I'm Confused: Should I Orchestrate My Containers on Service Fabric or AKS?

Alex Mang
@iAmAlexMang

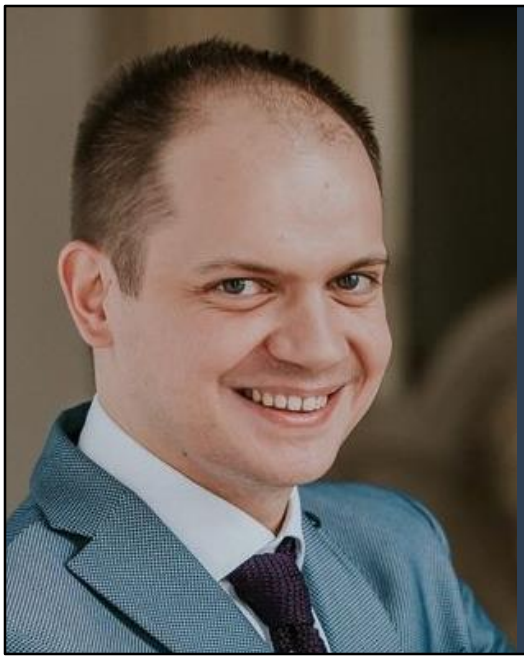
Unless you've been living under a rock…

- Containers are preferred choice of computing
  - 8.5% of time spent in the Valley is lost investment in HR
- Too many containers ➔ out of control
- Orchestrators to the rescue

Too many buzzwords?

- Microservice
- Container
- Docker
- Rkt
- Orchestrator
- Kubernetes
- Service Fabric

- …?



Alex Mang

@iAmAlexMang
www.alexmang.com

MVP
Microsoft
Most Valuable Professional

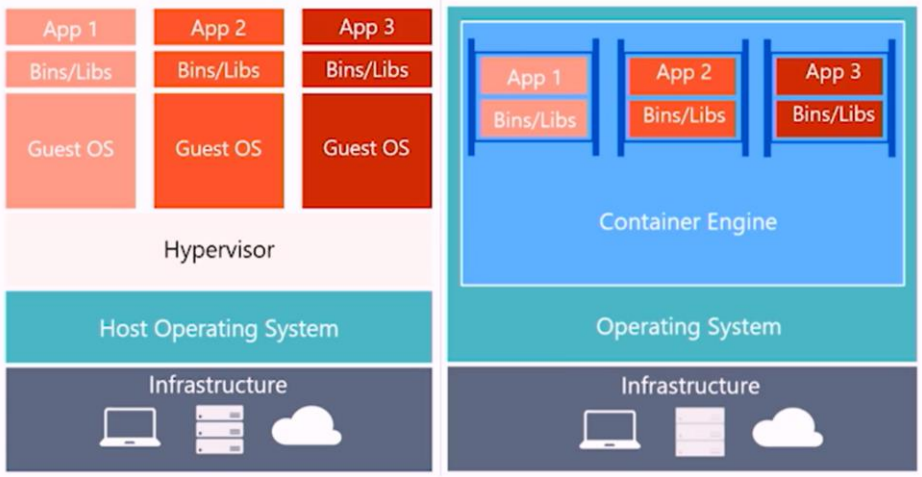
You!

Taboo subject

Let's remain friends
even after this session!

WELCOME CONTAINERS!

Virtual Machines vs. Containers



Docker

Docker is a platform for packaging, distributing and running applications.

Docker Image

The image contains the file system and the path to the application executable when the image is ran.

Docker Registry

Docker Registry is a repository of images which offers easy sharing of images between people and machines.

Docker Container

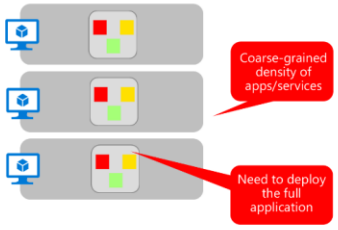
A running container is a process running on the host running Docker, but is completely isolated from both the host and all other processes running on it.

Once you have many containers…

- …many components, many moving parts
- …difficult inter-process communication
- …manual management can be difficult

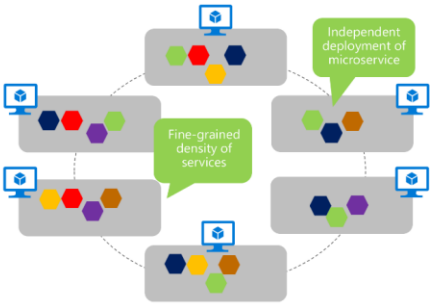
Monolithic deployment approach

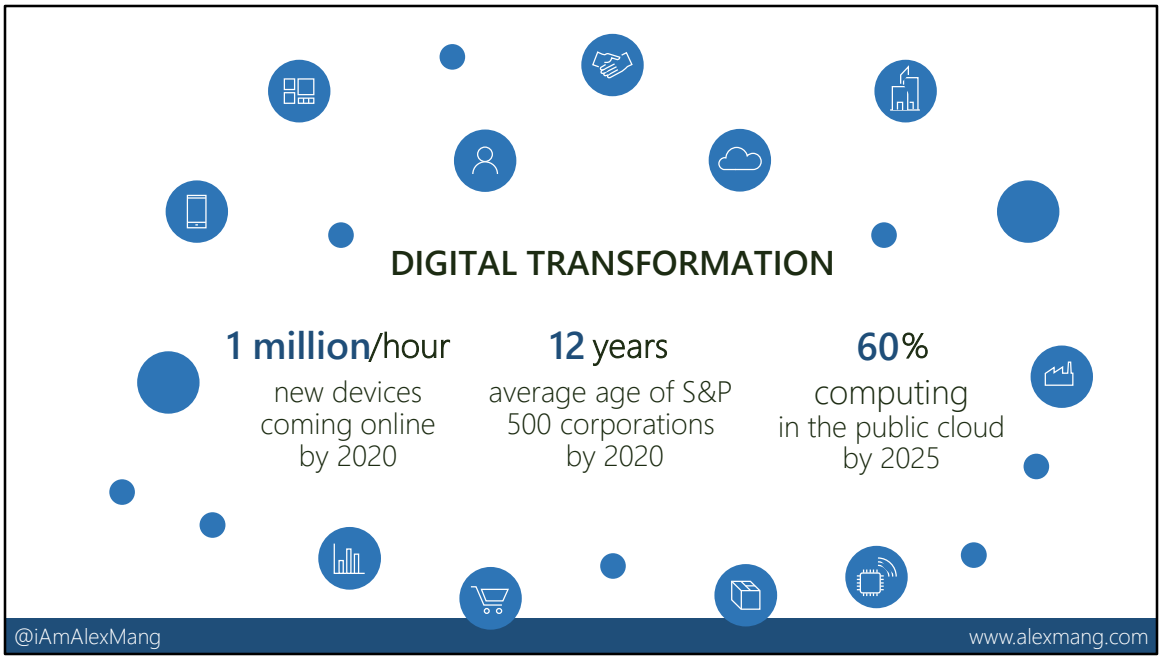
- A traditional application has most of its functionality within a few processes that are componentized with layers and libraries.
- Scales by cloning the app on multiple servers/VMs



Microservices application approach

- A microservice application segregates functionality into separate smaller services.
- Scales out by **deploying each service independently** with multiple instances across servers/VMs




DIGITAL TRANSFORMATION
1 million/hour
new devices coming online by 2020
12 years
average age of S&P 500 corporations by 2020
60%
computing in the public cloud by 2025
@iAmAlexMang
www.alexmang.com

WE NEED ORCHESTRATORS!

Kubernetes

- A software system that allows you to easily deploy and manage containerized applications on top of it
- Exposes the underlying infrastructure as a single computational resource
- Consistent deployment experience regardless of the size of the cluster

@iAmAlexMang www.alexmang.com

- Kubernetes is a software system that allows you to easily deploy and manage containerized applications on top of it.
- It enables you to run your software applications on thousands of computer nodes as if all these nodes were a single anonymous computer. It abstracts away the underlying infrastructure and doesn't care whether it contains physical machines or virtual machines.
- When you deploy your applications through Kubernetes, the process is always the same whether your cluster contains only a couple of nodes or thousands of them. The size of the cluster makes no difference at all. Additional cluster nodes simply translate to additional amount of resources available to your deployed applications.

Kubernetes System

- A developer submits apps through app descriptors
  - These are submitted to the master
  - The master analyzes them and deploys them to the workers
- As a developer, you don't care on which node the app runs on

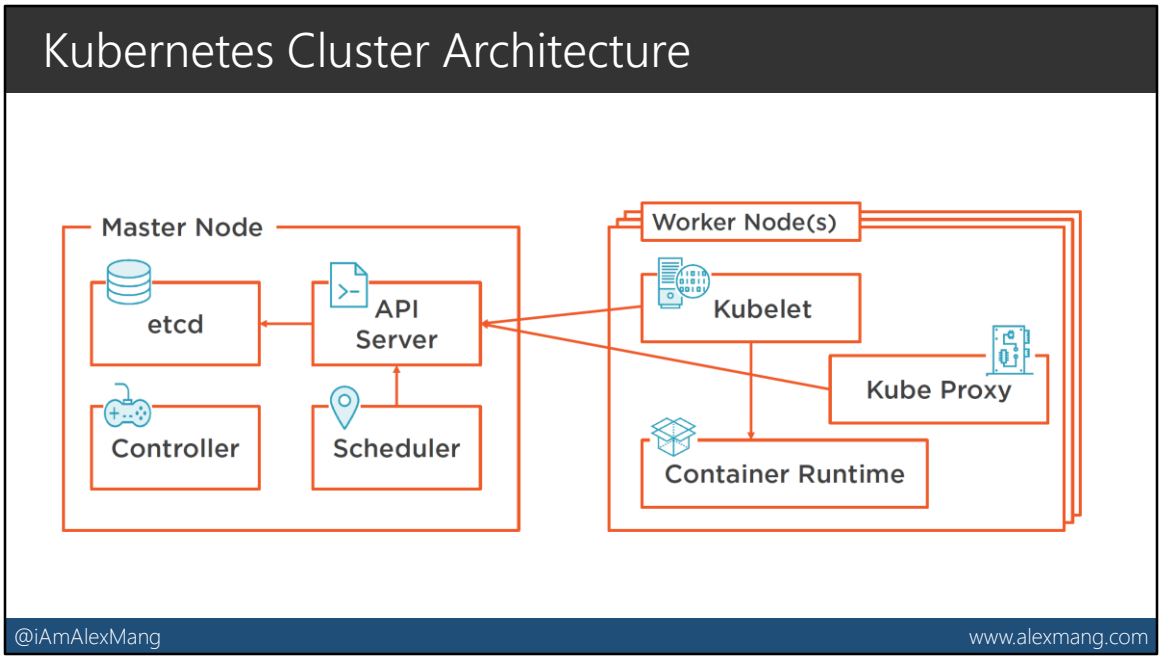
@iAmAlexMang www.alexmang.com

- It all starts with the developer. The developer submits the list of applications, typically called an app descriptor, to the Kubernetes master. The master looks at the app descriptor, figures out what to do, and then deploys those applications onto the worker nodes.
- As a developer, you don't care which node your application runs on. Your only concern is that you want X amount of copies of your application running, and Kubernetes handles that for you. The developer can specify that certain applications must run together, and Kubernetes will deploy them on the same worker node. Others will be spread out on the cluster, but they can still talk to each other in the same way regardless of where they are deployed

Kubernetes Cluster Node Types

- Master Nodes
- Worker Nodes
  - aka 'minions'

The system is composed of a master node and any number of worker nodes. The worker nodes were initially called as minions, but now just referred as worker nodes



The Kubernetes API server, which you and the other control plane components communicate with; the scheduler, which schedules your applications, basically assigning a worker node to each deployable component of your application; the controller manager, which performs the cluster-level functions, such as replicating component, keeping a track of worker nodes, handling node failures, and so on; etcd, which is a reliable distributed data store that persistently stores the cluster configuration.

Note that these components of the control plane hold and control the state of the cluster, but they don't run your applications.

This is done by the worker nodes. The worker nodes are the machines that run your containerized applications. It consists of three important components. The container runtime, which can be Docker or rkt; the kubelet, which talks to the API server and manages containers on its node; and the Kubernetes service proxy, which load balances network traffic between the application components

Steps to take for running an app in Kubernetes

1. Package the app into one or more containers
2. Push the images into an Image Registry
3. Post App Descriptor to the Kubernetes API Server
4. Scheduler schedules containers on available workers
5. Kubelet instructs nodes to download container images
6. Kubelet instructs nodes to run the containers

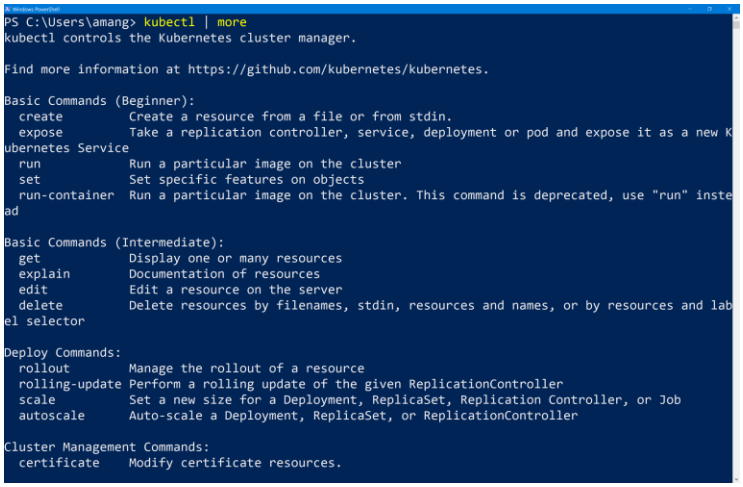
Kubernetes component - kubectl

```
kubectl <operation> <object> <resource name> <optional flags>
```

@iAmAlexMang www.alexmang.com

Whether you pronounce it as kube ctl or kube cutl, well, that's an eternal debate within the Kubernetes community. For the purpose of this course, I'm going to pronounce it as kube ctl.
Just like as you interact with Azure using the Azure CLI or using the Azure PowerShell module, kubectl is a command-line interface for running commands against Kubernetes clusters. To run kubectl commands from your terminal window, we use a syntax as shown on your screen. Kubectl followed by the command or the operation you want to perform, then the type of the Kubernetes object that you want to work with, which is then followed by the resource name and, optionally, any flags.

Kubernetes component - kubectl



For example, to get the list of nodes running in your Kubernetes cluster, you can run kubectl get nodes. To learn more about kubectl, you can simply refer to the help documentation. For example, to get more help on kubectl, you can type in kubectl, and then type in help, and then Enter.

Kubernetes component - pods

- Smallest unit that Kubernetes manages
- Is made up of one or more containers and information associated with those containers
- Querying a pod returns a data structure containing information about containers and its metadata

A pod is the smallest unit that Kubernetes manages and is the fundamental that the rest of the Kubernetes system is built on. It is made of one or more containers and the information associated with those containers.
When you ask Kubernetes about a pod, it'll return a data structure that includes a list of one or more containers along with the metadata that Kubernetes uses to coordinate the pod with other pods and policies of how Kubernetes should act and react if the program fails. The metadata can also define things such as affinity, which influences where a pod should be scheduled in a cluster, expectations around how to get the container images, and much more.

Kubernetes component - pods

- All the containers for a pod will run on the same node
- Any container running within a pod will share the node's network with any other containers in the same pod
- Containers within a pod can share files through volumes, attached to the containers
- A pod has an explicit life cycle, and will always remain on the node in which it was started

Now one thing to bear in mind is that the pod is not intended to be treated as a durable or a long-lived entity.
All the containers for a pod will be run on the same node. Any container running within a pod will share the node's network and any other containers in the same pod. Containers within a pod can share files through volumes attached to the containers. A pod has an explicit lifecycle and will always remain on the node in which it was started.
For all practical purposes, when you want to know what's running on a Kubernetes cluster, you are generally going to want to know about the pods running with your Kubernetes cluster and their state

Kubernetes component - namespaces

- Pods are collected into namespaces, which are used to group pods
- Can be used to provide quotas and limits on resource usage
- Impact DNS names that Kubernetes creates internally

@iAmAlexMang www.alexmang.com

- Pods are collected into namespaces, which are used to group pods together for a variety of purposes. For example, to see the status of all the pods in the cluster, you can run kubectl get pods with the --all-namespaces option.
- Namespaces can be used to provide quotas and limits around the resource usage, have an impact on the DNS names that Kubernetes creates internal to the cluster, and in future, even impact access control policies.
- If no namespace is specified when interacting with the Kubernetes through kubectl, the command assumes that you're working with the default namespace, which is typically called default.

Kubernetes component - networks

- All containers in a pod share the node's network
- All nodes in a Kubernetes cluster are expected to be connected to each other and share a private cluster-wide network
- Kubernetes runs containers within a pod within this isolated network

@iAmAlexMang www.alexmang.com

- All the containers in a pod share the node's network.
- In addition, all nodes in a Kubernetes cluster are expected to be connected to each other and share a private cluster-wide network. When Kubernetes runs containers in a pod, it does so within this isolated network.
- Kubernetes is also responsible for handling IP address, creating DNS entries, and making sure that a pod can communicate with the other pod in the same Kubernetes cluster.

Kubernetes component – Replica Set

- A Replica Set is associated with a Pod and indicates how many instances of that Pod should be running within a cluster
- A Replica Set also implies a controller that watches the ongoing state and knows how many of your Pods to keep running

- A ReplicaSet is associated with a pod and indicates how many instances of that pod should be running within that cluster. A ReplicaSet also implies that Kubernetes has a controller that watches the ongoing state and knows how many of your pod to keep running.
- This is where Kubernetes is really starting to do the work for you. If you specify three pods in a ReplicaSet and one failed, Kubernetes will automatically schedule and run another for you. A ReplicaSet is commonly wrapped in turn by a deployment

Kubernetes + Microsoft Azure

@iAmAlexMang www.alexmang.com

Kubernetes requires unnecessary underlying plumbing

➔Azure Kubernetes Services does that for you

The master node is fully managed by Microsoft!

Why AKS?

- Simplifies deployment, management and operations of Kubernetes
- Eliminates the burden of ongoing operations and maintenance by provisioning, upgrading and scaling resources on demand

Benefits of AKS

- Automated Kubernetes version upgrades and patching
- Easy cluster scaling (via Azure CLI or cmdlets)
- Self-healing hosted control plane (masters)
- Cost savings (only have to pay for running pool nodes)

And more…!

- Integrated identity and security management
- Integrated logging and monitoring
- Storage volume support
- Virtual network and ingress
  - DDOS protection, secure communication, ExpressRoute etc.
- SOC, ISO, and PCI DSS

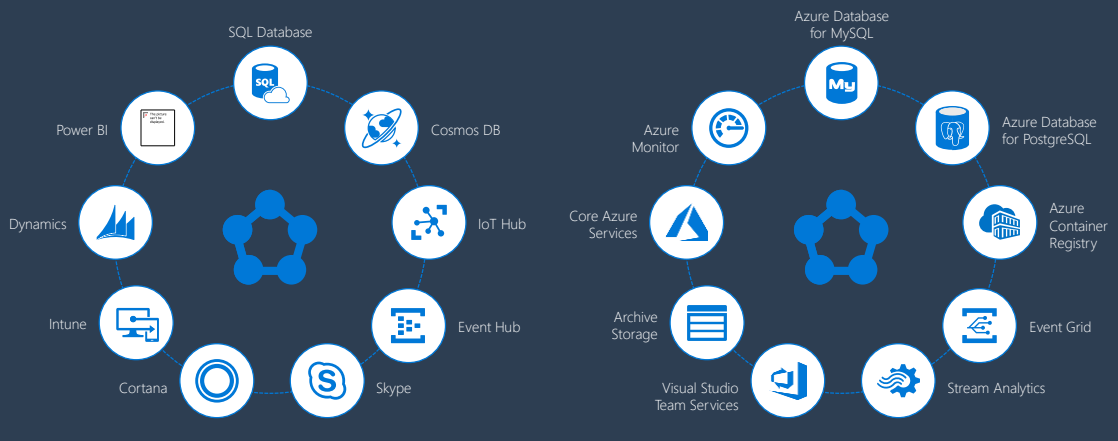
@iAmAlexMang www.alexmang.com

- AKS supports Kubernetes role-based access control (RBAC). RBAC lets you control how can access Kubernetes resources and namespaces, and what permissions that have on those resources. You can also configure an AKS cluster to integrate with Azure Active Directory (AD). With Azure AD integration, Kubernetes access can be configured based on existing identity and group membership.
- To understand how your AKS cluster and deployed applications are performing, Azure Monitor for container health collects memory and processor metrics from containers, nodes, and controllers. Container logs are available, and you can also review the Kubernetes master logs. This monitoring data is stored in an Azure Log Analytics workspace, and is available through the Azure portal, Azure CLI, or a REST endpoint.
- you can mount storage volumes for persistent data. Both static and dynamic volumes can be used. Depending on how many connected pods are to share the storage, you can use storage backed by either Azure Disks for single pod access, or Azure Files for multiple concurrent pod access.
- An AKS cluster can be deployed into an existing virtual network. In this configuration, every pod in the cluster is assigned an IP address in the virtual network, and can directly communicate with other pods in the cluster, and other nodes in the virtual network. Pods can connect also to other services in a peered

virtual network, and to on-premises networks over ExpressRoute or site-to-site (S2S) VPN connections.

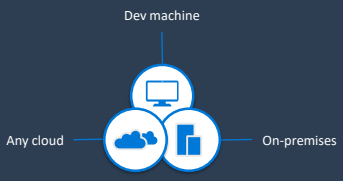
WHAT ABOUT SERVICE FABRIC?

Service Fabric Powers Azure and Microsoft services



@iAmAlexMang www.alexmang.com

Service Fabric Products



Service Fabric Standalone

Bring your own infrastructure

Azure Service Fabric

Dedicated Azure clusters

Azure Service Fabric Mesh

Fully managed microservices platform

Service Fabric Programming Models

- Reliable services
  - Stateless
  - Stateful
- Reliable actors
- Guest executables
- Containers

@iAmAlexMang www.alexmang.com

Reliable Service & Reliable Actor Services are built on top of .NET SDK frameworks provided by Azure Service Fabric. As such, they have programmatic access to service and service-fabric functionality. This includes accessing the Service Fabric Naming Service (resolve service instance URL by name) and integration with the instance lifecycle events, among others. Additionally, service instances are create as objects under the management of Service Fabric, instead of individual processes, which allows for higher density within the host cluster. These models will be discussed more in detail shortly

Guest Executable Services allow you to create a service that hosts an arbitrary executable. This executable can be written in any language you choose, and Service Fabric will take care of execution management tasks like ensuring the application is running and other orchestration tasks. Because guest executables are not built on top of the Service Fabric APIs, they have limited access internally to the Service Fabric functionality.

Container Services can be thought of as an application-within-an-application, in that multiple services can be placed within a container. This includes supporting Reliable Services which include Stateless Services within a container (Linux only), Stateful

Services within a container (Windows only), or arbitrary guest executables within a container.

Reliable Service Example

- Stateless
  - Any service which doesn't require state
  - For example, a calculator service
    - RunAsync() of the service can be empty
    - When created, will return an ICommunicationListener()
    - When call is made, an appropriate method is invoked and the service performs an action, without storing any data

@iAmAlexMang www.alexmang.com

A stateless service is one where there is literally no state maintained within the service, or the state that is present is entirely disposable and doesn't require synchronization, replication, persistence, or high availability.
For example, consider a calculator that has no memory and receives all terms and operations to perform at once.
In this case, the RunAsync() of the service can be empty, since there is no background task-processing that the service needs to do. When the calculator service is created, it will return an ICommunicationListener (for example Web API) that opens up a listening endpoint on some port. This listening endpoint will hook up to the different methods (example: "Add(n1, n2)") that define the calculator's public API.
When a call is made from a client, the appropriate method is invoked, and the calculator service performs the operations on the data provided and returns the result. It doesn't store any state.
Not storing any internal state makes this example calculator very simple. But most services aren't truly stateless. Instead, they externalize their state to some other store. (For example, any web app that relies on keeping session state in a backing store or cache is not completely stateless.)

Reliable Service Example

- Stateful
  - Must have some portion of state kept consistent and present
  - For example, calculating rolling average
    - Requires both the current set of incoming requests it needs to process,
    - as well as the current average
  - Any service that retrieves, processes, and stores information in an external store

Actor

- An implementation of a the Virtual Actor pattern
- Built on top of Stateful Reliable Services
  - Any actor can contain
    - Code: reliable service
    - State: reliable state
    - Mailbox: service remoting
- Designed for taking care of workflows, business logic orchestration, isolated data persistency based on the boundaries of the context

@iAmAlexMang www.alexmang.com

An Actor is an isolated Entity and is defined as a fundamental unit of computation, which embodies three essential parts, processing, storage, and communications. It sounds a bit confusing, abstract, and not very clear how it solves multithreading issues, so let's go over this. **Objective:** To introduce the Reliable Actors framework available in Azure Service Fabric development

Notes: The Reliable Actors framework is a Virtual Actor implementation that is built on top of stateful Reliable Services.
In a Virtual Actor implementation, an Actor is a unit of both logic and state that is managed by the framework. Client applications reference an actor by a unique ID. The framework manages the allocation and lifetime of individual actor instances, including “resurrecting” an actor instance which has gone dormant due to a period of inactivity.

An Actor is an abstract entity, which contains code to execute, this is the same code you would normally write using any methodology, language, or libraries you're used to. It has a persistent state, the state can only be accessed by the Actor and no one else. It's not accessible even by the Actor of the same type, but only by different instances of the Actor. An Actor has a mailbox, which means it's discoverable by other

Actors or different parts of the application like services. The mailbox stores messages for the Actor, which it will process at some point. And the last thing an Actor can do is send messages to other Actors, including itself.

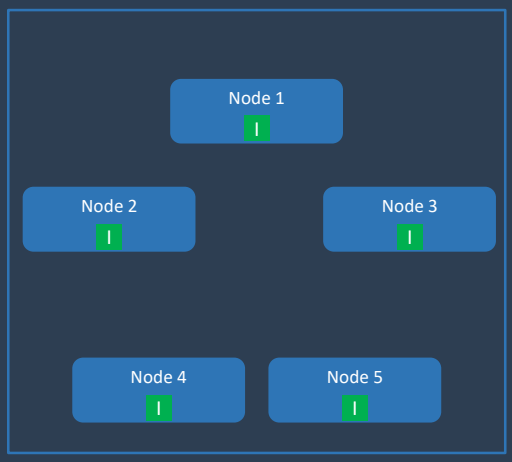
Actor (cont'd)

- Actors are instantiated upon reference and collected upon inactivity
  - State management can be used during reference to "restore" an actor
- Actors are turn-based (single-threaded) per instance
  - Supports Timers and Reminders
  - Also supports Events

Actor instances provide a turn-based, single-threaded access model, making it possible to not have to deal with concurrency issues in a distributed programming model. With this single-threaded behavior, Timers and Reminders both offer support for running code after a function call has completed. Timers and reminders differ mainly in that Timers cannot "re-awaken" a time-expired actor, whereas Reminders will continue to execute, "waking" a dormant actor instance as necessary, until the actor unregisters the reminder or the actor is explicitly deleted.

Finally, actors also support raising events back to the calling application. Because of the nature of distributed programming, these events are not strictly guaranteed to reach their listeners (if reliable messaging is required, a queue-based or similar solution is recommended.)

Partitioning – Stateless Services

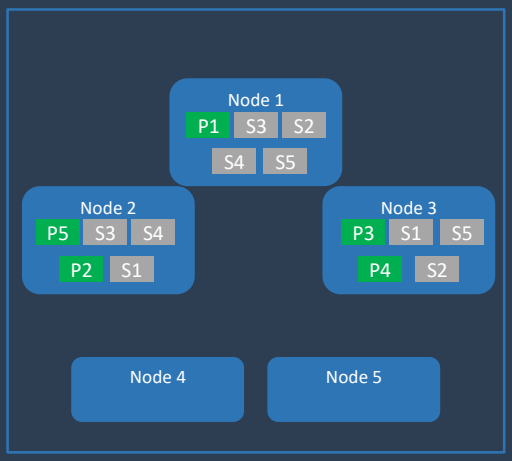


- In a stateless environment, scalability and availability generally achieved by adding instances
- Place an instance of the service in each Node
- Add Nodes to scale out

Here we have a Stateless Service running in 3 Partitions distributed across 3 Nodes. (Animation) If we add two more Nodes to the service, Service Fabric simply also create two additional service instances.

Partitioning – Stateful Services

- For stateful services
- Services are Distributed for scalability
- Service state is Replicated for availability
- Distribute primary replicas across the nodes in the cluster and also place secondary replicas
- Add Nodes, then rebalance to scale out



@iAmAlexMang www.alexmang.com

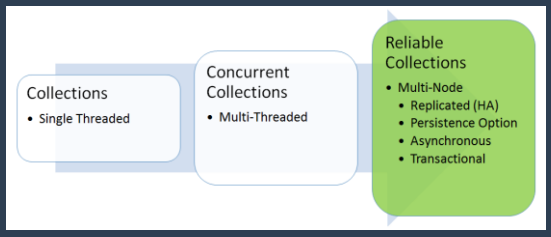
Providing scalability and reliability is a little trickier in a Stateful Service. In order to do so, we need to maintain distributed replicas of the state in each partition, keeping track of a primary Partition and a number of Secondary Partitions.

- Reliability exists because if Service Fabric is aware of a single node going down, it can rebalance using the Secondary partition replicas located on other nodes, promoting them to Primary nodes as necessary and creating new additional Secondary replicas.
- Scalability is present as long as the number of Nodes approaches the number of Partitions (when scaling up.) Once you have more Nodes than Partitions, the additional Nodes do not generally contribute to expanded capacity in the system.

Here we have a Stateful Service running in 5 Partitions with 3 Replicas per partition, distributed across 3 Nodes. (Animation) If we add two more Nodes to the service, Service Fabric rebalances the existing partitions, easing the load on the original 3 Nodes.

Reliable Collections

- Natural evolution of System.Collection classes
  - Replicated
  - Persisted
  - Asynchronous
  - Transactional
- Reliable:
  - Dictionary
  - Queue
  - Concurrent Queue



@iAmAlexMang www.alexmang.com

Reliable Collections can be thought of as the natural evolution of the **System.Collections** classes: a new set of collections that are designed for the cloud and multi-computer applications without increasing complexity for the developer. As such, Reliable Collections are:
Replicated: State changes are replicated for high availability.
Persisted: Data is persisted to disk for durability against large-scale outages (for example, a datacenter power outage).
Asynchronous: APIs are asynchronous to ensure that threads are not blocked when incurring IO.
Transactional: APIs utilize the abstraction of transactions so you can manage multiple Reliable Collections within a service easily.

Today, **Microsoft.ServiceFabric.Data.Collections** contains three collections:
Reliable Dictionary: Represents a replicated, transactional, and asynchronous collection of key/value pairs. Similar to **ConcurrentDictionary**, both the key and the value can be of any type.
Reliable Queue: Represents a replicated, transactional, and asynchronous strict first-in, first-out (FIFO) queue. Similar to **ConcurrentQueue**, the value can be of any type.
Reliable Concurrent Queue: Represents a replicated, transactional, and asynchronous

best effort ordering queue for high throughput. Similar to the **ConcurrentQueue**, the value can be of any type.

https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-concepts-replica-lifecycle

From monolith to microservice

5 stages in a continuum…

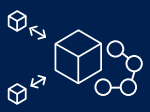
1

Traditional app

2

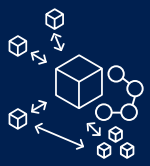
Monolith Hosted as guest executable or container

3



Existing Monolith + new microservices

4



Parts of existing monolith extracted

5

New or transformed microservices app

… support any stage you choose

APPLES-TO-APPLES COMPARISON (?!)

Kubernetes' matrix

The good parts	The bad parts
Everything is customizable	Very steep learning curve
Stable	Complexity

Service Fabric's matrix

The good parts	The bad parts
Good support for .NET framework	Still very complex
Supports lift-and-shift strategy for legacy .NET stack	Contrainers are second-class citizens

Service Fabric's history

Software system that weaves together machines across datacenters into logical, scale-free, geo-distributed, hierarchical and consistent distributed system

https://www.youtube.com/watch?v=MrfcP6dS6mU
Gopal Kakivaya, Service Fabric team founder

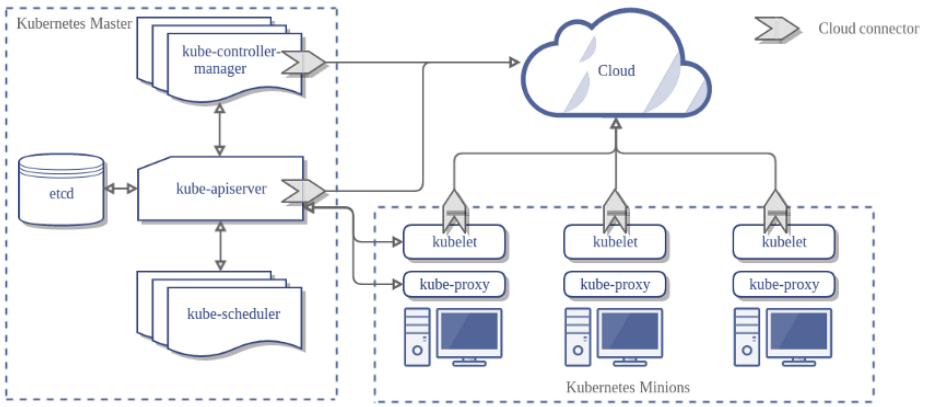
Service Fabric's history

- Technology was first disclosed at Microsoft PDC 2008 ("Distributed Fabric")
  - Basis for SQL Data Services, launched in 2010
  - https://channel9.msdn.com/Blogs/pdc2008/BB03
- Lack of SSDs meant optimizations were required on magnetic disks
  - Ktlogger
- First boxed product was Lync Server 2013

Kubernetes' history

- Kubernetes is a fairly young technology and its first stable release dates to July 21, 2015.
- Conceptually based on and strongly influenced by Borg
  - the orchestrator responsible for managing the entire fleet of machines in Google DCs internally
- Differences Kubernetes and Borg:
  - Borg is focused on power-users
  - Borg orchestrates plain old processes running huge, statically linked executables
  - Kubernetes orchestrates containers
  - Kubernetes API server is designed to be much more general and microservices based

Main architectural difference (Kubernetes)



@iAmAlexMang www.alexmang.com

Paradoxically, Kubernetes is a heavily centralized system. It has an API Server in the middle, and agents called Kubelets installed on all worker nodes. All Kubelets communicate with the API Server, which keeps the state of the cluster persisted in a centralized repository: the etcd cluster. The etcd cluster is a fairly simple Raft-backed distributed KV-store. Most production configurations use between 3 and 7 nodes for etcd clusters.

To maintain the cluster membership information, all the Kubelets are required to maintain a connection with the API Server and send heartbeats every period of time (for example 10 seconds).

Such an approach can have negative effects on the scalability.

https://github.com/kubernetes/community/blob/master/keps/sig-node/0009-node-heartbeat.md

Main architectural difference (Service Fabric)

"*The design of fault-tolerant systems will be simple if faulty processes can be reliably detected. […] Distributed consensus, which is at the heart of numerous coordination problems, has trivially simple solution if there is a reliable failure detection service.*"

Sukumar Ghosh, Distributed Systems - An Algorithmic Approach, Second Edition

The authors of Service Fabric, instead of using a distributed consensus protocol like raft or paxos (which does not scale well with growing number of nodes), or using a centralized store for cluster state (which might use distributed consensus underneath, and which represents a separate point of failure), have focused on designing a fully distributed system. Within Service Fabric's architecture, this is referred to as the Federation Subsystem.
The Federation Subsystem is responsible for giving a consistent answer to the core question: is the specific node part of the system? In the centralized approach discussed earlier, this answer was given by the API Server based on the heartbeats received from all the nodes. In Service Fabric, the nodes are organized in a ring and heartbeats are only sent to a small subset of nodes called the neighborhood.

Core infrastructure services

- In Kubernetes, everything goes into the master
- Service Fabric uses a different approach, where most system services are standard Service Fabric Reliable Services and are distributed among the entire cluster respecting the placement constraints

- Due to its centralized nature, most of the core infrastructure services in Kubernetes are placed on the master nodes.
- Service Fabric uses a different approach, where most system services are standard Service Fabric Reliable Services and are distributed among the entire cluster respecting the placement constraints. For example Azure-hosted Service Fabric clusters run system services on the primary node type only.

Stateful services

- Service Fabric was designed to run both stateless and stateful workloads
  - It tries to detect dead nodes ASAP.
  - Its design allows sending heartbeats much more often in order to detect and correct failure.
  - 5 minutes of unavailability is unacceptable
  - The default lease duration in Service Fabric is 30 seconds,
    - though different types of failures can be detected sooner, and most network partitions are detected within 10 to 15 seconds.
- Eventually consistent databases like Riak are much better candidates to run on K8S clusters than the strong consistent databases

@iAmAlexMang www.alexmang.com

Service Fabric has no competition when it comes to orchestrating mission-critical stateful workloads. It is a strongly consistent platform that is able to provide guarantees like RTO < 30 seconds in case of single rack/availability zone failure, or RTO < 60 seconds in case of entire region outage, with RPO = 0 guarantee, while maintaining high performance and extremely low latency from extensive use of RAM and ephemeral SSDs.
Because Service Fabric was designed to run both stateless and stateful workloads, it tries to detect dead nodes ASAP. Its design allows sending heartbeats much more often in order to detect and correct failure. 5 minutes of unavailability is unacceptable for a stateful service in many scenarios. The default lease duration in Service Fabric is 30 seconds, though different types of failures can be detected sooner, and most network partitions are detected within 10 to 15 seconds.
Eventually consistent databases like Riak (especially with facilities like good CRDT-support) are much better candidates to run on K8S clusters than the strong consistent databases.

Hyperconvergence and IoT Edge

- Service Fabric = key element of the hyperconvergence puzzle
  - Effective failure detection
  - replication mechanisms
  - strong decentralization
- https://youtu.be/t3Vo37V9oU8?t=3103, Mark Russinovich's "Die Hard"

Effective failure detection and replication mechanisms combined together with strong decentralization enable efficient use of ephemeral storage, networking resources, and computing resources, making Service Fabric a key element of the hyperconvergence puzzle. Service Fabric can efficiently run on the commodity hardware or even can form clusters using the small IoT Edge devices. This capability was recently demonstrated in Mark Russinovich's "Die Hard" demo.

Extreme scalability

- Many stories about etcd tuning exposing their scalability issues ← **this is related to Kubernetes**
- Subramanian Ramaswamy demoed orchestration of 1 million containers on Service Fabric
  - more than 8x the OpenShift limit (120 000).
  - https://www.youtube.com/watch?v=OjhOZkql4uE

@iAmAlexMang www.alexmang.com

- If you browse the web, there are many stories about etcd tuning exposing their scalability issues.
- Meanwhile, Subramanian Ramaswamy ran the following demo during the Ignite conference – orchestrating 1 million containers, which is more than 8x the OpenShift limit (120 000).

Failover so fast it can control the network itself

- Service Fabric can be tuned to be used as a Network Controller for Software Defined Networking, which presents fast failover capabilities.

https://docs.microsoft.com/en-us/windows-server/networking/sdn/technologies/network-controller/network-controller-high-availability

On the other hand, Service Fabric can be tuned to be used as a Network Controller for Software Defined Networking, which presents fast failover capabilities.

Geographical distribution

- Cortana is a good example
- SF's hierarchical fault domains (e.g. *fd:/westeurope/1*)
  - Enables the failure detection to differentiate between intra- and inter-region heartbeats
  - Limitations on SF:
    - Requires at least 3 regions
    - SF is strongly consistent
- Etcd scalability issues and centralized design make strongly consistent cross-regional configurations unfeasible.

@iAmAlexMang www.alexmang.com

Cortana already uses Service Fabric in such a manner. The main facility enabling this capability are the hierarchical fault domains, for example: *fd:/westeurope/1*. This distinction enables the failure detection to differentiate between intra- and inter-region heartbeats, being a bit more forgiving to cross regional networking errors.

The most important limitations of Service Fabric's geographically distributed approach are:

- There must be at least 3 regions, because no region is privileged and there must be a quorum to be able to determine which region is down
- Service Fabric is strongly consistent – which means that if an eventually consistent geographically distributed solution (one that must survive longer connectivity problems across the regions) is required – multiple separate clusters should be created – which gives Kubernetes the upper hand for eventually consistent systems

Interesting points

Both solutions are lacking
a cluster-level SLA on Azure!

Other Azure services for container needs

Find the Azure service for your container needs

YOU WANT TO:	USE THIS:
Scale and orchestrate Linux containers using Kubernetes	Azure Kubernetes Service (AKS)
Deploy web apps or APIs using Linux containers in a PaaS environment	Azure App Service
Elastically burst from your Azure Kubernetes Service (AKS) cluster	Azure Container Instances
Run repetitive compute jobs using containers	Azure Batch
Lift, shift, and modernize .NET applications to microservices using Windows Server containers	Azure Service Fabric
Store and manage container images across all types of Azure deployments	Azure Container Registry

https://azure.microsoft.com/en-us/overview/containers/

Azure Container Instance

Develop apps fast without managing virtual machines or having to learn new tools—it's just your application, in a container, running in the cloud

- No VMs, No hypervisors
- Designed for targeted use-cases and ephemeral runtime
- Elastic bursting with AKS with virtual Kubeletes

Azure App Service for Containers

- Easily deploy and run containerized applications that scale with your business
- Use a fully-managed platform to perform infrastructure maintenance
- Take advantage of built-in auto scaling and load balancing

Azure Batch

- Extremely intriguing as a container hosting option
  - Especially due to the NVIDIA GPU-enabled VM families and low-priority VM families
- Perfect fit if the container based application has schedule-based runs

https://docs.microsoft.com/en-us/azure/batch/batch-docker-container-workloads

Azure Service Fabric Mesh

Azure Service Fabric Mesh is a fully managed service that enables developers to deploy microservices applications without managing virtual machines, storage, or networking

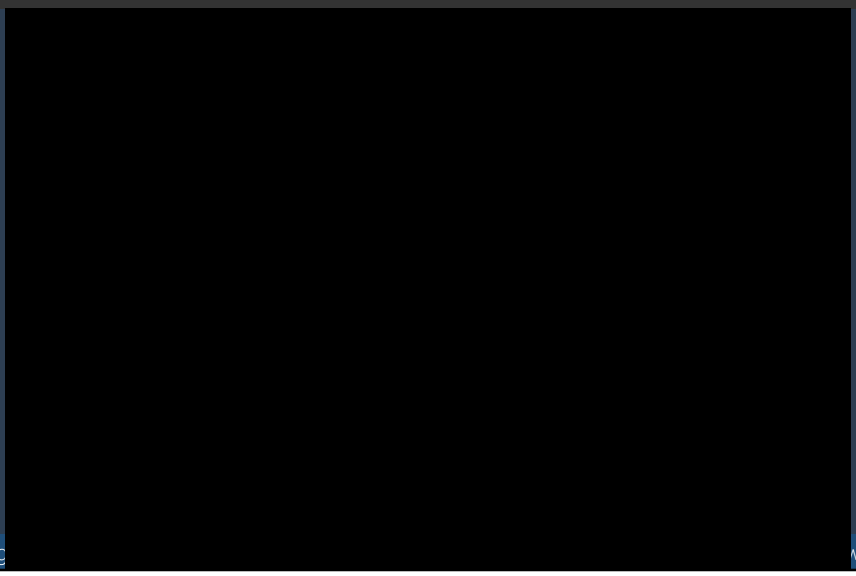
- HA, DR, horizontal scaling, discoverability, orchestration, message routing, reliable messaging, no-downtime upgrades, secrets management, state management, config management and distributed transactions

Azure Service Fabric Mesh

- Modernize everything: The way to cloud-native applications - THR3069

https://www.youtube.com/watch?v=A1yZNbcrGg8

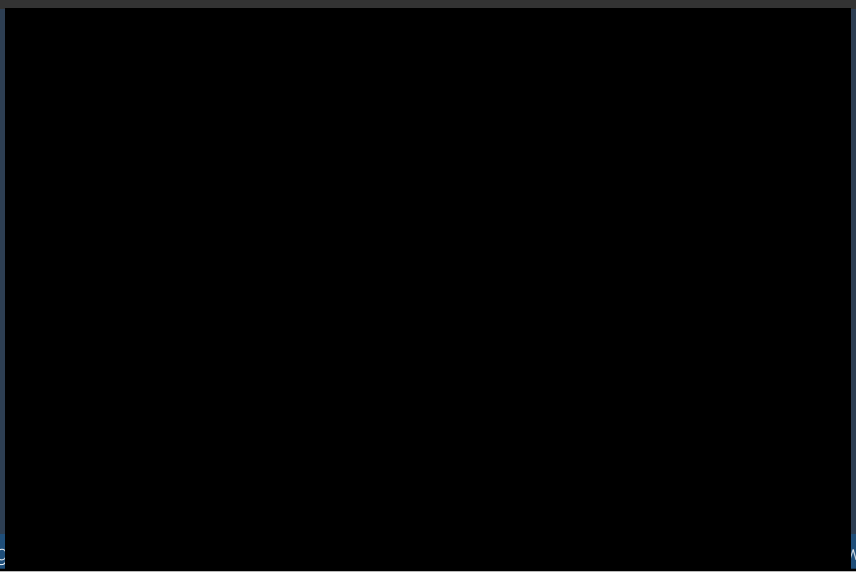
Developing Cloud-Native Apps with SFM



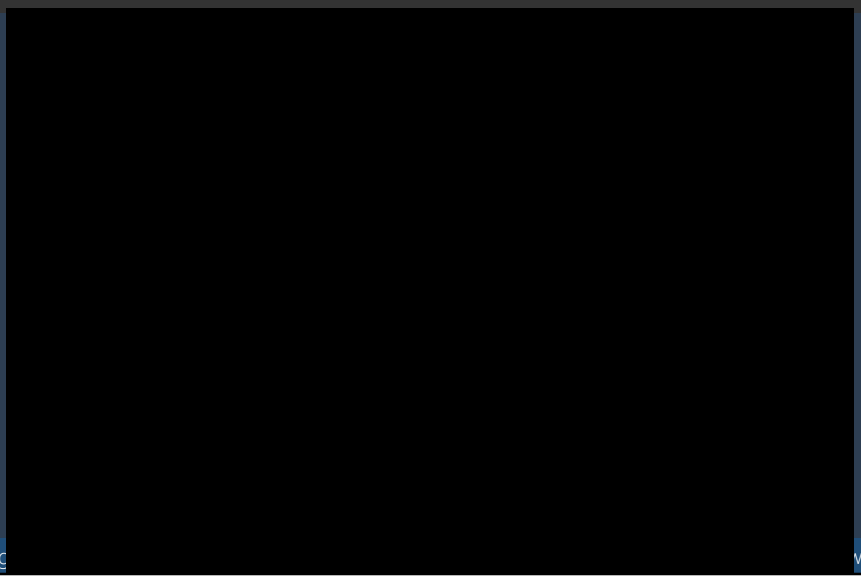
@iAmAlexMang

w.alexmang.com

Persisting data with SFM



n-tier with Mesh in Visual Studio with SFM



Final thoughts

- If you
  - only have a handful of services…
  - don't need A/B testing, canary releases, ring-based deployments…
  - don't care about money…
- …don't use containers

Beer discussion facts

- Service Fabric:
  - First class citizen for Azure
- AKS:
  - Much larger community
- Service Fabric:
  - Great Visual Studio tooling
- AKS:
  - Ops-specific experience

Picking the 'better' option

- Kubernetes clearly wins Mrs. Popularity prize
- Limitations pick the "winner"
  - Real question remains: Linux or Windows?
- More alternatives exist!
  - Outside the Azure specific sphere: Nomad by HashiCorp
  - DO YOU EVEN WANT CONTAINER ORCHESTRATORS?

Favorite?


@iAmAlexMang
That's all Folks!