Kubernetes - Beyond a Black Box

A humble peek into one level below your running application in production

Part II

About The Author

- **Hao (Harry) Zhang**
- **Currently:** Software Engineer, LinkedIn, Team Helix @ Data Infrastructure
- **Previously:** Member of Technical Staff, Applatix, Worked with Kubernetes, Docker and AWS
- **Previous blog series:** “Making Kubernetes Production Ready”
  - Part 1, Part 2, Part 3
  - Or just Google “Kubernetes production”, “Kubernetes in production”, or similar
- **Connect with me on LinkedIn**

Previously in Part I

- A high level idea about what is Kubernetes
- Components, functionalities, and design choice
  - API Server
  - Controller Manager
  - Scheduler
  - Kubelet
  - Kube-proxy
- Put it together, what happens when you do `kubectl create`
- [SlideShare link for Part I]

Outline - Part II

- An analysis of controlling framework design philosophy
  - **Choreography, not Orchestration**
  - **Level Driven, not Edge Triggered**
  - **Generalized Workload and Centralized Controller**
- Scheduling - Limitation and next steps
- Interfaces for production environments
- High level workload abstractions
  - Strength and limitations
- Conclusion

Part II

Controlling Framework

Choreography, Level-driven, Centralized Controller

Micro-service Choreography

- Orchestration: one gigantic controller trying to make everything correct at the same time
- Choreography: Desired state in cluster is achieved by **collaborations of separate autonomous entities reacting on changes of one or more API object(s) they are interested in**
  - Separation of concern
  - More flexibility to extend different types of semantics (CRD / TPR are great examples)
  - Develop a controller is easy!

More Readings:

✤ Third Party Resource (TPR): https://kubernetes.io/docs/tasks/access-kubernetes-api/extend-api-third-party-resource/
✤ Customer Resource Definition (CRD): https://kubernetes.io/docs/tasks/access-kubernetes-api/extend-api-custom-resource-definitions/
✤ TPR Design Spec: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/api-machinery/extending-api.md

Micro-service Choreography

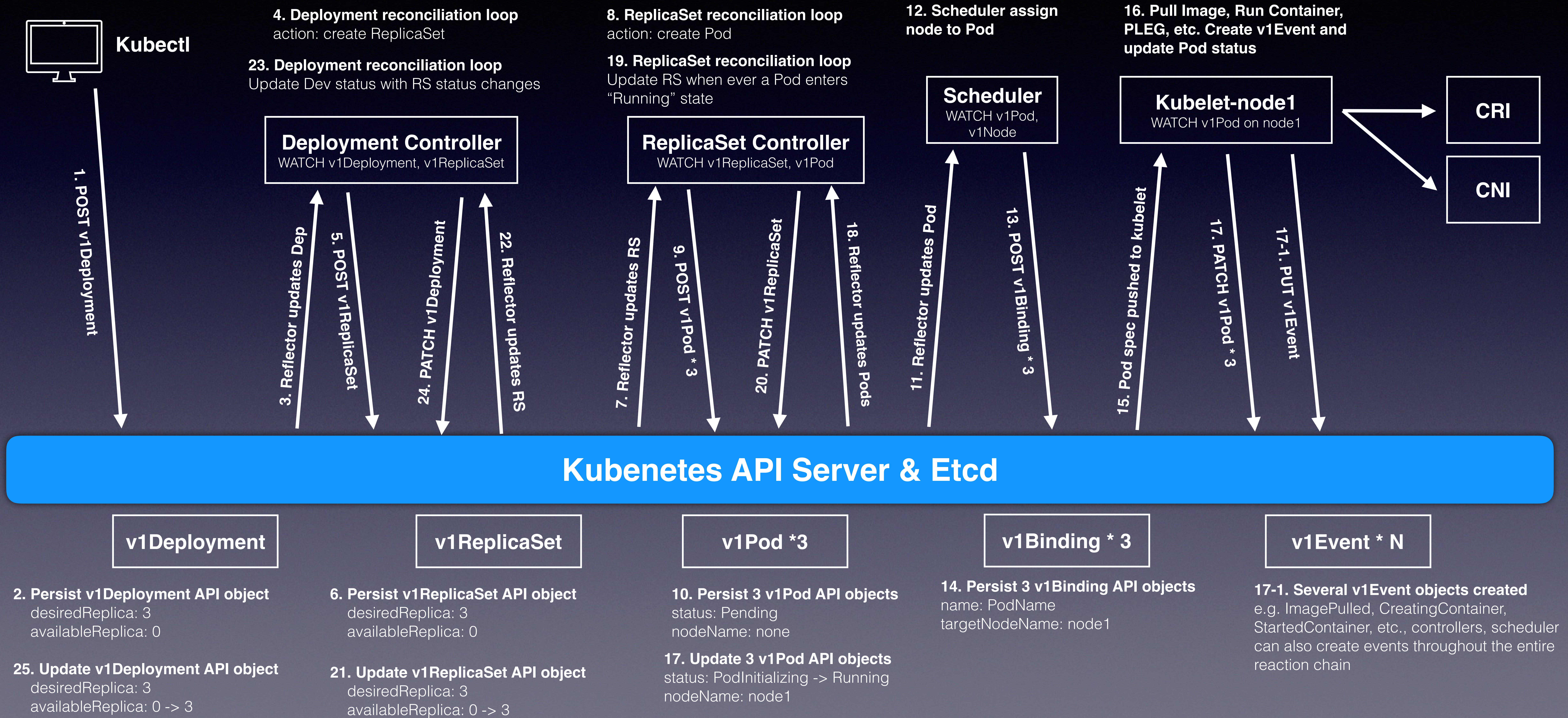
- Separation of Concern via **Micro-service Choreography** (Deployment example)
  - **ReplicaSet (RS)** ensures given # of Pod is always running
    - Create / Delete Pod API object if existing number is not matching spec
    - Scheduler knows where to put it; Kubelet knows Pod life cycle details
  - **Deployment (DEP)** provides higher level of semantics
    - ***Scale:*** tell RS to change # of Pods by modifying RS spec, RS knows what to do
    - ***Upgrade:*** delete current RS, create new RS with new Pod spec
    - ***Rolling-Upgrade:*** create new RS with new Pod spec, scale down old RS by 1, scale out new RS by 1, repeat it until no old Pod remains (Same for roll back)
  - **HorizontalPodAutoscaler (HPA)** manipulates deployment
    - It polls Pod metrics (from a source such as Heapster or Prometheus), compare it with user defined metric specs, and make scaling decision
    - Tell DEP to scale up/down by modifying DEP spec, DEP controller knows what to do

Micro-service Choreography

- See Next Slide: An illustration about how different autonomous entities reacting on API objects they are interested in can move cluster to a desired state
  - Using Deployment as example, duplicated from Part I

Micro-service Choreography

Note: this is a rather high-level idea about Kubernetes reaction chain. It hides some details for illustration purpose and is different than actual behavior



Micro-service Choreography

- Choreography - **some down sides**
  - Long reaction chain
    - Multiple layers of reconciliation stages can be error prone (i.e. racing conditions, component down, etc), it has been improved a lot in late Kubernetes versions
    - Latency could be long as pipeline needs to persist after every stage, might not fit high QoS requirements
  - Debugging is hard
    - i.e. I forget to update image in registry
    - GET Deployment — found desired replica is not up
    - LIST Pods with label — found image pull backoff in container status
    - LIST events with involved object as this Pod — found message “Image does not exist”
    - Operations can be heavy if you want to automate failure recovery / detailed error reporting, unless you cache a lot (Efficient caching is another huge topic…)

Level-Driven Control

- State is considered as "***level***" while state change resembles "***edge***"
- Desired state and current state are persisted
- Controller can always re-check object state and make action to move object towards desired state
- Atomic ListAndWatch (see Part I) assures controllers react based on state changes pushed from API server (watch) while not miss any event (list)

More Readings:
✤ Kubernetes: edge vs level triggered logics: https://speakerdeck.com/thockin/edge-vs-level-triggered-logic

Centralized Controller

- **Generalized Workload and Centralized Controller**
  - Kubernetes generalize applications into limited types of semantics
    - Job, Deployment, Node, StatefulSet, DaemonSet, …
  - Reduced control overhead, i.e., 1 type of controller manages all instances of workloads in one category
    - i.e. Deployment controller will control all deployments in the cluster
    - Compared with 1 controller controls 1 deployment instance (*O(n) space complexity*), Kubernetes' control overhead is constant
  - Reconciliation control loops: things will ultimately become correct
  - **Some down sides:** Stateful applications are hard to generalize (More discussions later)

Centralized Controller

- Two opposite design choices from state-of-art cluster management frameworks
  - Apache Hadoop Yarn, Apache Mesos
    - User have full freedom to implement their workload controlling logics
    - One app controller per app, all controllers talk to master
  - Kubernetes
    - Pre-defined very generic workload abstractions
    - Workloads of same type share 1 controller
    - Centralized management of controllers with optimized master access

Scheduling

Limitations and Next Steps

Scheduling

- Default sequential scheduler - **Limitation**
  - Scoring system can be hard to tune
  - Sequential scheduling might not be ideal for batch workloads
  - Assumes launching more is ALWAYS better than launching less
    - I need 3 replicas to form a quorum, then what's the point of starting 2 upon insufficient resource?
  - No global re-balancing mechanism
    - Upon new node join
      - Moving things around can reduce node resource pressure as we over-provision
      - But this rebalance would make scale-down even harder
    - Upon insufficient resource (moving things around might fit)
    - Hacks available such as re-scheduling and Kubelet preemption

Scheduling

- Some advanced scheduling problems people are trying to solve:
  - If *m* Pods cannot fit onto *n* Nodes, how to choose which ones to run
  - If not all Pods in an application can get scheduled
    - Can we just schedule some of them?
  - What if an application needs to create other resources?
    - i.e. workflow has a big parallel step which cannot be serialized, if this step cannot fit in, it'd be better to fail the entire workflow
  - Cluster-wide resource isolation mechanism other then ResourceQuota per namespace
  - Bin-pack apps and scale back to release some resources

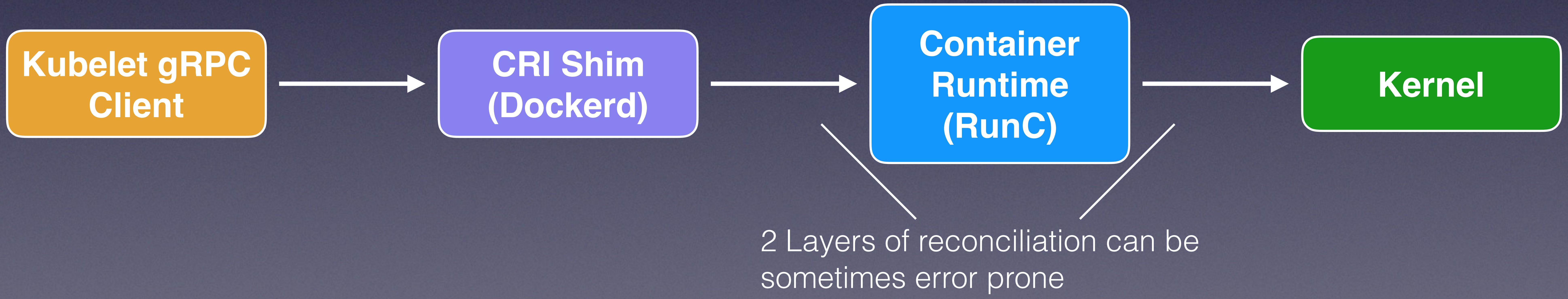
More Readings:
✤ Resource Sharing / scheduling design spec: https://docs.google.com/document/d/1-H2hnZap7gQivcSU-9j4ZrJ8wE_WwcfOkTeAGjzUyLA
✤ Scheduling Feature Proposals: https://github.com/kubernetes/community/tree/master/contributors/design-proposals/scheduling

Environment Interfaces

CNI, CRI and Flex Volume

Production Interfaces

- Container Runtime Interfaces (CRI)
  - Implemented based on Open Container Initiative (OCI)
  - Runtime Service for container lifecycle management
  - Image Service for image management
  - Runtime plugin can be customized
    - You can even write your plug-in for VM



More Readings:
✤ Docker reconciliation bug: https://github.com/moby/moby/issues/32413

Production Interfaces

- Container Network Interface (CNI)
  - Pluggable interface for cluster networking layer
  - Used to setup/teardown Pod network
- Flex Volume (user-defined volumes)
  - Interface for data volume plugin
  - Need to implement methods such as attach/detach, mount/unmount
- With these CRI, CNI, and Flex Volume, you can make kubelet a "dumb", environment agnostic worker, which is extremely flexible to fit any production environment

Production Interfaces

More Readings:

- CRI
  - Official blog introducing CRI: http://blog.kubernetes.io/2016/12/container-runtime-interface-cri-in-kubernetes.html
  - CRI Spec: https://github.com/kubernetes/community/blob/master/contributors/devel/container-runtime-interface.md
  - CRI Container Stats Proposal: https://github.com/kubernetes/community/blob/master/contributors/devel/cri-container-stats.md
  - Open Container Initiative (OCI): https://www.opencontainers.org
  - Open Container Initiative (OCI) Runtime/Image spec, and RunC: https://github.com/opencontainers

- CNI
  - Pod Networking Design Proposal: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/network/networking.md
  - Kubernetes Networking Plugin Introduction: https://kubernetes.io/docs/concepts/cluster-administration/network-plugins/#cni
  - Linux CNI Spec: https://github.com/containernetworking/cni/blob/master/SPEC.md#network-configuration
  - CNCF CNI Project: https://github.com/containernetworking/cni
  - Kubelet CNI Usage: https://github.com/kubernetes/kubernetes/blob/master/pkg/kubelet/network/cni/cni.go
  - Kubenet - Kubernetes' default network plugin: https://github.com/kubernetes/kubernetes/tree/master/pkg/kubelet/network/kubenet

- Volume
  - Flex Volume: https://github.com/kubernetes/community/blob/master/contributors/devel/flexvolume.md
  - Kubernetes Volume Documentation: https://kubernetes.io/docs/concepts/storage/volumes/

Workload Abstractions

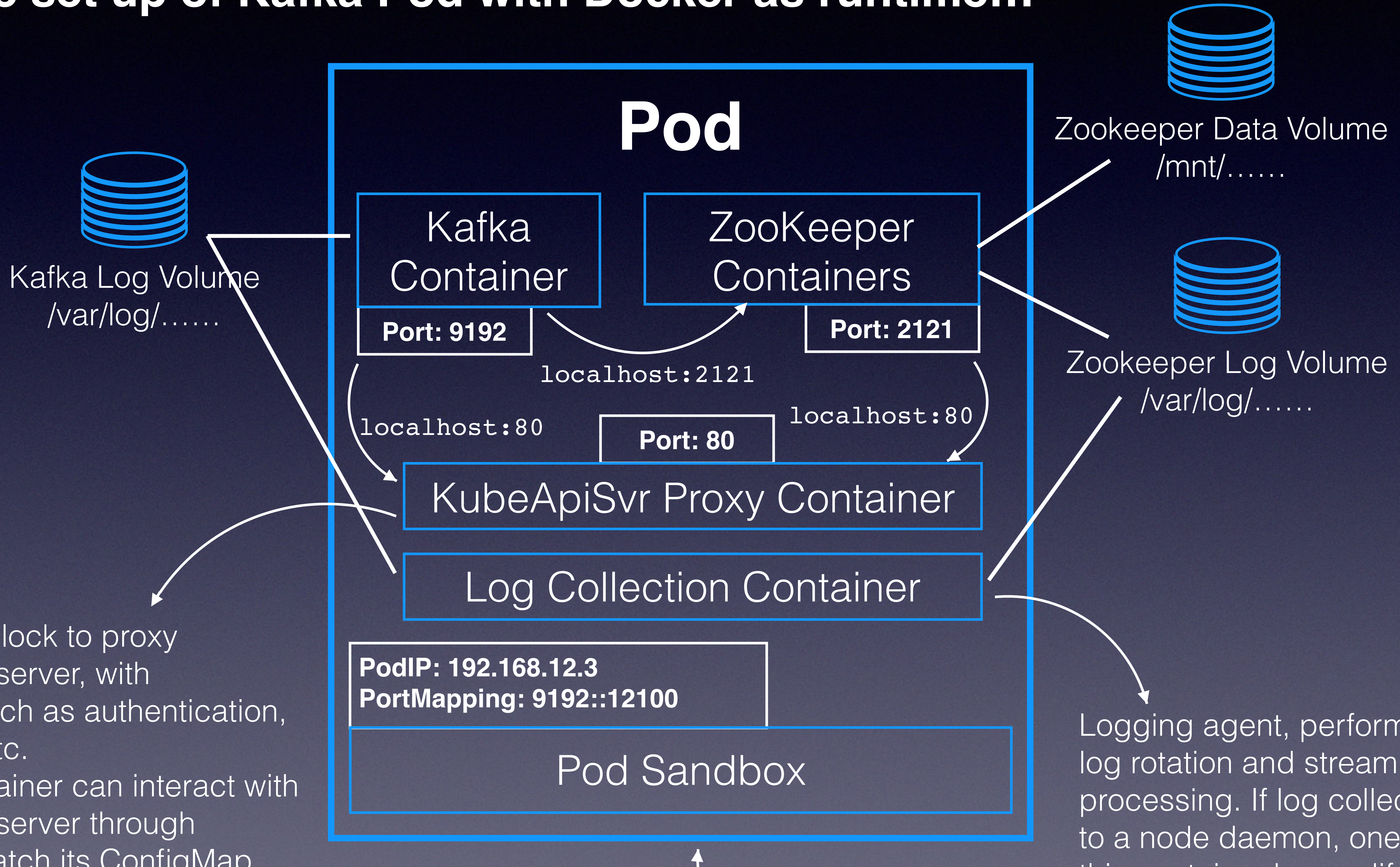
Pod, Job, Deployment, DaemonSet, StatefulSet - Strength and limitations

Workload Abstractions

- Pod - Basic Workload Unit in Kubernetes
  - An isolated environment with resource constraints
    - With docker as run time, it is a group of containers under Infra container Cgroup
    - User can implement their own runtime through CRI
  - Pod = atomic scheduling unit, ensures co-location
  - Container = single-function building block running in isolated env
    - Just modify Pod spec to plug/unplug functionalities, no code change
    - Spin up in milliseconds (it’s just a process)
  - Containers in Pod can share directory, volume, localhost, etc.
  - Crashed container will be restarted on same node

Workload Abstractions

A possible set up of Kafka Pod with Docker as runtime…



- Small building block to proxy Kubernetes API server, with functionalities such as authentication, API versioning etc.
- Application container can interact with Kubernetes API server through localhost, i.e., watch its ConfigMap change and react to config changes in runtime

Logging agent, perform works such as log rotation and stream logs for backend processing. If log collection is switched to a node daemon, one can easily unplug this container by modifying Pod spec

`Talk to Kafka: curl 192.168.12.3:12100`

Workload Abstractions

- Job
  - Run to complete (i.e. container exit code is 0)
  - Cron job, batch job, etc…
- Deployment
  - Maintain replicas of **stateless** applications (not managing volume)
  - High-level interfaces such as rollout, rollback
- DaemonSet
  - One replica per node, primary use cases include:
    - Log collection daemon (fluentd)
    - Node monitoring daemon (node-exporter)
    - Cluster storage daemon (gclusterd)

Workload Abstractions

- StatefulSet
  - Use case prototypes:
    - Quorum with leader election: MongoDB, Zookeeper, Etcd
    - De-centralized quorum: Cassandra
    - Active-active (multiple masters): Galera
  - Besides features of deployment, StatefulSet also provides:
    - Every replica has persistent identifier for network (Pod name is formatted as "podName-{0..N-1}"), which might help identifying peers
    - Every replica has persistent storage (data persist even after deletion)
    - Supports automatic storage provisioning
    - Ordered deploy, shutdown and upgrade (from {0..N-1})

Workload Abstractions

More Readings:

- Typically used workloads
  - Pod: https://kubernetes.io/docs/concepts/workloads/pods/pod/
  - Deployment: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
  - Job: https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/
  - CronJob: https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/
  - Deamonset: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/
- Other Kubernetes Concepts (Network/Config/Volume, etc.): https://kubernetes.io/docs/concepts/
- Design Specs
  - Workloads Design Specs (Interfaces, behaviors and updates): https://github.com/kubernetes/community/tree/master/contributors/design-proposals/apps
  - Autoscaling Design Specs: https://github.com/kubernetes/community/tree/master/contributors/design-proposals/autoscaling
- Very good example of running master-slave MySQL example on Kubernetes using StatefulSet: https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/

Abstraction Limitations

- Some thoughts about Kubernetes workload abstractions
  - **Pod** and **Job** are perfect for small execution unit
  - **Deployment** can fit most use cases of stateless application
  - **DaemonSet** can fit most use cases of one-per-node applications
  - **StatefulSet** might be useful for small-medium scale peer-to-peer apps
    - i.e. Leader election can be done through DNS, and does not need StatefulSet controller’s help
    - Over-generalized StatefulSet controller is application-agnostic and therefore cannot control complicated application state
  - Good thing is that Kubernetes makes it easy enough to define customer resources and write controllers

Abstraction Limitations

- Complicated Workload Case 1: **Workflow**
  - Stateful and run-to-complete
  - There used to be an official DAG workflow implementation, but was finally moved out from core API
  - Even usually defined as DAGs, Workflow can be complicated in many different ways and is very hard to generalize as core API
    - i.e. A deep-learning workflow can be totally different from a DevOps workflow
    - It's not always just DAG, it can also contain FOR-loops, If-Else, etc
  - Impl Discussions: https://github.com/kubernetes/kubernetes/pull/17787
  - Design Spec: https://github.com/kubernetes/kubernetes/pull/18827
  - Implementation: https://github.com/kubernetes/kubernetes/pull/24781
  - Discussion to Remove: https://github.com/kubernetes/kubernetes/issues/25067

Abstraction Limitations

- Complicated Workload Case 2: **Semi-stateful Apps**
  - DevOps use case: a pool of workers working on building docker images
  - Need a persistent volume as graph storage to cache docker image layers (if layer is not updated, don’t need to pull from remote)
  - Data (image layers) can be huge so not a good idea to use node’s root storage or memory
  - Losing data is fine as this is just a cache
  - But you can also say, it’s persisting data so its stateful…
  - StatefulSet is too heavy but ReplicationSet does not support dynamically provisioned data volumes

Abstraction Limitations

- Complicated Workload Case 3: **Sharded DB with Master/Slave of Each Shard**
  - Master RW, Slave RO
    - Master usually handle more workload
    - Need global view to balance load among multiple such applications
      - Node will suffer if too many replica on it become master
    - StatefulSet is NOT application-aware so additional work is needed
  - When a particular shard has request spike, what's better?
    - Possible action 1: Scale **horizontally** and re-shard
      - Horizontal-scaler might help
      - Overhead in data migration and re-sharding
      - Scale back is also challenging
    - Possible action 2: Scale **vertically** (up to node capacity) and evict
      - Might remove less important Pods from node and they can hopefully get re-scheduled

Conclusion

What makes Kubernetes successful

Take-aways from Kube's Design

- API Server and Versioned API Groups
  - Easy upgrade / backward compatibility
  - API server performs conversion between client and metadata store
- Protect cluster, every where
  - Throttle, resource quota limit @ API server
    - Important production question: **Given a pre-provisioned metadata store, what is a reasonable amount of mutating / non-mutating operations I can smoothly handle in parallel based on my SLA?**
  - QPS, exponential backoff with jittered retry @ client

Take-aways from Kube's Design

- Optimize Etcd READs
  - Cache @ API server - serve all READs from memory
    - It's just meta data, size can be reasonable for memory
  - Shared Informer - aggregate READs from all controllers
- Micro-service choreography
  - Everyone focus on their own API objects and different pipelines are formed automatically
  - Atomic commits on single API objects, reconciliation loops will finally make things right
- Level-triggered control
  - Controller always go back and assert state, so nothing can be missing

Why Cluster Management

- As tedious as endless on-calls
- As sexy as and as important as an orchestrator that
  - Increases resource utilization and save money
  - Automates resource provisioning / scheduling / scaling (both up and down) / failure detection and recovery
  - Provides methods to debug from outside the cluster
    - i.e. execute debug commands in your container
  - Plugs solutions such as Logging, Monitoring, Dashboard, Security, Admission Control, etc.
  - Release labor for feature development

Why is Kubernetes Successful

- Great abstraction for plug-and-play simple workload
  - Writing control logics can somewhat be a burden, especially for startups, and Kubernetes made it much easier
- Environment-agnostic interfaces
  - CRI, CNI, Flex Volume makes it possible to handle hybrid environment
- Native support for popular public cloud
  - Out-of-box resource provisioning and management
- Plug-and-play cluster management solutions from community
  - Cluster autoscaler, logging, monitoring, admission, etc.
  - Just run `kubectl create` you will have your app

Borg, Omega, Kubernetes

- Borg @ Google: https://research.google.com/pubs/pub43438.html
- Omega @ Google: https://research.google.com/pubs/pub41684.html
- From Borg, Omega to Kubernetes: http://queue.acm.org/detail.cfm?id=2898444