Container-less Development or Immutable Containers?

Mark Little, VP, Red Hat

HPTS September 2015

Distributed systems archeology

1970's-1990's client-server rules Typically single-threaded Multi-threading in languages rare setjmp/longjump anyone? Core services/capabilities begin to emerge Transactions, messaging, storage, ...



CORBA (other architectures are available)



And then ...

 Late 90's/2000's
 New generation of chips, e.g., M68030, SPARC, Xeon, Itanium Multi-core, hyper-threads RAM sizes "explode"; access times too 64 Meg in Sun 3/80, 512 Meg Pentium 3 Network speeds improve more slowly





The application container

Encourages co-location of capabilities/ services

Improve performance & memory footprint

Application containers encapsulate and abstract

Thread pooling, transaction management, security, connection pooling, ...



"Make everything as simple as possible, but not simpler." – Albert Einstein

Java EE components



Application container backlash

- Not as simple to develop the "easy stuff"
- Application containers begin to be viewed as bloated
- Not everyone wants all enterprise services
 All-or-nothing approach to capabilities
 OSGi or MSC evolve to address dynamic updates

Java EE stripped down

Many developers are happy with Java EE Robust and mature components; well understood
 Scalable, standards compliant, integrates well Not everyone wants to use all of Java EE Stripping down is common Ditch the container to use components "raw" Ø WildFly-Swarm

Microservices

A monolithic application puts all its functionality into a single process...



... and scales by replicating the monolith on multiple servers









A microservices architecture puts each element of functionality into a separate service...

... and scales by distributing these services across servers, replicating as needed.











Along come Linux Containers!

VIRTUALIZATION



CONTAINERS





Such as Docker

 Solve the problem of moving applications between infrastructures

Ø Docker is disruptive:

 Technology Advantages – componentised applications packaged and separated in their own containers

Business Benefits – "standard" container means faster delivery as only one mechanism for packaging

Ecosystem – Docker, Google, OpenStack, Red Hat, RackSpace, IBM, VMWare, Microsoft and Amazon



Kubernetes

Open source project from Google

- The de facto standard for cluster management for Docker containers
- Packages Orchestration, service discovery, load balancing – all behind a simple rest API
- Backing from Google, IBM, Red Hat, Microsoft, Rackspace, Cloudbees etc.

Kubernetes and Containers

Substitution Kubernetes requires immutability of images Any changes to a running image are lost Can still be made but volatile
 State must be stored off image Shared (persistent) volumes, non-Container services etc.

Immutability simplifies architectures

Opposition of the second se New images can be created quickly Memory footprint still small Rip out some application container code Opdate? Create new image and redeploy! Keep connection pools, thread pools, dependency injection, ...

Enterprise capabilities

However, the need for transactions, reliable messaging etc. doesn't go away

Applications still need them

Application containers breaking into pieces

Independently deployable (Container) services

Available to different language clients using REST/HTTP and other protocols

Balls of mud made of services

"If you're building a monolithic system and it's turning into a big ball of mud, perhaps you should consider whether you're taking enough care of your software architecture. Do you really understand what the core structural abstractions are in your software? Are their interfaces and responsibilities clear too? If not, why do you think moving to a microservices architecture will help? Sure, the physical separation of services will force you to not take some shortcuts, but you can achieve the same separation between components in a monolith." http://www.infog.com/news/2014/08/ microservices ballmud