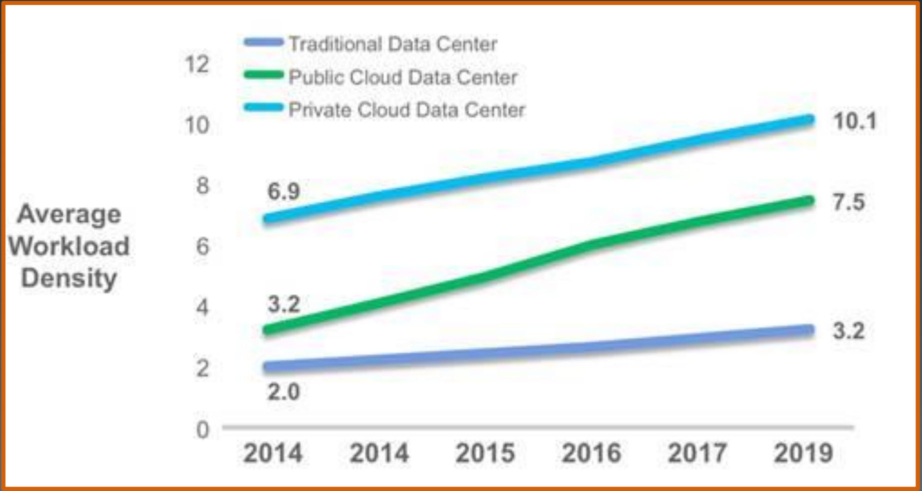


VMs, why do I care?

- VM's allow new machine instances to be deployed in real time (seconds-minutes)
  - Not months, as is typical with physical server purchase/deploy cycles
- VM's enable migration and elasticity
  - Machines can be created, moved and deleted rapidly
- VM's allow physical resources to be fully utilized
  - Physical CPU/RAM use can be maximized without mixing logical machine roles
  - Increased server density
- VM's enable repeatable static system environments to be used for development, testing and deployment
  - The same machine can be launched by Vagrant, OpenStack, Amazon EC2, Microsoft Azure, Google Cloud, etc.
- VM's enable cloud computing
  - Pay as you go
  - Self service

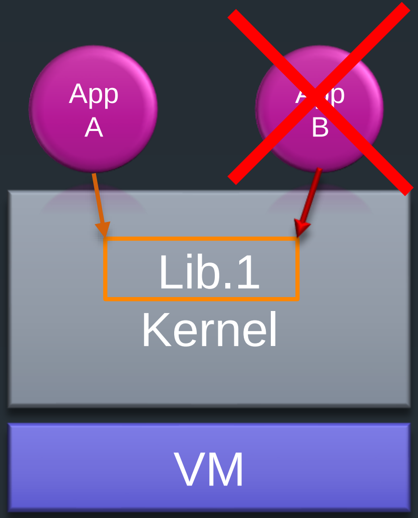
IaaS

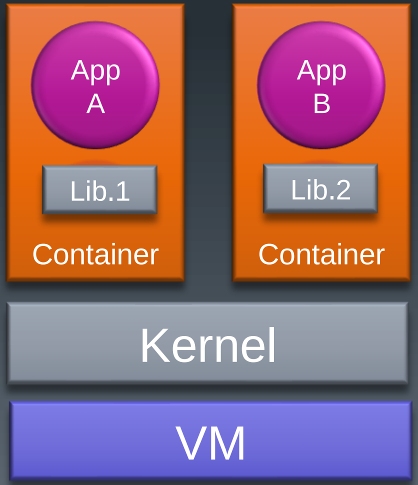




Containers, why do I care?

- Containers provide a static application runtime environment creating reliable deployments
  - Fundamentally changes the approach to operations
  - Removes an entire class of extremely complex operational problems
- VMs are typically used to host infrastructure roles (e.g. Web Server, Application Server, Database Server, …)
  - Applications running on such systems can have complex interactions with system services and other applications
  - This complexity makes it difficult to guarantee identical dev/test/prod environments, making application deployment complex and error prone
- Containers create a new layer of abstraction at the operating system level for application roles (web server, logging system, security tools, monitoring software, etc.)
  - Isolation
    - Allows multiple containerized applications to run on the same VM
  - Encapsulation
    - Each container is encapsulated with its own unique dependencies
  - Portability
    - Repeatable deployment across dev/test/prod environments and clouds





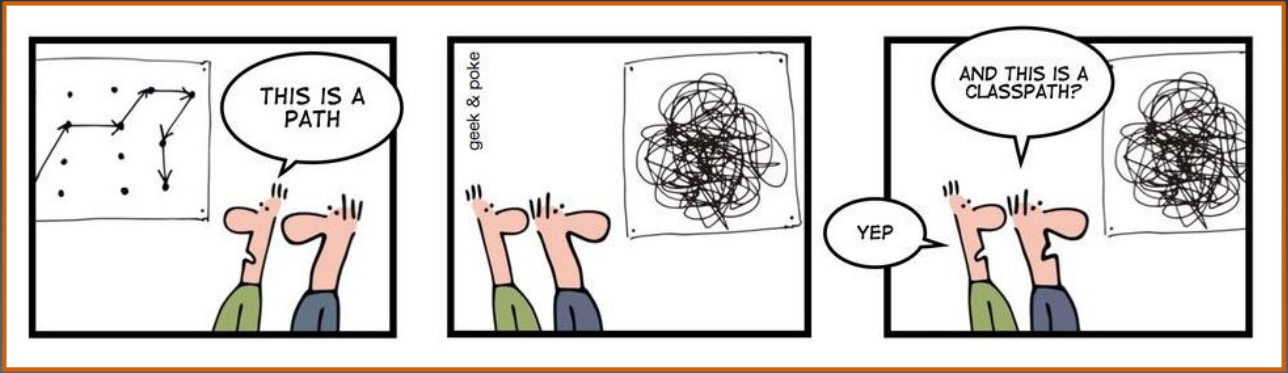
PaaS

Encapsulation

- Containers can encapsulate:
  - Data
  - Code
  - Configuration files
  - Frameworks
  - Libraries
  - System Dependencies
  - Packaging
  - Linux Distributions
  - Environment Variables
  - Command line arguments
  - and more
- Virtually everything needed by an application can be packaged within the application’s container
- We can ignore where and how the container runs
  - The container’s internals do not interact with external aspects of the environment, removing an entire class of deployment problems

Empowering Agile Processes

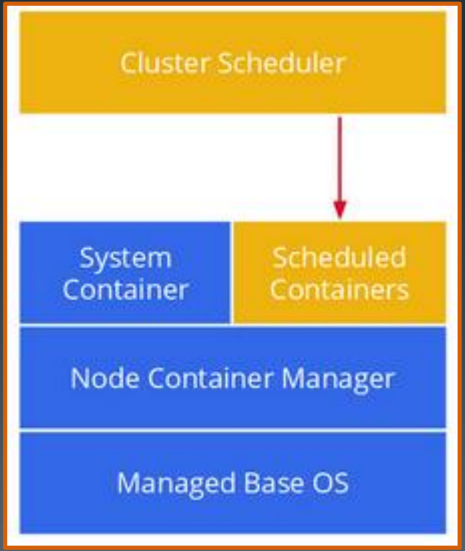
- Microservices
  - The more atomic the service:
    1. The more likely it is to be reusable
    2. The more easily it can be encapsulated
    3. The more of them you need to do something useful (!)
- Micro services and VMs have different cardinalities
  - 10:1 Ten services running on a single VM creates an ops integration challenge across all services
- Micro services and Containers have the same cardinality
  - 1:1 One service in one container, requires only ops support for the service encapsulated
- Containers allow each service to be packaged with its own dependencies
  - 10 containerized services map directly to 10 individual, isolated, reliable, repeatable ops events
  - Gives devops teams autonomy
    - Team owns software and configuration when using containers
  - Empowers CI/CD and incremental application evolution
    - Integration challenges are reduced to inter service communications, networking and volumes



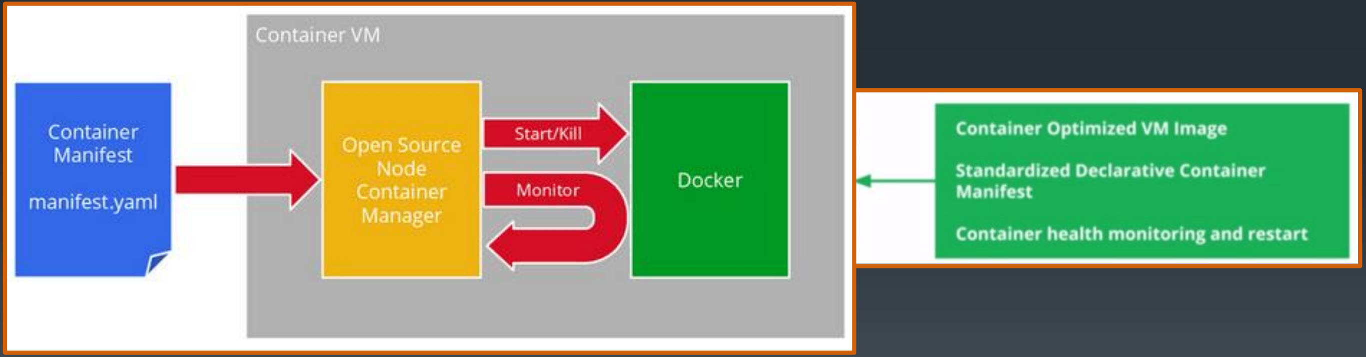
Containers at scale

- Containers - a key enabler of PaaS cloud environments
  - Google App Engine is one of the most visible container based cloud systems
- Everything at Google, from Search to Gmail, is packaged and run in a Linux container
  - Google's Borg system (the basis for Kubernetes) is a cluster manager that ran hundreds of thousands of jobs, from many thousands of different containerized applications, across a number of clusters each with up to tens of thousands of machines (Borg is now replaced within Google by the next gen Omega system)
  - Over 2 billion containers are started per week at Google (over 3,000 per second on average)
- lmctfy ("Let Me Contain That For You") open source version of Google's container stack
  - Google has ported the core lmctfy concepts and abstractions to Docker's libcontainer
  - libcontainer is now the basis for much of the OCI container standard

Google Container Infrastructure



Google Cloud Platform
Docker containers in Google Compute Engine

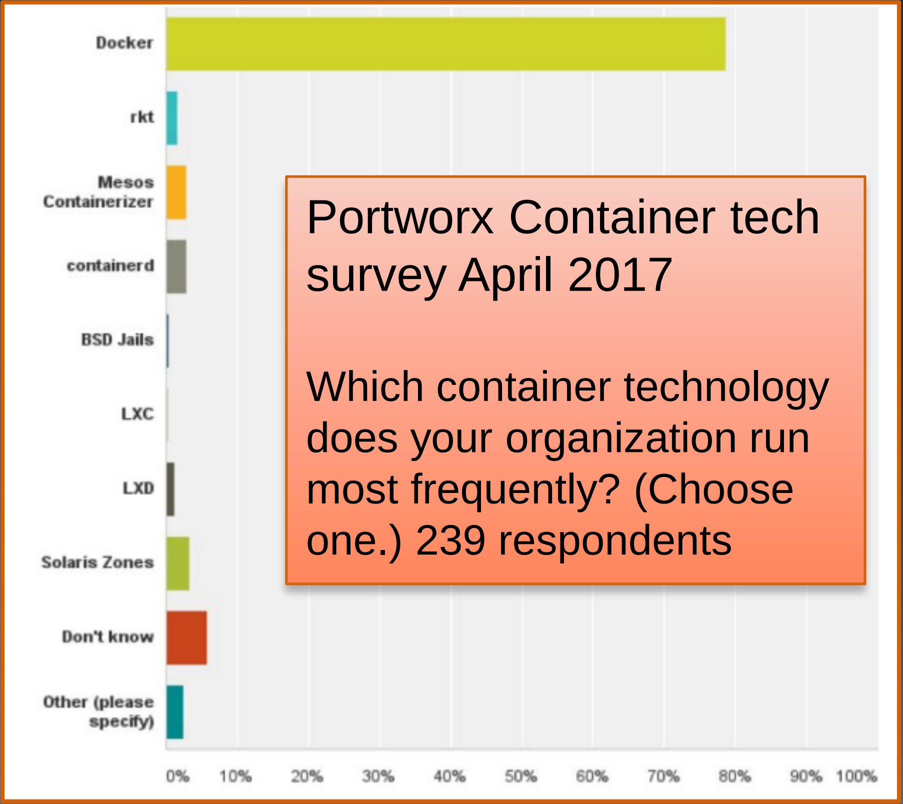

Container VM
Container Manifest
manifest.yaml
Open Source Node Container Manager
Start/Kill
Monitor
Docker
Container Optimized VM Image
Standardized Declarative Container Manifest
Container health monitoring and restart

Containers At Scale by Joe Beda
Published May 22, 2014 in
Technology

Borg Paper: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43438.pdf

Docker Alternatives

- Virtual Machines
  - Better isolation, worse performance, more complex deployment
  - VMware, AWS, Google, and Azure all run Docker-based workloads on behalf of cloud customers in a multi-tenant environment, but do so by putting each customer's Docker containers inside the logical boundaries of virtual machines
  - VMware VIC (VM Integrated Containers)
  - Kata Containers (combined projects: Hyper RunV and Intel Clear Containers)
- RedHat Solutions
  - CoreOS Inc. (acquired by RedHat) Tectonic is a minimal Linux OS with direct support for containers (similar to RHEL Atomic, Ubuntu Core and Windows 2016 Nano)
  - Rocket (rkt) lightweight docker like platform
  - CRI-O is an open source RedHat backed container manager
- Joyent SmartOS & Triton
  - Triton uses the docker client but supplies SmartOS (an OpenSolaris/KVM hypervisor) with Solaris Zone isolation (using Illumos Zones) for containers offering strong security
- Mesos
  - Distributed OS running ”frameworks”
- LXC
  - Original Linux container library (v2.0.3 – 6/2016)
- LXD/OpenVZ
  - Systems in containers (lightvisors)
- BSD Jails
  - Isolation features of BSD Unix
- OpenSolaris Container
  - Zones based isolation model



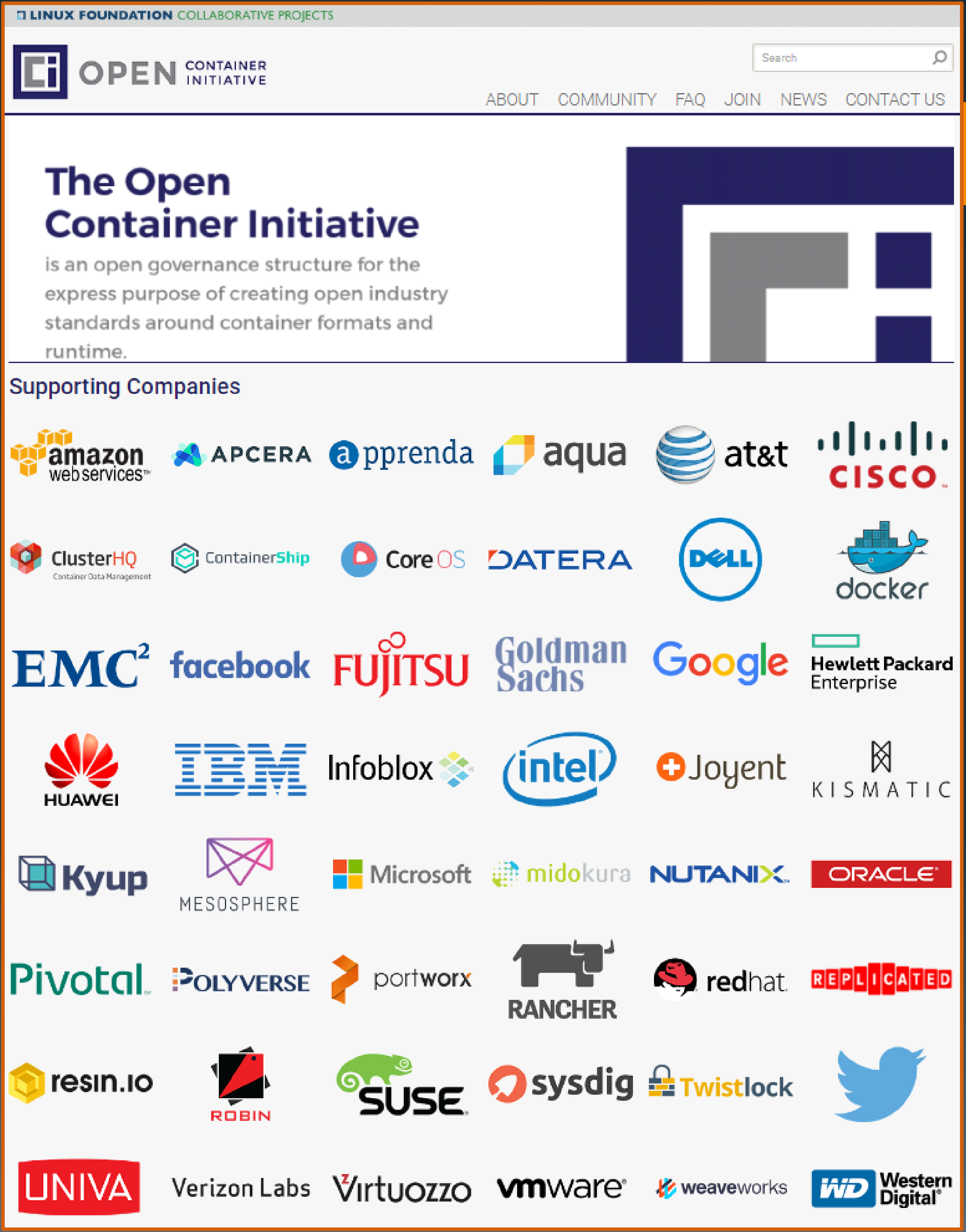
Portworx Container tech survey April 2017

Which container technology does your organization run most frequently? (Choose one.) 239 respondents

http://portworx.com/wp-content/uploads/2017/04/Portworx_Annual_Container_Adoption_Survey_2017_Report.pdf

Container Standardization

- OCI [circa 6/2015]
  - Open Container Initiative
    - Formerly Open Container Project
  - The OCI contains three specifications:
    - Runtime Specification (runtime-spec) 1.0
      - How to run a "filesystem bundle" that is unpacked on disk
    - Image Specification (image-spec) 1.0
      - The format of an OCI Image and how to unpack that image into an OCI Runtime filesystem bundle
    - Distribution Specification (distribution-spec) in progress
      - Standardizes container image distribution based on Docker Registry API v2 protocol
  - Run by the Linux Foundation
    - Non-profit which oversees the Linux open source operating system
  - Docker donated container format, runtime code (libcontainer), specifications and registry API
  - Members (All of the tech companies in the Fortune 100 except Apple, all of the key container startups):
    - Amazon, Apcera, Apprenda, Aqua, AT&T, ClusterHQ, Cisco, CoreOS, ContainerShip, Datera, Dell, Docker, EMC, Facebook, Fujitsu Limited, Goldman Sachs, Google, Hewlett Packard Enterprise, Huawei, IBM, Infoblox, Intel, Joyent, Kismatic, Kyup, Mesosphere, Microsoft, Midokura, Nutanix, Odin (Virtuozzo), Oracle, Pivotal, Polyverse, Portworx, Rancher Labs, Red Hat, Replicated, Resin.io, Robin, Scalock, Sysdig, SUSE, Twistlock, Twitter, Univa, Verizon Labs, Vmware, Weaveworks and WD


LINUX FOUNDATION COLLABORATIVE PROJECTS

OPEN CONTAINER INITIATIVE

Search

ABOUT COMMUNITY FAQ JOIN NEWS CONTACT US

The Open Container Initiative

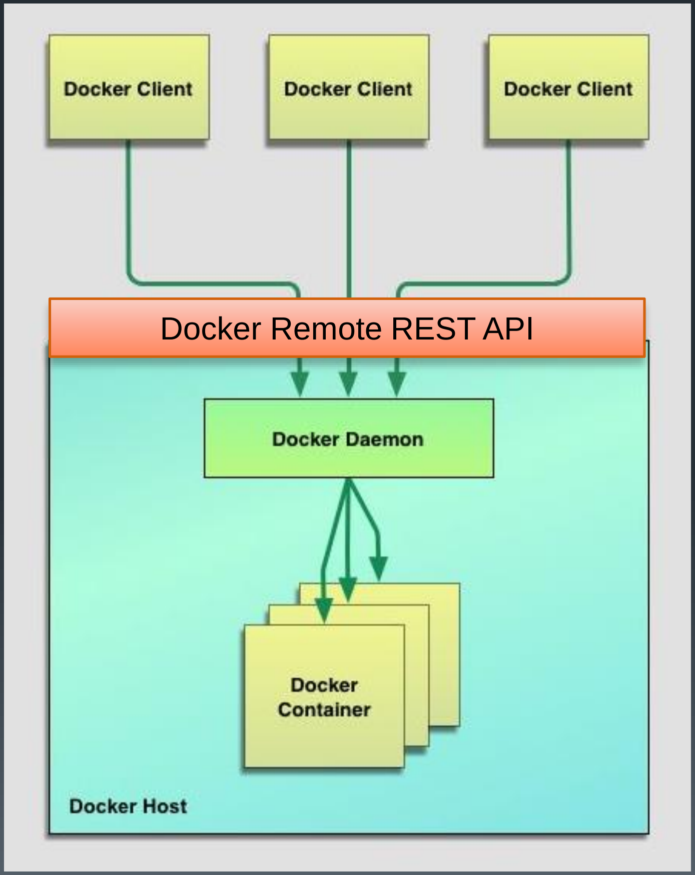
is an open governance structure for the express purpose of creating open industry standards around container formats and runtime.

Supporting Companies

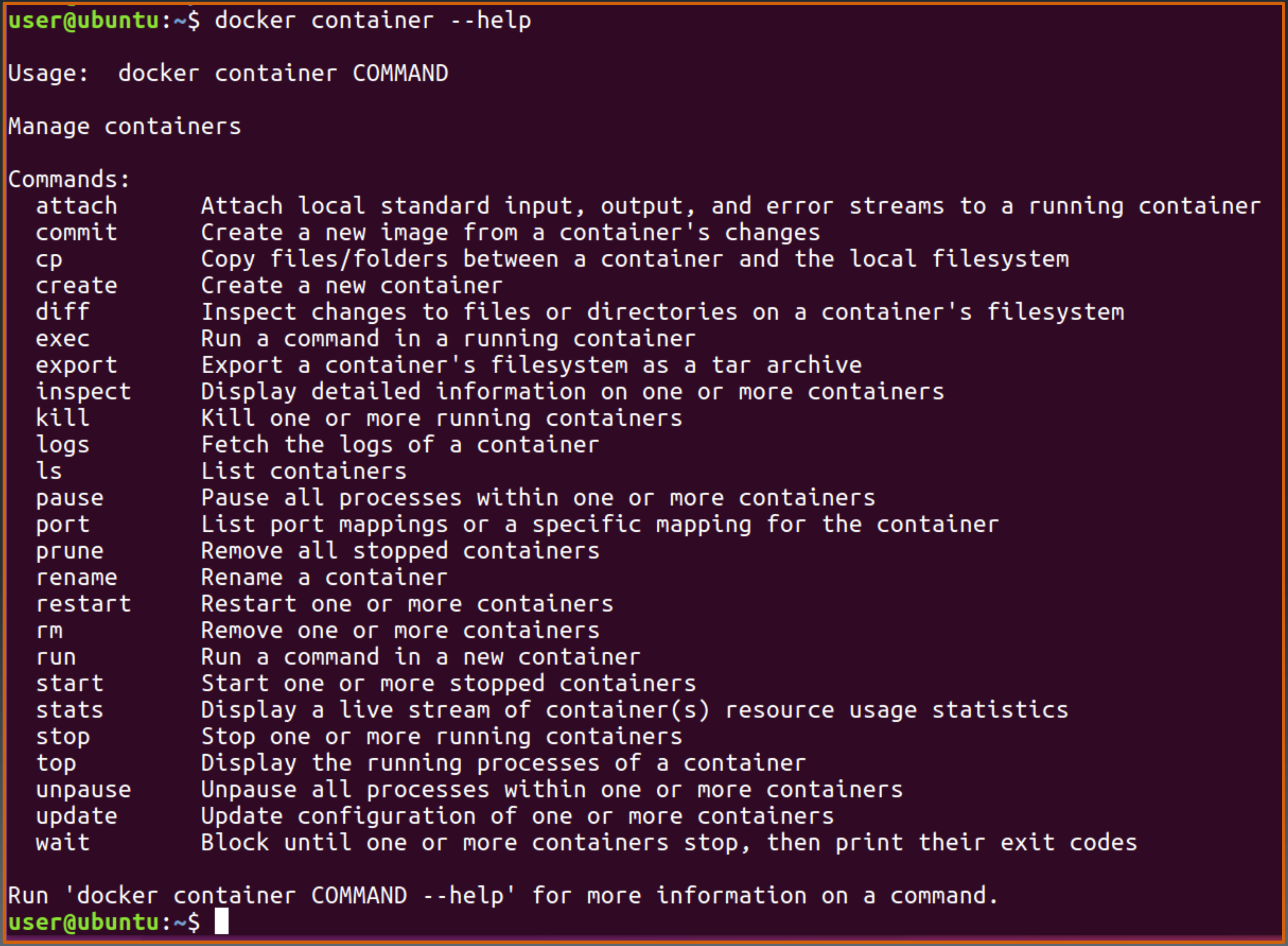
amazon web services, APCERA, apprenda, aqua, at&t, CISCO, ClusterHQ Container Data Management, ContainerShip, CoreOS, DATERA, DELL, docker, EMC², facebook, FUJITSU, Goldman Sachs, Google, Hewlett Packard Enterprise, HUAWEI, IBM, Infoblox, intel, Joyent, KISMATIC, Kyup, MESOSPHERE, Microsoft, midokura, NUTANIX, ORACLE, Pivotal, POLYVERSE, portworx, RANCHER, redhat, REPLICATED, resin.io, ROBIN, SUSE, sysdig, Twistlock, UNIVA, Verizon Labs, Virtuozzo, vmware, weaveworks, WD Western Digital

Core Docker Components

- Docker Engine
  - Daemon on the Linux host supporting docker container execution: /usr/bin/dockerd
    - Docker manages containers, the linux kernel runs containers
- Docker Client
  - The Docker client sends requests to local and remote Docker daemons: /usr/bin/docker (or docker.exe)
- Docker Remote API
  - The Docker client talks to the Docker daemon through the Docker Remote RESTful API
- Docker Images
  - Images are used to generate containers
    - Just as a VM image can create multiple VM instances
    - Just as an executable (/usr/bin/vi) can create multiple processes
- Registries
  - Registries are network services from which Docker Images can be saved and retrieved
  - The Docker Hub is an internet based registry with support for public and private images
  - Private registry servers can be created for an organization's internal use
- Docker Containers
  - A container is a software package generated from an image
  - Said to be running when processes are executing within it
  - Can be stopped and started



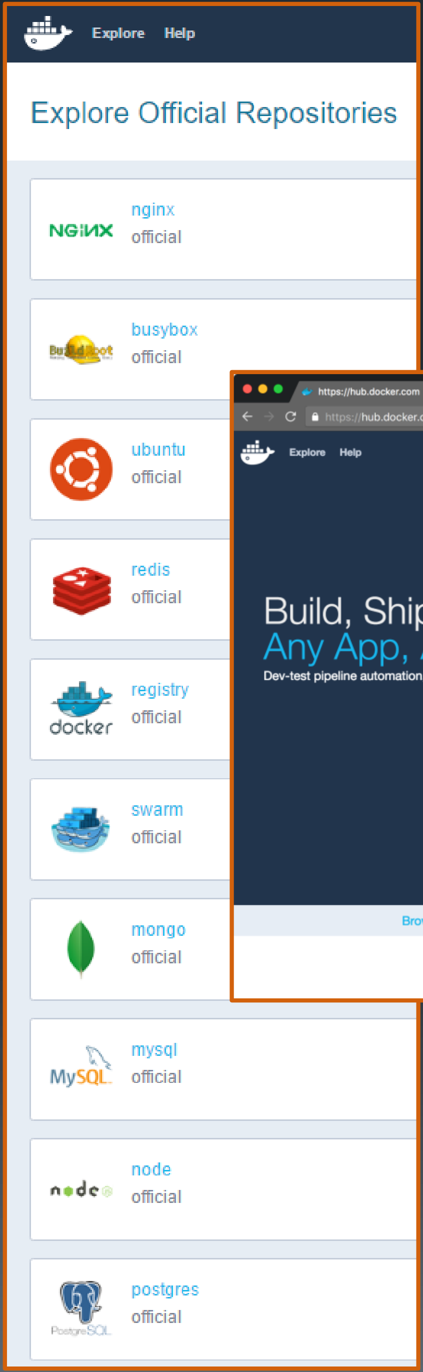
Container Control



```
user@ubuntu:~$ docker container --help

Usage:  docker container COMMAND

Manage containers

Commands:
  attach      Attach local standard input, output, and error streams to a running container
  commit      Create a new image from a container's changes
  cp          Copy files/folders between a container and the local filesystem
  create      Create a new container
  diff        Inspect changes to files or directories on a container's filesystem
  exec        Run a command in a running container
  export      Export a container's filesystem as a tar archive
  inspect     Display detailed information on one or more containers
  kill        Kill one or more running containers
  logs        Fetch the logs of a container
  ls          List containers
  pause       Pause all processes within one or more containers
  port        List port mappings or a specific mapping for the container
  prune       Remove all stopped containers
  rename      Rename a container
  restart     Restart one or more containers
  rm          Remove one or more containers
  run         Run a command in a new container
  start       Start one or more stopped containers
  stats       Display a live stream of container(s) resource usage statistics
  stop        Stop one or more running containers
  top         Display the running processes of a container
  unpause     Unpause all processes within one or more containers
  update      Update configuration of one or more containers
  wait        Block until one or more containers stop, then print their exit codes

Run 'docker container COMMAND --help' for more information on a command.
user@ubuntu:~$ 
```

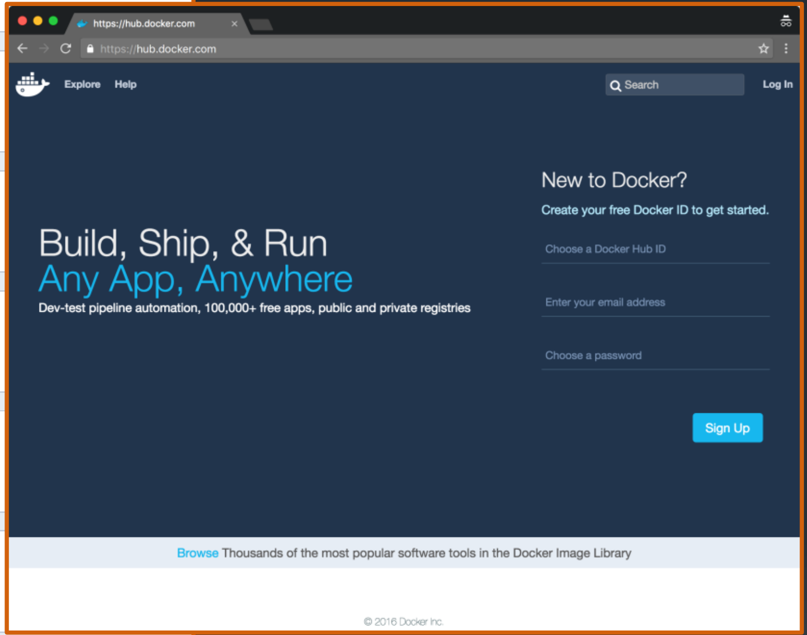
The Docker Hub

- The Docker Hub is a registry supporting dynamic container image download
- Docker Hub hosts many base images
  - Cirros
  - Ubuntu
  - Fedora
  - Debian
  - Centos
  - etc .
- These images can be used as the basis for building custom application service images
  - Our prior example automatically downloaded the minimal Cirros Linux cloud image and then ran a Bourne shell within the container
- When docker can not find an image locally it looks for the image in a registry, like DockerHub
- Docker Hub also hosts ready to run service images
  - MongoDB, Cassandra, Apache, Nginx, Redis, WordPress, …
- And dev platform images
  - Java, NodeJS, Ruby, Python, …



Explore Official Repositories

nginx official
busybox official
ubuntu official
redis official
registry official
swarm official
mongo official
mysql official
node official
postgres official



Build, Ship, & Run
Any App, Anywhere
Dev-test pipeline automation, 100,000+ free apps, public and private registries

New to Docker?
Create your free Docker ID to get started.
Choose a Docker Hub ID
Enter your email address
Choose a password

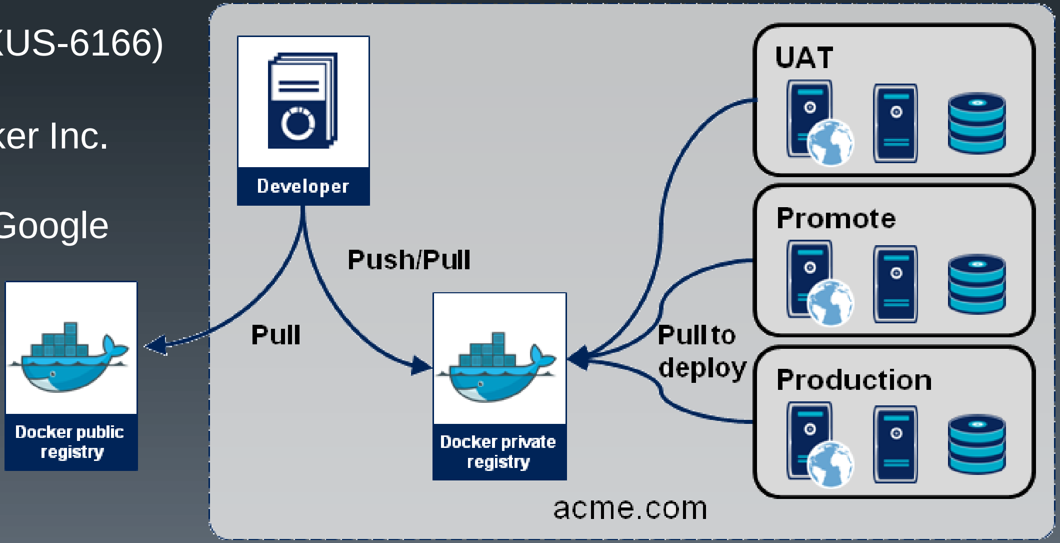
Browse Thousands of the most popular software tools in the Docker Image Library

© 2016 Docker Inc.

Registries, Repos and Tags

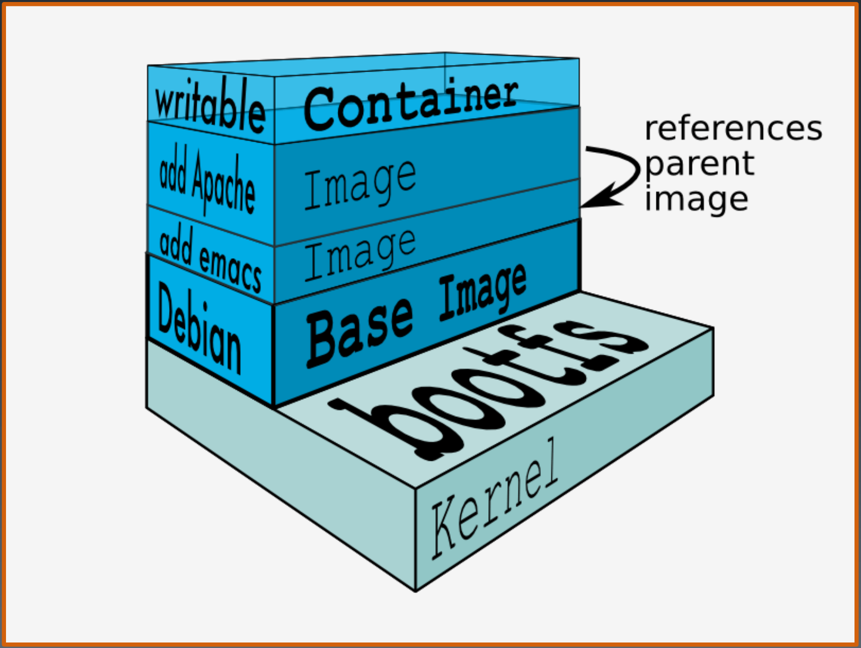
- Images house file system layers and metadata
- Repositories are named collections of images
- Tags are strings used to identify individual images within a repository
- Registries are services that allow you to store and retrieve images by repository:tag name using a REST (HTTP) API
  - Docker Hub is the central public registry
  - Private open source and commercial registries
    - Docker Trusted Registry (formerly known as: Docker Hub Enterprise) from Docker Inc. (commercial product)
    - Quay Enterprise part of the Tectonic platform from CoreOS (commercial product)
    - Docker Registry from Docker Inc. (free open source) (https://github.com/docker/docker-registry)
    - Harbor open source enterprise-class container registry server from Vmware (https://github.com/vmware/harbor)
    - Artifactory 3.4+ (http://www.jfrog.com/open-source/#os-arti)
    - Nexus 3 Milestone 5 (in preview) (https://issues.sonatype.org/browse/NEXUS-6166)
  - Hosted cloud solutions
    - Docker Hub (hub.docker.com) from Docker Inc.
    - Quay (quay.io), acquired by CoreOS Inc.
    - Google Container Registry (gcr.io) from Google
    - Amazon EC2 Container Registry [ECR]

Public registry images should be used with caution by the security minded



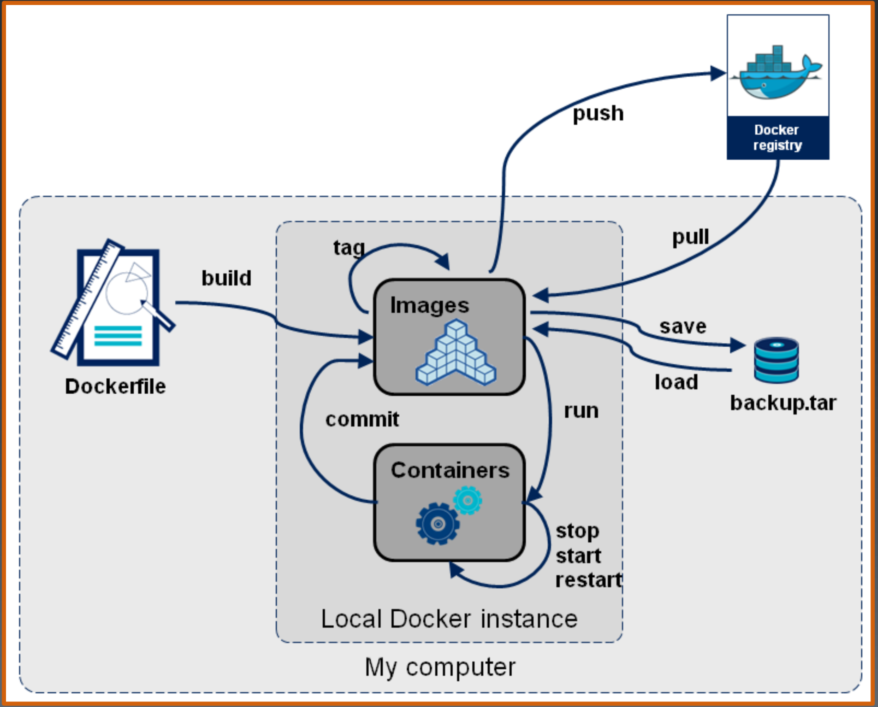
Docker Images

- Docker containers are constructed by sequentially applying meta data and file system layers from one or more images
- A Docker image includes metadata and an optional file system layer
  - Image meta data includes the ID of the image's parent, the default command to run, etc.
    - In Docker Engine <= v1.9 image IDs were random (UUIDs)
    - In Docker Engine >= v1.10 image IDs are SHA256 hashes of the image (content addressable)
  - Layered images tend to supply one specific feature on top of the parent image
  - Upper layer metadata and files mask metadata and files at lower layers with the same name/pathname
- An image with no parent image is called a Base Image
  - Base images tend to be largish and house operating system libraries and executables (e.g. Debian, Ubuntu, Centos, etc.)
- Image metadata and files are immutable
  - Allows one Image to support multiple Container instances with repeatable results
  - Reduces the disk and memory footprint of a given set of containers which share the same read only images
- Containers have a writable file system layer
  - The container file system layer is initially empty
  - All writes go to this file system layer and overlay any matching underlying image files
  - In this way, container file systems contain only the delta between their file system state and that of their underlying images
  - The view from the top down, including all file system layers in the stack, is called the union file system



Creating Images

- Docker images are the basis of containers
- Docker stores downloaded images on the Docker host
  - If a required image isn't already present on the host it is downloaded from a registry
- There are two ways to create new images
  - Update a container created from an image and commit the results to a new image
  - Use a Dockerfile to specify instructions to create an image



Dockerfiles

- docker commit
  - A practical way to create images interactively
  - An impractical way to create identical or incrementally improved images
- docker build
  - The docker build command creates images automatically from a Dockerfile
- Dockerfile
  - A text document containing all the commands you would normally execute manually in order to build a Docker image with docker commit
  - To build an image you can place a file called “Dockerfile” at the root of your repository and call “docker build” with the path to your repository
    - $ docker build .
  - Each command in a Dockerfile creates a new image layer

```
user@ubuntu:~$ mkdir thriftdev
user@ubuntu:~$ cd thriftdev/
user@ubuntu:~/thriftdev$ vim dockerfile
user@ubuntu:~/thriftdev$ cat dockerfile
# Version: 1.0.0
FROM ubuntu:14.04
RUN apt-get update
RUN apt-get install -y git
RUN echo 'thrift dev image v1.0.0' > /README.md
LABEL license "Apache 2.0"
user@ubuntu:~/thriftdev$ docker build -t my-image .
Sending build context to Docker daemon  2.048kB
Step 1/5 : FROM ubuntu:14.04
...
 ---> 3b853789146f
Step 2/5 : RUN apt-get update
 ---> Running in a9f3916d7afa
...
Removing intermediate container a9f3916d7afa
 ---> 13834b4d4d37
Step 3/5 : RUN apt-get install -y git
 ---> Running in d1957635c842
...
Removing intermediate container d1957635c842
 ---> 499d7350e19a
Step 4/5 : RUN echo 'thrift dev image v1.0.0' > /README.md
 ---> Running in 503ca71c7085
Removing intermediate container 503ca71c7085
 ---> 80787dfa4e12
Step 5/5 : LABEL license "Apache 2.0"
 ---> Running in 84267ff31dfe
Removing intermediate container 84267ff31dfe
 ---> 450665b7c688
Successfully built 450665b7c688
Successfully tagged my-image:latest
user@ubuntu:~/thriftdev$
```

```
user@ubuntu:~/thriftdev$ docker image ls -a
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
my-image            latest              450665b7c688        42 minutes ago      282MB
<none>              <none>              80787dfa4e12        42 minutes ago      282MB
<none>              <none>              499d7350e19a        42 minutes ago      282MB
<none>              <none>              13834b4d4d37        43 minutes ago      244MB
ubuntu              14.04               3b853789146f        4 days ago          223MB
user@ubuntu:~/thriftdev$
```

Summary

- Application containers are packaging systems and runtime environments that isolate and constrain the resources of a single service
- Docker is the prevalent container technology though several others are available and new options are emerging
- Docker automates the process of running containerized services
- Virtual Machines and containers have some features in common but are significantly different technologies
  - VMs are more heavy weight but provide better isolation
- Docker Images offer a layered system for creating containers
  - This may enable software sharing without strong coupling

Lab 3

- Container packaging

4: Communication II

Objectives

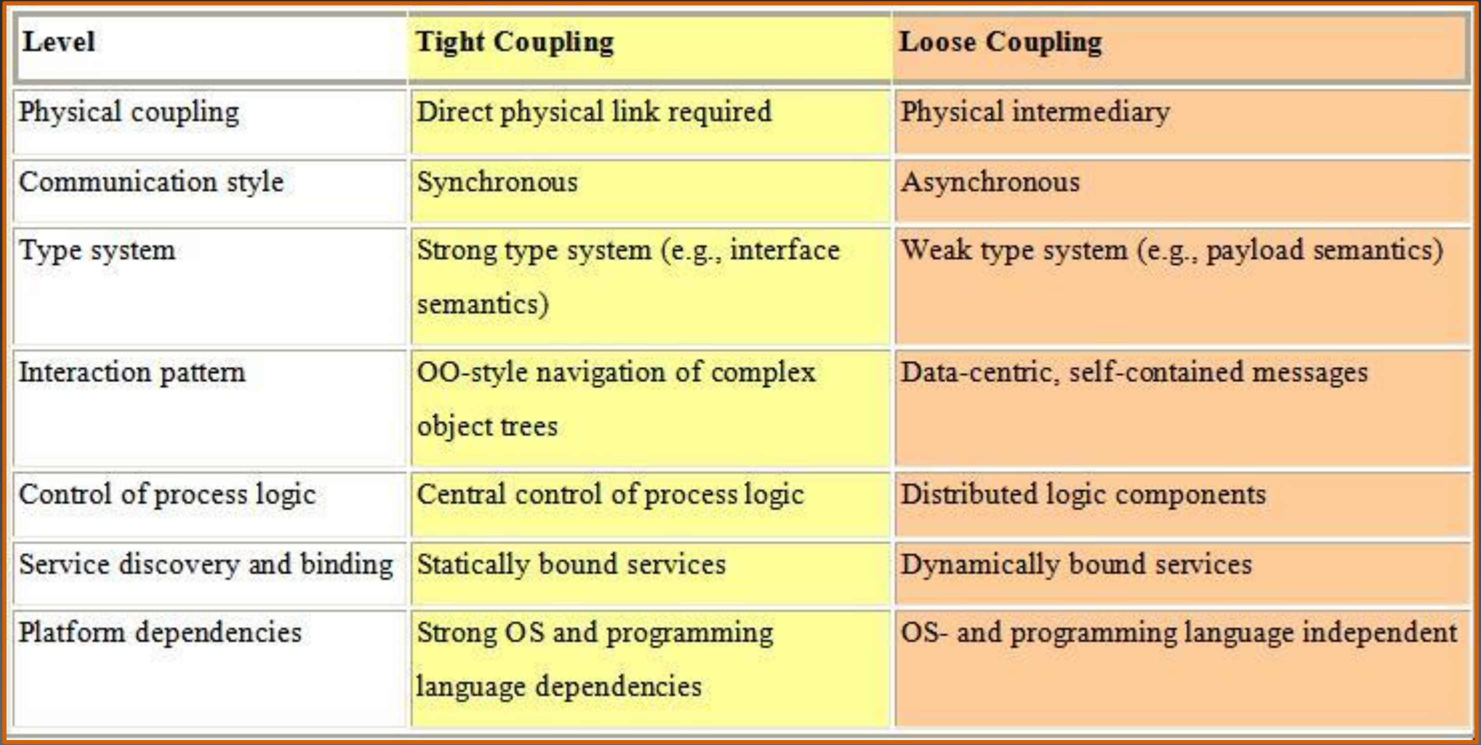
- Explain the benefits of loosely coupled systems
- Describe the difference between asynchronous and synchronous communications
- Map out the behavior of event based systems
- List some of the more important messaging platforms

Loosely Coupled Systems

- A loosely coupled system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components
- One of the most critical enablers of loosely coupled systems is messaging

Loose Coupling

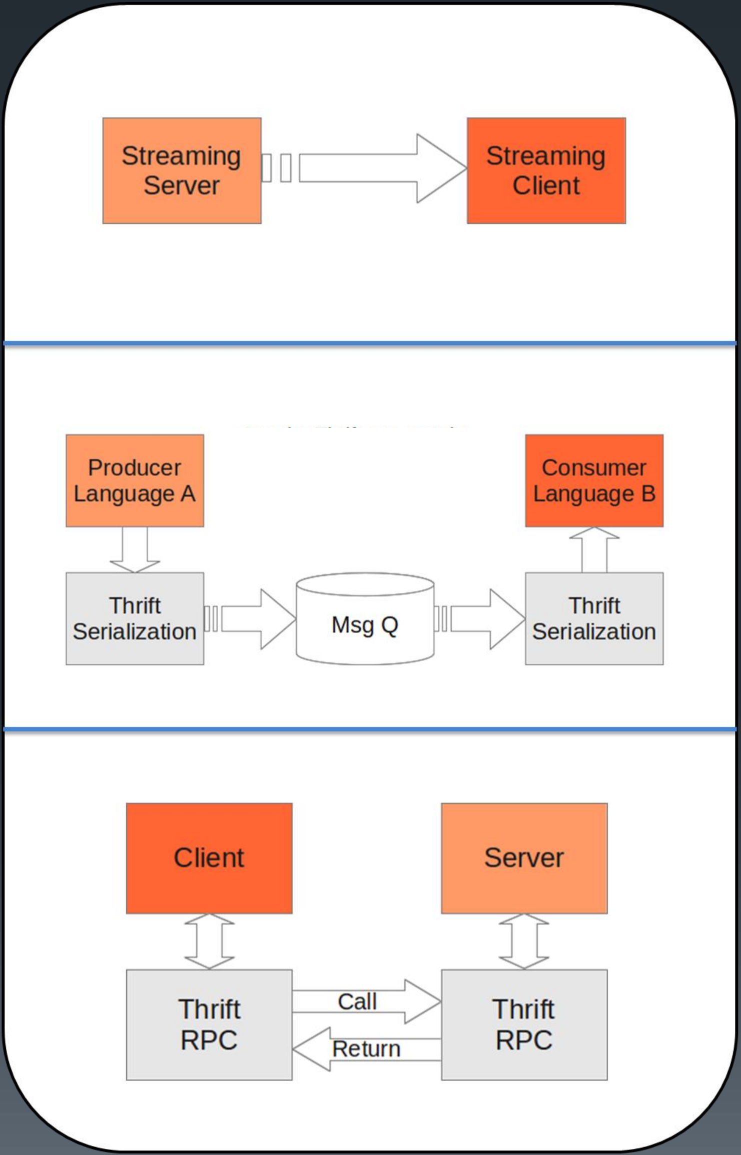
- Systems that are loosely coupled have independent lifespans
  - If service A crashes or is taken down, service B should not crash
  - Rather service B should either
    - Not know (as would be the case in many messaging scenarios)
    - Rediscover and reconnect (trying repeatedly over time as necessary)
  - Services directly depending upon other service (e.g. client/server relationships) should degrade favorably until the dependency is resolved
- Loose coupling is critical to overall application safety
  - Isolating failure modes rather than propagating them



Level	Tight Coupling	Loose Coupling
Physical coupling	Direct physical link required	Physical intermediary
Communication style	Synchronous	Asynchronous
Type system	Strong type system (e.g., interface semantics)	Weak type system (e.g., payload semantics)
Interaction pattern	OO-style navigation of complex object trees	Data-centric, self-contained messages
Control of process logic	Central control of process logic	Distributed logic components
Service discovery and binding	Statically bound services	Dynamically bound services
Platform dependencies	Strong OS and programming language dependencies	OS- and programming language independent

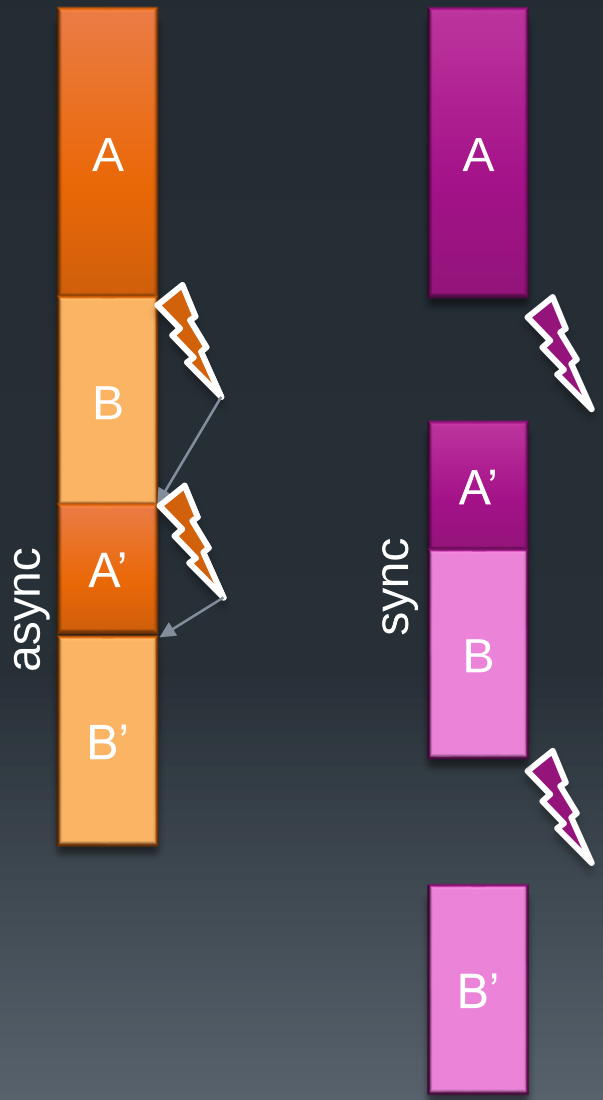
Communications Schemes

- **Streaming** – Communications characterized by an ongoing flow of bytes from a server to one or more clients.
  - Example: An internet radio broadcast where the client receives bytes over time transmitted by the server in an ongoing sequence of small packets.
- **Messaging** – Message passing involves one way asynchronous, often queued, communications, producing loosely coupled systems.
  - Example: Sending an email message where you may get a response or you may not, and if you do get a response you don't know exactly when you will get it.
- **RPC** – Remote Procedure Call systems allow function calls to be made between processes on different computers.
  - Example: An iPhone app calling a service on the Internet which returns the weather forecast.



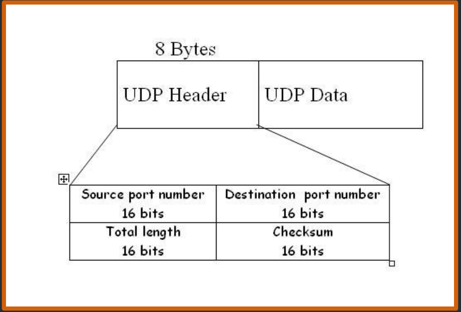
Asynchrony

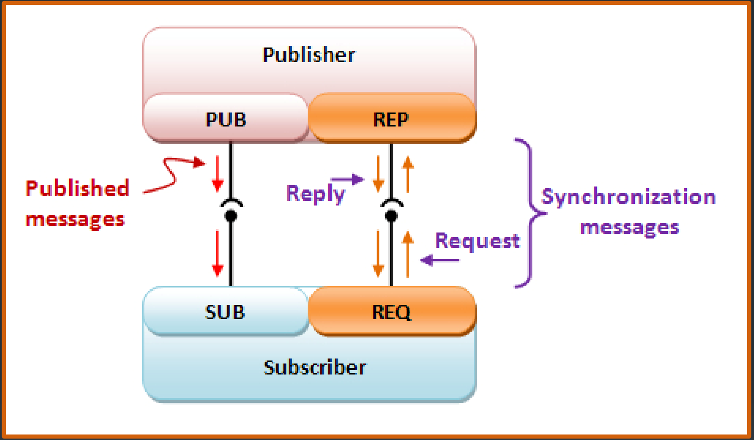
- Synchronous communication
  - Call block until the operation completes
  - Easy to reason about
  - One knows when things complete and what the status is
- Asynchronous communication
  - The caller doesn't wait for the operation to complete
  - May not even care whether or not the operation completes
  - Useful for long-running jobs
  - Good for low latency operations
- Streaming can be sync but is usually async
- Messaging is almost always async
- RPC can be either sync or async
- REST is sync by definition when over HTTP
  - Schemes like chunked responses and long polling attempt to alleviate

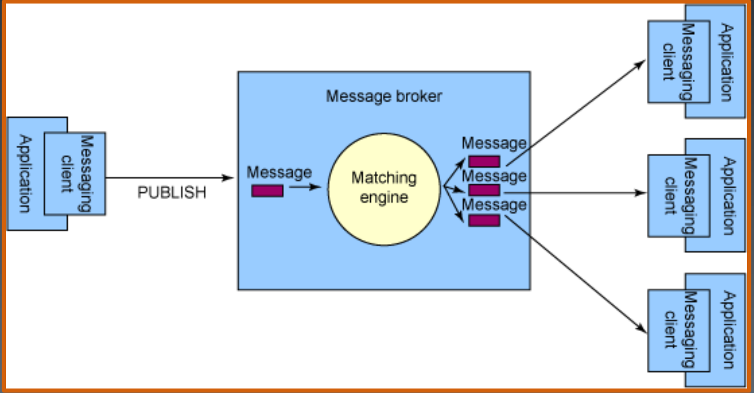


Types of Messaging

- Raw Network Messaging
  - UDP
  - No store, no forward
  - True Multicast, True Broadcast
  - Fast as it gets
- Library Based
  - Can offer some delivery assurances (retry)
  - Deploys in process
  - No server or middleware required
  - Examples
    - Nanomsg
    - ZeroMQ
- Message Oriented Middleware (MOM)
  - Store and forward
  - Pub/sub
  - Perhaps various Transaction Levels
  - Routing
  - More complicated deployment
  - Not the fastest messaging solution
  - Examples
    - NATS
    - Kafka
    - RabbitMQ
    - ActiveMQ
    - MSMQ
    - Redis







AMQP

- Advanced Message Queuing Protocol (AMQP)
- An OASIS open standard
- Application layer protocol for message-oriented middleware
  - message orientation
  - Queuing
  - routing (including point-to-point and publish-and-subscribe)
  - reliability
  - security
- AMQP mandates the behavior of the messaging provider and client to the extent that implementations from different vendors are truly interoperable, in the same way as SMTP, HTTP, FTP, etc. have created interoperable systems
- Previous attempts to standardize middleware have happened at the API level (e.g. JMS) and thus did not ensure interoperability
- Unlike JMS, which merely defines an API, AMQP is a wire-level protocol
- Any tool that can create and interpret messages that conform to this data format can interoperate with any other compliant tool irrespective of implementation language
- Broker Implementations
  - SwiftMQ, a commercial JMS, AMQP 1.0 and AMQP 0.9.1 broker and a free AMQP 1.0 client
  - Windows Azure Service Bus, Microsoft's cloud based messaging service
  - Apache Qpid, an open-source project at the Apache Foundation
  - Apache ActiveMQ, an open-source project at the Apache Foundation
  - Apache Apollo, open-source modified version of the ActiveMQ project at the Apache Foundation (threading functionality replaced and non-blocking techniques implemented more widely)
  - RabbitMQ an open-source project sponsored by Pivotal, supports AMQP 1.0 and other protocols via plugins

Java JMS 2.0

- Java Message Service (JMS) originally released in 2001
- Java based Message Oriented Middleware (MOM) API
- Used to send messages between two or more clients
- JMS is a part of the Java Platform, Enterprise Edition
  - JSR 914
- Allows the communication between different components of a distributed application to be loosely coupled, reliable, and asynchronous
- JMS 2.0 is the first update in over 10 years
  - Simplified API
  - Async send
  - Delayed delivery
  - Scaling through shared topics
- Not the best choice for polyglot microservices as it is Java specific

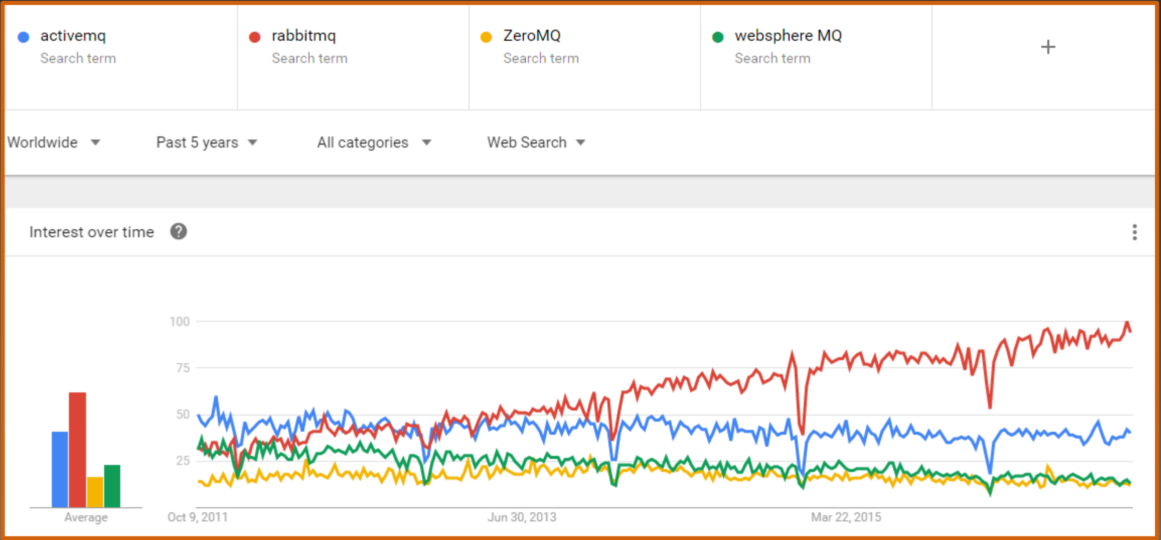
- Many Provider implementations
  - Apache ActiveMQ
  - Apache Qpid, using AMQP[5]
  - Oracle Weblogic and AQ from Oracle
  - EMS from TIBCO
  - FFMQ, GNU LGPL licensed
  - JBoss Messaging and HornetQ
  - JORAM, from the OW2 Consortium
  - Open Message Queue, from Oracle
  - OpenJMS, from The OpenJMS Group
  - Solace JMS from Solace Systems
  - RabbitMQ by Pivotal
  - SAP Process Integration ESB
  - SonicMQ from Progress Software
  - SwiftMQ
  - Tervela
  - Ultra Messaging from Informatica
  - webMethods from Software AG
  - WebSphere MQ FioranoMQ

MSMQ

- Microsoft Message Queuing [MSMQ]
- A message queue implementation developed by Microsoft and deployed in its Windows Server operating systems since Windows NT 4 and Windows 95
- The latest Windows 10/Server 2016 also includes this component
- MSMQ has been incorporated into Microsoft Embedded platforms since 1999 and the release of Windows CE 3.0
- Support for three transmission modes
  - Express – not written to disk, no ack
  - Reliable – written to disk, acked
  - Transactional – ACID, DTC eligible
- In .Net Systems, implemented through WCF Reliable Messaging
- Not the best choice for cross platform microservices as it is Windows specific

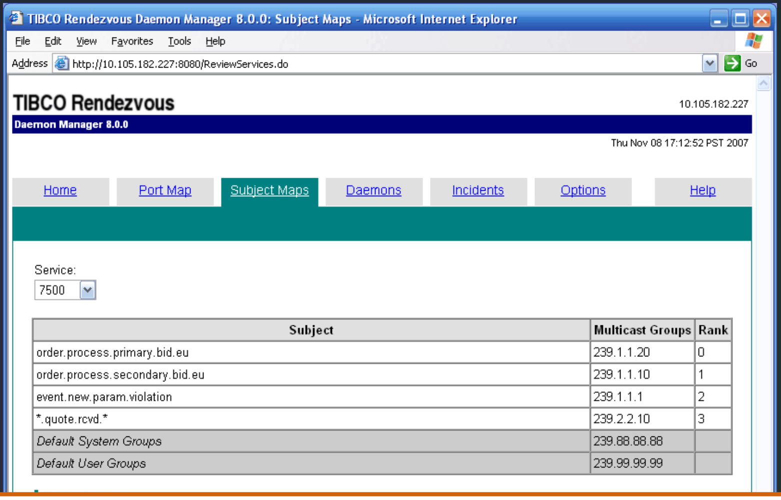
High Profile Messaging Platforms

- Websphere MQ – The first MOM system released in 1992
  - Originally known as MQ Series
  - A revelation
  - Still important but loosing the spotlight to open source and others
- ActiveMQ – Most popular community open source MQ system
  - Supports everything (JMS, JDBC, STOMP, XMPP, MQTT, REST, OpenWrite, …)
  - **Apache Apollo** – rewrite of ActiveMQ in Scala, much simpler and faster
- Apache QPID – AMQP driven messaging (100% AMQP support)
  - RedHat MRG enterprise messaging product is based on this
- RabbitMQ – Most popular commercially backed open source MQ system
  - Erlang based, elegant, fast, reliable
- ZeroMQ – Library based messaging, thus very fast, but brokerless, requiring rendezvous



High Performance Messaging Systems

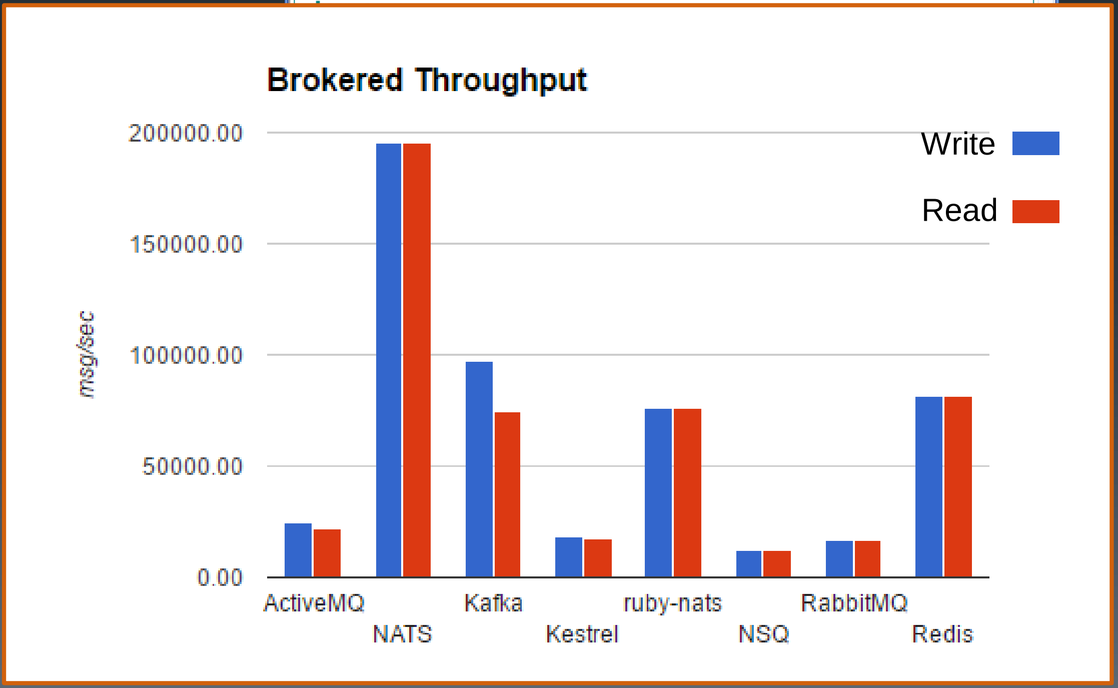
- Tibco Rendezvous [RV] – frequently used in financial applications, fast, expensive
- ZeroMQ – High performance library only messaging, open source
- Informatica Ultra Messaging – Formerly 29West, fast, expensive (a little cheaper than Tibco)
- NATS - open source messaging broker designed for microservices and cloud native applications
- Apache Kafka - a distributed platform supporting pub/sub, processing and storage for streams of data in a distributed, replicated cluster


TIBCO Rendezvous Daemon Manager 8.0.0: Subject Maps - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://10.105.182.227:8080/ReviewServices.do
Go
Home
Port Map
Subject Maps
Daemons
Incidents
Options
Help

TIBCO Rendezvous
Daemon Manager 8.0.0
10.105.182.227
Thu Nov 08 17:12:52 PST 2007

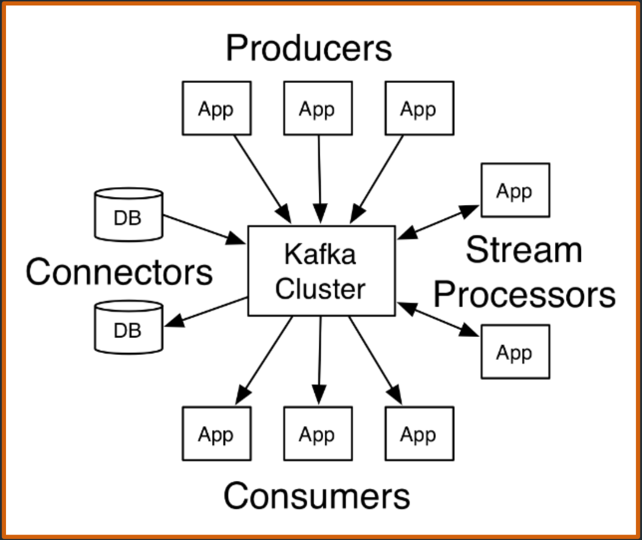
Service: 7500

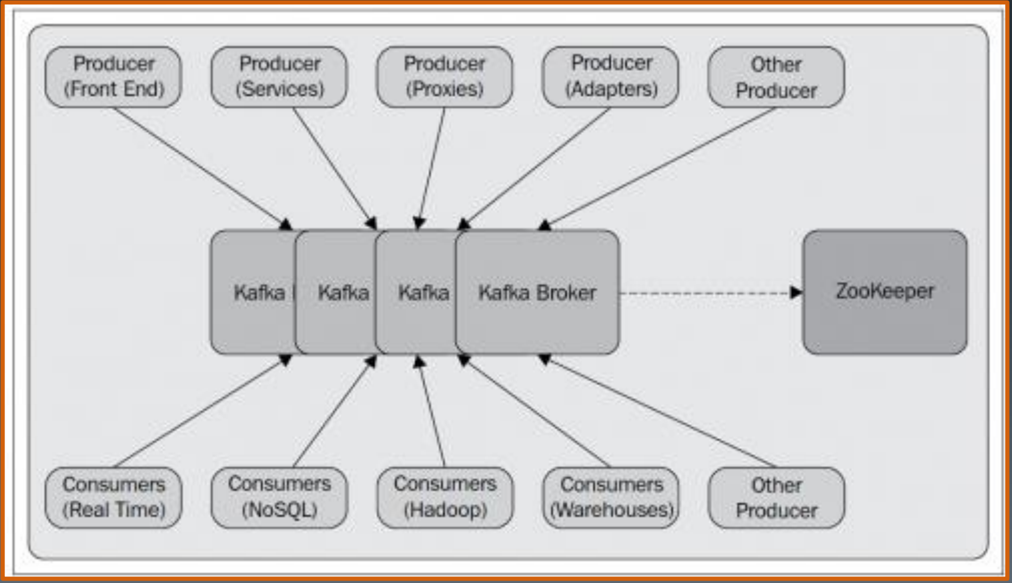
Subject	Multicast Groups	Rank
order.process.primary.bid.eu	239.1.1.20	0
order.process.secondary.bid.eu	239.1.1.10	1
event.new.param.violation	239.1.1.1	2
.quote.rcvd.	239.2.2.10	3
Default System Groups	239.88.88.88	
Default User Groups	239.99.99.99	



Distributed Messaging

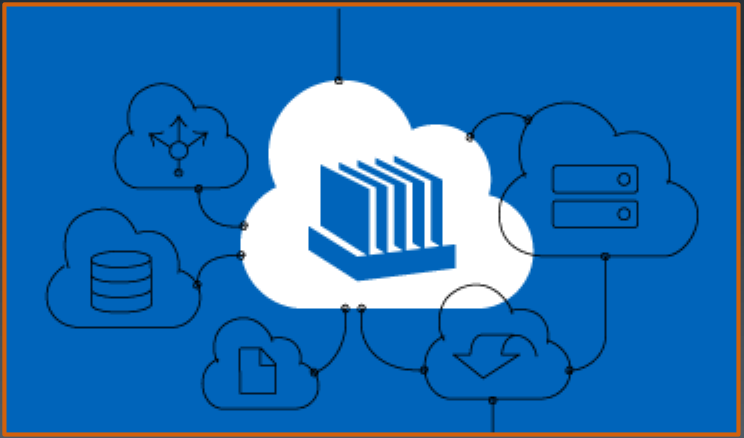
- Apache Kafka
  - Open source, publish-subscribe message broker
  - Amazon Kinesis Streams, managed Kafka
- Implemented as a distributed commit log
- Written in Scala
- Unified, high-throughput, low-latency platform for handling real-time data feeds
- Developed at LinkedIn, open sourced in 2011
- Designed to allow a single cluster to serve as the central data backbone for an organization
- Can be elastically and transparently expanded without downtime
- A single Kafka broker can handle hundreds of megabytes of reads and writes per second from thousands of clients
- Data streams are partitioned and replicated over a cluster of machines
  - Allows data streams larger than the capability of any single machine and to allow clusters of coordinated consumers





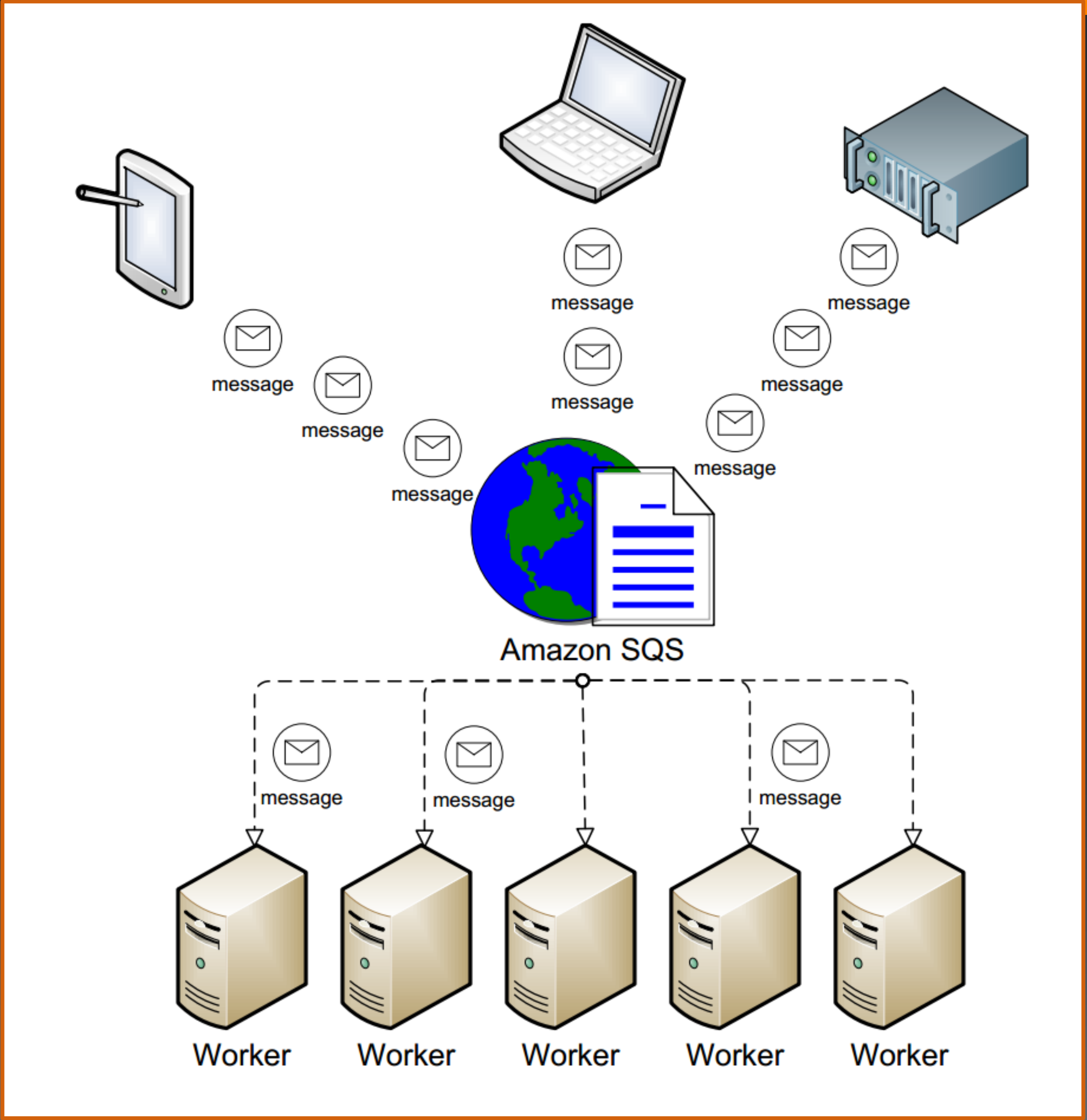
Cloud Based Messaging

- Amazon Simple Queue Service (SQS) is a fast, reliable, scalable, fully managed message queuing service
  - SQS is simple and can transmit large volumes of data, at any level of throughput, without losing messages or requiring other services to be available
- Amazon Simple Notification Service
- Amazon Kinesis Streams
- Azure Queue
- Google Cloud Pub/Sub
- Rackspace offers CloudQueues
- Heroku offers RabbitMQ
- Etc.



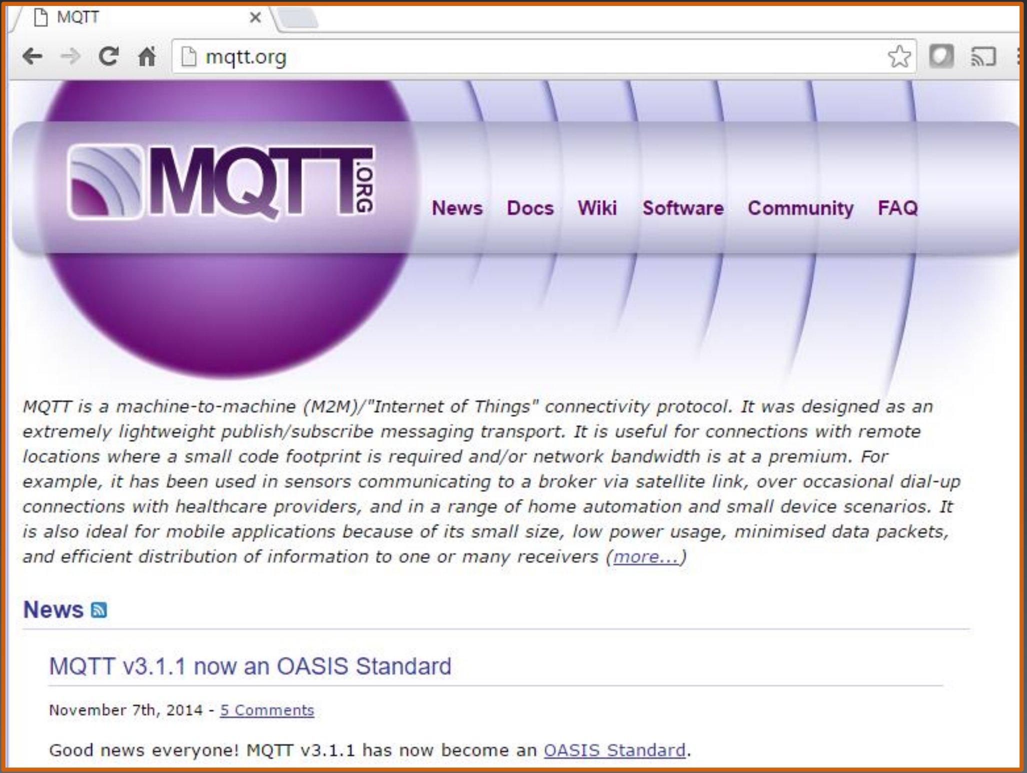
Amazon SQS

- Amazon Simple Queue Service (SQS) is a reliable distributed messaging system
  - A reasonable choice for fault-tolerant applications
  - At least once or at most once delivery
- Messages are stored in user defined queues
  - Each queue is accessed by URL
  - Available to the Internet
  - An Access Control List (ACL) determines access to the queue
- Any messages that you send to a queue are retained for up to four days (or until they are read and deleted by an application)
  - Messages read are hidden for the hide window time until acked
  - Message delivery can have a preconfigured delay
- Amazon Kinesis, a fully managed real-time streaming data service
  - Hosted Apache Kafka
  - Designed for streaming big data
  - Multiple applications can receive the same data
  - Data is delivered in order and cached for up to 24 hours
    - i.e. A client can replay data from the prior 24 hours



IoT Messaging

- MQTT (formerly MQ Telemetry Transport)
  - ISO standard PRF 20922
  - Publish-subscribe-based "lightweight" messaging protocol for use on top of TCP/IP
  - Designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited
- Powers Facebook Messenger
- The publish-subscribe messaging pattern requires a message broker
  - EMQ: an open source highly scalable MQTT broker
  - MQTT can also work with or without a broker
- 1999: Cirrus Link Solutions Stanford-Clark/Nipper develop MQTT
- 2013: IBM submitted MQTT v3.1 to OASIS
  - The "MQ" in "MQTT" came from IBM's MQ Series message queuing product
- MQTT-SN is a variation of the main protocol aimed at embedded devices on non-TCP/IP networks, such as ZigBee
- Alternative protocols include:
  - Advanced Message Queuing Protocol
  - IETF Constrained Application Protocol
  - XMPP
  - Web Application Messaging Protocol (WAMP)



MQTT is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol. It was designed as an extremely lightweight publish/subscribe messaging transport. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium. For example, it has been used in sensors communicating to a broker via satellite link, over occasional dial-up connections with healthcare providers, and in a range of home automation and small device scenarios. It is also ideal for mobile applications because of its small size, low power usage, minimised data packets, and efficient distribution of information to one or many receivers (more...)

News

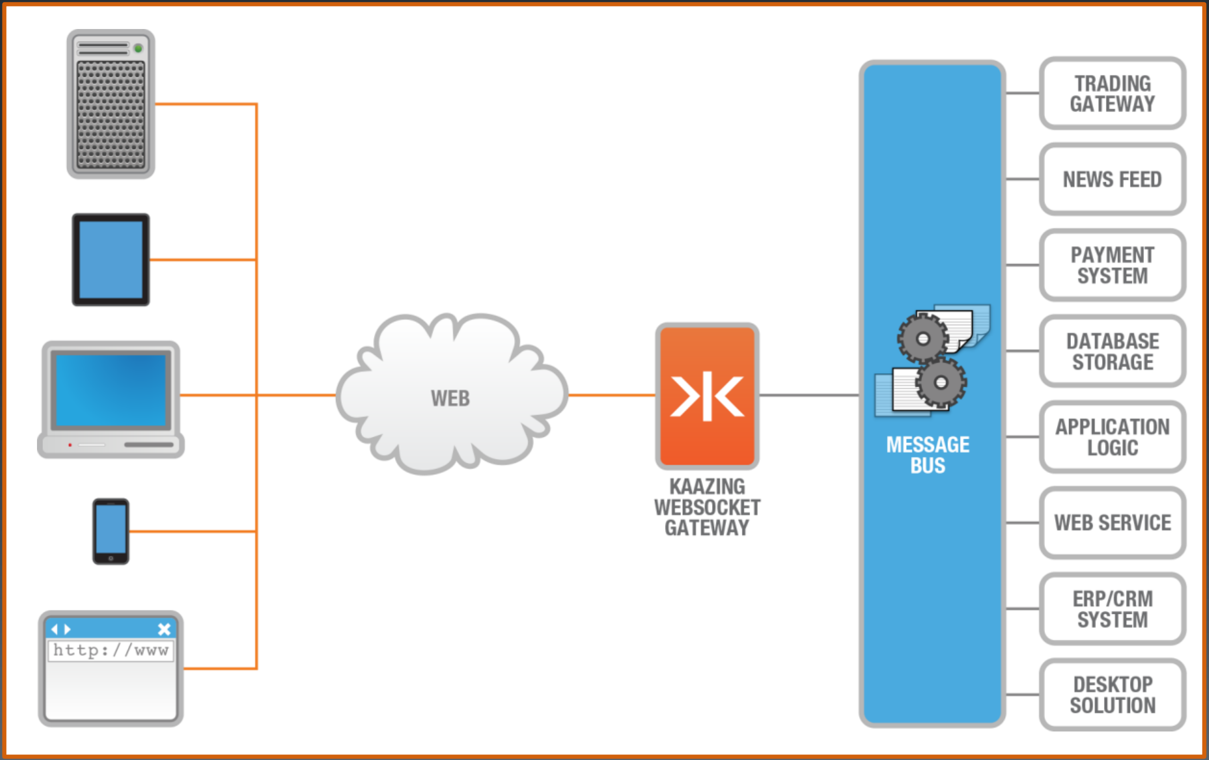
MQTT v3.1.1 now an OASIS Standard

November 7th, 2014 - 5 Comments

Good news everyone! MQTT v3.1.1 has now become an OASIS Standard.

Web Socket

- Web Socket brings high performance messaging to the Web
  - Suitable for Messaging and Streaming
  - Supports async communications in both directions
- Web Socket brokers/gateways like Kaazing provides a messaging protocol which operates over Web Sockets
- Apache Thrift RPC can operate over Web Socket



Summary

- Loosely coupled systems promote service independence
- Asynchronous systems do not wait for operations to complete before returning
- Event based systems allow subscribers to come and go without disturbing the sender
- A wide range of messaging platforms with several key target markets exist, many of which are useful in microservice based applications

Lab 4

- Building an RPC service

5: Cloud Native Transactions and Event Sourcing

Objectives

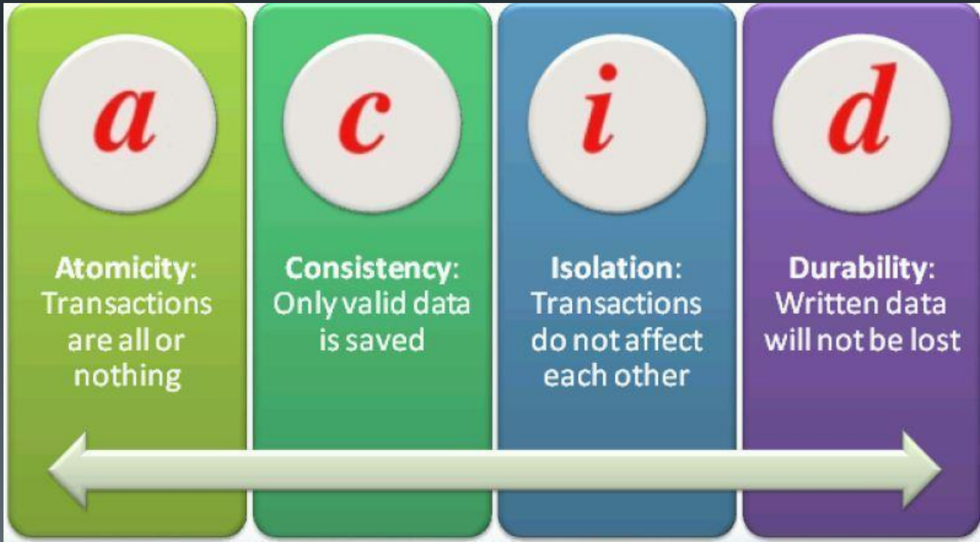
- Define transaction
- Discuss the challenges facing distributed transactions
- Contrast ACID transactions with eventual consistency
- Explain the CAP theorem
- Describe the value of Etags, Event Sourcing and CQRS

State

- State def:
  - All the stored information, at a given instant in time, to which the program has access
- The output of a computer program at any time is completely determined by its current inputs and its state
  - If your state goes bad, your application breaks
- State management presents the single most challenging facet of distributed application design

ACID

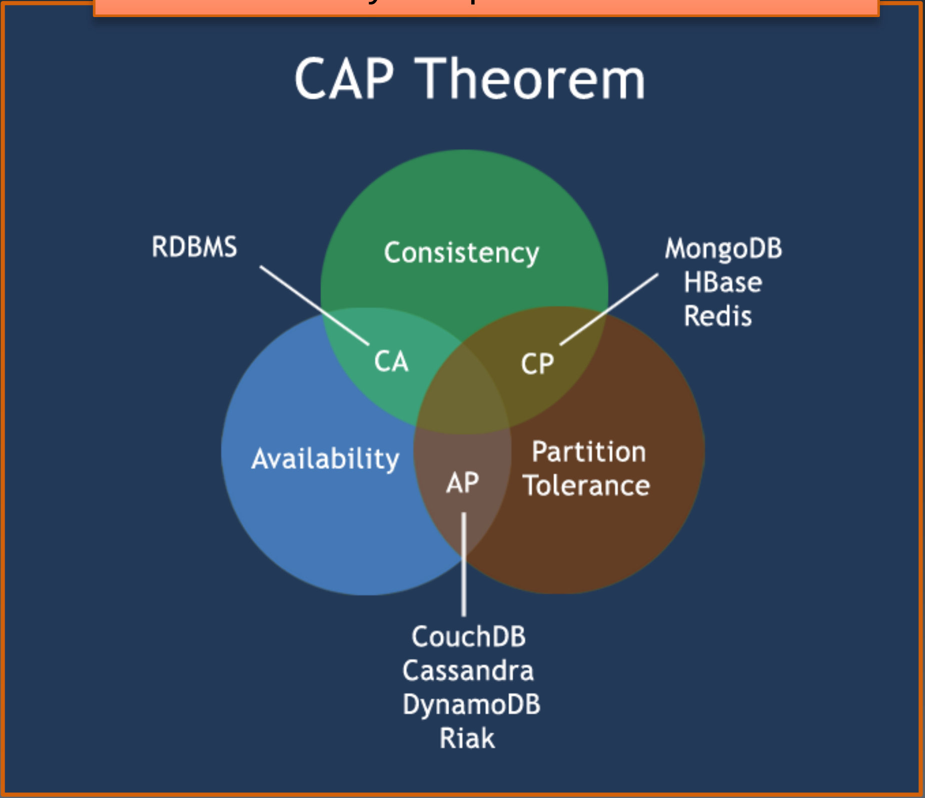
- ACID is a set of properties that guarantee that database transactions are processed reliably
  - Atomicity - requires that each transaction is "all or nothing"
  - Consistency - ensures that any transaction will bring the database from one valid state to another
  - Isolation - ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially
  - Durability - means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors
- Transactions - In the context of databases, a single logical operation on the data is called a transaction
- Jim Gray defined the properties of a reliable transaction system in the late 1970s and developed technologies to achieve them automatically
- In 1983, Andreas Reuter and Theo Härder coined the acronym ACID to describe them


a
c
i
d
Atomicity: Transactions are all or nothing
Consistency: Only valid data is saved
Isolation: Transactions do not affect each other
Durability: Written data will not be lost

The CAP Theorem

- The CAP theorem states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:
  - Consistency - all nodes see the same data at the same time
  - Availability - guarantees every request receives a response whether successful or failed
  - Partition tolerance - the system operates despite message loss or failure of part of the system
- CAP consistency means that any data item has a value reached by applying all the prior updates in some agreed-upon order
  - A consistent service must never forget an update once it has been accepted and the client has been sent a reply
- Availability is a mixture of performance and fault-tolerance
  - A service should keep running and offer rapid responses even if a few replicas have crashed or are unresponsive, and even if some of the data sources it needs are inaccessible
  - No client is ever left waiting, even if we cannot get the needed data
- Partition tolerance means that a system should be able to keep running even if the network itself fails, cutting off some nodes from the others
  - Partitioning is not an issue within modern data centers only across data centers

http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf



The CAP impact on the Cloud

- The CAP Principle was developed by Berkeley Professor Eric Brewer (Brewer 2000)
  - A CAP Theorem was thereafter proved by MIT researchers Seth Gilbert and Nancy Lynch (Gilbert and Lynch 2002)
  - The theorem addresses a strict ACID view of consistency
  - Brewer has since made three key points
    1. Because partitions are rare, there is little reason to forfeit C or A when the system is not partitioned
    2. The choice between C and A can occur many times within the same system at very fine granularity
       - Not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved
    3. All three properties are more continuous than binary
- Brewer argues that the value of quick responses is so high that even a response based on stale data is often preferable to leaving the external client waiting

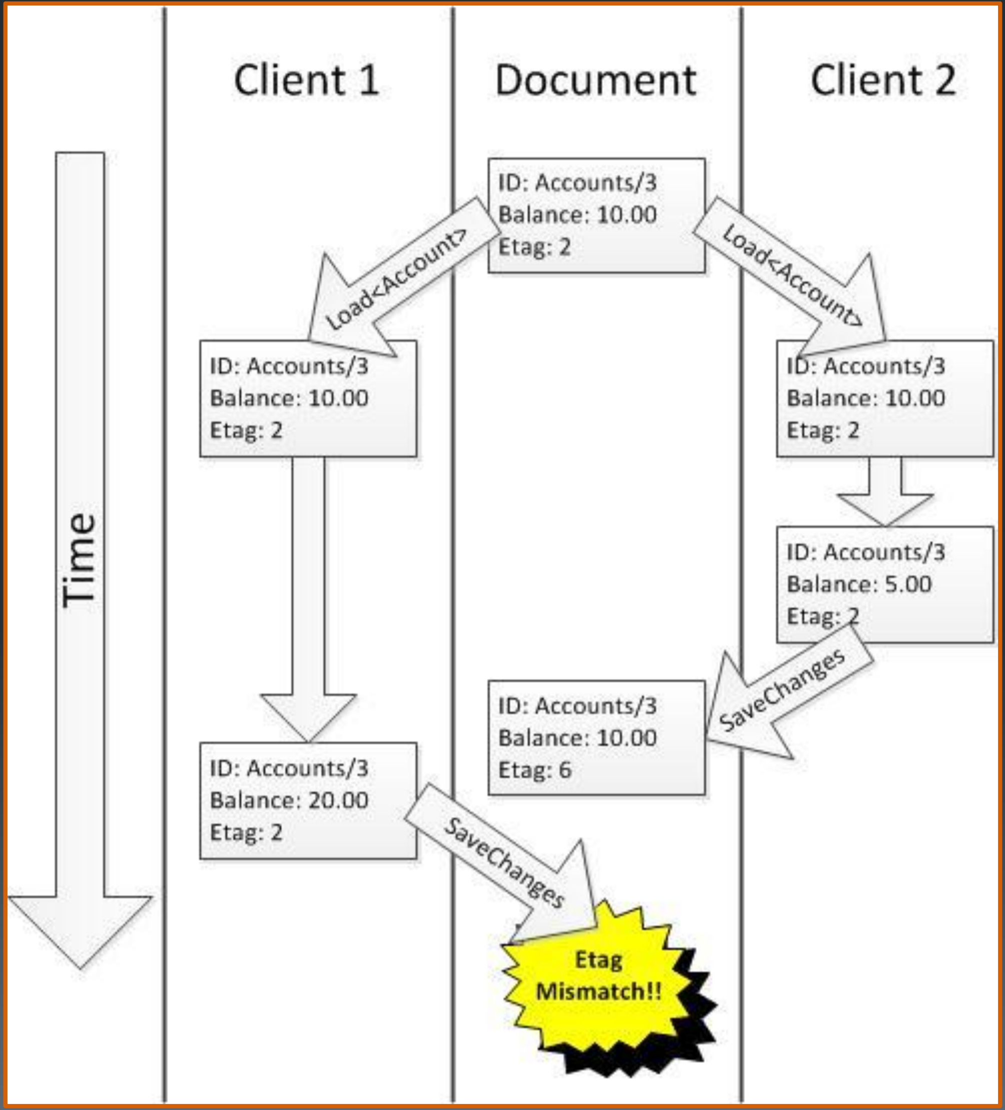
When a web page renders some content with a “broken” icon to designate missing or unavailable content, we are seeing CAP in action

How CAP plays out in application design

- Brewer argues, that it is better to initially run in an optimistic mode when designing large scale applications
  - E.g. Booking the sale without checking the remaining inventory stock
  - A question of priorities
    - scale and performance versus absolute accuracy
- The CAP theorem suggests that sometimes a system should run acceptable risks if by doing so it can offer better responsiveness and scalability
- Consistency in CAP is really a short-hand for two properties
  - An order-based consistency property
    - Updates to data replicated in the system will be applied in the same order at all replicas
  - A durability property
    - Once the system has committed to do an update, it will not be lost or rolled back
- This conflation of consistency with durability is important
  - Durability is expensive and turns out to be of limited value in the first tier of the cloud where services do not keep permanent state

Distributed State

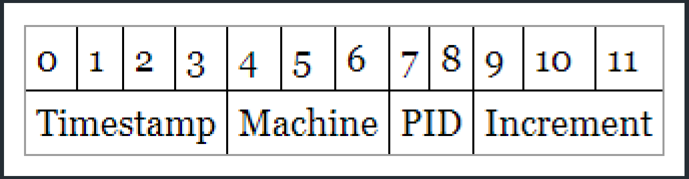
- The web works so well because it is based on:
  - Representations of resources
    - You never have the real resource state, only a representation
    - Your representation may be old the moment you receive it
  - Caching
    - Many web servers would be crushed if they had to directly support all of the clients using pages that they are responsible for
    - The fabric of caches on the web is massive and offloads many servers by one or more orders of magnitude
- The idea that the data you receive might be stale as soon as you receive it is tied up with eventual consistency
- Etags can be used as a state versioning mechanism to ensure that state changes are only applied if you are changing the most recent state



Key Generation

- Keys are a critical feature of distributed data systems
- Keys allow objects to be uniquely identified
  - This is usually a prerequisite for Idempotence
- For this reason keys are best generated at the time of datum/entity creation or ingestion
- Potential Key Algorithm Input Factors:
  - Explicitly assigned prefix
  - Machine ID
    - Process ID
      - Thread ID
  - Time
    - Perhaps highly granular (microsecond, nanosecond, etc.)
  - Monotonically increasing counters
  - Foreign/External Keys
- Unique keys require a set of factors sufficient to eliminate all orthogonal vectors of replication
- Critical to many Idempotent interface implementations

Considerations:
- Key compare time/rate
- Key generation time/rate
- Key storage (data & index)
- Key clustering
- Composite keys and implicit foreign keys

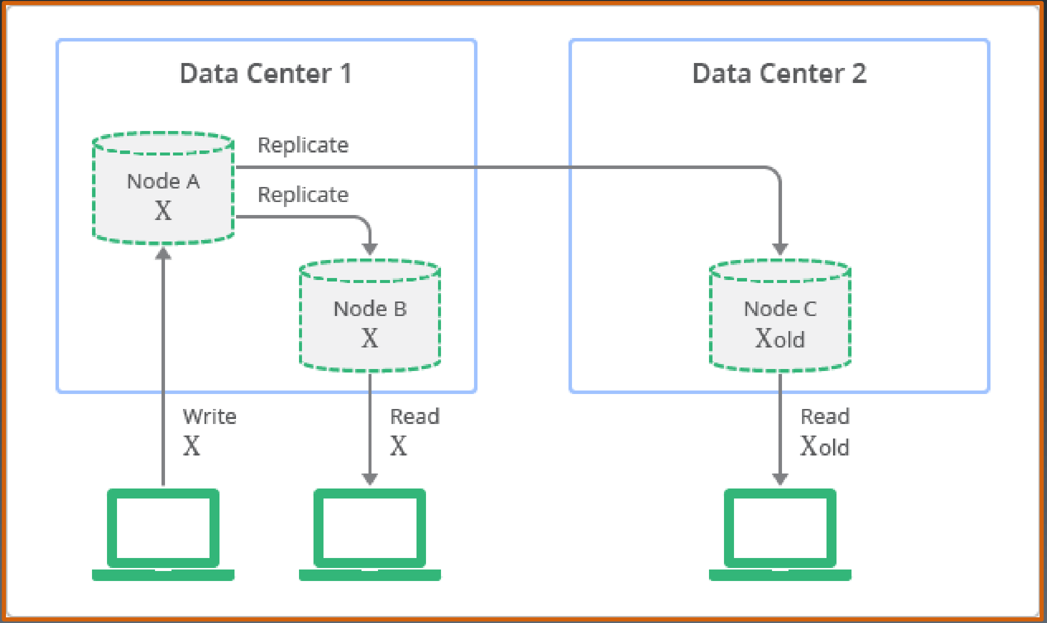


0	1	2	3	4	5	6	7	8	9	10	11
Timestamp				Machine			PID		Increment		

- MongoDB ObjectID (OID) example
- 12 bytes
- 0-3: timestamp in seconds since epoch
  - Provides uniqueness at the granularity of a second
  - The timestamp comes first so OIDs will sort in roughly insertion order, making OIDs efficient to index
  - Implicit creation timestamp
- 4-6: unique machine identifier (usually a hash of the machine's hostname)
  - Statistically ensures that OIDs on different machines will not collide
- 7-8: Process ID
  - Ensures concurrent processes generate unique IDs on the same machine
- 9-11: Increment responsible for uniqueness within a second in a single process
  - Allows up to 256^3 (16,777,216) unique OIDs to be generated per process in a single second
  - Must be synchronized in a multithreaded process

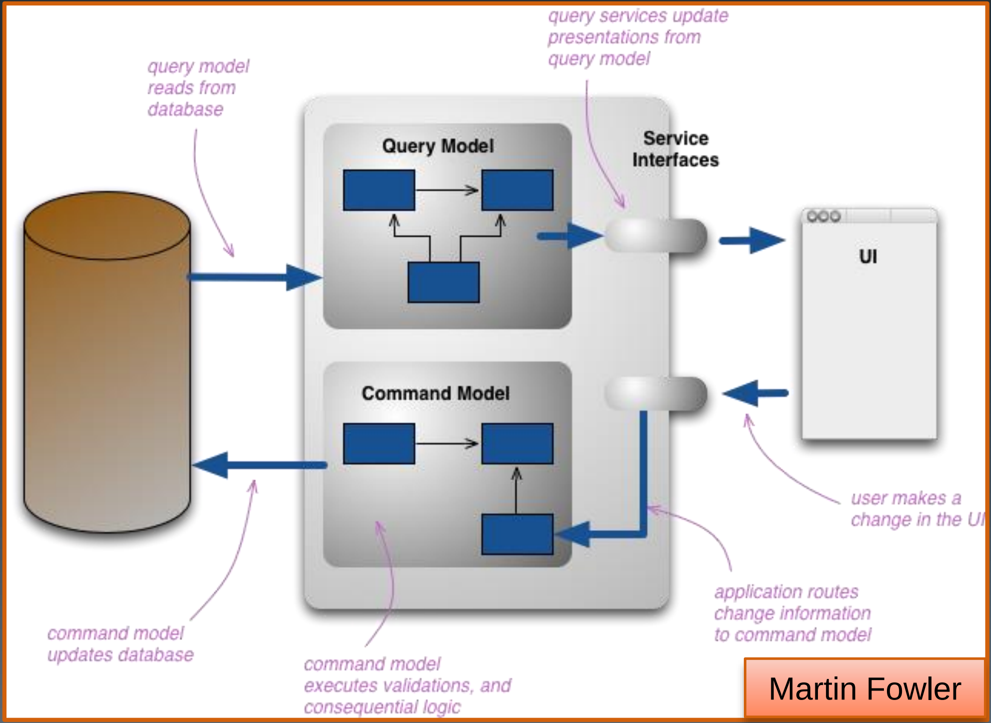
Eventual consistency

- Eventual Consistency
  - A consistency model used in distributed computing
  - Achieves high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value
- Quorum based systems can ensure strong consistency
  - Consul, etcd, zookeeper, etc.
  - Do not scale well due to the high communications overhead between nodes
  - Good for special purposes
    - Leader election
    - Small key/value bits of cluster state
  - Bad for large data storage
- Many times you can trade things you do not need for things you do
  - Read your own writes
    - Trading global consistency for local consistency
    - Others may not see your writes immediately but you will



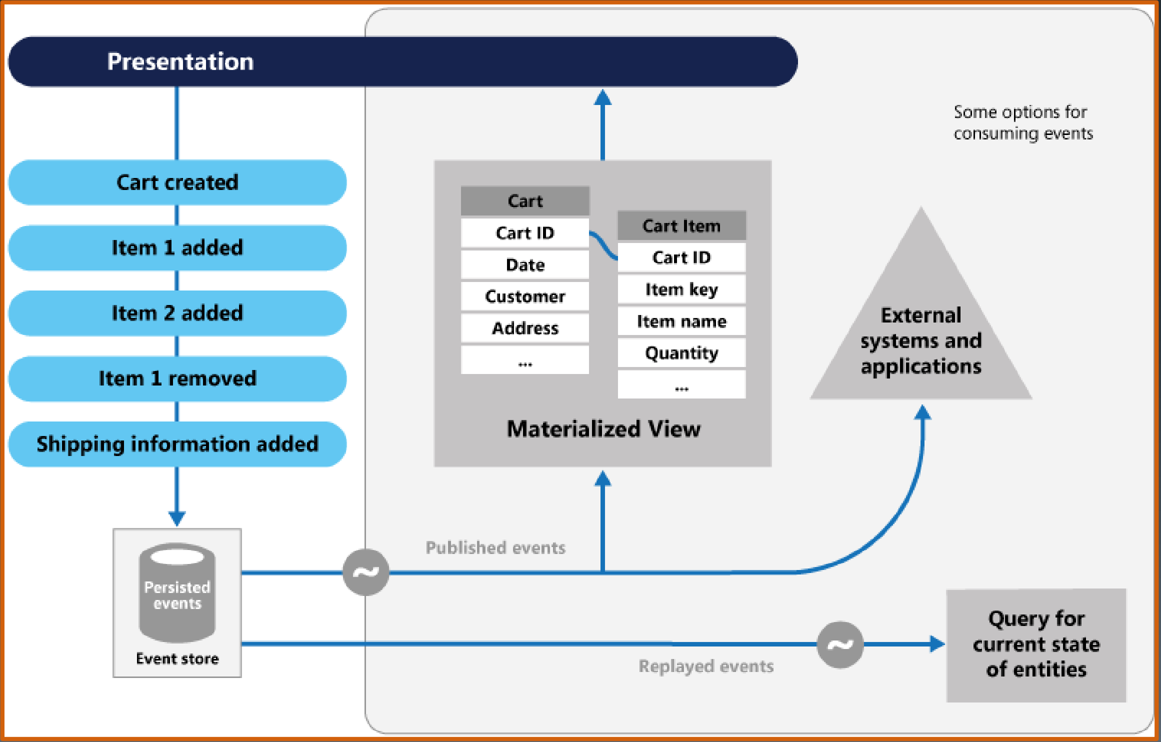
CQRS

- Command Query Responsibility Segregation (CQRS)
  - A pattern which uses one model to modify state and another to retrieve it
- CQRS can solve intractable problems or make things intractable
  - Should only be used when appropriate (the tax on incorrect usage is much high than with most patterns)
- CQRS differs from CRUD, which uses a single (typically synchronous) model for all operations
  - CRUD = create/read/update/delete operations on (typically) records in a database
- CQRS Commands
  - Modify state
  - Queued and asynchronous
  - Are more expensive
    - Often need to be atomic and/or serialized, perhaps involving some locking
- CQRS Queries
  - Retrieve data
  - Typically synchronous in CQRS
  - Cheap and fast
    - Can often be performed lock less
  - Must tolerate eventual consistency
    - Read your own writes becomes difficult


query services update presentations from query model
query model reads from database
Query Model
Service Interfaces
UI
Command Model
user makes a change in the UI
application routes change information to command model
command model updates database
command model executes validations, and consequential logic
Martin Fowler

Event sourcing

- Event Sourcing is the process of capturing all inputs as events
- Often combined with eager read derivation
  - Reads don't touch the main database
  - Reporting databases service most reads and are structured to accept all state change events and derive the typical read data model in advance of any user requests for data
  - This makes Queries even faster
- The state of an entity is established by looking at its event history
  - Each event is immutable
  - Balances and other "summary" type states can be computed and optionally cached if fast access is required
    - However the event history is always canonical



The Saga Pattern

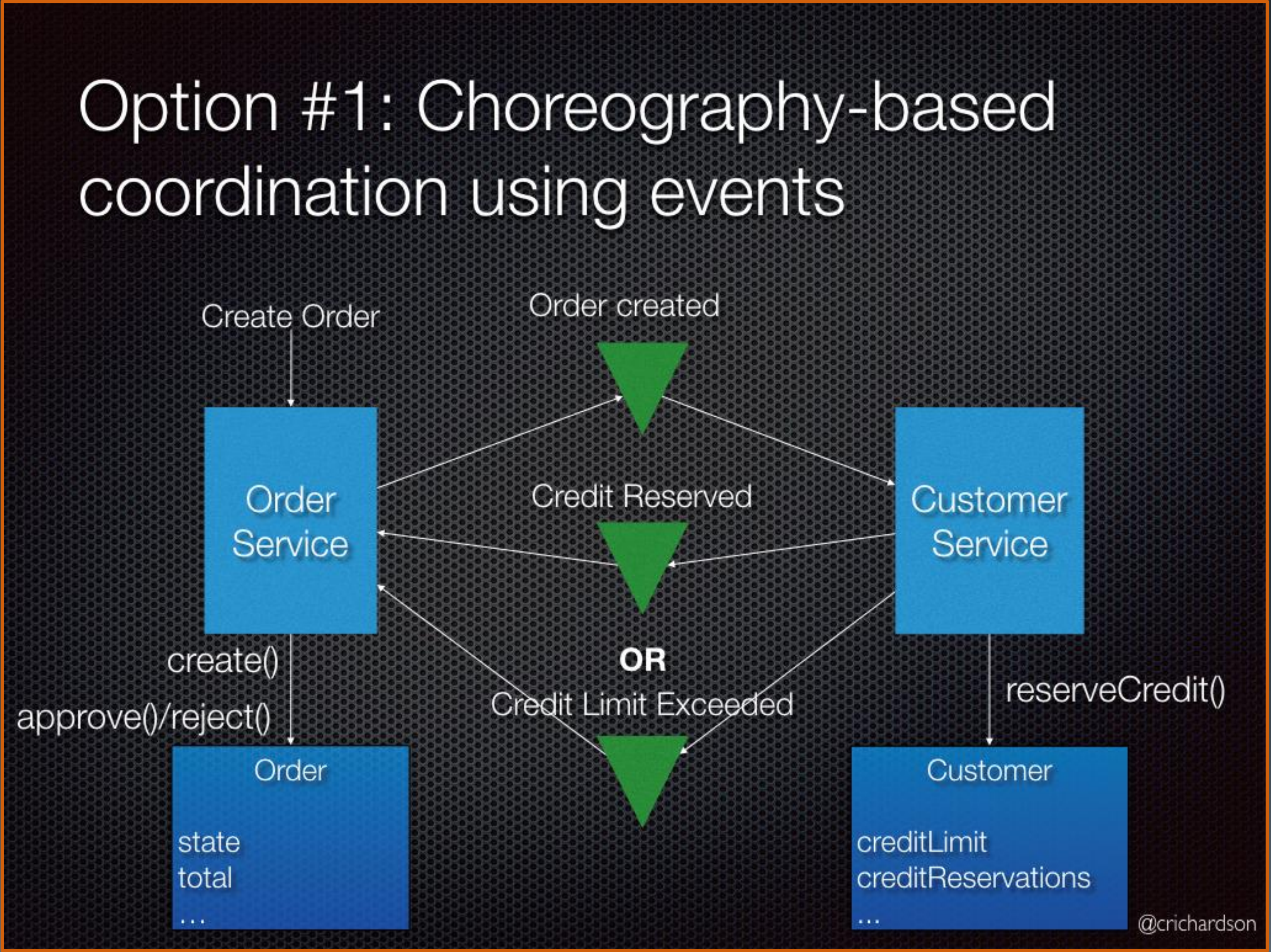
- Microservices are typically designed such that each service has its own database if it is stateful
- Some business transactions span multiple service
- A mechanism may be needed to ensure data consistency across services
  - E.g. an e-commerce store where customers have a credit limit, the application must ensure that a new order will not exceed the customer's credit limit, requiring coordination between the Order and Customer services
- A saga is a sequence of local transactions
  - Each local transaction updates the local state and publishes a message or event to trigger the next local transaction in the saga
  - If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions
- There are two common ways to coordinate sagas:
  - Choreography - each local transaction publishes domain events that trigger local transactions in other services
  - Orchestration - an orchestrator (object) tells the participants what local transactions to execute

Choreography

1. The Order Service creates an Order in a pending state and publishes an OrderCreated event

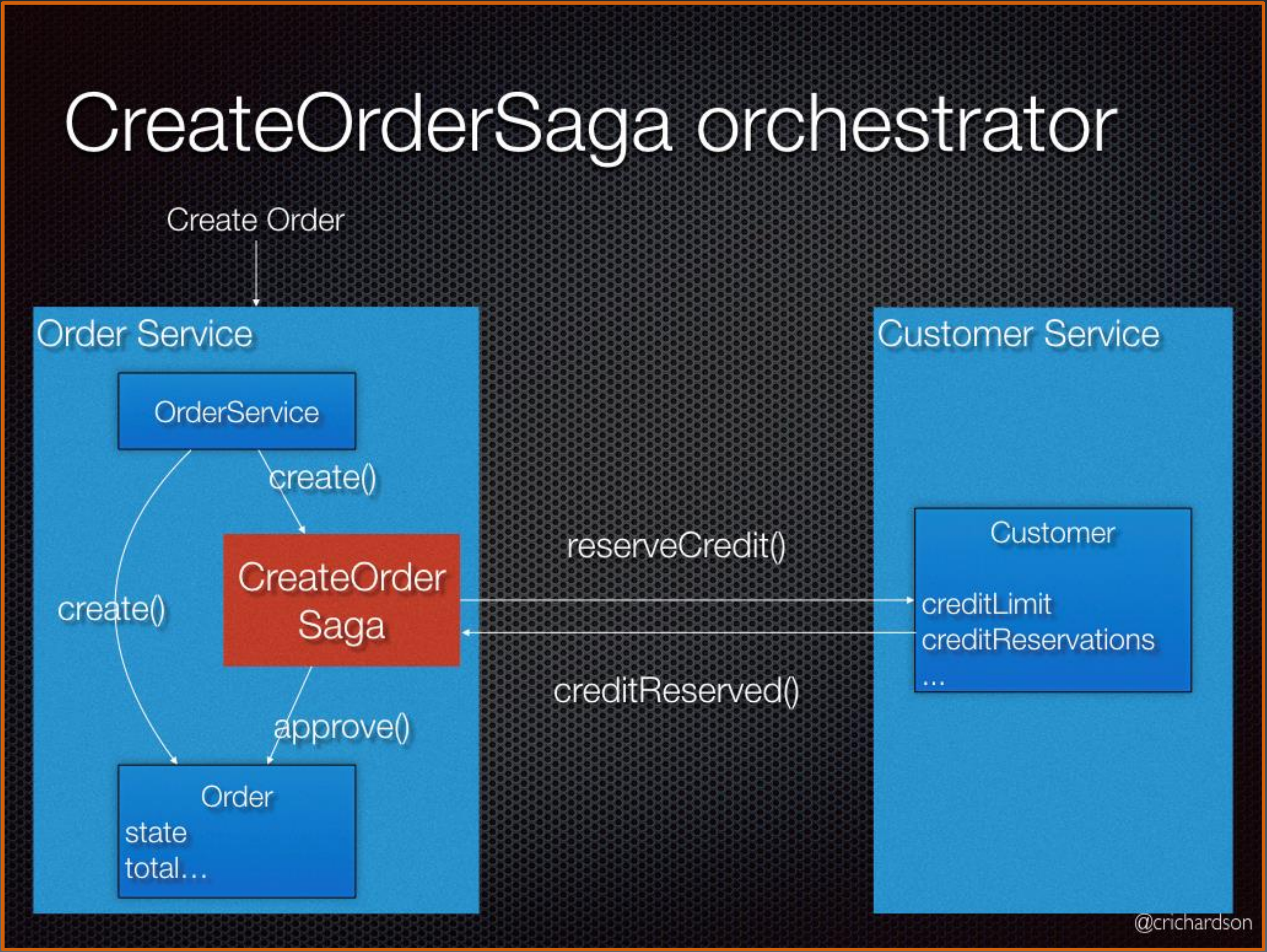
2. The Customer Service receives the event attempts to reserve credit for that Order. It publishes either a Credit Reserved event or a CreditLimitExceeded event.

3. The Order Service receives the event and changes the state of the order to either approved or cancelled



Orchestration

1. The Order Service creates an Order in a pending state and creates a CreateOrderSaga
2. The CreateOrderSaga sends a ReserveCredit command to the Customer Service
3. The Customer Service attempts to reserve credit for that Order and sends back a reply
4. The CreateOrderSaga receives the reply and sends either an ApproveOrder or RejectOrder command to the Order Service
5. The Order Service changes the state of the order to either approved or cancelled



Summary

- A transaction is a sequence of operations performed as a single logical unit of work
- Per the CAP theorem, distributed systems must balance consistency with availability when partitioned
- Eventual consistency implies that the state of an object at any given moment may be viewed differently in different parts of a distributed system but will eventually converge
- Etags are a web centric means of ensuring that one only updates a remote object if the object is in the expected pre-state
- Event Sourcing is a means to decouple state managers by generating broadcast events associated with all state changes
- CQRS – Command Query Responsibility Separation is a process where expensive state change commands are queued and fast/cheap state queries are synchronous

Lab 5

- Create a message driven loosely coupled service