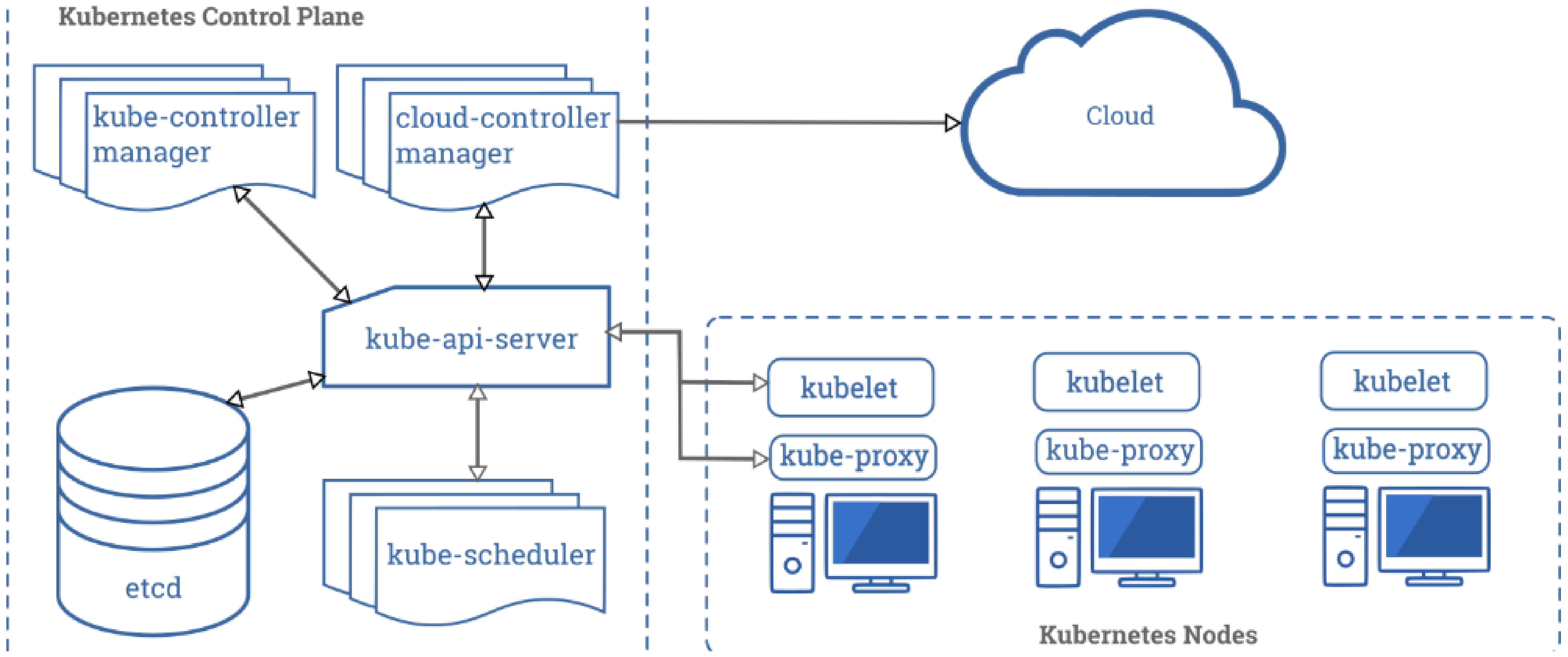
Kubernetes (V1.18) Study

Shenjie Yao
x/xx/2020

Components

- Control Plane
  - api-server
  - kube-controller-manager
  - cloud-controller-manager
  - scheduler
  - etcd
- Node
  - kubelet
  - kube-proxy
  - Container-Runtime

Components

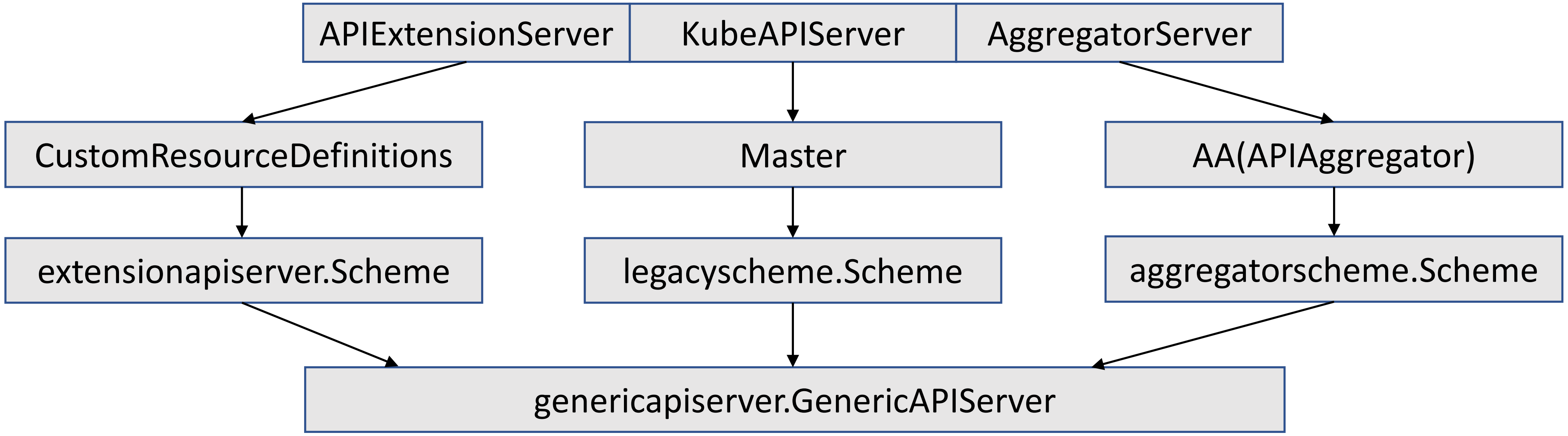

Kubernetes Control Plane
kube-controller manager
cloud-controller manager
Cloud
kube-api-server
etcd
kube-scheduler
kubelet
kube-proxy
kubelet
kube-proxy
kubelet
kube-proxy
Kubernetes Nodes

What is kube-apiserver?

- The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.
- The main implementation of a Kubernetes API server is kube-apiserver. kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

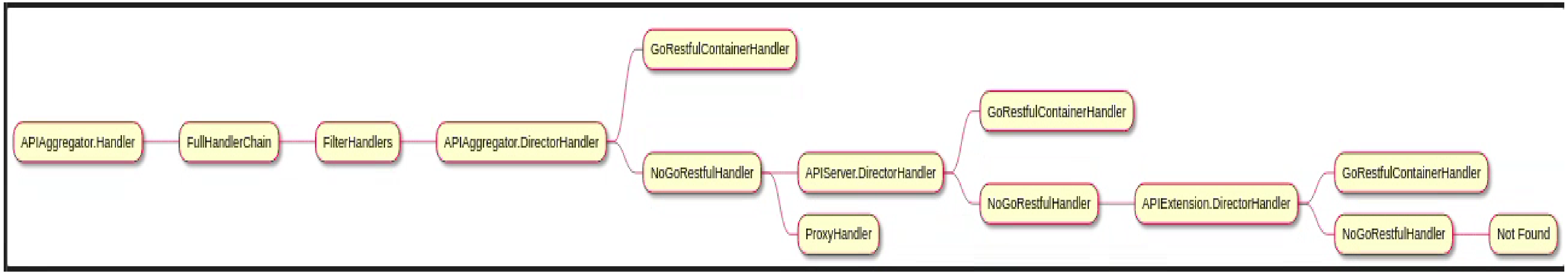
Kube-apiserver

- Kube-apiserver supplied three HTTP Server services for enriching ecosystem. These three HTTP Servers respectively is APIExtensionServer, KubeAPIServer and AggregatorServer. whatever which one Server is, it will depend on GenericAPIServer.
- Kube-apiserver's architecture design as following diagram:



Kube-apiserver

- This is an invoking chain for these three HTTP Servers:
  - AggregatorServer -> APIServer -> APIExtensionServer
- Following diagram shows that when received a HTTP request will invoke which one Server's handler to handle it.


APIAggregator.Handler
FullHandlerChain
FilterHandlers
APIAggregator.DirectorHandler
GoRestfulContainerHandler
NoGoRestfulHandler
APIServer.DirectorHandler
ProxyHandler
GoRestfulContainerHandler
NoGoRestfulHandler
APIExtension.DirectorHandler
GoRestfulContainerHandler
NoGoRestfulHandler
Not Found

APIAggregator

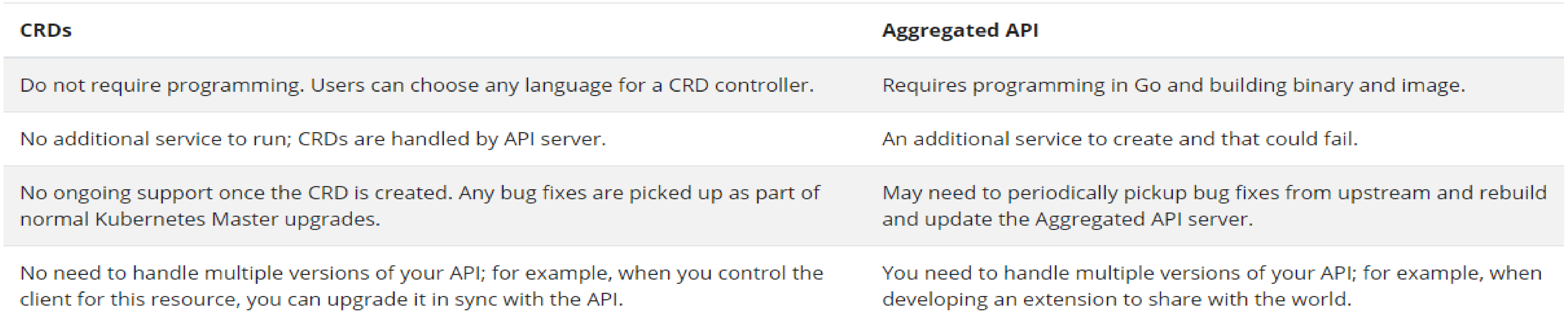
- API Aggregation requires programming, but allows more control over API behaviors like how data is stored and conversion between API versions.
- The aggregation layer allows Kubernetes to be extended with additional APIs, beyond what is offered by the core Kubernetes APIs. The additional APIs can either be ready-made solutions such as service-catalog, or APIs that you develop yourself.
- The aggregation layer is different from Custom Resources, which are a way to make the kube-apiserver recognise new kinds of object.
- The aggregation layer allows you to provide specialized implementations for your custom resources by writing and deploying your own standalone API server. The main API server delegates requests to you for the custom resources that you handle, making them available to all of its clients.

APIExtension

- The CustomResourceDefinition API resource allows you to define custom resources. Defining a CRD object creates a new custom resource with a name and schema that you specify. The Kubernetes API serves and handles the storage of your custom resource. The name of a CRD object must be a valid DNS subdomain name.
- This frees you from writing your own API server to handle the custom resource, but the generic nature of the implementation means you have less flexibility than with API server aggregation.
- Refer to the custom controller example for an example of how to register a new custom resource, work with instances of your new resource type, and use a controller to handle events.

APIAggregator VS APIExtension

- CRDs are easier to use. Aggregated APIs are more flexible. Choose the method that best meets your needs.
- Typically, CRDs are a good fit if:
  - You have a handful of fields
  - You are using the resource within your company, or as part of a small open-source project (as opposed to a commercial product)



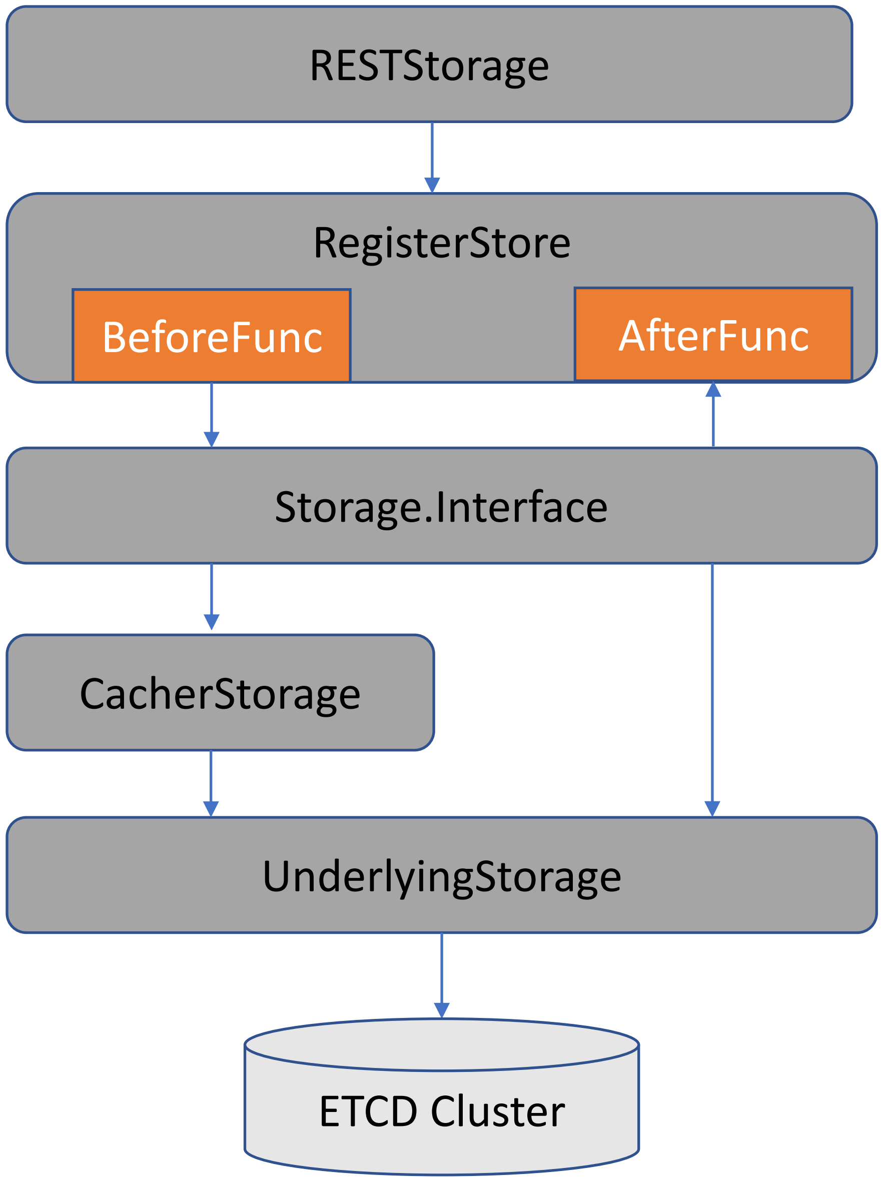
CRDs	Aggregated API
Do not require programming. Users can choose any language for a CRD controller.	Requires programming in Go and building binary and image.
No additional service to run; CRDs are handled by API server.	An additional service to create and that could fail.
No ongoing support once the CRD is created. Any bug fixes are picked up as part of normal Kubernetes Master upgrades.	May need to periodically pickup bug fixes from upstream and rebuild and update the Aggregated API server.
No need to handle multiple versions of your API; for example, when you control the client for this resource, you can upgrade it in sync with the API.	You need to handle multiple versions of your API; for example, when developing an extension to share with the world.

APIServer

- APIServer is a core service. It supplied Kubernetes in-built core resources services, and it doesn’t allow developer to change relevant resources.
- API core services is managed by Master object. And through legacyscheme.Scheme resource regedit manage Master relevant resources.

ETCD Storage

- Kubernetes makes a plenty of encapsulation for ETCD storage. And it split this encapsulation to multiple layers, and each layer supports high extensibility.
- The right diagram shows relationship of among these layers

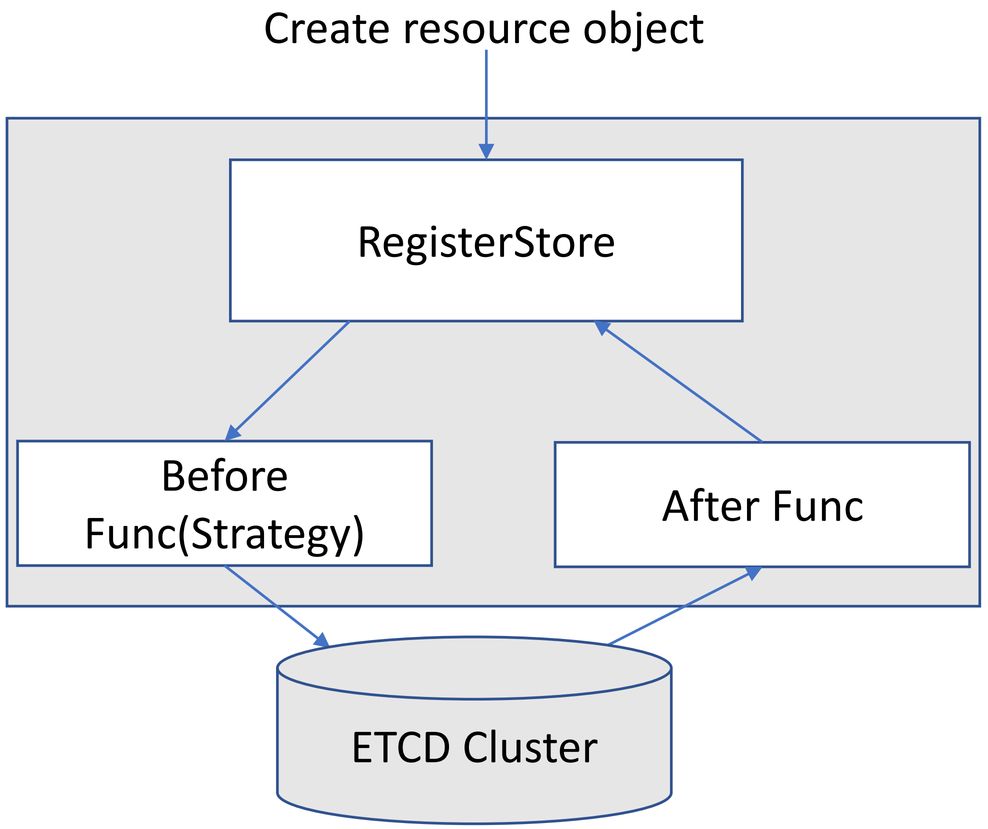


RESTStorage

- RESTStorage is a storage service general interface.
- So Every resources of the Kubernetes supplied RESTful API interface to external resources(RESTStorage interface). And all of resources which expose to external through RESTful API must implement this RESTStorage interface.
- Each RESTStorage interface encapsulated RegisterStore.

RegisterStore

- RegisterStore implemented general operation of the resource storage. Such as, execute some function before or after store resource object.
- Before Func(aka Strategy preprocess): it aims to do some preprocess works before store resource object.
- After Func: it aims to do some post-process works after store resource object.
- Last, it encapsulated Storage.Interface.



RegisterStore Strategy

- **CreateStrategy** : is a preprocess operation as creating a resource object. It defined a BeforeCreate function.
- **UpdateStrategy**: is a preprocess operation as updating a resource object. It defined a BeforeUpdate function.
- **DeleteStrategy**: is a preprocess operation as deleting a resource object. It defined a BeforeDelete function.
- **ExportStrategy**: is a preprocess operation as exporting a resource object. It defined an Export method. only some part of resources implemented this method.

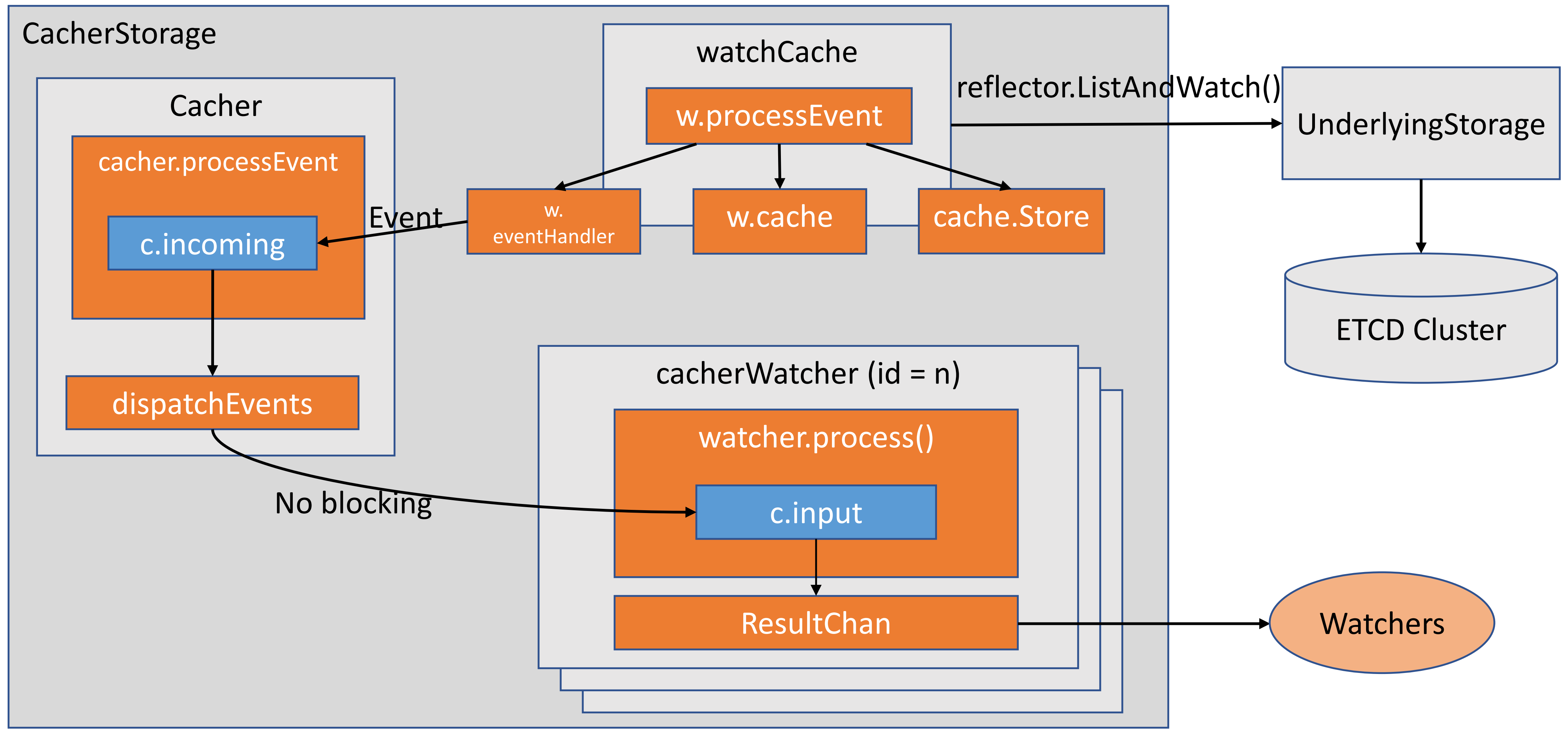
Storage.Interface

- Storage.Interface is a general storage interface and it defined operation method of the resources.
- CacherStorage and UnderlyingStorage implemented all of this interfaces.

CacherStorage

- CacherStorage supplied cache service of the DB data layer. It aims to quickly response and reduce pressure of the DB layer.
- The advantage of this design is that quickly response and return required data. It can reduce connection amount of the ETCD cluster and keep the data consistent between returned data and ETCD cluster.
- CacherStorage is not cache data for all of operations. For some operations ,in order to keep data consistent , so we don’t necessary add this cacher layer. such as Create, Delete and Count operation, we can use UnderlyingStorage sending require to ETCD cluster directly.

CacherStorage



CacherStorage

- cacheWatcher: Watcher observer management.
- watchCache: interacting with UnderlyingStorage layer through Reflector framework, let UnderlyingStorage to interact directly with ETCD cluster, and respectively put callback event into w.eventHandler, w.cache and cache.Store.
- Cacher: dispatching events to all of observers who has connected, and It implemented by non-blocking mechanism in dispatching process.

UnderlyingStorage

- UnderlyingStorage also know as BackendStorage in the code. It is a real resource storage object interacting with ETCD cluster.
- UnderlyingStorage encapsulated official ETCD client library. And it also is an implementation of Storage.Interface general storage interface.

References

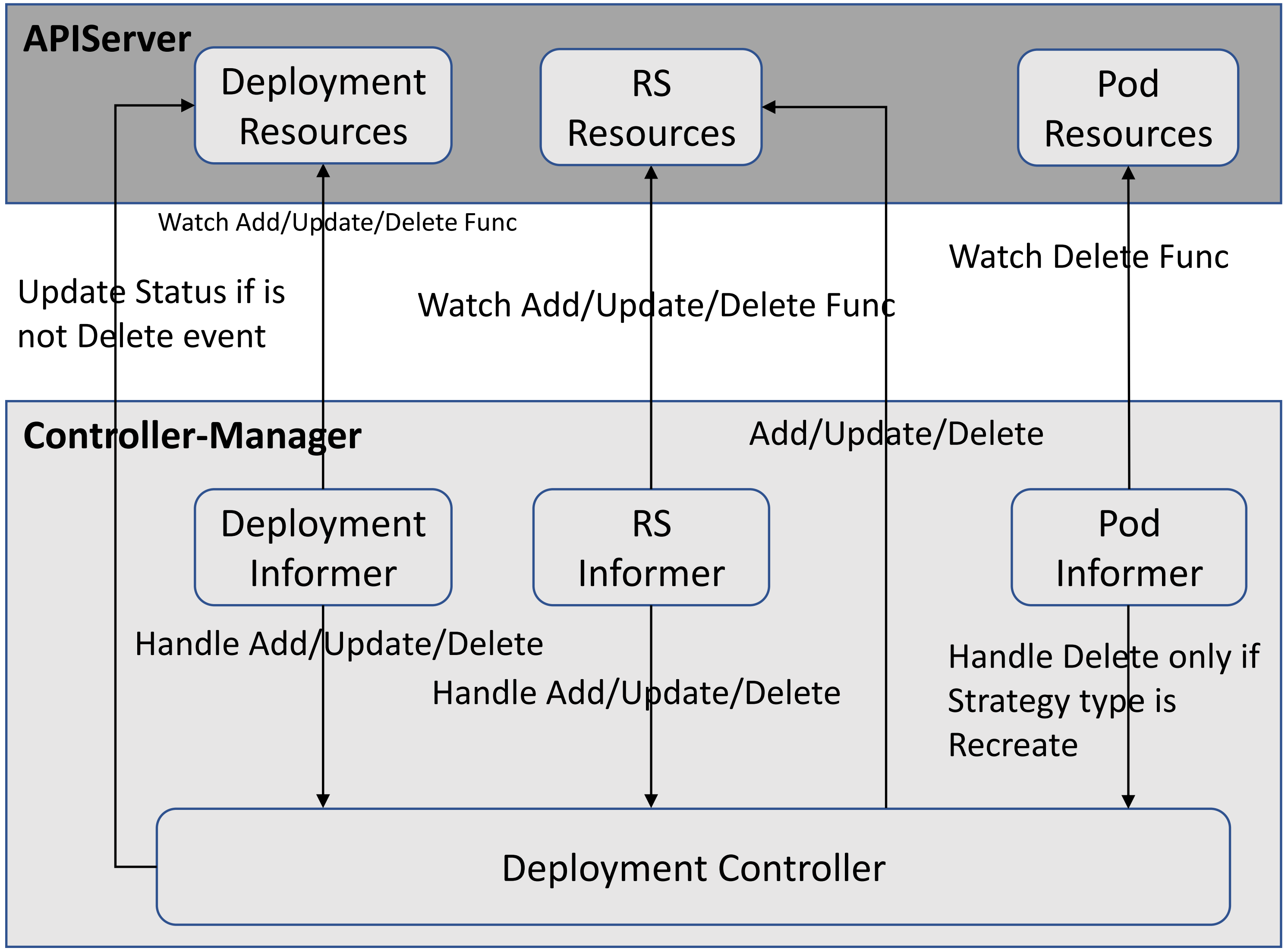
- https://kubernetes.io/docs/reference/access-authn-authz/controlling-access/
- https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/apiserver-aggregation/
- https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/
- https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/
- https://kubernetes.io/docs/concepts/extend-kubernetes/

Controller-manager

- Kube-controller-manager
  - Control Plane component that runs controller (**a control loop that watches the shared state of the cluster through the api-server and make changes attempting to move current state towards desired state**)processes.
  - Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.
- Cloud-controller-manager

Deployment Controller

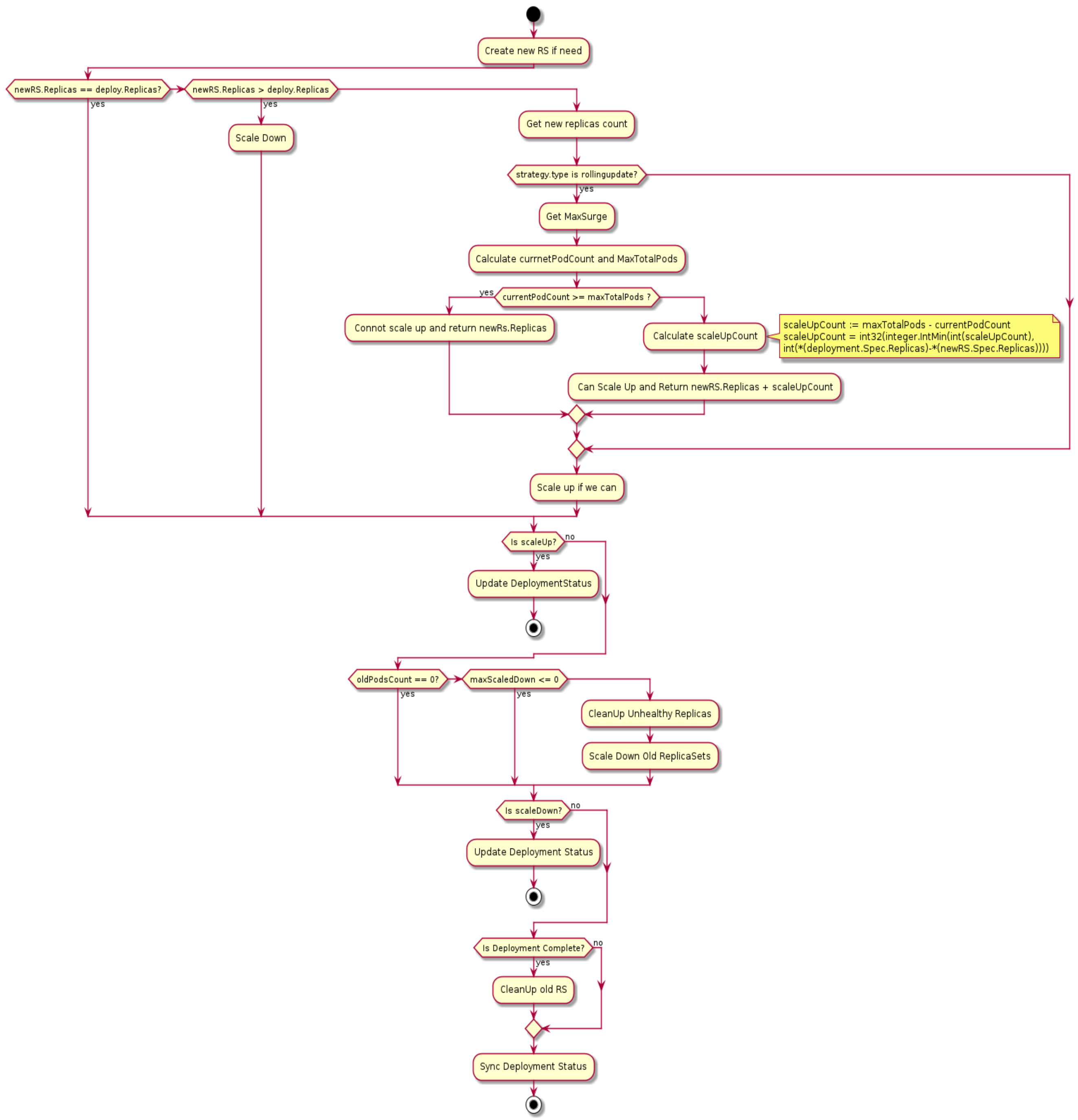
- A *Deployment* provides declarative updates for Pods ReplicaSets.
- You describe a *desired state* in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.
- Deployment ensures that only a certain number of Pods are down while they are being updated. By default, it ensures that at least 75% of the desired number of Pods are up (25% max unavailable).
- Deployment also ensures that only a certain number of Pods are created above the desired number of Pods. By default, it ensures that at most 125% of the desired number of Pods are up (25% max surge).
- https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment



Deployment Controller

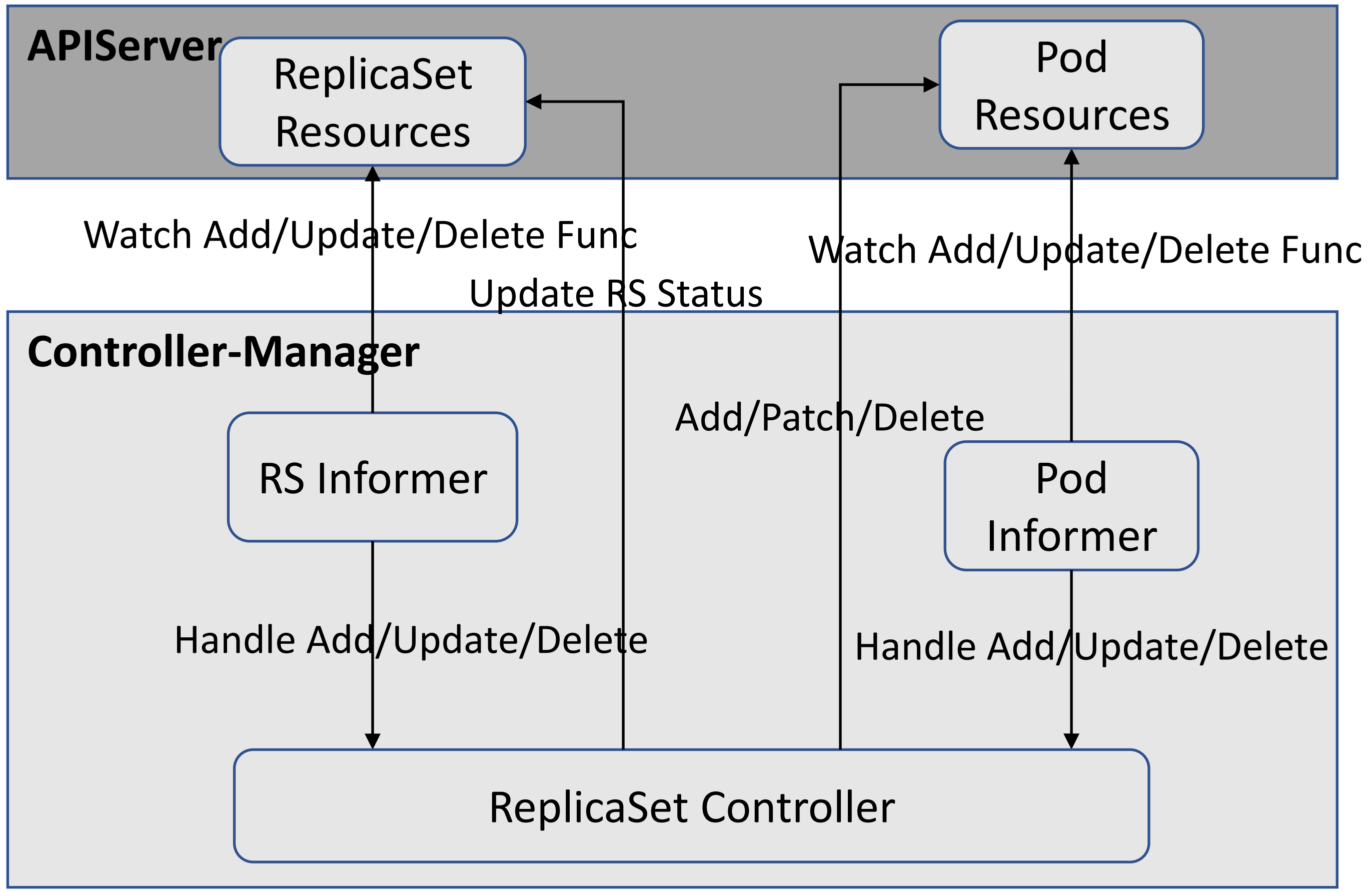
- Strategy Type
  - Recreate – scale down old replica set first then scale up new replica set.
  - RollingUpdate – scale up the new replica set if we can or scale down the old replica set if we can.

    maxScaleDown = allPodsCount – (*(deployment.Spec.Rplicas) - maxUnavailable) – (*(newRS.Spec.Replicas) – newRS.Status.AvailableReplics)


Create new RS if need
newRS.Replicas == deploy.Replicas?
yes
newRS.Replicas > deploy.Replicas
yes
Scale Down
Get new replicas count
strategy.type is rollingupdate?
yes
Get MaxSurge
Calculate currnetPodCount and MaxTotalPods
currentPodCount >= maxTotalPods ?
yes
Connot scale up and return newRs.Replicas
Calculate scaleUpCount
scaleUpCount := maxTotalPods - currentPodCount
scaleUpCount = int32(integer.IntMin(int(scaleUpCount),
int(*(deployment.Spec.Replicas)-*(newRS.Spec.Replicas))))
Can Scale Up and Return newRS.Replicas + scaleUpCount
Scale up if we can
Is scaleUp?
no
yes
Update DeploymentStatus
oldPodsCount == 0?
yes
maxScaledDown <= 0
yes
CleanUp Unhealthy Replicas
Scale Down Old ReplicaSets
Is scaleDown?
no
yes
Update Deployment Status
Is Deployment Complete?
no
yes
CleanUp old RS
Sync Deployment Status

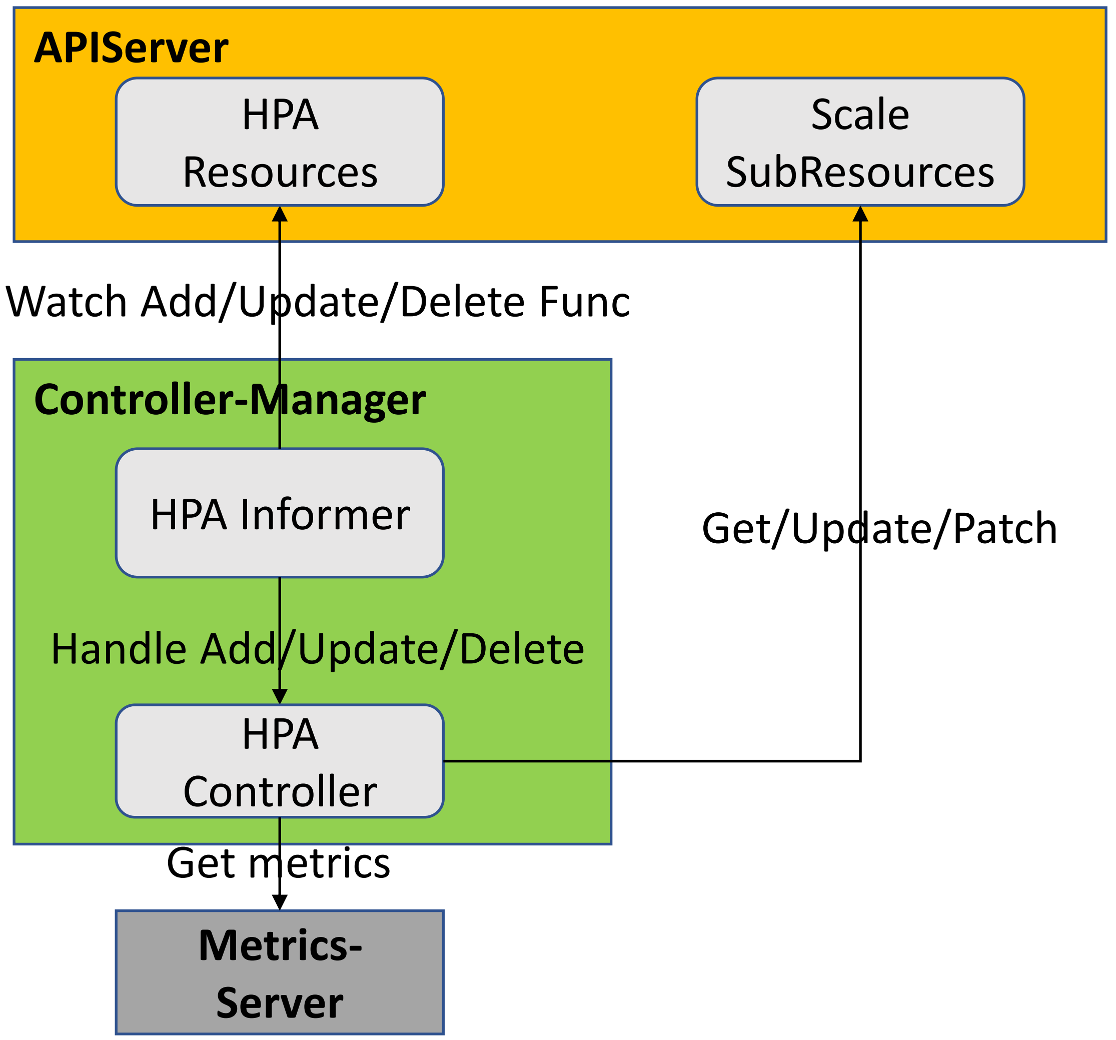
ReplicaSet Controller

- A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.
- A ReplicaSet is defined with fields, including a selector that specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintaining, and a pod template specifying the data of new Pods it should create to meet the number of replicas criteria. A ReplicaSet then fulfills its purpose by creating and deleting Pods as needed to reach the desired number. When a ReplicaSet needs to create new Pods, it uses its Pod template.
- A ReplicaSet is linked to its Pods via the Pods' metadata.ownerReferences field, which specifies what resource the current object is owned by. All Pods acquired by a ReplicaSet have their owning ReplicaSet's identifying information within their ownerReferences field. It's through this link that the ReplicaSet knows of the state of the Pods it is maintaining and plans accordingly.
- A ReplicaSet identifies new Pods to acquire by using its selector. If there is a Pod that has no OwnerReference or the OwnerReference is not a Controller and it matches a ReplicaSet's selector, it will be immediately acquired by said ReplicaSet.
- https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/



PodAutoScaler Controller

- The Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller, deployment, replica set or stateful set based on observed CPU utilization (or, with custom metrics support, on some other application-provided metrics). Note that Horizontal Pod Autoscaling does not apply to objects that can't be scaled, for example, DaemonSets.
- The Horizontal Pod Autoscaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. The controller periodically adjusts the number of replicas in a replication controller or deployment to match the observed average CPU utilization to the target specified by user.
- Algorithm details please reference this https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#algorithm-details.

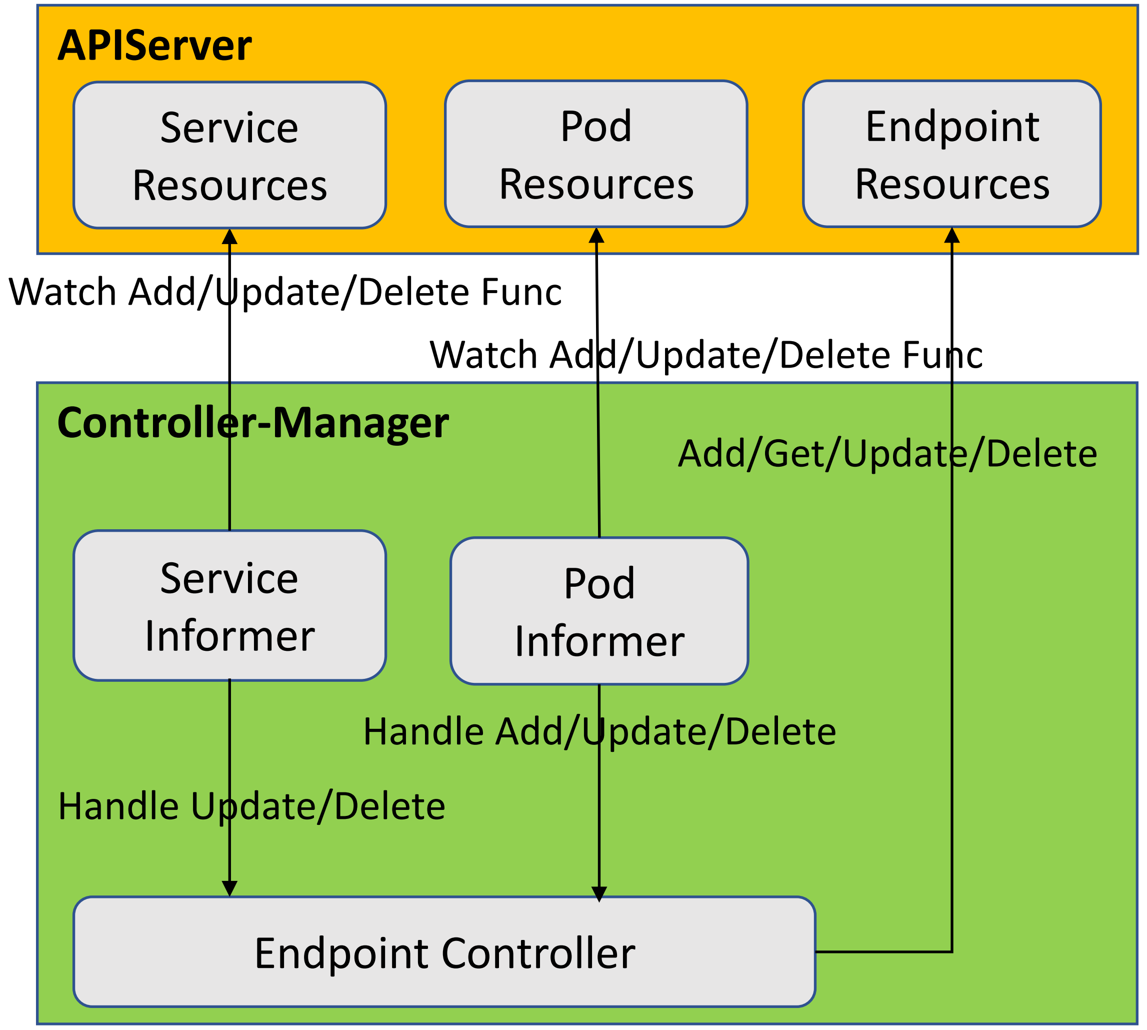


Service Controller

- TODO

Endpoint Controller

- Endpoint Controller is a mainly method to create and delete endpoint resource. It will obtain all of relevant pods according to Selector in the Service object. And mapping port in the between Service and Pod to generate a EndpointPort struct.
- Endpoint Controller will generate a new EndpointSubset for each Pod, it including Pod's IP and Port , and import Port and target Port in the Service. And at last it will according to EndpointSubset to repackage and to create a new endpoint resource.



Garbage Collector Controller

- TODO

Custom controller

- Using **client-go** to write a controller directly.
  - This is an official example for using client-go to write a controller. https://github.com/kubernetes/sample-controller.
- Using **controller-runtime** to write a controller.
  - TODO
- Using **kubebuilder** or **operator SDK** to generate a controller.
  - These two tools can generate a CRD and its controller. Hence if you want to write a CRD, you can choose one of these two tools. But if you only want to write a controller to control k8s resources. I recommend you directly use client-go or controller-runtime.

References

- https://kubernetes.io/docs/concepts/architecture/controller/
- https://kubernetes.io/docs/concepts/architecture/cloud-controller/
- https://github.com/kubernetes/sample-controller
- https://kubernetes.io/docs/concepts/workloads/controllers/
- https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

Scheduler

- In Kubernetes, *scheduling* refers to making sure that Pods are matched to Nodes so that Kubelet can run them.
- A scheduler watches for newly created Pods that have no Node assigned. For every Pod that the scheduler discovers, the scheduler becomes responsible for finding the best Node for that Pod to run on.
- kube-scheduler is the default scheduler for Kubernetes and runs as part of the control plane. kube-scheduler is designed so that, if you want and need to, you can write your own scheduling component and use that instead.
- For every newly created pod or other unscheduled pods, kube-scheduler selects an optimal node for them to run on. However, every container in pods has different requirements for resources and every pod also has different requirements. Therefore, existing nodes need to be filtered according to the specific scheduling requirements.
- In a cluster, Nodes that meet the scheduling requirements for a Pod are called *feasible* nodes. If none of the nodes are suitable, the pod remains unscheduled until the scheduler is able to place it.
- The scheduler finds feasible Nodes for a Pod and then runs a set of functions to score the feasible Nodes and picks a Node with the highest score among the feasible ones to run the Pod. The scheduler then notifies the API server about this decision in a process called *binding*.
- Factors that need taken into account for scheduling decisions include individual and collective resource requirements, hardware / software / policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and so on.

Scheduler

- kube-scheduler selects a node for a pod in following 2-step operation:
  - Filtering
  - Scoring
- The **filtering** step finds the set of Nodes where it’s feasible to schedule the Pod. For example, the PodFitsResources filter checks whether a candidate Node has enough available resource to meet a Pod’s specific resource requests. After this step, the node list contains any suitable Nodes; often, there will be more than one. If the list is empty, that Pod isn’t (yet) schedulable.
- The **scoring** step, the scheduler ranks the remaining nodes to choose the most suitable Pod placement. The scheduler assigns a score to each Node that survived filtering, basing this score on the active scoring rules.
- Finally, kube-scheduler assigns the Pod to the Node with the highest ranking. If there is more than one node with equal scores, kube-scheduler selects one of these at random.

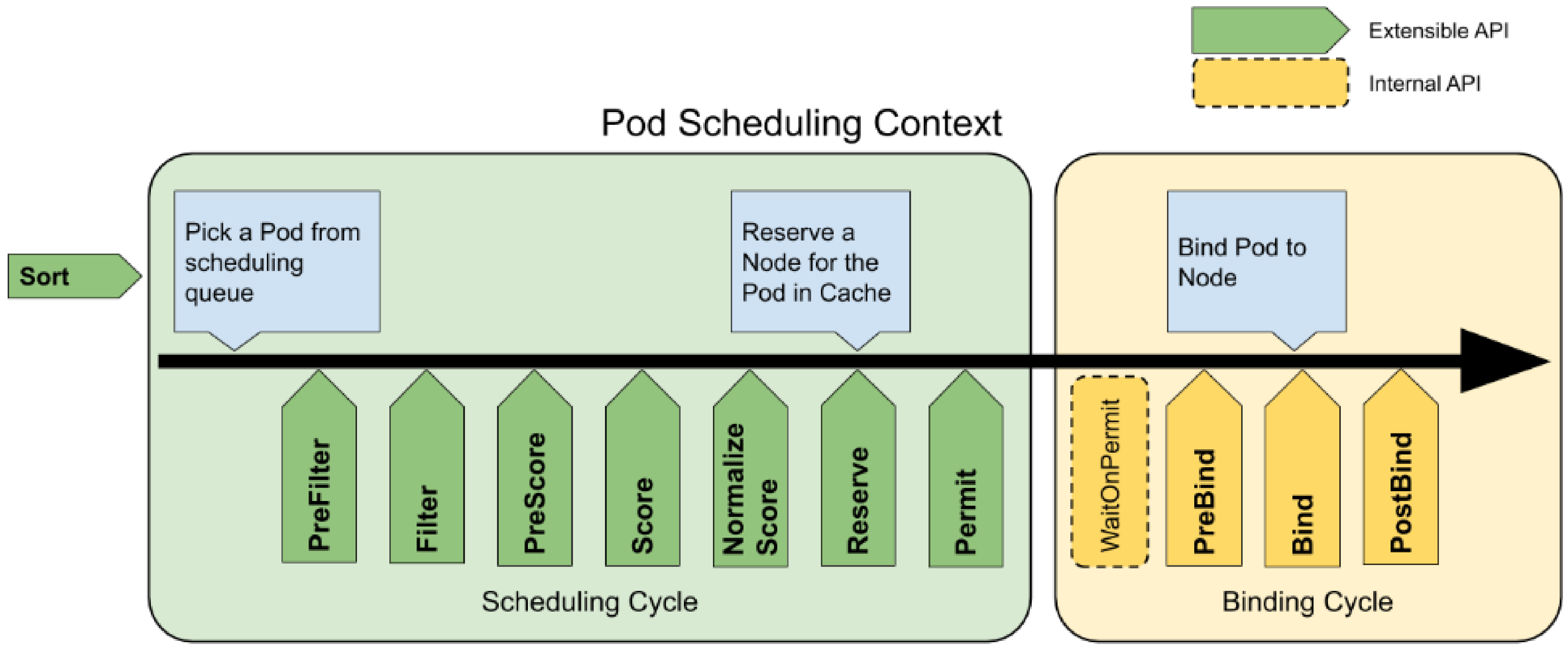
Scheduler

- There are two supported ways to configure the filtering and scoring behavior of the scheduler:
- Scheduling Policies allow you to configure *Predicates* for filtering and *Priorities* for scoring.
- Scheduling Profiles allow you to configure Plugins that implement different scheduling stages, including : QueueSort, Filter, Score, Bind, Reserve, Permit, and others. You can also configure the kube-scheduler to run different profiles.

Scheduling Framework

- The scheduling framework is a pluggable architecture for Kubernetes Scheduler that makes scheduler customizations easy. It adds a new set of "plugin" APIs to the existing scheduler. Plugins are compiled into the scheduler. The APIs allow most scheduling features to be implemented as plugins, while keeping the scheduling "core" simple and maintainable. Refer to the design proposal of the scheduling framework for more technical information on the design of the framework.
- The Scheduling Framework defines a few extension points. Scheduler plugins register to be invoked at one or more extension points. Some of these plugins can change the scheduling decisions and some are informational only.
- Each attempt to schedule one Pod is split into two phases, the **scheduling cycle** and the **binding cycle**.

Extension Points



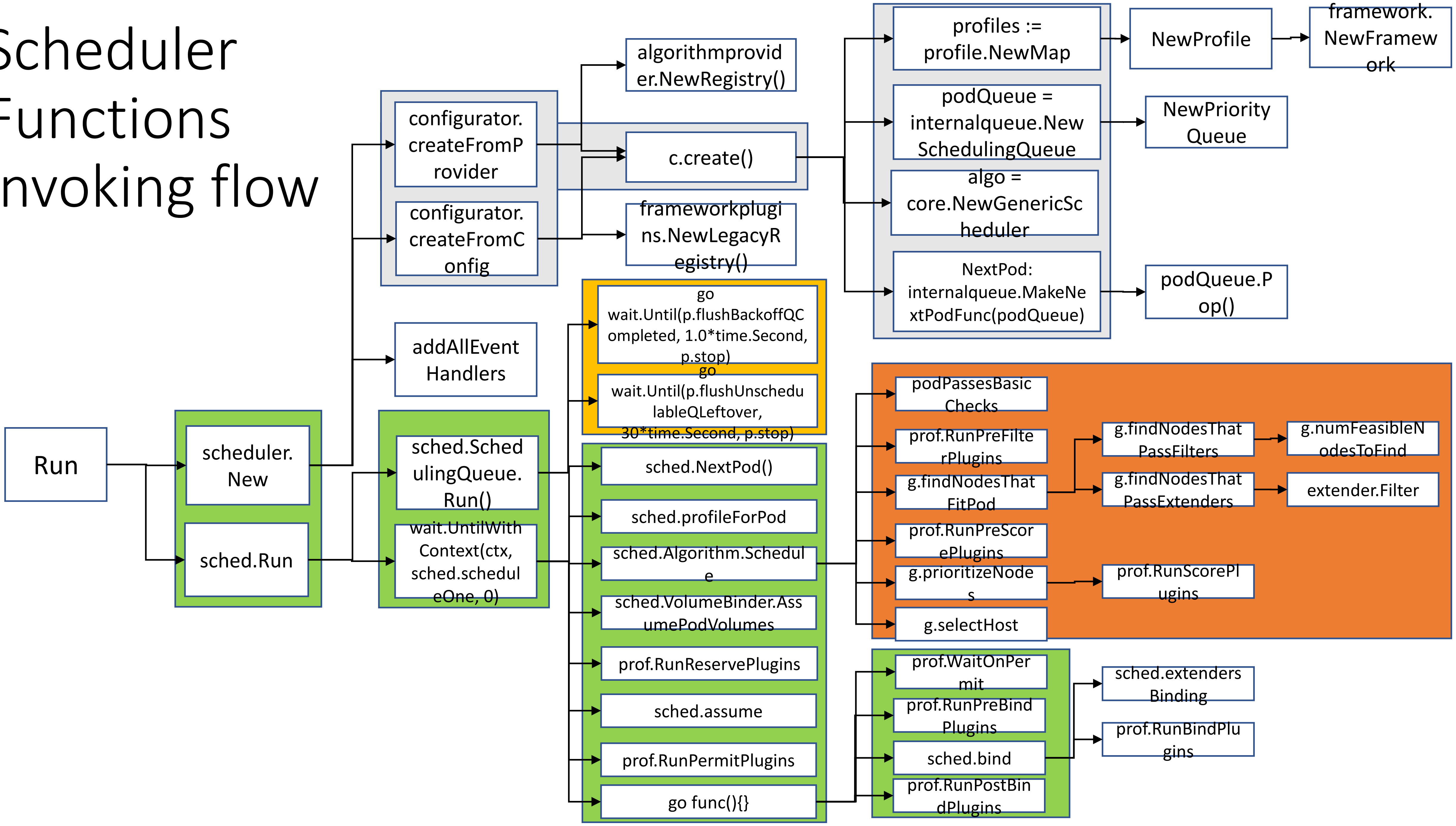
Pod Scheduling Context

- **Queue sort -** These plugins are used to sort Pods in the scheduling queue. A queue sort plugin essentially provides a Less(Pod1, Pod2) function. Only one queue sort plugin may be enabled at a time.
- **PreFilter -** These plugins are used to pre-process info about the Pod, or to check certain conditions that the cluster or the Pod must meet. If a PreFilter plugin returns an error, the scheduling cycle is aborted.
- **Filter -** These plugins are used to filter out nodes that cannot run the Pod. For each node, the scheduler will call filter plugins in their configured order. If any filter plugin marks the node as infeasible, the remaining plugins will not be called for that node. Nodes may be evaluated concurrently.
- **PreScore -** These plugins are used to perform "pre-scoring" work, which generates a sharable state for Score plugins to use. If a PreScore plugin returns an error, the scheduling cycle is aborted.
- **Score -** These plugins are used to rank nodes that have passed the filtering phase. The scheduler will call each scoring plugin for each node. There will be a well defined range of integers representing the minimum and maximum scores. After the NormalizeScore phase, the scheduler will combine node scores from all plugins according to the configured plugin weights.
- **Reserve -** This is an informational extension point. Plugins which maintain runtime state (aka "stateful plugins") should use this extension point to be notified by the scheduler when resources on a node are being reserved for a given Pod. This happens before the scheduler actually binds the pod to the Node, and it exists to prevent race conditions while the scheduler waits for the bind to succeed.
- **Permit -** *Permit* plugins are invoked at the end of the scheduling cycle for each Pod, to prevent or delay the binding to the candidate node.
- **PreBind -** These plugins are used to perform any work required before a Pod is bound. For example, a pre-bind plugin may provision a network volume and mount it on the target node before allowing the Pod to run there.
- **Bind -** These plugins are used to bind a Pod to a Node. Bind plugins will not be called until all PreBind plugins have completed. Each bind plugin is called in the configured order. A bind plugin may choose whether or not to handle the given Pod. If a bind plugin chooses to handle a Pod, **the remaining bind plugins are skipped**.
- **PostBind -** This is an informational extension point. Post-bind plugins are called after a Pod is successfully bound. This is the end of a binding cycle, and can be used to clean up associated resources.

Default Schedule Algorithms

- Defaultpodtopologyspread
- Imagelocality
- Tainttoleration
- Nodename
- Nodeports
- Nodepreferavoidpods
- Nodeaffinity
- Podtopologyspread
- Nodeunschedulable
- Noderesources
    - NodeResourcesFit
    - NodeResourcesBalancedAllocation
    - NodeResourcesLeastAllocated
    - NodeResourcesMostAllocated
    - RequestedToCapacityRatio
    - NodeResourceLimits
- Volumebinding
- Volumerestrictions
- Volumezone
- Nodevolumelimits
    - NodeVolumeLimits(CSI)
    - AzureDiskLimits
    - CinderLimits
    - EBSLimits
    - GCEPDLimits
- Interpodaffinity
- Nodelabel
- Serviceaffinity
- Queuesort
- Defaultbinder

Scheduler Functions invoking flow


Run
scheduler.New
sched.Run
configurator.createFromProvider
configurator.createFromConfig
addAllEventHandlers
sched.SchedulingQueue.Run()
wait.UntilWithContext(ctx, sched.scheduleOne, 0)
algorithmprovider.NewRegistry()
c.create()
frameworkplugins.NewLegacyRegistry()
go wait.Until(p.flushBackoffQCompleted, 1.0*time.Second, p.stop)
go wait.Until(p.flushUnschedulableQLeftover, 30*time.Second, p.stop)
sched.NextPod()
sched.profileForPod
sched.Algorithm.Schedule
sched.VolumeBinder.AssumePodVolumes
prof.RunReservePlugins
sched.assume
prof.RunPermitPlugins
go func(){}
profiles := profile.NewMap
podQueue = internalqueue.NewSchedulingQueue
algo = core.NewGenericScheduler
NextPod: internalqueue.MakeNextPodFunc(podQueue)
NewProfile
framework.NewFramework
NewPriorityQueue
podQueue.Pop()
podPassesBasicChecks
prof.RunPreFilterPlugins
g.findNodesThatFitPod
prof.RunPreScorePlugins
g.prioritizeNodes
g.selectHost
g.findNodesThatPassFilters
g.findNodesThatPassExtenders
g.numFeasibleNodesToFind
extender.Filter
prof.RunScorePlugins
prof.WaitOnPermit
prof.RunPreBindPlugins
sched.bind
prof.RunPostBindPlugins
sched.extendersBinding
prof.RunBindPlugins

Sample-Scheduler

- If you want to write a custom scheduler, you can implement some of following interfaces defined by Kubernetes. https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/v1alpha1/interface.go. Such as you can reference these examples provided by k8s scheduler framework resource https://github.com/kubernetes/kubernetes/blob/release-1.19/pkg/scheduler/framework/plugins/examples. And also you can reference this example to implement a custom scheduler https://github.com/Luffy110/k8s-studies/tree/master/playground/sample-scheduler.

References

- https://kubernetes.io/docs/reference/scheduling/policies/
- https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/
- https://github.com/kubernetes/enhancements/blob/master/keps/sig-scheduling/20180409-scheduling-framework.md
- https://www.servicemesher.com/blog/202003-k8s-scheduling-framework/
- https://www.qikqiak.com/post/custom-kube-scheduler/
- https://github.com/kubernetes/enhancements/blob/master/keps/sig-scheduling/624-scheduling-framework/README.md

Kube-proxy

- Kube-proxy is a core component of the K8s, it be deployed in every Node to implement communication and load balancing of the between services.
- Kube-proxy is mainly responsible for creating proxy service for pod, and obtaining all of services information from APIserver, then according to these information to create relevant proxy service so that to implement request route and forward from Server to Pod.
- Now It supports **user space** proxy mode, **iptables** proxy mode and **ipvs** proxy mode.

What is Iptables?

- **iptables** is the user space command line program used to configure the Linux 2.4.x and later packet filtering ruleset. It is targeted towards system administrators.
- Since Network Address Translation(NAT) is also configured from the packet filter ruleset, **iptables** is used for this, too.
- Features:
    - listing the contents of the packet filter ruleset
    - adding/removing/modifying rules in the packet filter ruleset
    - listing/zeroing per-rule counters of the packet filter ruleset

What is IPVS?

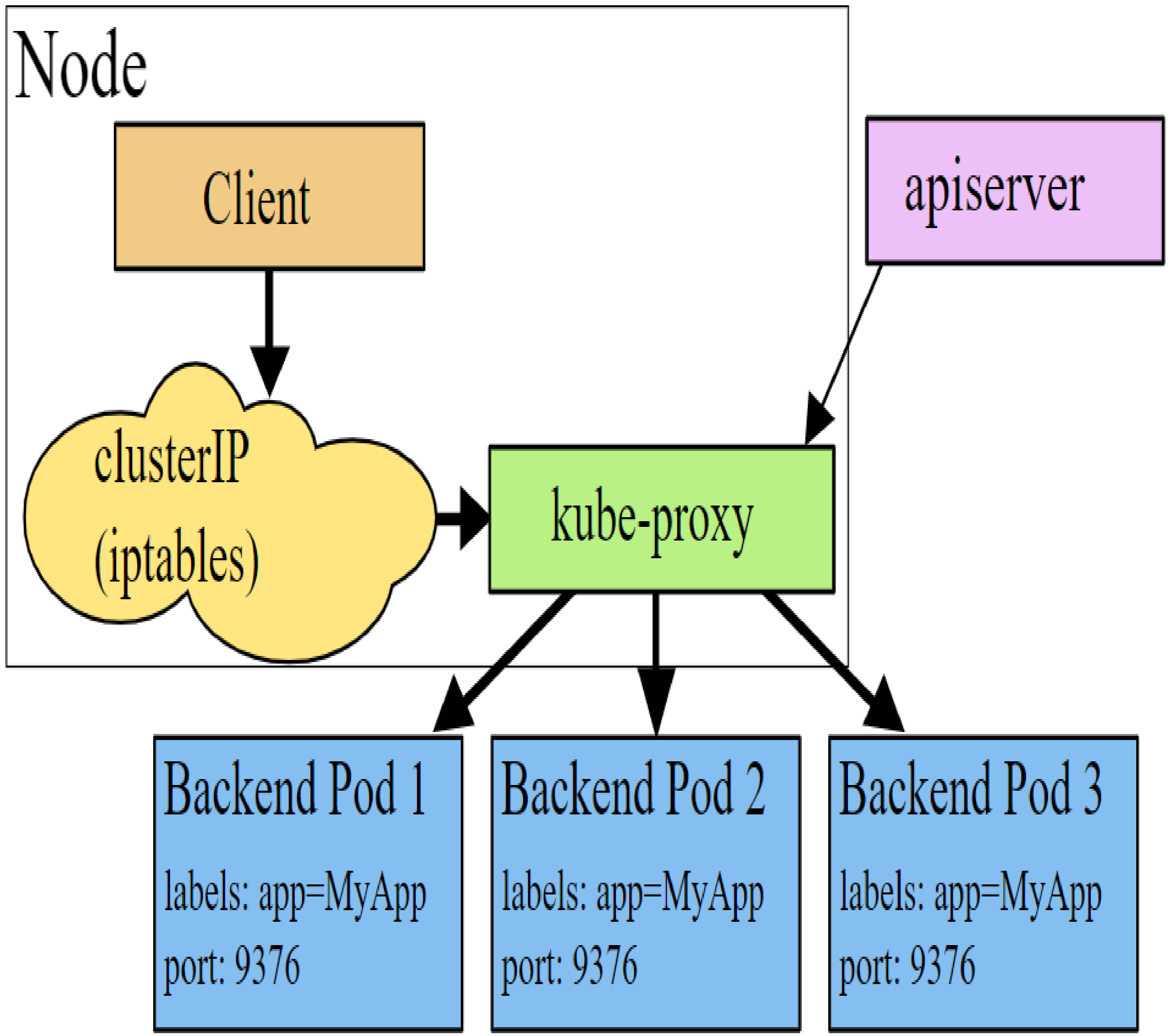
- **IPVS** (**IP Virtual Server**) implements transport-layer load balancing, usually called Layer 4 LAN switching, as part of the Linux kernel. It's configured via the user-space utility ipvsadm(8) tool.
- IPVS is incorporated into the Linux Virtual Server (LVS), where it runs on a host and acts as a load balancer in front of a cluster of real servers. IPVS can directly request for TCP- and UDP-based services to the real servers, and making services of the real servers appear as virtual services on a single IP address. IPVS is built on top of the Netfilter.[1]

What is Conntrack?

- Please reference this blog: http://arthurchiao.art/blog/conntrack-design-and-implementation-zh/.

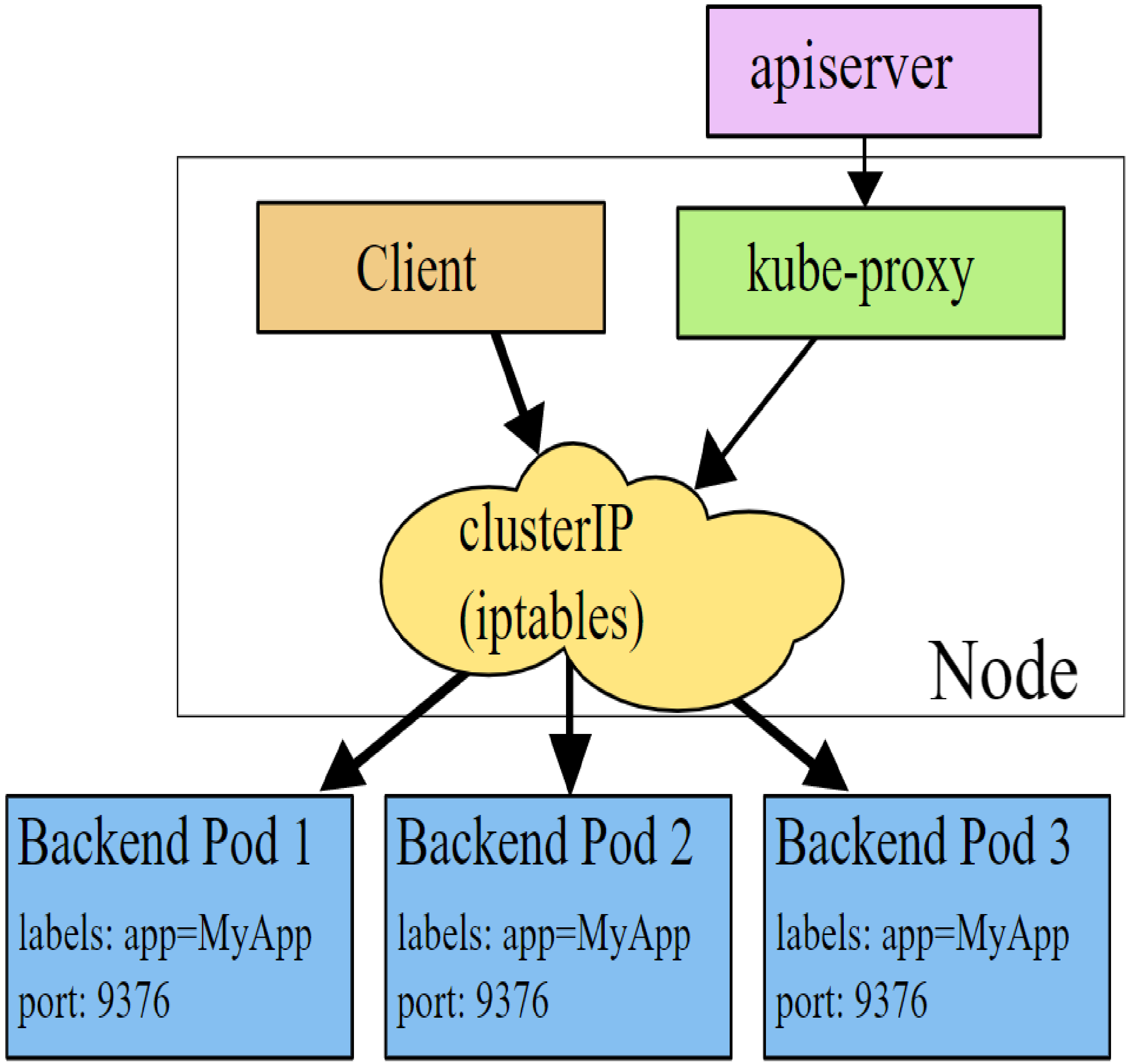
User Space Proxy Mode

- In this mode, kube-proxy watches the Kubernetes master for the addition and removal of Service and Endpoint objects. For each Service it opens a port (randomly chosen) on the local node. Any connections to this "proxy port" are proxied to one of the Service's backend Pods (as reported via Endpoints). kube-proxy takes the **SessionAffinity** setting of the Service into account when deciding which backend Pod to use.
- Lastly, the user-space proxy installs iptables rules which capture traffic to the Service's **clusterIP** (which is virtual) and **port**. The rules redirect that traffic to the proxy port which proxies the backend Pod.
- By default, kube-proxy in userspace mode chooses a backend via a round-robin algorithm.


Node
Client
apiserver
clusterIP (iptables)
kube-proxy
Backend Pod 1 labels: app=MyApp port: 9376
Backend Pod 2 labels: app=MyApp port: 9376
Backend Pod 3 labels: app=MyApp port: 9376

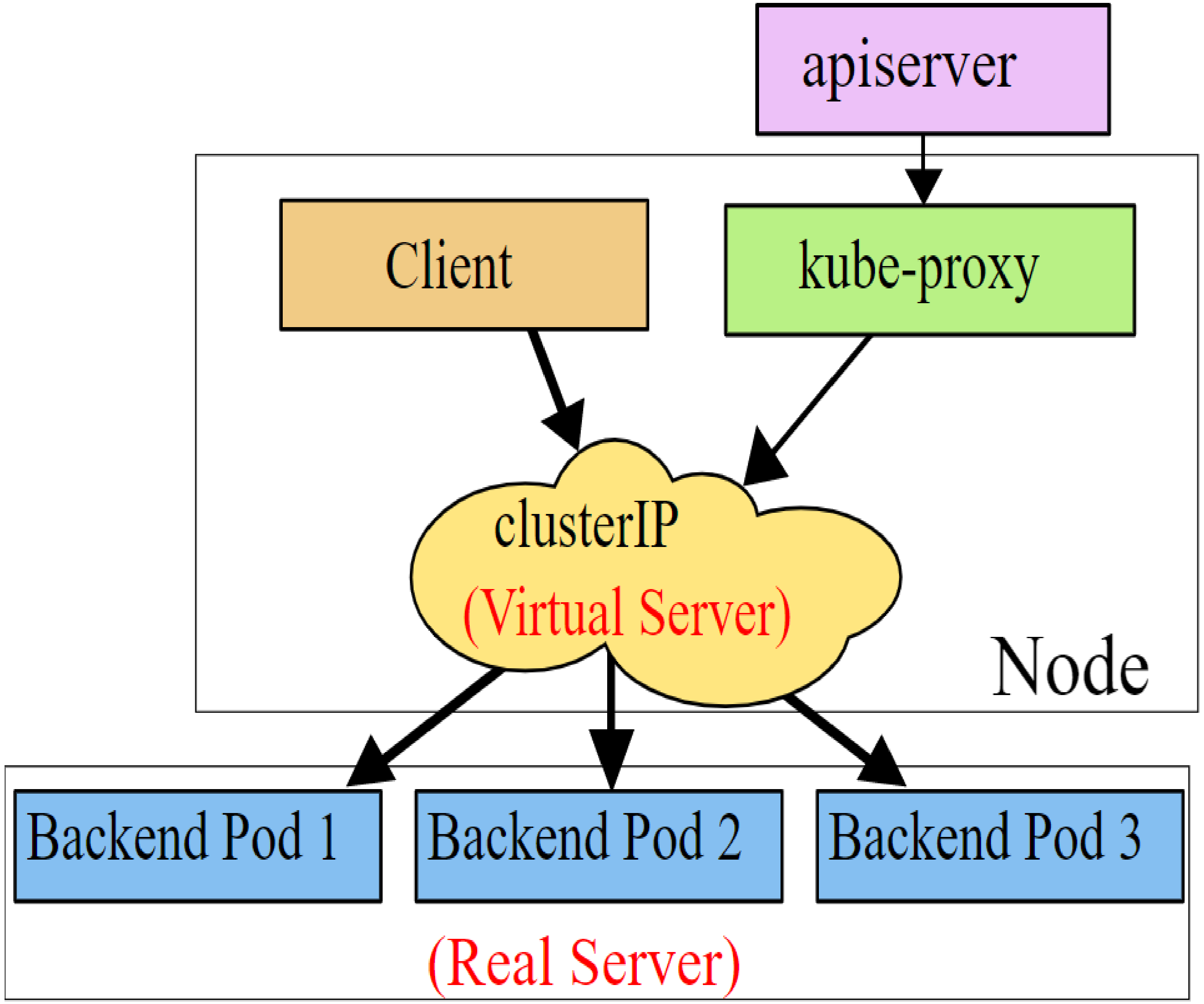
Iptables Proxy Mode

- In this mode, kube-proxy watches the Kubernetes control plane for the addition and removal of Service and Endpoint objects. For each Service, it installs iptables rules, which capture traffic to the Service's **clusterIP** and **port**, and redirect that traffic to one of the Service's backend sets. For each Endpoint object, it installs iptables rules which select a backend Pod.
- Using iptables to handle traffic has a lower system overhead, because traffic is handled by Linux netfilter without the need to switch between userspace and the kernel space. This approach is also likely to be more reliable.
- The way kube-proxy programs the iptables rules means that it is nominally an **O(n)** style algorithm, where n grows roughly in proportion to your cluster size



IPVS Proxy Mode

- In **IPVS** mode, kube-proxy watches Kubernetes Services and Endpoints, calls **netlink** interface to create IPVS rules accordingly and synchronizes IPVS rules with Kubernetes Services and Endpoints periodically. This control loop ensures that IPVS status matches the desired state. When accessing a Service, IPVS directs traffic to one of the backend Pods.
- The IPVS proxy mode is based on netfilter hook function that is similar to iptables mode, but uses a hash table as the underlying data structure and works in the kernel space. That means kube-proxy in IPVS mode redirects traffic with lower latency than kube-proxy in iptables mode, with much better performance when synchronising proxy rules. Compared to the other proxy modes, IPVS mode also supports a higher throughput of network traffic.
- IPVS provides more options for balancing traffic to backend Pods; these are round-robin, least connection, destination hashing, source hashing etc.
- The result is that kube-proxy's connection processing in IPVS mode has a nominal computational complexity of **O(1)**. In other words, in most scenarios, its connection processing performance will stay constant independent of your cluster size.

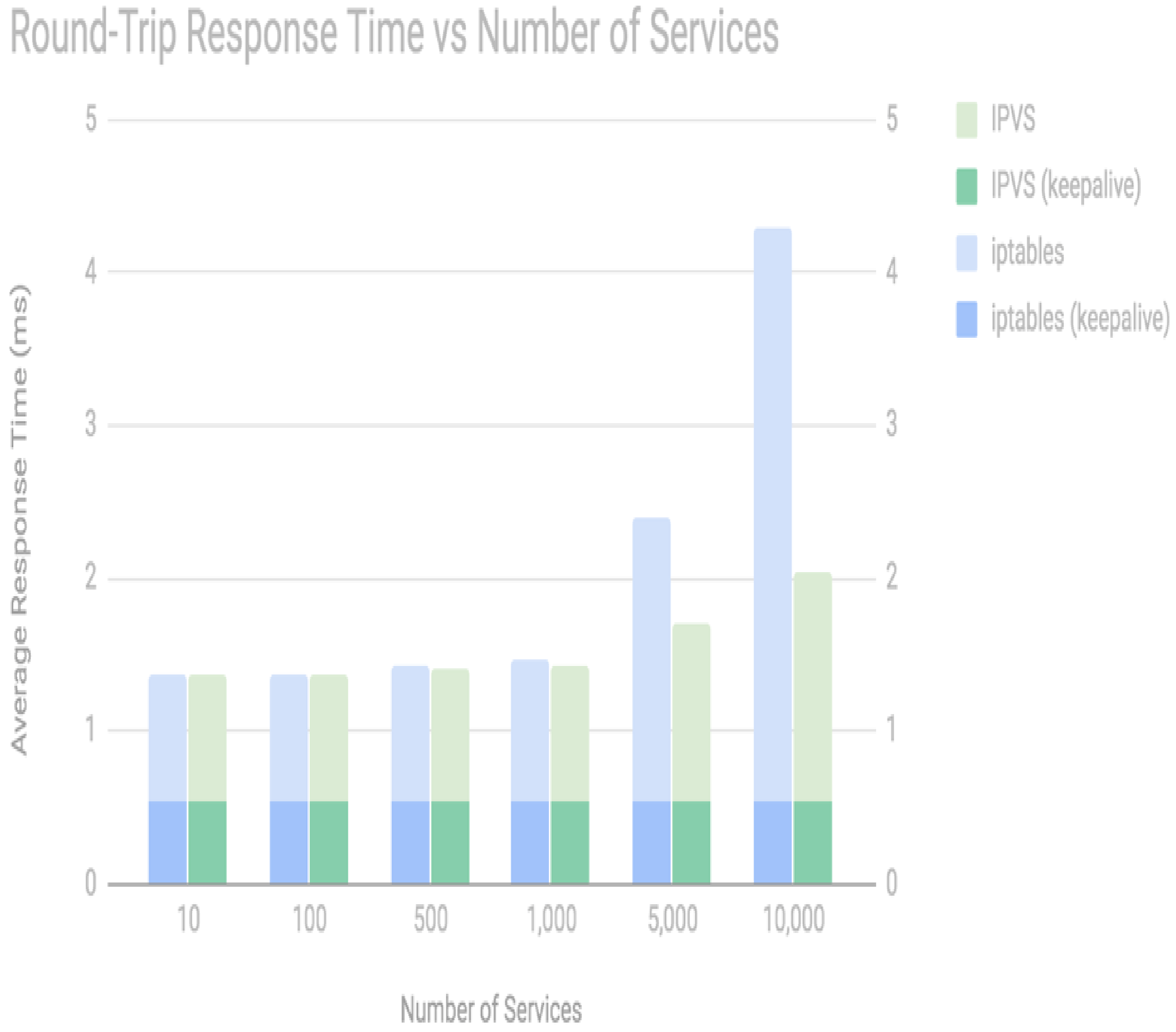


Iptables vs IPVS

- IPVS supplied better extendibility and performance for large cluster.
- IPVS supports more complex balancing algorithms than iptables.
- IPVS supports service healthy check and connection retry functionalities and so on.

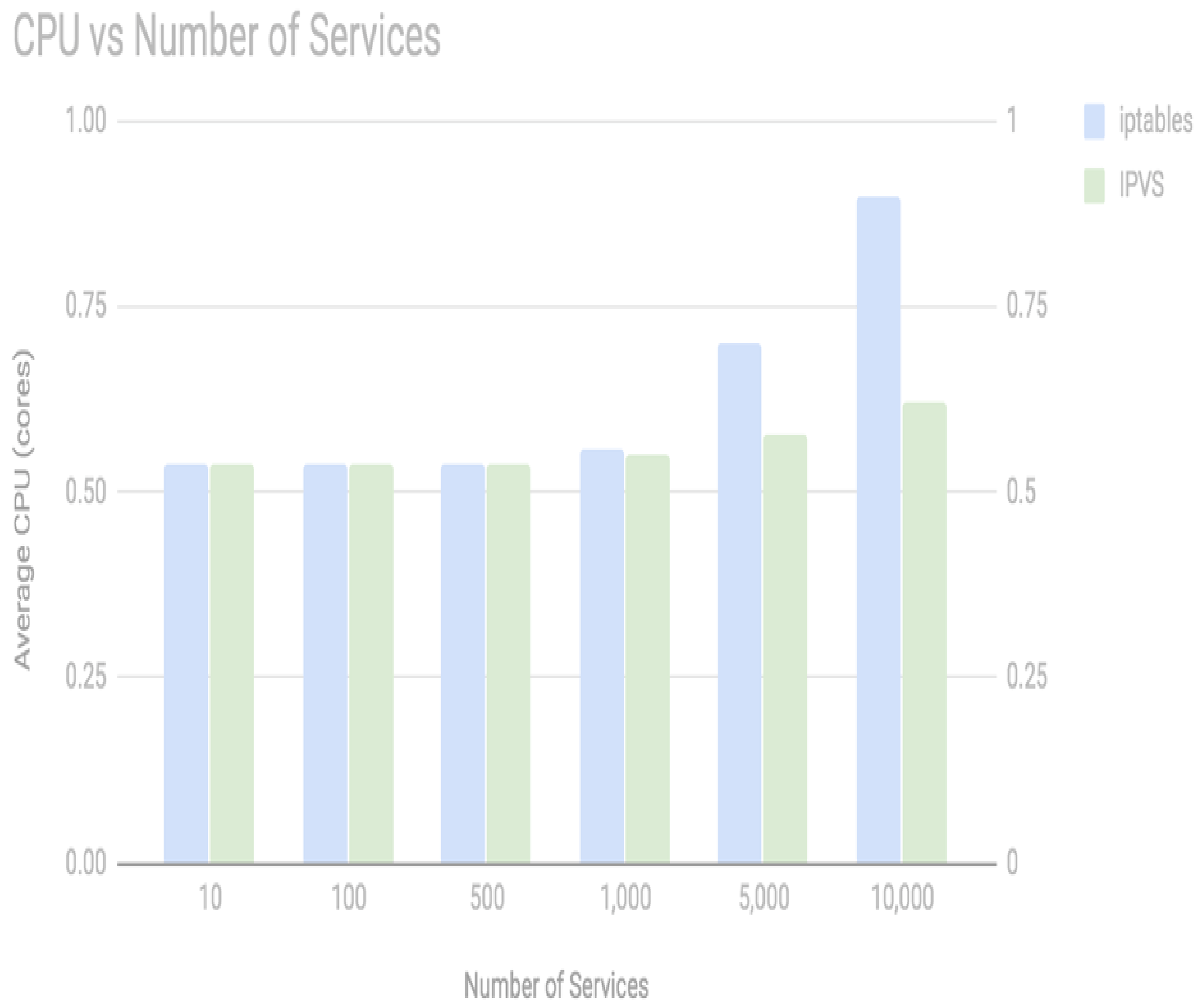
Iptables vs IPVS
- Round-Trip Response Times

- The right chart shows two key things:
  - The difference in average round-trip response times between iptables and IPVS is trivially insignificant until you get beyond 1,000 services (10,000 backend pods).
  - The difference in average round-trip response times is only discernible when not using keepalive connections. i.e. when using a new connection for every request.

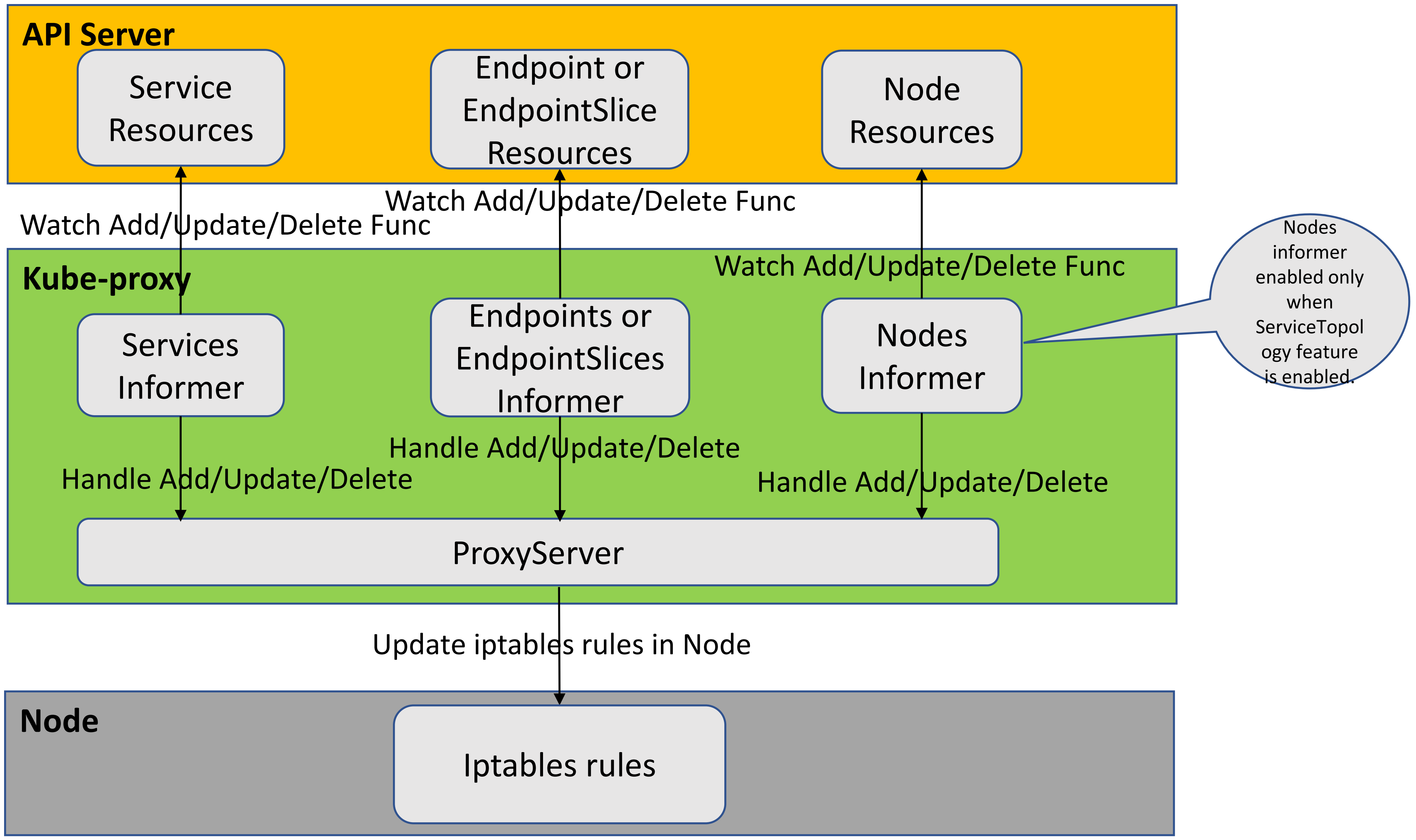


Iptables vs IPVS
- Total CPU Usage

- The right chart shows two key things:
  - The difference in CPU usage between iptables and IPVS is relatively insignificant until you get beyond 1,000 services (with 10,000 backend pods).
  - At 10,000 services (with 100,000 backend pods), the increase in CPU with iptables is ~35% of a core, and with IPVS is ~8% of a core.

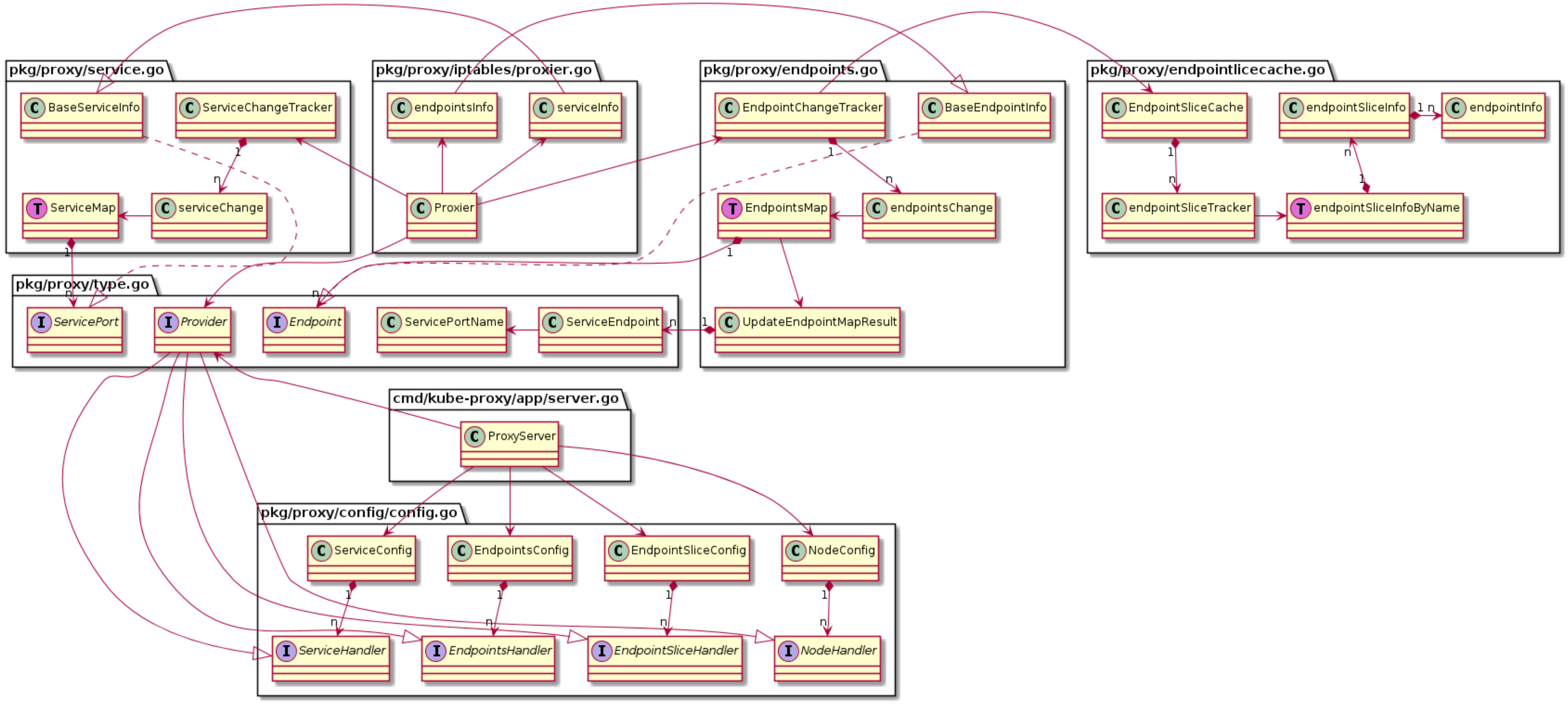


Kube-proxy informer flow

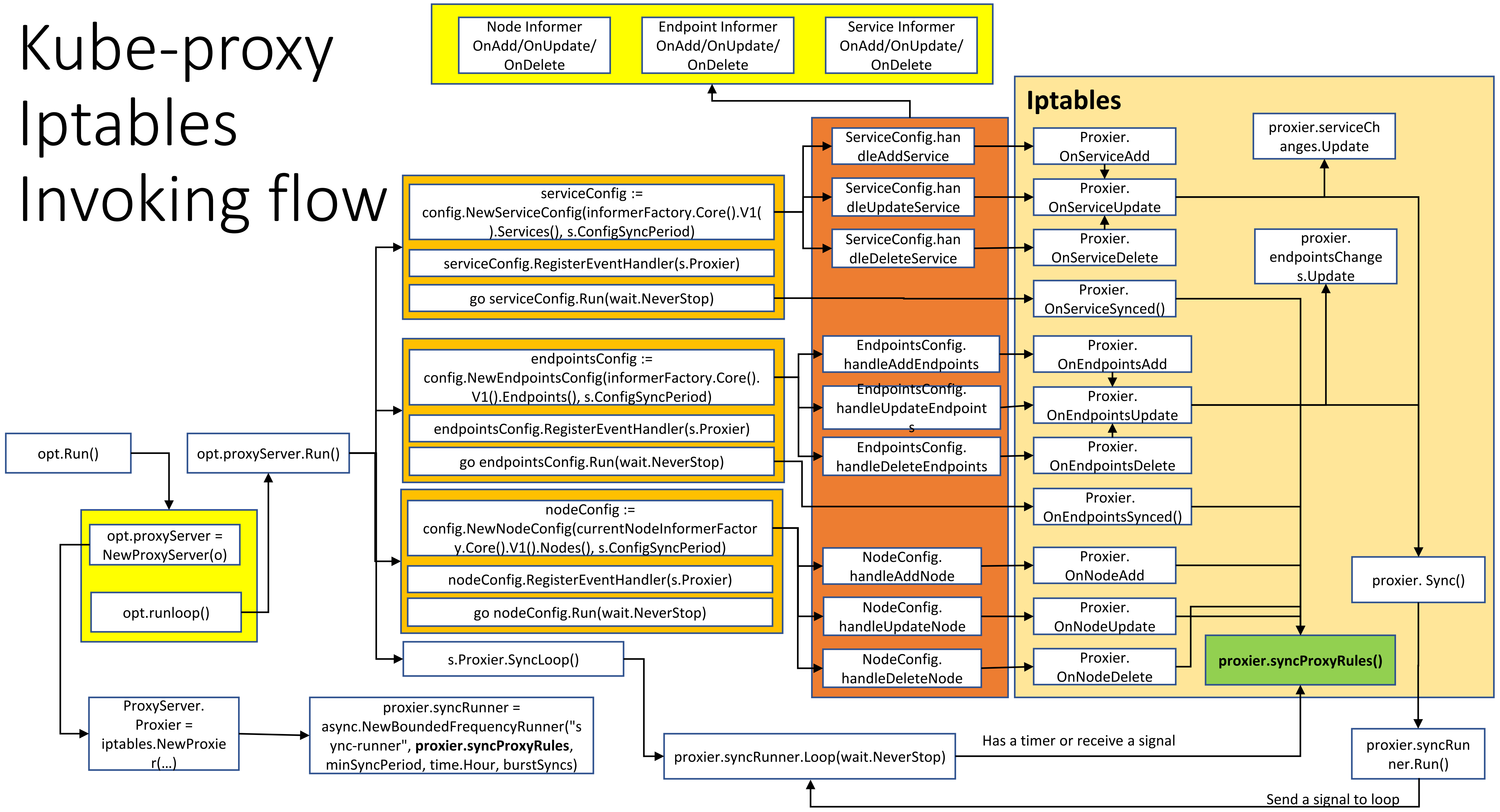


API Server
Service Resources
Endpoint or EndpointSlice Resources
Node Resources
Watch Add/Update/Delete Func
Watch Add/Update/Delete Func
Watch Add/Update/Delete Func
Kube-proxy
Services Informer
Endpoints or EndpointSlices Informer
Nodes Informer
Nodes informer enabled only when ServiceTopology feature is enabled.
Handle Add/Update/Delete
Handle Add/Update/Delete
Handle Add/Update/Delete
ProxyServer
Update iptables rules in Node
Node
Iptables rules

Kube-proxy class diagram



Kube-proxy Iptables Invoking flow



Node Informer OnAdd/OnUpdate/ OnDelete
Endpoint Informer OnAdd/OnUpdate/ OnDelete
Service Informer OnAdd/OnUpdate/ OnDelete
Iptables
ServiceConfig.handleAddService
ServiceConfig.handleUpdateService
ServiceConfig.handleDeleteService
Proxier. OnServiceAdd
Proxier. OnServiceUpdate
Proxier. OnServiceDelete
Proxier. OnServiceSynced()
proxier.serviceChanges.Update
proxier. endpointsChanges.Update
serviceConfig := config.NewServiceConfig(informerFactory.Core().V1().Services(), s.ConfigSyncPeriod)
serviceConfig.RegisterEventHandler(s.Proxier)
go serviceConfig.Run(wait.NeverStop)
endpointsConfig := config.NewEndpointsConfig(informerFactory.Core().V1().Endpoints(), s.ConfigSyncPeriod)
endpointsConfig.RegisterEventHandler(s.Proxier)
go endpointsConfig.Run(wait.NeverStop)
EndpointsConfig. handleAddEndpoints
EndpointsConfig. handleUpdateEndpoints
EndpointsConfig. handleDeleteEndpoints
Proxier. OnEndpointsAdd
Proxier. OnEndpointsUpdate
Proxier. OnEndpointsDelete
Proxier. OnEndpointsSynced()
opt.Run()
opt.proxyServer.Run()
opt.proxyServer = NewProxyServer(o)
opt.runloop()
nodeConfig := config.NewNodeConfig(currentNodeInformerFactory.Core().V1().Nodes(), s.ConfigSyncPeriod)
nodeConfig.RegisterEventHandler(s.Proxier)
go nodeConfig.Run(wait.NeverStop)
NodeConfig. handleAddNode
NodeConfig. handleUpdateNode
NodeConfig. handleDeleteNode
Proxier. OnNodeAdd
Proxier. OnNodeUpdate
Proxier. OnNodeDelete
proxier. Sync()
proxier.syncProxyRules()
s.Proxier.SyncLoop()
ProxyServer. Proxier = iptables.NewProxier(...)
proxier.syncRunner = async.NewBoundedFrequencyRunner("sync-runner", **proxier.syncProxyRules**, minSyncPeriod, time.Hour, burstSyncs)
proxier.syncRunner.Loop(wait.NeverStop)
Has a timer or receive a signal
proxier.syncRunner.Run()
Send a signal to loop

References

- https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/
- https://kubernetes.io/docs/concepts/services-networking/service/
- https://cloud.tencent.com/developer/article/1097449
- https://www.tigera.io/blog/comparing-kube-proxy-modes-iptables-or-ipvs/
- https://netfilter.org/projects/iptables/index.html
- https://en.wikipedia.org/wiki/IP_Virtual_Server
- https://www.qikqiak.com/post/how-to-use-ipvs-in-kubernetes/
- https://opengers.github.io/openstack/openstack-base-netfilter-framework-overview/
- https://www.frozentux.net/iptables-tutorial/chunkyhtml/x1309.html
- http://arthurchiao.art/blog/conntrack-design-and-implementation-zh/
- http://arthurchiao.art/blog/deep-dive-into-iptables-and-netfilter-arch-zh/
- https://conntrack-tools.netfilter.org/manual.html
- http://arthurchiao.art/blog/conntrack-design-and-implementation-zh/

kubelet

- The kubelet is the primary "node agent" that runs on each node. It makes sure that containers are running in a Pod. It can register the node with the apiserver using one of: the hostname; a flag to override the hostname; or specific logic for a cloud provider.
- The kubelet works in terms of a PodSpec. A PodSpec is a YAML or JSON object that describes a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms (primarily through the apiserver) and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.
- Other than from a PodSpec from the apiserver, there are three ways that a container manifest can be provided to the Kubelet.
  - **File**: Path passed as a flag on the command line. Files under this path will be monitored periodically for updates. The monitoring period is 20s by default and is configurable via a flag.
  - **HTTP endpoint**: HTTP endpoint passed as a parameter on the command line. This endpoint is checked every 20 seconds (also configurable with a flag).
  - **HTTP server**: The kubelet can also listen for HTTP and respond to a simple API (underspec'd currently) to submit a new manifest.

Kubelet - containerManager

- Source code in /pkg/kubelet/cm/

Kubelet - containerGC

- containerGC provides two methods GarbageCollect() and DeleteAllUnusedContainers() to collect containers and deletes all unused containers respectively. And the real implementation is the implementor of Runtime interfaces. The implementor is KubeRuntimeManager in kubelet source code.
- GC has three types Policy:
  - **MinAge** - Minimum age at which a container can be garbage collected
  - **MaxPerPodContainer** - Max number of dead containers any single pod (UID, container name) pair is allowed to have
  - **MaxContainers** - Max number of total dead containers
- Source code in /pkg/kubelet/container/container_gc.go

Kubelet - imageManager

- imageManager is for pulling images, it has two methods to pull images(**serial** and **parallel**).
- imageManager mainly encapsulated **Image Service** interface of CRI, and also it provides a way of throttle image pulling, and it used a **TokenBucketRateLimiter** to control flow if provided qps and burst parameters when invoking **NewImageManager** function.
- Source code in /pkg/kubelet/images/image_manager.go
- /pkg/kubelet/images/image_gc_manager.go

Kubelet – imageGCManager

- ImageGCManager is mainly for removing unused images if over the max threshold for freeing enough place, also it uses some interfaces of CRI.
- GC Policies:
  - **HighThresholdPercent** - Any usage above this threshold will always trigger garbage collection(This is the highest usage we will allow).
  - **LowThresholdPercent** - Any usage below this threshold will never trigger garbage collection(This is the lowest threshold we will try to garbage collect to).
  - **MinAge** - Minimum age at which an image can be garbage collected.
- Source code in /pkg/kubelet/images/image_gc_manager.go

Kubelet - statusManager

- The statusManager is mainly for updating pod statuses in APIServer, and it writes only when the new status has changed.
- Source code in /pkg/kubelet/status/

Kubelet - probeManager

- Manager manages pod probing. It creates a probe "worker" for every container that specifies a probe (AddPod). The worker periodically probes its assigned container and caches the results. The manager use the cached probe results to set the appropriate Ready state in the PodStatus when requested (UpdatePodStatus). Updating probe parameters is not currently supported.
- In implementation there are a concept of **worker**, it mainly for handling he periodic probing of its assigned container. Each worker has a go-routine associated with it which runs the probe loop until the container permanently terminates, or the stop channel is closed. The worker uses the probe Manager's statusManager to get up-to-date container IDs
- It has readinessManager, livenessManager and startupManager. Respectively for managing the results of readiness probes, the results of liveness probes and the results of startup probes.
- The prober provided following three types prober. it will select one that you provided in v1.Probe to send probe.
  - Exec
  - HTTP
  - tcp
- Source code in /pkg/kubelet/prober/

Kubelet - evictionManager

- Memory pressure
- DIsk pressure
- PID pressure
- Local storage eviction (if enabled LocalStorageCapacityIsolation feature)
  - containerEphemeralStorageLimitEviction
  - podEphemeralStorageLimitEviction
  - emptyDirLimitEviction
- Source code in /pkg/kubelet/eviction/

Kubelet - volumeManager

- https://draveness.me/kubernetes-volume/

Kubelet - runtimeManager

- runtimeManager provides a KubeGenericRuntimeManager. It mainly implemented all of container runtime interfaces(detailed interface, you can look at /pkg/kubelet/container/runtime.go).
- The runtimeManager is a real implementor of containerGC mechanism.
- Source code in /pkg/kubelet/kuberuntime/

Kubelet - podManager

- Manager stores and manages access to pods, maintaining the mappings between static pods and mirror pods. The kubelet discovers pod updates from 3 sources: file, http, and apiserver. Pods from non-apiserver sources are called static pods, and API server is not aware of the existence of static pods. In order to monitor the status of such pods, the kubelet creates a mirror pod for each static pod via the API server.
- A mirror pod has the same pod full name (name and namespace) as its static counterpart (albeit different metadata such as UID, etc). By leveraging the fact that the kubelet reports the pod status using the pod full name, the status of the mirror pod always reflects the actual status of the static pod. When a static pod gets deleted, the associated orphaned mirror pod will also be removed.

Kubelet - Pod Lifecycle Event Generator(PLEG)

- Getting pod current status from runtime, generating a pod lifecycle event and syncing it to kubelet pods cache.

Kubelet - cAdvisor

- Implemented all of interfaces of cAdvisor, so the real implementation, you should look at cAdvisor lib.

Kubelet - OOMWatcher

- Only start cAdvisor oomparser to do OOM, and this watcher only record events.
- So detailed OOM handling, please look atStreamOoms(chan<- *oomparser.OomInstance) method of github.com/google/cadvisor/utils/oomparser

References

- https://www.infoq.cn/article/odSLClSjvO8BNx*MbRbK
- https://feisky.xyz/posts/kubernetes-container-runtime/
- https://github.com/opencontainers/runtime-spec/blob/master/config-linux.md
- https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/
- https://www.bookstack.cn/read/source-code-reading-notes/kubernetes-kubelet-modules.md

Container-Runtime

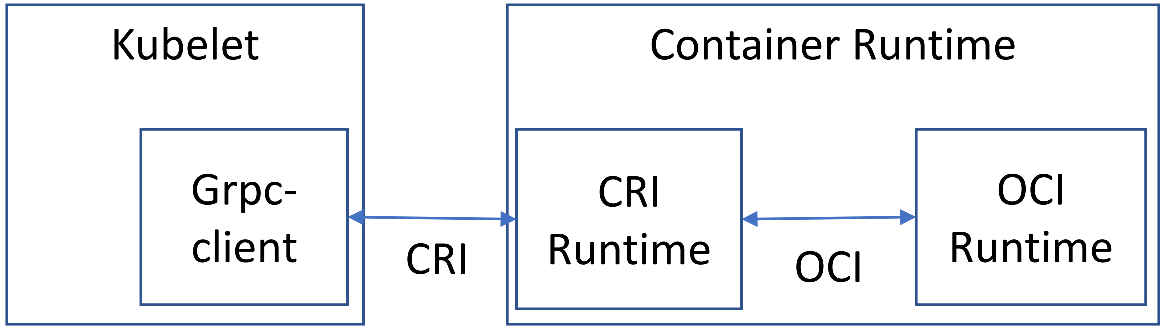
- Low-level container runtimes are responsible for setting up namespaces and cgroups for containers, and then running commands inside those namespaces and cgroups. Low-level runtimes support using these operating system features.
- High-level runtimes are responsible for transport and management of container images, unpacking the image, and passing off to the low-level runtime to run the container. Typically, high-level runtimes provide a daemon application and an API that remote applications can use to logically run containers and monitor but they sit on top of and delegate to low-level runtimes or other high-level runtimes for the actual work.

Container-Runtime

- Kubernetes runtimes are high-level container runtimes that support the Container Runtime Interface (CRI). and acts as a bridge between the kubelet and the container runtime. High-level container runtimes that want to integrate with Kubernetes are expected to implement CRI. The runtime is expected to handle the management of images and to support Kubernetes pods, as well as manage the individual containers.

Container-Runtime

- The kubelet uses CRI to communicate with the container runtime running on that same node. In this way CRI is simply an abstraction layer or API that allows you to switch out container runtime implementations instead of having them built into the kubelet.



Container Runtime Interface(CRI)

- CRI (*Container Runtime Interface*) consists of a protobuf API, specifications/requirements (to-be-added), and [libraries] (https://github.com/kubernetes/kubernetes/tree/master/pkg/kubelet/server/streaming) for container runtimes to integrate with kubelet on a node.
- https://github.com/kubernetes/cri-api/blob/master/pkg/apis/runtime/v1alpha2/api.proto

CRI Implementations

- Dockershim
  - Embedded into kubelet. And Dockershim communicates with docker.
- CRI-O
- Containerd
- Frakti
- rktlet

Open Container Initiative(OCI)

- The Open Container Initiative (OCI) is a lightweight, open governance structure (project), formed under the auspices of the Linux Foundation, for the express purpose of creating open industry standards around container formats and runtime.
- The OCI currently contains two specifications: **the Runtime Specification (runtime-spec)** and **the Image Specification (image-spec)**. The Runtime Specification outlines how to run a "filesystem bundle" that is unpacked on disk. At a high-level an OCI implementation would download an OCI Image then unpack that image into an OCI Runtime filesystem bundle. At this point the OCI Runtime Bundle would be run by an OCI Runtime.

OCI Runtime Specification

- The Open Container Initiative Runtime Specification aims to specify the configuration, execution environment, and lifecycle of a container.
- A container's configuration is specified as the config.json for the supported platforms and details the fields that enable the creation of a container. The execution environment is specified to ensure that applications running inside a container have a consistent environment between runtimes along with common actions defined for the container's lifecycle.

OCI Image Format Specification

- This specification defines an OCI Image, consisting of a manifest, an image index (optional), a set of filesystem layers, and a configuration.
- The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run.

OCI Runtime Implementations

- runc
- Kata containers

Container

- Containers are implemented using Linux namespaces and cgroups.
    - Namespaces let you virtualize system resources, like the file system or networking, for each container.
    - Cgroups provide a way to limit the amount of resources like CPU and memory that each container can use.

Kernel Namespaces

- **Namespaces** are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources. The feature works by having the same namespace for a set of resources and processes, but those namespaces refer to distinct resources. Resources may exist in multiple spaces. Examples of such resources are process IDs, hostnames, user IDs, file names, and some names associated with network access, and interprocess communication.

Kernel Namespaces

- Process ID(PID) namespace
  - The PID namespace provides processes with an independent set of process IDs (PIDs) from other namespaces. PID namespaces are nested, meaning when a new process is created it will have a PID for each namespace from its current namespace up to the initial PID namespace. Hence the initial PID namespace is able to see all processes, albeit with different PIDs than other namespaces will see processes with.
  - The first process created in a PID namespace is assigned the process id number 1 and receives most of the same special treatment as the normal init process, most notably that orphaned processes within the namespace are attached to it. This also means that the termination of this PID 1 process will immediately terminate all processes in its PID namespace and any descendants.[5]
- Mount namespace
  - Mount namespaces control mount points. Upon creation the mounts from the current mount namespace are copied to the new namespace, but mount points created afterwards do not propagate between namespaces (using shared subtrees, it is possible to propagate mount points between namespaces[4]).
  - The clone flag used to create a new namespace of this type is CLONE_NEWNS - short for "NEW NameSpace". This term is not descriptive (as it doesn't tell which kind of namespace is to be created) because mount namespaces were the first kind of namespace and designers did not anticipate there being any others.
- Network namespace
  - Network namespaces virtualize the network stack. On creation a network namespace contains only a loopback interface.
  - Each network interface (physical or virtual) is present in exactly 1 namespace and can be moved between namespaces.
  - Each namespace will have a private set of IP addresses, its own routing table, socket listing, connection tracking table, firewall, and other network-related resources.
  - Destroying a network namespace destroys any virtual interfaces within it and moves any physical interfaces within it back to the initial network namespace.
- Inter Process Communication(IPC) namespace
- User ID namespace
- Control group(cgroup) namespace
- Time namespace
- UTS(UNIX Time Sharing) namespace

Control Groups (cgroups)

- **cgroups** (abbreviated from **control groups**) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.
- One of the design goals of cgroups is to provide a unified interface to many different use cases, from controlling single processes (by using nice, for example) to full operating system-level virtualization (as provided by OpenVZ, Linux-VServer or LXC, for example).
- Cgroups provides:
  - **Resource limiting**
    - groups can be set to not exceed a configured memory limit, which also includes the file system cache
  - **Prioritization**
    - some groups may get a larger share of CPU utilization[11] or disk I/O throughput[12]
  - **Accounting**
    - measures a group's resource usage, which may be used, for example, for billing purposes[13]
  - **Control**
    - freezing groups of processes, their checkpointing and restarting[13]

Container Image

- A **Docker image** is an immutable (unchangeable) file that contains the source code, libraries, dependencies, tools, and other files needed for an application to run.

UnionFS

- **Unionfs** is a filesystem service for Linux, FreeBSD and NetBSD which implements a union mount for other file systems. It allows files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system. Contents of directories which have the same path within the merged branches will be seen together in a single merged directory, within the new, virtual filesystem.
- When mounting branches, the priority of one branch over the other is specified. So when both branches contain a file with the same name, one gets priority over the other.
- The different branches may be either *read-only* or *read/write* file systems, so that writes to the virtual, merged copy are directed to a specific real file system. This allows a file system to appear as writable, but without actually allowing writes to change the file system, also known as copy-on-write. This may be desirable when the media is physically read-only, such as in the case of Live CDs.

References

- https://www.ianlewis.org/en/container-runtimes-part-1-introduction-container-r
- https://jvns.ca/blog/2016/10/10/what-even-is-a-container/
- https://en.wikipedia.org/wiki/Linux_namespaces
- https://en.wikipedia.org/wiki/Cgroups
- https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction/
- https://github.com/opencontainers/runtime-spec/blob/master/spec.md
- https://github.com/opencontainers/image-spec/blob/master/spec.md
- https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/
- https://github.com/kubernetes/kubernetes/blob/242a97307b34076d5d8f5bbeb154fa4d97c9ef1d/docs/devel/container-runtime-interface.md
- https://www.jianshu.com/p/3ba255463047
- https://en.wikipedia.org/wiki/UnionFS
- https://unionfs.filesystems.org/
- https://phoenixnap.com/kb/docker-image-vs-container
- https://xuanwo.io/2019/08/06/oci-intro/

Client-go

- Go clients for talking to a kubernetes cluster.
- What's included
  - The kubernetes package contains the clientset to access Kubernetes API.
  - The discovery package is used to discover APIs supported by a Kubernetes API server.
  - The dynamic package contains a dynamic client that can perform generic operations on arbitrary Kubernetes API objects.
  - The plugin/pkg/client/auth packages contain optional authentication plugins for obtaining credentials from external sources.
  - The transport package is used to set up auth and start a connection.
  - The tools/cache package is useful for writing controllers.

REST Client

- This is a RESTful Client which supports json and protobuf and native resources and CRD resources.

Dynamic Client

- dynamicClient is a dynamic client, in fact, it encapsulated rest client. It supports dynamic configuration resource Group, version and which resource used.
- Example: https://github.com/kubernetes/client-go/tree/master/examples/dynamic-create-update-delete-deployment
- NOTE: This example is based on out of cluster env. If you want to use it in cluster env(such as you want to use it in pod container). You need use "rest.InClusterConfig()" to generate a configuration. More detailed example please reference https://github.com/kubernetes/client-go/tree/master/examples/in-cluster-client-configuration.

Typed ClientSet

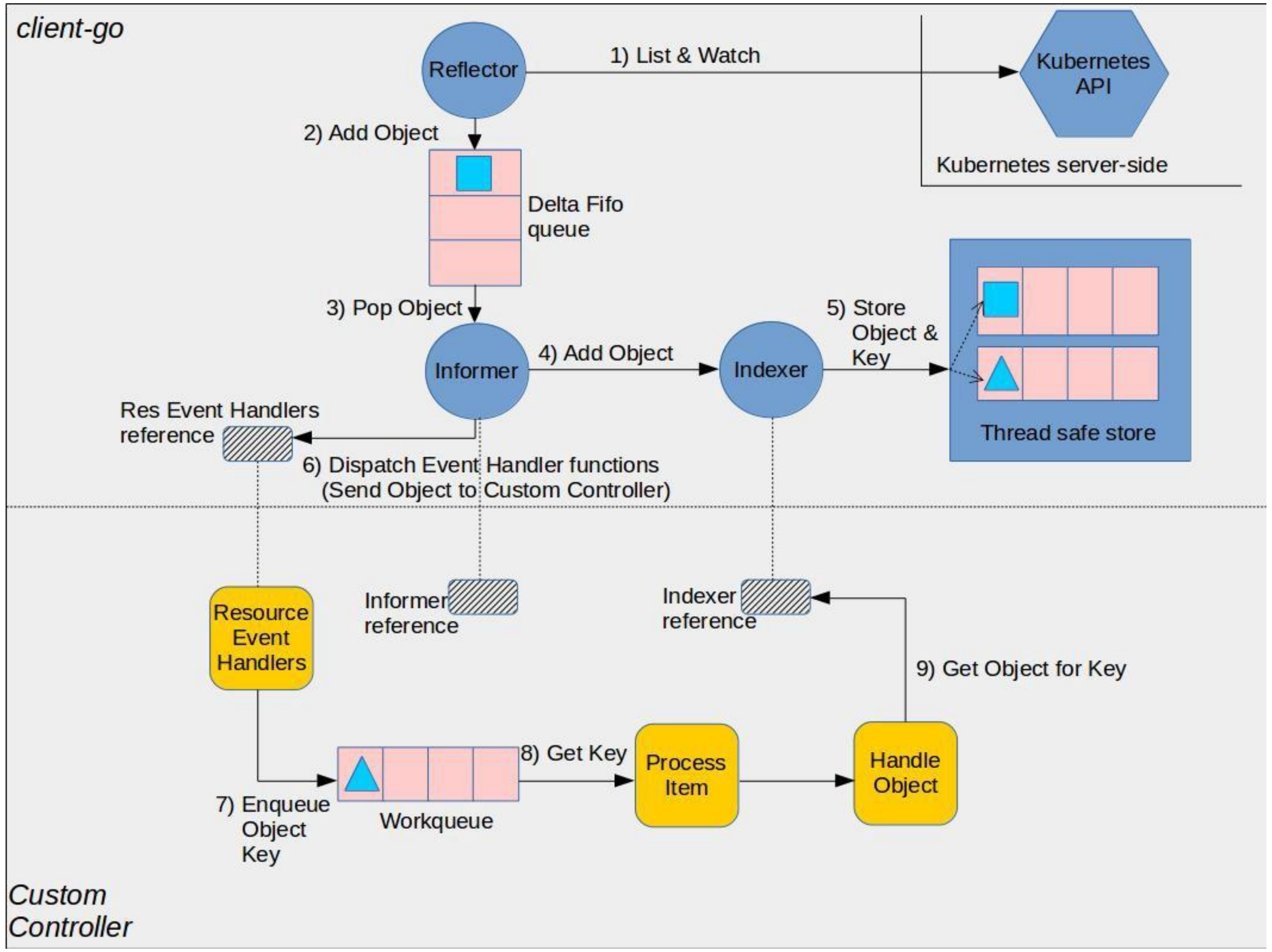
- Typed clientSet is a kubernetes client. It defined all resources API operations. So you can directly use it to operate resources of the all versions.
- Example: https://github.com/kubernetes/client-go/tree/master/examples/create-update-delete-deployment

Typed Vs Dynamic

- The typed client sets make it simple to communicate with the API server using pre-generated local API objects to achieve an RPC-like programming experience. Typed clients uses program compilations to enforce data safety and some validation. However, when using typed clients, programs are forced to be tightly coupled with the version and the types used.
- The **dynamic** package on the other hand, uses a simple type, **unstructured.Unstructured**, to represent all object values from the API server. Type **Unstructured** uses a collection of nested **map[string]inferface{}** values to create an internal structure that closely resamble the REST payload from the server.
- The dynamic package defers all data bindings until runtime. This means programs that use the dynamic client will not get any of the benefits of type validation until the program is running. This may be a problem for certain types of applications that require strong data type check and validation.
- Being loosely coupled, however, means that programs that uses the **dynamic** package do not require recompilation when the client API changes. The client program has more flexibility in handling updates to the API surface without knowing ahead of time what those changes are.

Informer

This is a pictorial representation showing how the various components in the client-go library work and their interaction points with the custom controller code that you will write.



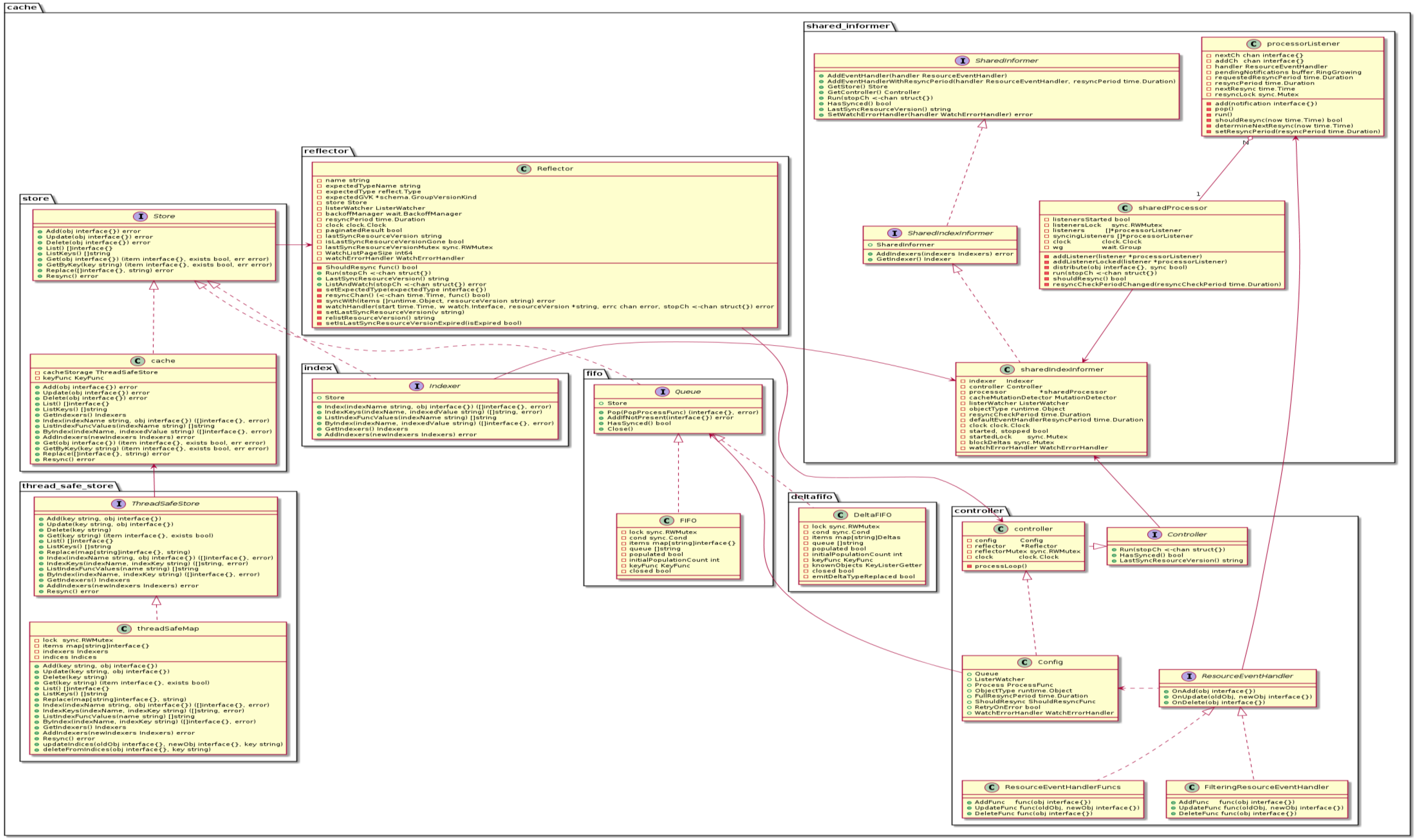
Informer Mechanism

- **client-go components**
  - **Reflector**: A reflector, which is defined in type *Reflector* inside package *cache*, watches the Kubernetes API for the specified resource type (kind). The function in which this is done is *ListAndWatch*. The watch could be for an in-built resource or it could be for a custom resource. When the reflector receives notification about existence of new resource instance through the watch API, it gets the newly created object using the corresponding listing API and puts it in the Delta Fifo queue inside the *watchHandler* function.
  - **Informer**: An informer defined in the base controller inside package *cache* pops objects from the Delta Fifo queue. The function in which this is done is *processLoop*. The job of this base controller is to save the object for later retrieval, and to invoke our controller passing it the object.
  - **Indexer**: An indexer provides indexing functionality over objects. It is defined in type *Indexer* inside package *cache*. A typical indexing use-case is to create an index based on object labels . Indexer can maintain indexes based on several indexing functions. Indexer uses a thread-safe data store to store objects and their keys. There is a default function named *MetaNamespaceKeyFunc* defined in type Store inside package cache that generates an object's key as <namespace>/<name> combination for that object.

Informer Mechanism

- **Custom Controller components**
  - **Informer reference**: This is the reference to the Informer instance that knows how to work with your custom resource objects. Your custom controller code needs to create the appropriate Informer.
  - **Indexer reference**: This is the reference to the Indexer instance that knows how to work with your custom resource objects. Your custom controller code needs to create this. You will be using this reference for retrieving objects for later processing.
  - **Resource Event Handlers**: These are the callback functions which will be called by the Informer when it wants to deliver an object to your controller. The typical pattern to write these functions is to obtain the dispatched object's key and enqueue that key in a work queue for further processing.
  - **Work queue**: This is the queue that you create in your controller code to decouple delivery of an object from its processing. Resource event handler functions are written to extract the delivered object's key and add that to the work queue.
  - **Process Item**: This is the function that you create in your code which processes items from the work queue. There can be one or more other functions that do the actual processing. These functions will typically use the Indexer reference, or a Listing wrapper to retrieve the object corresponding to the key.

Informer class diagram

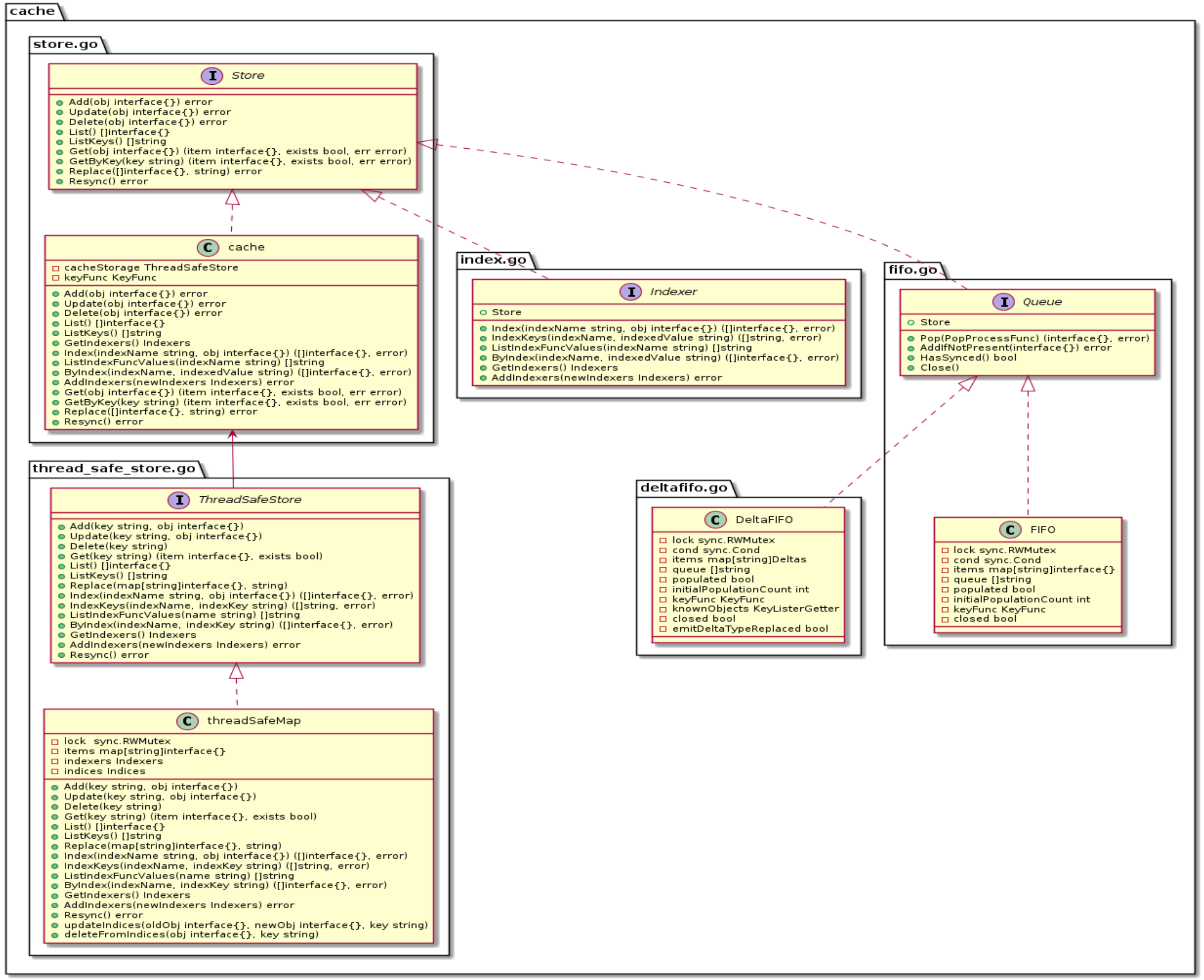

cache
shared_informer
SharedInformer
AddEventHandler(handler ResourceEventHandler)
AddEventHandlerWithResyncPeriod(handler ResourceEventHandler, resyncPeriod time.Duration)
GetStore() Store
GetController() Controller
Run(stopCh <-chan struct{})
HasSynced() bool
LastSyncResourceVersion() string
SetWatchErrorHandler(handler WatchErrorHandler) error
processorListener
nextCh chan interface{}
addCh chan interface{}
handler ResourceEventHandler
pendingNotifications buffer.RingGrowing
requestedResyncPeriod time.Duration
resyncPeriod time.Duration
nextResync time.Time
resyncLock sync.Mutex
add(notification interface{})
pop()
run()
shouldResync(now time.Time) bool
determineNextResync(now time.Time)
setResyncPeriod(resyncPeriod time.Duration)
N
1
SharedIndexInformer
SharedInformer
AddIndexers(indexers Indexers) error
GetIndexer() Indexer
sharedProcessor
listenersStarted bool
listenersLock sync.RWMutex
listeners []*processorListener
syncingListeners []*processorListener
clock clock.Clock
wg wait.Group
addListener(listener *processorListener)
addListenerLocked(listener *processorListener)
distribute(obj interface{}, sync bool)
run(stopCh <-chan struct{})
shouldResync() bool
resyncCheckPeriodChanged(resyncCheckPeriod time.Duration)
sharedIndexInformer
indexer Indexer
controller Controller
processor *sharedProcessor
cacheMutationDetector MutationDetector
listerWatcher ListerWatcher
objectType runtime.Object
resyncCheckPeriod time.Duration
defaultEventHandlerResyncPeriod time.Duration
clock clock.Clock
started, stopped bool
startedLock sync.Mutex
blockDeltas sync.Mutex
watchErrorHandler WatchErrorHandler
reflector
Reflector
name string
expectedTypeName string
expectedType reflect.Type
expectedGVK *schema.GroupVersionKind
store Store
listerWatcher ListerWatcher
backoffManager wait.BackoffManager
resyncPeriod time.Duration
clock clock.Clock
paginatedResult bool
lastSyncResourceVersion string
isLastSyncResourceVersionGone bool
lastSyncResourceVersionMutex sync.RWMutex
WatchListPageSize int64
watchErrorHandler WatchErrorHandler
ShouldResync func() bool
Run(stopCh <-chan struct{})
LastSyncResourceVersion() string
ListAndWatch(stopCh <-chan struct{}) error
setExpectedType(expectedType interface{})
resyncChan() (<-chan time.Time, func() bool)
syncWith(items []runtime.Object, resourceVersion string) error
watchHandler(start time.Time, w watch.Interface, resourceVersion *string, errc chan error, stopCh <-chan struct{}) error
setLastSyncResourceVersion(v string)
relistResourceVersion() string
setIsLastSyncResourceVersionExpired(isExpired bool)
store
Store
Add(obj interface{}) error
Update(obj interface{}) error
Delete(obj interface{}) error
List() []interface{}
ListKeys() []string
Get(obj interface{}) (item interface{}, exists bool, err error)
GetByKey(key string) (item interface{}, exists bool, err error)
Replace([]interface{}, string) error
Resync() error
cache
cacheStorage ThreadSafeStore
keyFunc KeyFunc
Add(obj interface{}) error
Update(obj interface{}) error
Delete(obj interface{}) error
List() []interface{}
ListKeys() []string
GetIndexers() Indexers
Index(indexName string, obj interface{}) ([]interface{}, error)
ListIndexFuncValues(indexName string) []string
ByIndex(indexName, indexedValue string) ([]interface{}, error)
AddIndexers(newIndexers Indexers) error
Get(obj interface{}) (item interface{}, exists bool, err error)
GetByKey(key string) (item interface{}, exists bool, err error)
Replace([]interface{}, string) error
Resync() error
index
Indexer
Store
Index(indexName string, obj interface{}) ([]interface{}, error)
IndexKeys(indexName, indexedValue string) ([]string, error)
ListIndexFuncValues(indexName string) []string
ByIndex(indexName, indexedValue string) ([]interface{}, error)
GetIndexers() Indexers
AddIndexers(newIndexers Indexers) error
fifo
Queue
Store
Pop(PopProcessFunc) (interface{}, error)
AddIfNotPresent(interface{}) error
HasSynced() bool
Close()
FIFO
lock sync.RWMutex
cond sync.Cond
items map[string]interface{}
queue []string
populated bool
initialPopulationCount int
keyFunc KeyFunc
closed bool
deltafifo
DeltaFIFO
lock sync.RWMutex
cond sync.Cond
items map[string]Deltas
queue []string
populated bool
initialPopulationCount int
keyFunc KeyFunc
knownObjects KeyListerGetter
closed bool
emitDeltaTypeReplaced bool
thread_safe_store
ThreadSafeStore
Add(key string, obj interface{})
Update(key string, obj interface{})
Delete(key string)
Get(key string) (item interface{}, exists bool)
List() []interface{}
ListKeys() []string
Replace(map[string]interface{}, string)
Index(indexName string, obj interface{}) ([]interface{}, error)
IndexKeys(indexName, indexKey string) ([]string, error)
ListIndexFuncValues(name string) []string
ByIndex(indexName, indexKey string) ([]interface{}, error)
GetIndexers() Indexers
AddIndexers(newIndexers Indexers) error
Resync() error
threadSafeMap
lock sync.RWMutex
items map[string]interface{}
indexers Indexers
indices Indices
Add(key string, obj interface{})
Update(key string, obj interface{})
Delete(key string)
Get(key string) (item interface{}, exists bool)
List() []interface{}
ListKeys() []string
Replace(map[string]interface{}, string)
Index(indexName string, obj interface{}) ([]interface{}, error)
IndexKeys(indexName, indexKey string) ([]string, error)
ListIndexFuncValues(name string) []string
ByIndex(indexName, indexKey string) ([]interface{}, error)
GetIndexers() Indexers
AddIndexers(newIndexers Indexers) error
Resync() error
updateIndices(oldObj interface{}, newObj interface{}, key string)
deleteFromIndices(obj interface{}, key string)
controller
controller
config Config
reflector *Reflector
reflectorMutex sync.RWMutex
clock clock.Clock
processLoop()
Controller
Run(stopCh <-chan struct{})
HasSynced() bool
LastSyncResourceVersion() string
Config
Queue
ListerWatcher
Process ProcessFunc
ObjectType runtime.Object
FullResyncPeriod time.Duration
ShouldResync ShouldResyncFunc
RetryOnError bool
WatchErrorHandler WatchErrorHandler
ResourceEventHandler
OnAdd(obj interface{})
OnUpdate(oldObj, newObj interface{})
OnDelete(obj interface{})
ResourceEventHandlerFuncs
AddFunc func(obj interface{})
UpdateFunc func(oldObj, newObj interface{})
DeleteFunc func(obj interface{})
FilteringResourceEventHandler
AddFunc func(obj interface{})
UpdateFunc func(oldObj, newObj interface{})
DeleteFunc func(obj interface{})

DeltaFIFO ReSync

- ReSync Indexer data into DeltaFIFO.
  - This mechanism can ensure events will not be lost, but it will trigger some redundancy events.
- Why:
  - Q1: controller has already implemented work queue mechanism, why need this resync operation?
  - A1: not all of controllers have implemented this work queue. And work queue has a limitation which is that it has a size restriction(default value is 100). So if controller didn't implement work queue or work queue overloaded due to too many failures. We need this mechanism to avoid event lost.
  - Q2: how to ensure resync will not overwrite event that it had already existed in DeltaFIFO?
  - A2: If the event has already existed in DeltaFIFO, then ignore this resync operation. If not, add this event into DeltaFIFO. Read source code https://github.com/kubernetes/client-go/blob/release-1.18/tools/cache/delta_fifo.go#L367
  - Q3: why resync operation will trigger an onUpdate event?
  - A3: read source code https://github.com/kubernetes/client-go/blob/release-1.18/tools/cache/shared_informer.go#L494 and https://github.com/kubernetes/client-go/blob/release-1.18/tools/cache/delta_fifo.go#L595
  - Q4: if a listener has not event in a long time, then this resync mechanism will always trigger some redundancy events?
  - A4: Yes.

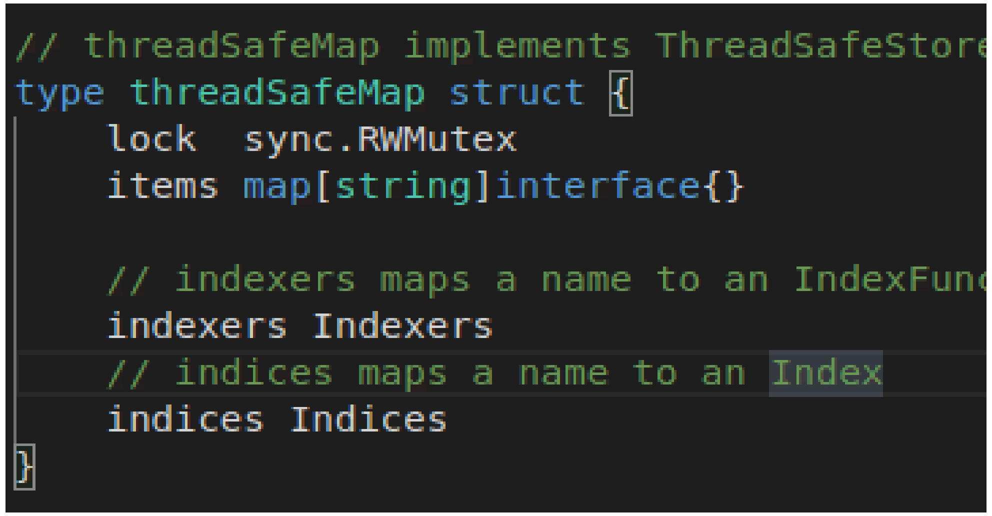
Indexer

- **Store** is a generic object storage and processing interface
- **Indexer** extends Store with multiple indices and restricts each accumulator to simply hold the current object (and be empty after Delete).
- **cache** implements Indexer in terms of a ThreadSafeStore and an associated KeyFunc.
- **ThreadSafeStore** is an interface that allows concurrent indexed access to a storage backend. It is like Indexer but does not (necessarily) know how to extract the Store key from a given object.
- **threadSafeMap** implements ThreadSafeStore


cache
store.go
Store
Add(obj interface{}) error
Update(obj interface{}) error
Delete(obj interface{}) error
List() []interface{}
ListKeys() []string
Get(obj interface{}) (item interface{}, exists bool, err error)
GetByKey(key string) (item interface{}, exists bool, err error)
Replace([]interface{}, string) error
Resync() error
cache
cacheStorage ThreadSafeStore
keyFunc KeyFunc
Add(obj interface{}) error
Update(obj interface{}) error
Delete(obj interface{}) error
List() []interface{}
ListKeys() []string
GetIndexers() Indexers
Index(indexName string, obj interface{}) ([]interface{}, error)
ListIndexFuncValues(indexName string) []string
ByIndex(indexName, indexedValue string) ([]interface{}, error)
AddIndexers(newIndexers Indexers) error
Get(obj interface{}) (item interface{}, exists bool, err error)
GetByKey(key string) (item interface{}, exists bool, err error)
Replace([]interface{}, string) error
Resync() error
index.go
Indexer
Store
Index(indexName string, obj interface{}) ([]interface{}, error)
IndexKeys(indexName, indexedValue string) ([]string, error)
ListIndexFuncValues(indexName string) []string
ByIndex(indexName, indexedValue string) ([]interface{}, error)
GetIndexers() Indexers
AddIndexers(newIndexers Indexers) error
fifo.go
Queue
Store
Pop(PopProcessFunc) (interface{}, error)
AddIfNotPresent(interface{}) error
HasSynced() bool
Close()
thread_safe_store.go
ThreadSafeStore
Add(key string, obj interface{})
Update(key string, obj interface{})
Delete(key string)
Get(key string) (item interface{}, exists bool)
List() []interface{}
ListKeys() []string
Replace(map[string]interface{}, string)
Index(indexName string, obj interface{}) ([]interface{}, error)
IndexKeys(indexName, indexKey string) ([]string, error)
ListIndexFuncValues(name string) []string
ByIndex(indexName, indexKey string) ([]interface{}, error)
GetIndexers() Indexers
AddIndexers(newIndexers Indexers) error
Resync() error
deltafifo.go
DeltaFIFO
lock sync.RWMutex
cond sync.Cond
items map[string]Deltas
queue []string
populated bool
initialPopulationCount int
keyFunc KeyFunc
knownObjects KeyListerGetter
closed bool
emitDeltaTypeReplaced bool
FIFO
lock sync.RWMutex
cond sync.Cond
items map[string]interface{}
queue []string
populated bool
initialPopulationCount int
keyFunc KeyFunc
closed bool
threadSafeMap
lock sync.RWMutex
items map[string]interface{}
indexers Indexers
indices Indices
Add(key string, obj interface{})
Update(key string, obj interface{})
Delete(key string)
Get(key string) (item interface{}, exists bool)
List() []interface{}
ListKeys() []string
Replace(map[string]interface{}, string)
Index(indexName string, obj interface{}) ([]interface{}, error)
IndexKeys(indexName, indexKey string) ([]string, error)
ListIndexFuncValues(name string) []string
ByIndex(indexName, indexKey string) ([]interface{}, error)
GetIndexers() Indexers
AddIndexers(newIndexers Indexers) error
Resync() error
updateIndices(oldObj interface{}, newObj interface{}, key string)
deleteFromIndices(obj interface{}, key string)

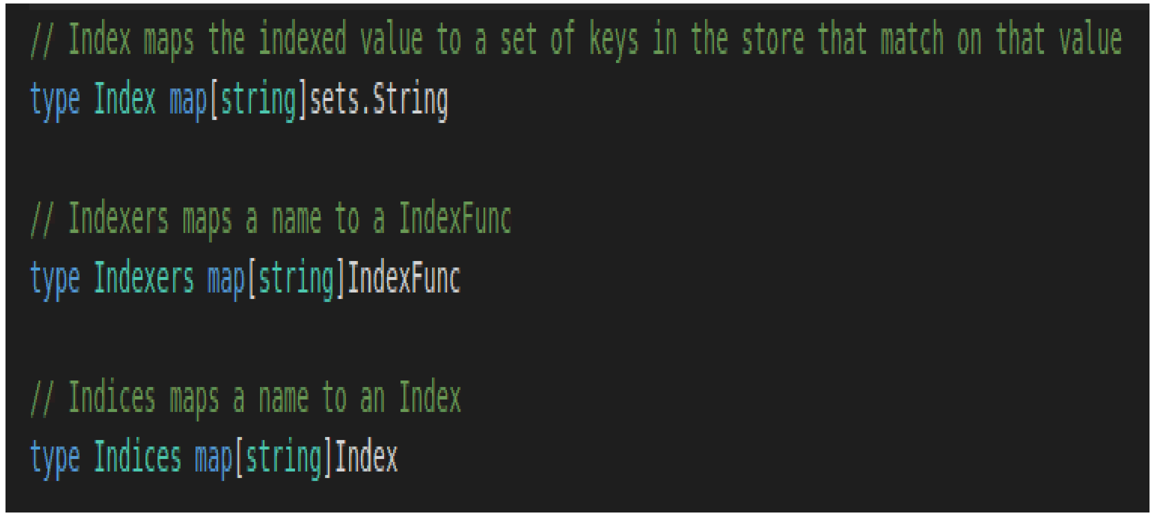
threadSafeMap

- **threadSafeMap** includes following attributes:
  1. **items** which is saved all of data
  2. **indexers** which is mapping a name to an IndexFunc
  3. **indices** which is mapping a name to an Index.



```
// threadSafeMap implements ThreadSafeStore
type threadSafeMap struct {
    lock  sync.RWMutex
    items map[string]interface{}

    // indexers maps a name to an IndexFunc
    indexers Indexers
    // indices maps a name to an Index
    indices Indices
}
```

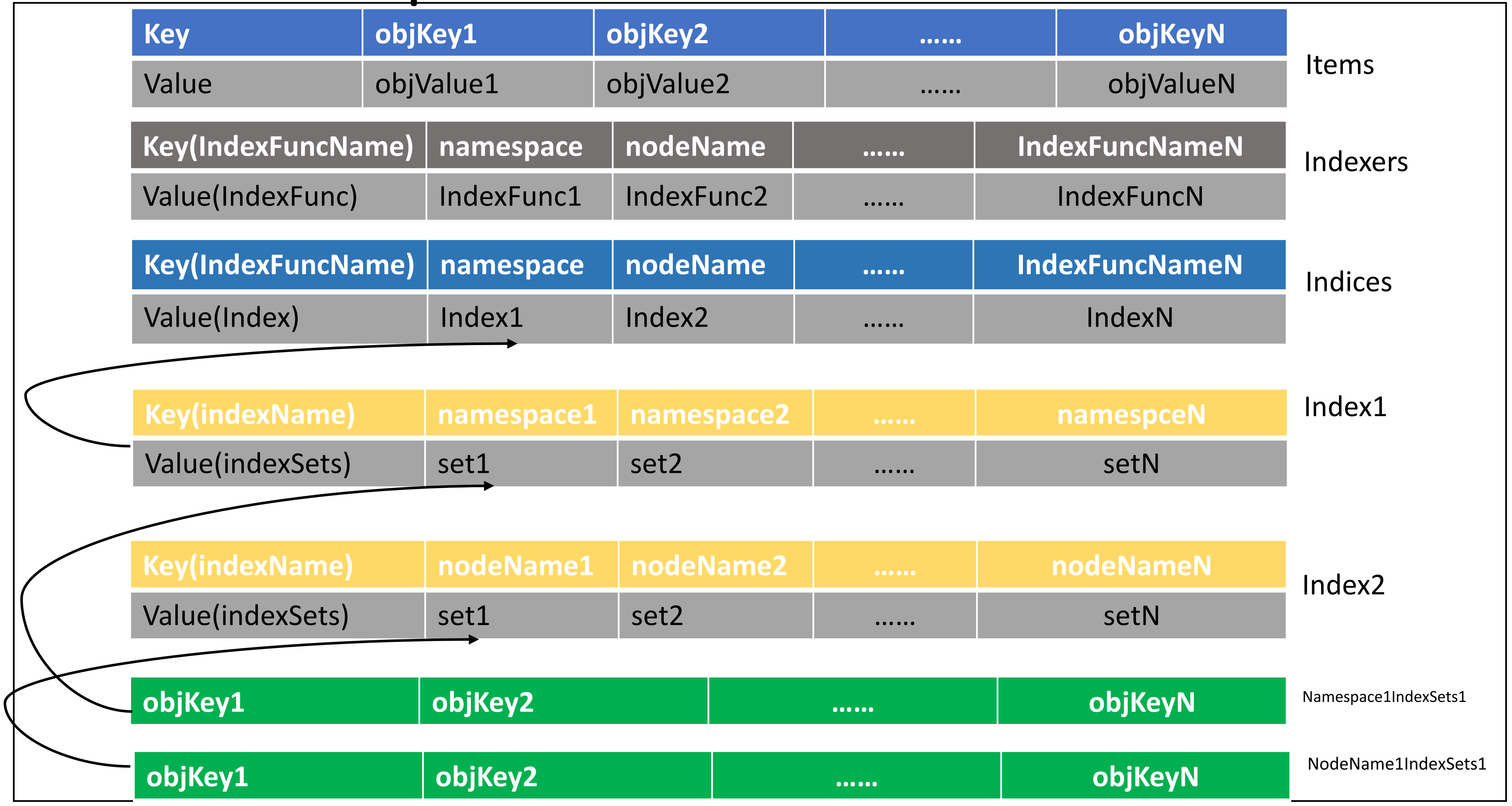


```
// Index maps the indexed value to a set of keys in the store that match on that value
type Index map[string]sets.String

// Indexers maps a name to a IndexFunc
type Indexers map[string]IndexFunc

// Indices maps a name to an Index
type Indices map[string]Index
```

threadSafeMap



Items

Key	objKey1	objKey2	……	objKeyN
Value	objValue1	objValue2	……	objValueN

Indexers

Key(IndexFuncName)	namespace	nodeName	……	IndexFuncNameN
Value(IndexFunc)	IndexFunc1	IndexFunc2	……	IndexFuncN

Indices

Key(IndexFuncName)	namespace	nodeName	……	IndexFuncNameN
Value(Index)	Index1	Index2	……	IndexN

Index1

Key(indexName)	namespace1	namespace2	……	namespceN
Value(indexSets)	set1	set2	……	setN

Index2

Key(indexName)	nodeName1	nodeName2	……	nodeNameN
Value(indexSets)	set1	set2	……	setN

Namespace1IndexSets1

objKey1	objKey2	……	objKeyN

NodeName1IndexSets1

objKey1	objKey2	……	objKeyN

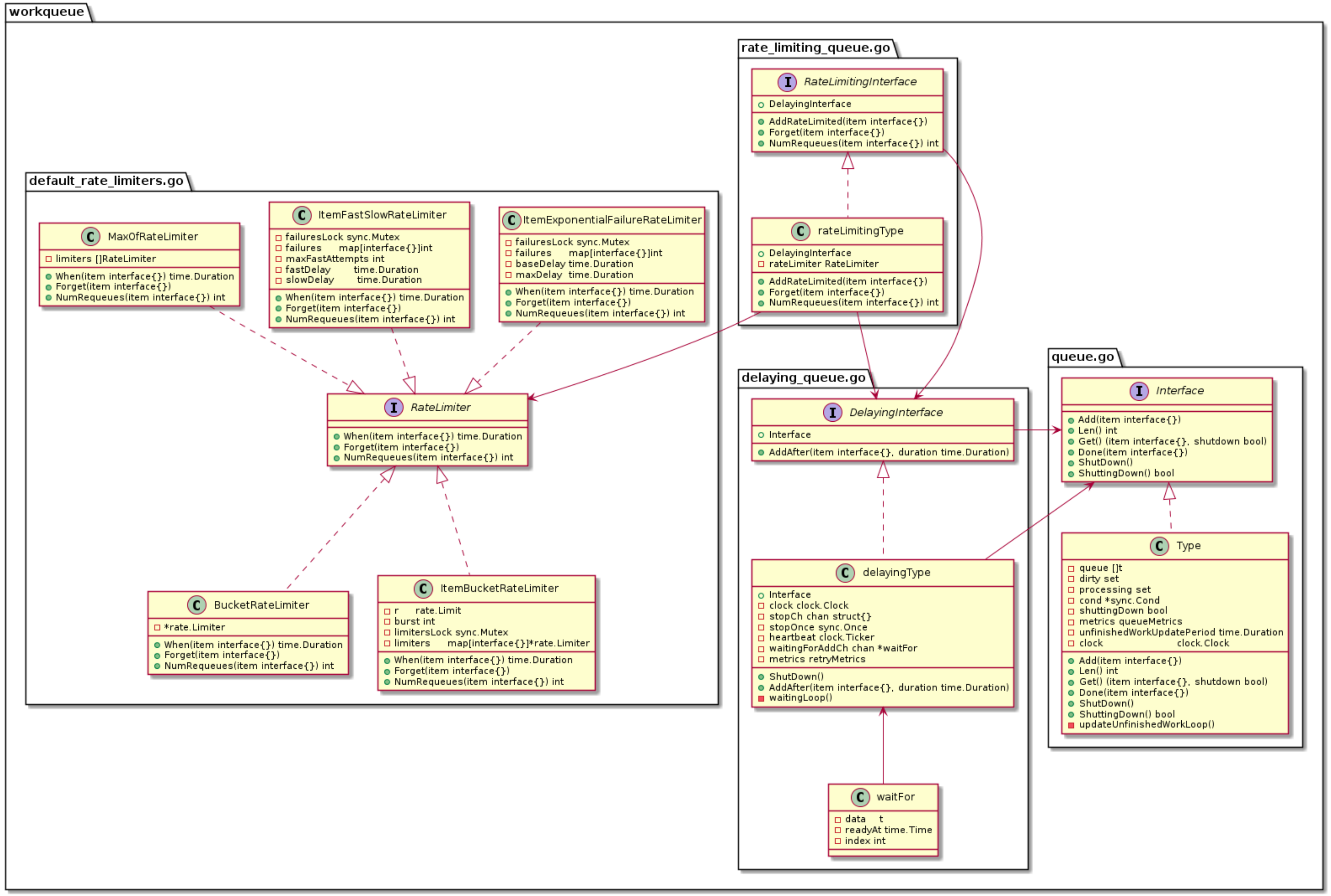
threadSafeMap

Example: Add a newObj into map

1. Add a newObj into items that specified key
2. Update this new key into indices
    1. Delete oldObj from indices if it exists.
    2. Iterate indexers to find indexValues of the newObj.
    3. Use indexer name to find index from indices.
    4. Create a new index if index is not existed. Otherwise next step.
    5. Iterate indexValues to find set from index.
    6. Create a new set if set is not existed.
    7. Add the new specified key into set.

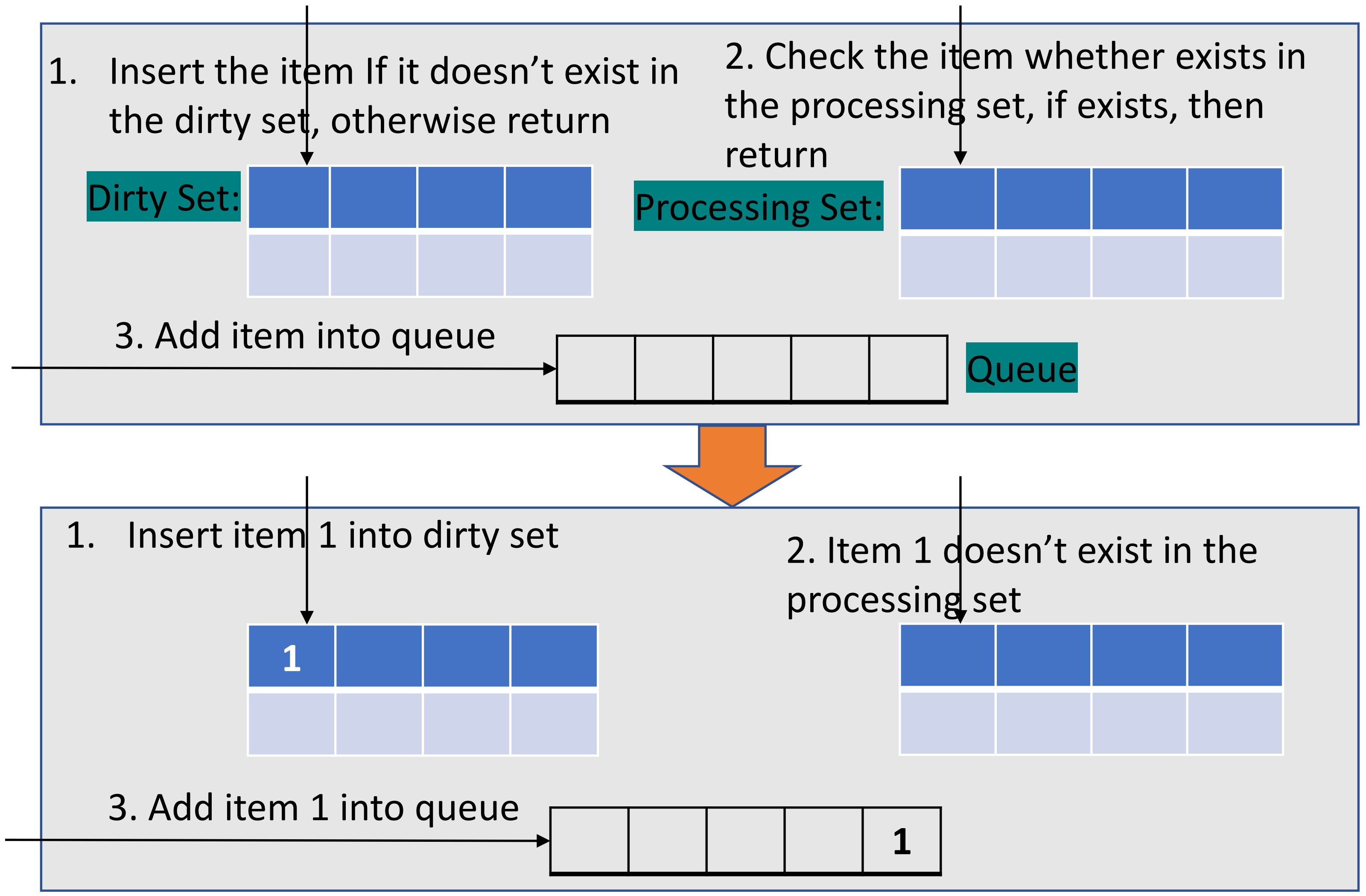
WorkQueue

- **Interface** . A FIFO queue interface , supports deduplication.
- **DelayingInterface.** A delay queue interface, encapsulated based on **Interface** interface.
- **RateLimitingInterface**. A rate limiting interface, encapsulated based on **DelayingInterface .**



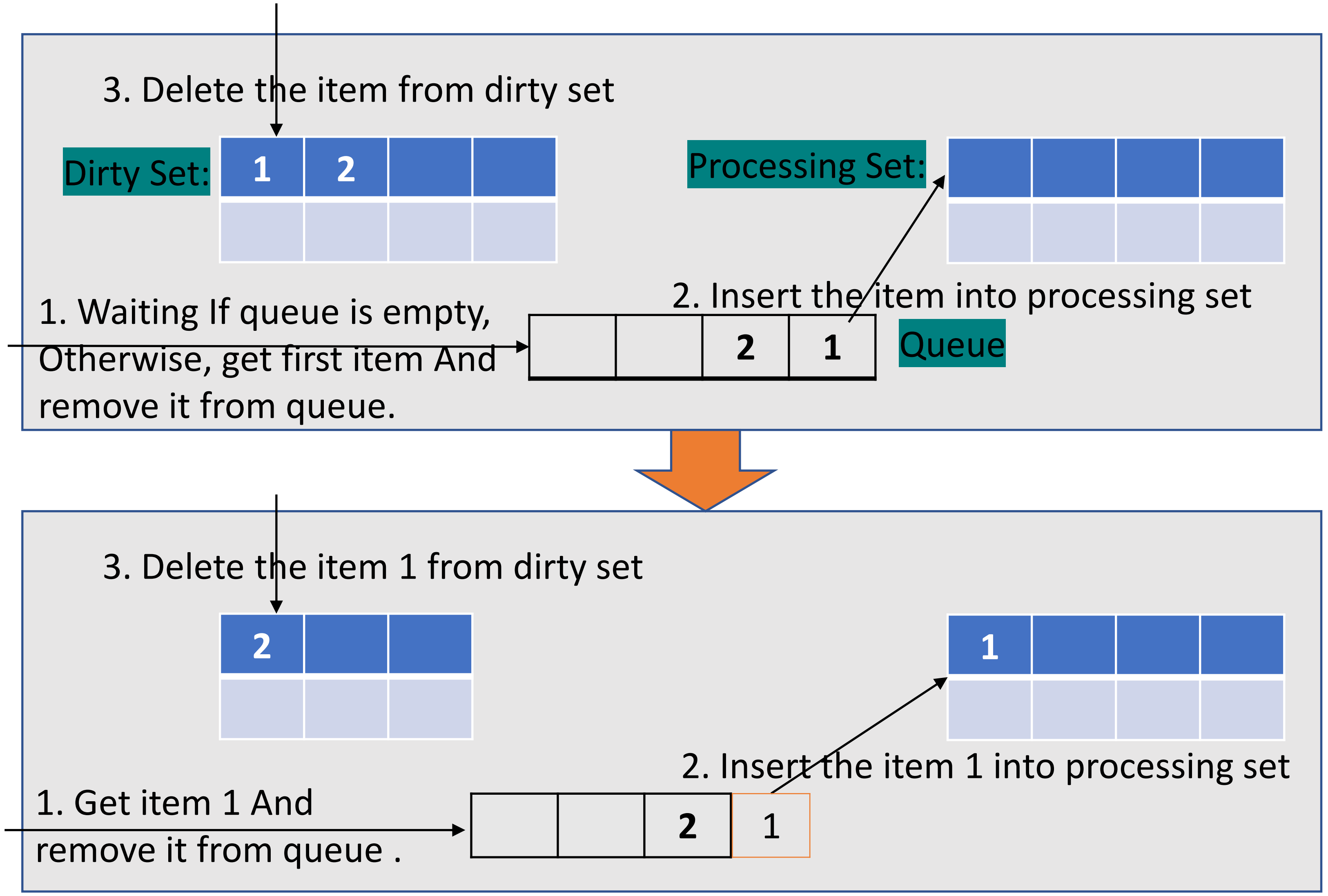
Basic Queue

- **Add** an item handling.
- **Add** marks item as needing processing.


1. Insert the item If it doesn't exist in the dirty set, otherwise return
2. Check the item whether exists in the processing set, if exists, then return
Dirty Set:
Processing Set:
3. Add item into queue
Queue
1. Insert item 1 into dirty set
2. Item 1 doesn't exist in the processing set
1
3. Add item 1 into queue
1

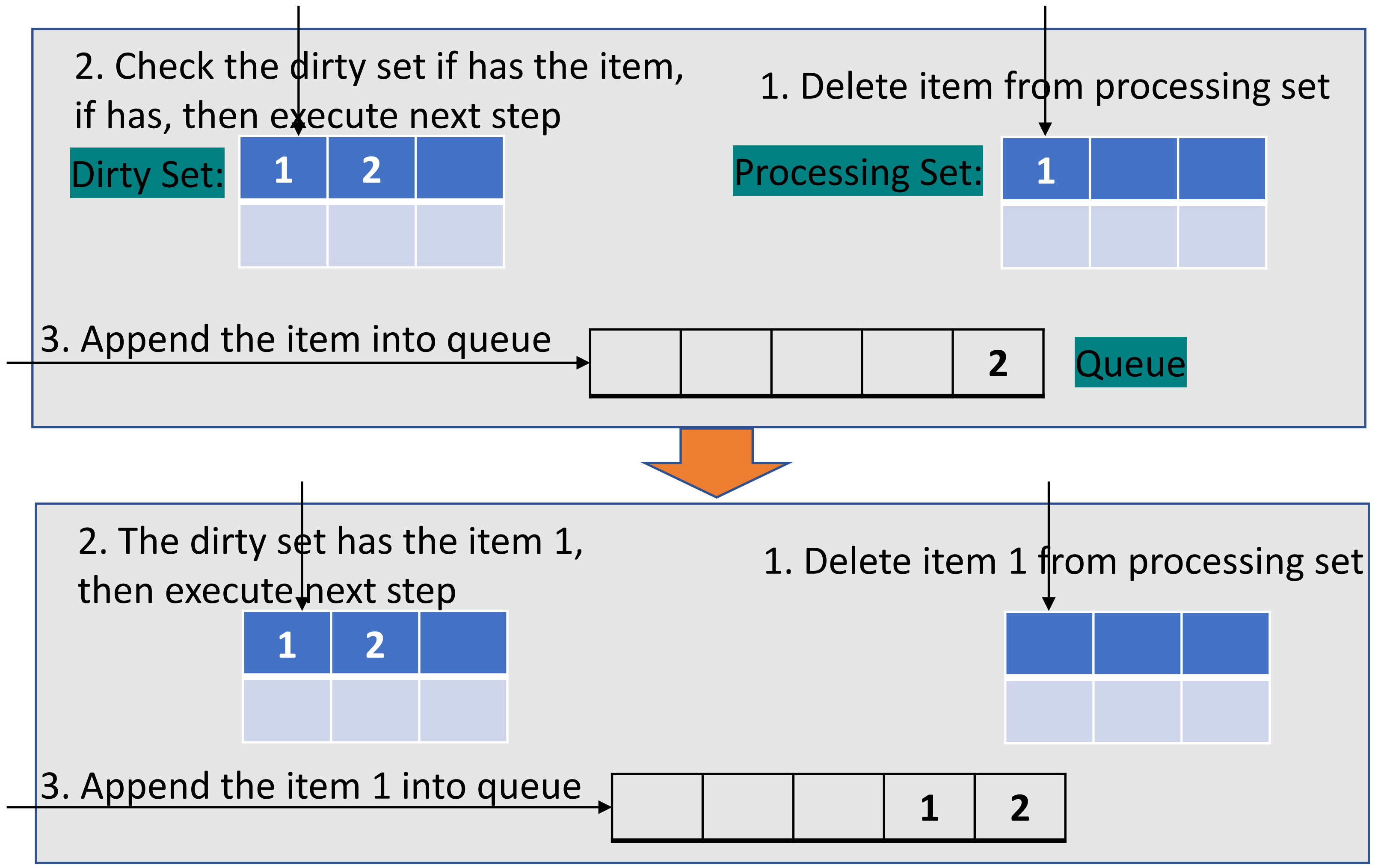
Basic Queue

- **Get** an item handling.
- **Get** blocks until it can return an item to be processed. If shutdown = true, the caller should end their goroutine. You must call Done with item when you have finished processing it.


3. Delete the item from dirty set
Dirty Set:
1
2
Processing Set:
1. Waiting If queue is empty, Otherwise, get first item And remove it from queue.
2. Insert the item into processing set
2
1
Queue
3. Delete the item 1 from dirty set
2
1
2. Insert the item 1 into processing set
1. Get item 1 And remove it from queue .
2
1

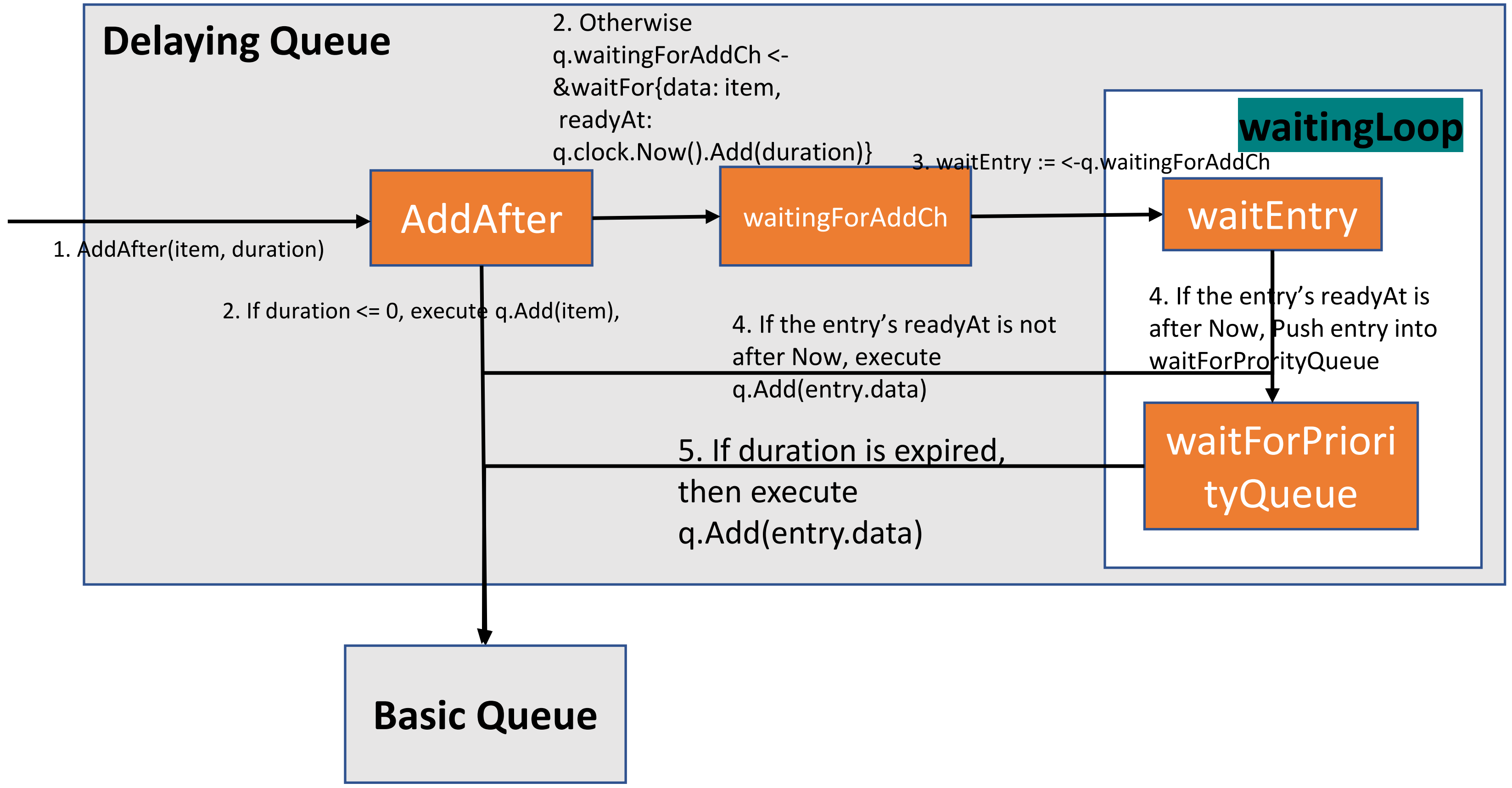
Basic Queue

- **Done** an item handling.
- **Done** marks item as done processing, and if it has been marked as dirty again while it was being processed, it will be re-added to the queue for re-processing.


2. Check the dirty set if has the item, if has, then execute next step
1. Delete item from processing set
Dirty Set:
1
2
Processing Set:
1
3. Append the item into queue
2
Queue
2. The dirty set has the item 1, then execute next step
1. Delete item 1 from processing set
1
2
3. Append the item 1 into queue
1
2

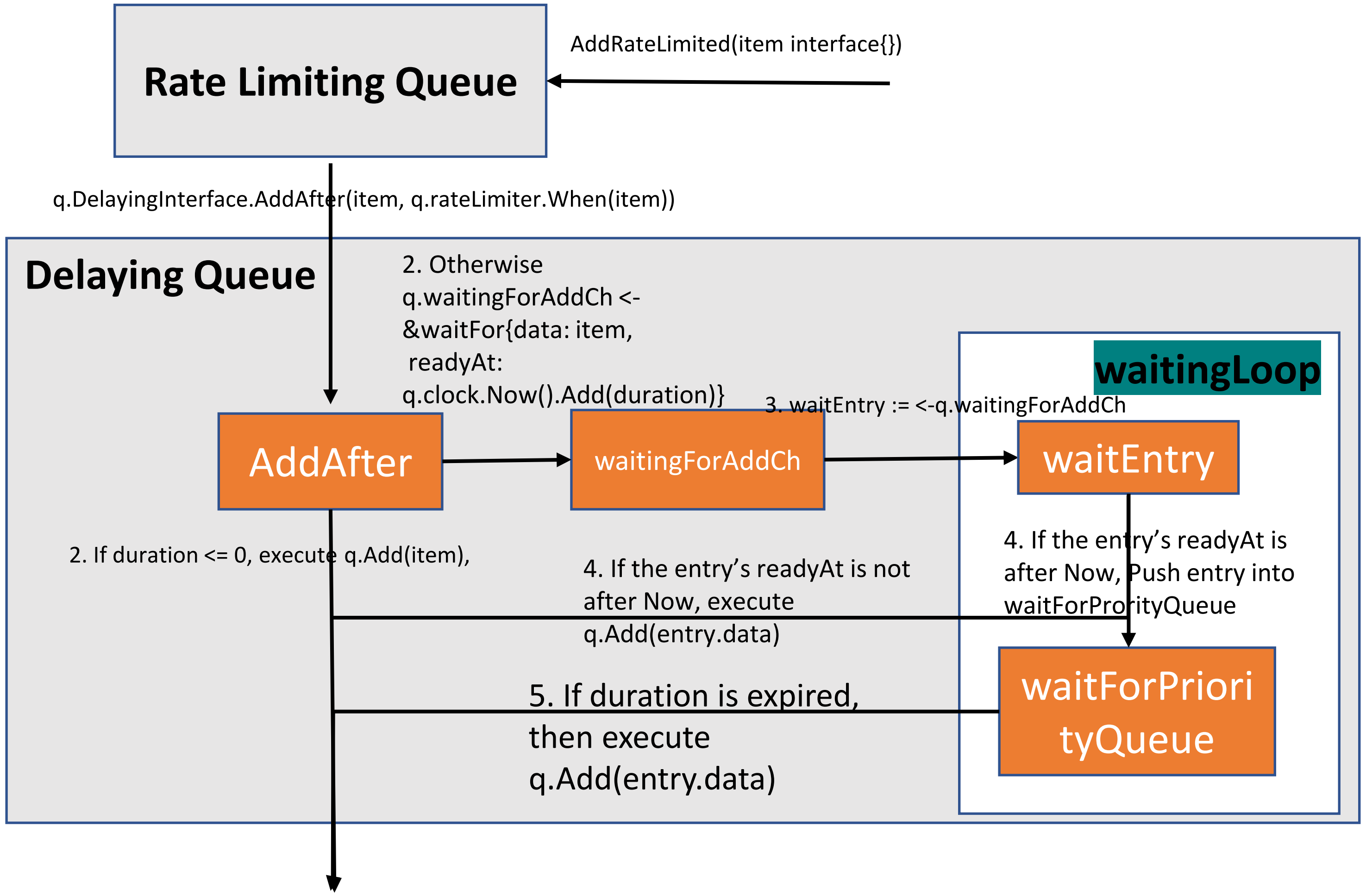
Delaying Queue

- **DelayingInterface** is an Interface that can Add an item at a later time. This makes it easier to requeue items after failures without ending up in a hot-loop.


Delaying Queue
1. AddAfter(item, duration)
AddAfter
2. Otherwise q.waitingForAddCh <- &waitFor{data: item, readyAt: q.clock.Now().Add(duration)}
waitingForAddCh
3. waitEntry := <-q.waitingForAddCh
waitingLoop
waitEntry
2. If duration <= 0, execute q.Add(item),
4. If the entry's readyAt is not after Now, execute q.Add(entry.data)
4. If the entry's readyAt is after Now, Push entry into waitForProrityQueue
waitForPriorityQueue
5. If duration is expired, then execute q.Add(entry.data)
Basic Queue

Rate limiting Queue

- **RateLimitingInterface** is an interface that rate limits items being added to the queue.
- Rate limiting queue is an interface, so if you don't want using default queue, then you can define a custom rate limiting queue.
- Default rate limiting queue:
    - BucketRateLimiter
    - ItemBucketRateLimiter
    - ItemExponentialFailureRateLimiter
    - ItemFastSlowRateLimiter
    - MaxOfRateLimiter

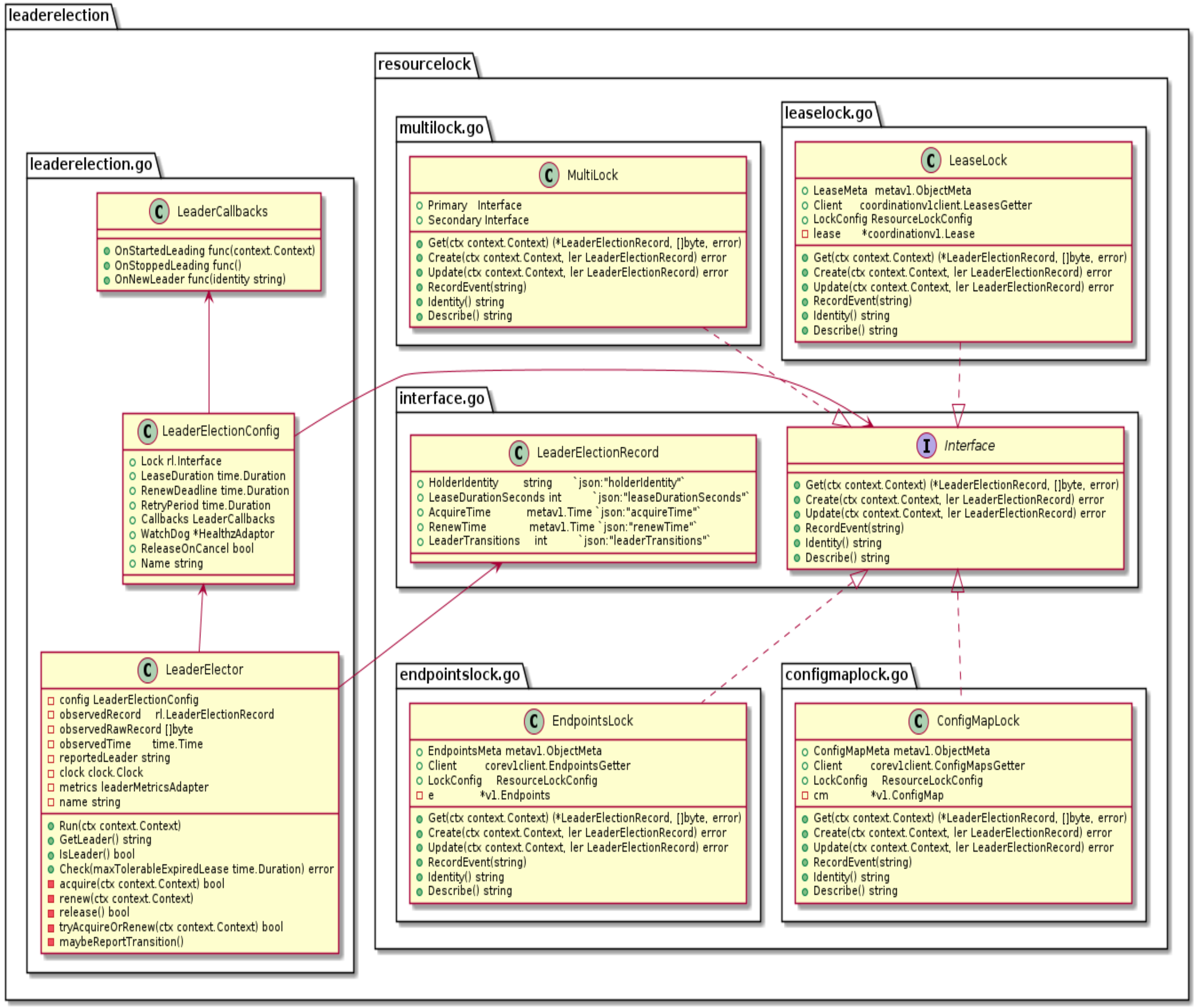

Rate Limiting Queue
AddRateLimited(item interface{})
q.DelayingInterface.AddAfter(item, q.rateLimiter.When(item))
Delaying Queue
2. Otherwise q.waitingForAddCh <- &waitFor{data: item, readyAt: q.clock.Now().Add(duration)}
waitingLoop
3. waitEntry := <-q.waitingForAddCh
AddAfter
waitingForAddCh
waitEntry
2. If duration <= 0, execute q.Add(item),
4. If the entry's readyAt is not after Now, execute q.Add(entry.data)
4. If the entry's readyAt is after Now, Push entry into waitForProrityQueue
5. If duration is expired, then execute q.Add(entry.data)
waitForPriorityQueue

Informer example

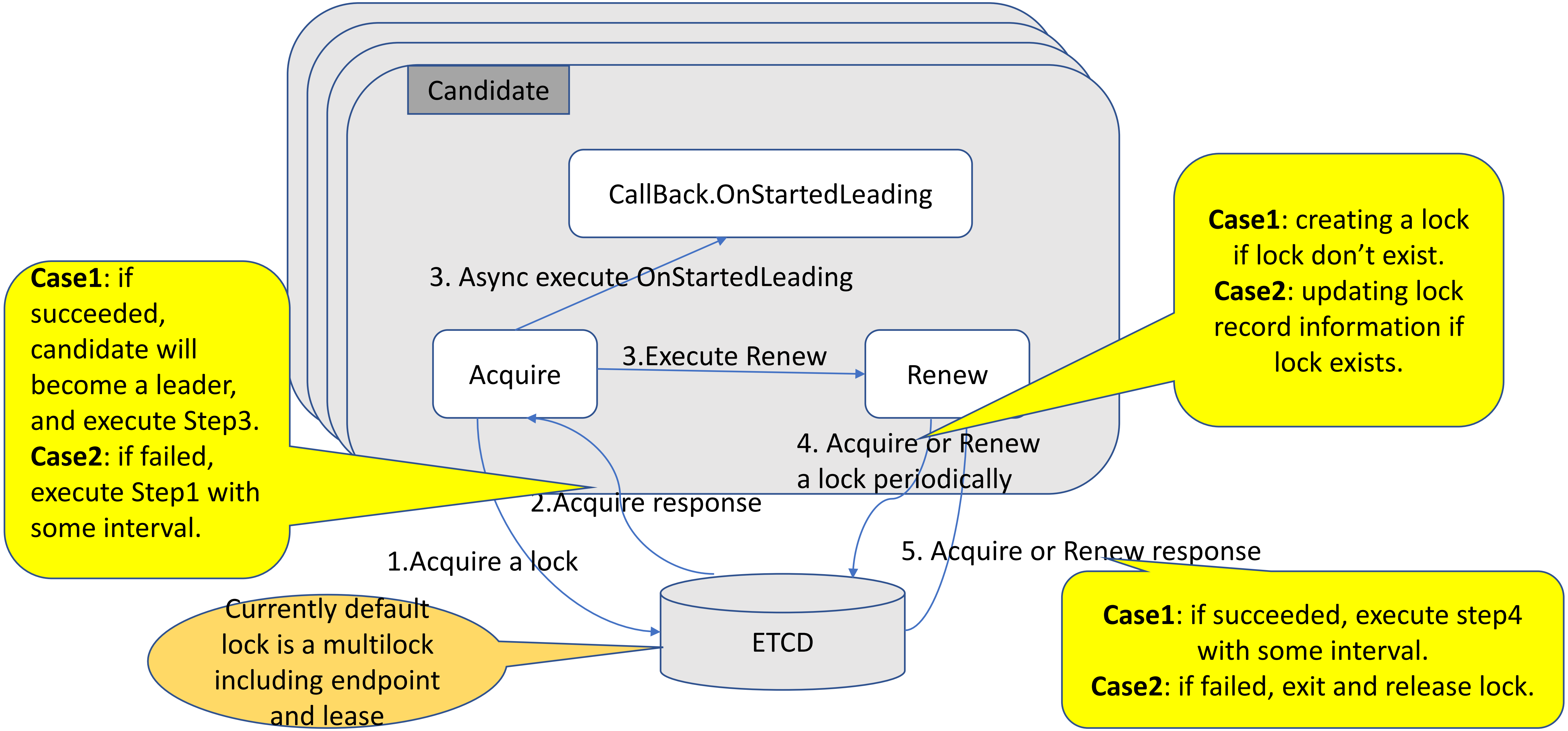
- You can reference k8s official guide(Kubernetes/sample-controller https://github.com/kubernetes/sample-controller)
- You can also reference client-go official example(workqueue https://github.com/kubernetes/client-go/tree/master/examples/workqueue). This is a typically example use client-go informer, it is fully based on this client-go informer architecture. And almost all of K8S controllers are implemented based on this architecture.

Leader Election

- leaderelection implements leader election of a set of endpoints. It uses an annotation in the endpoints object to store the record of the election state. This implementation does not guarantee that only one client is acting as a leader (a.k.a. fencing).


leaderelection
resourcelock
leaderelection.go
LeaderCallbacks
OnStartedLeading func(context.Context)
OnStoppedLeading func()
OnNewLeader func(identity string)
LeaderElectionConfig
Lock rl.Interface
LeaseDuration time.Duration
RenewDeadline time.Duration
RetryPeriod time.Duration
Callbacks LeaderCallbacks
WatchDog *HealthzAdaptor
ReleaseOnCancel bool
Name string
LeaderElector
config LeaderElectionConfig
observedRecord rl.LeaderElectionRecord
observedRawRecord []byte
observedTime time.Time
reportedLeader string
clock clock.Clock
metrics leaderMetricsAdapter
name string
Run(ctx context.Context)
GetLeader() string
IsLeader() bool
Check(maxTolerableExpiredLease time.Duration) error
acquire(ctx context.Context) bool
renew(ctx context.Context)
release() bool
tryAcquireOrRenew(ctx context.Context) bool
maybeReportTransition()
multilock.go
MultiLock
Primary Interface
Secondary Interface
Get(ctx context.Context) (*LeaderElectionRecord, []byte, error)
Create(ctx context.Context, ler LeaderElectionRecord) error
Update(ctx context.Context, ler LeaderElectionRecord) error
RecordEvent(string)
Identity() string
Describe() string
leaselock.go
LeaseLock
LeaseMeta metav1.ObjectMeta
Client coordinationv1client.LeasesGetter
LockConfig ResourceLockConfig
lease *coordinationv1.Lease
Get(ctx context.Context) (*LeaderElectionRecord, []byte, error)
Create(ctx context.Context, ler LeaderElectionRecord) error
Update(ctx context.Context, ler LeaderElectionRecord) error
RecordEvent(string)
Identity() string
Describe() string
interface.go
LeaderElectionRecord
HolderIdentity string `json:"holderIdentity"`
LeaseDurationSeconds int `json:"leaseDurationSeconds"`
AcquireTime metav1.Time `json:"acquireTime"`
RenewTime metav1.Time `json:"renewTime"`
LeaderTransitions int `json:"leaderTransitions"`
Interface
Get(ctx context.Context) (*LeaderElectionRecord, []byte, error)
Create(ctx context.Context, ler LeaderElectionRecord) error
Update(ctx context.Context, ler LeaderElectionRecord) error
RecordEvent(string)
Identity() string
Describe() string
endpointslock.go
EndpointsLock
EndpointsMeta metav1.ObjectMeta
Client corev1client.EndpointsGetter
LockConfig ResourceLockConfig
e *v1.Endpoints
Get(ctx context.Context) (*LeaderElectionRecord, []byte, error)
Create(ctx context.Context, ler LeaderElectionRecord) error
Update(ctx context.Context, ler LeaderElectionRecord) error
RecordEvent(string)
Identity() string
Describe() string
configmaplock.go
ConfigMapLock
ConfigMapMeta metav1.ObjectMeta
Client corev1client.ConfigMapsGetter
LockConfig ResourceLockConfig
cm *v1.ConfigMap
Get(ctx context.Context) (*LeaderElectionRecord, []byte, error)
Create(ctx context.Context, ler LeaderElectionRecord) error
Update(ctx context.Context, ler LeaderElectionRecord) error
RecordEvent(string)
Identity() string
Describe() string

Leader Election


Candidate
CallBack.OnStartedLeading
3. Async execute OnStartedLeading
Acquire
3.Execute Renew
Renew
4. Acquire or Renew a lock periodically
2.Acquire response
1.Acquire a lock
5. Acquire or Renew response
ETCD
Case1: if succeeded, candidate will become a leader, and execute Step3. **Case2**: if failed, execute Step1 with some interval.
Case1: creating a lock if lock don't exist. **Case2**: updating lock record information if lock exists.
Case1: if succeeded, execute step4 with some interval. **Case2**: if failed, exit and release lock.
Currently default lock is a multilock including endpoint and lease

Leader Election

- Currently kube-controller-manager is using “endpointlease” resource lock as a default value. So it will have two resource locks, the primary one is endpoint lock, the secondary one is lease lock.
- Q1: why introduce multilock?
- Q2: how to ensure one client is acting as a leader?

References

- Source code: https://github.com/kubernetes/client-go
- Article :
  - https://github.com/opsnull/kubernetes-dev-docs/tree/master/client-go
  - https://tangxusc.github.io/blog/2019/05/client-go-informer%E6%9C%BA%E5%88%B6/
  - https://www.jianshu.com/p/76e7b1a57d2c
- Informer analysis:
  - https://xigang.github.io/2019/09/21/client-go/.
  - http://www.programmersought.com/article/6135240470/.
  - https://www.huweihuang.com/kubernetes-notes/code-analysis/kube-controller-manager/sharedIndexInformer.html
- Sample-controller : https://github.com/kubernetes/sample-controller/blob/master/docs/controller-client-go.md
- Indexer https://studygolang.com/articles/20402
- Leader election:
- https://medium.com/michaelbi-22303/deep-dive-into-kubernetes-simple-leader-election-3712a8be3a99
- https://kubernetes.io/blog/2016/01/simple-leader-election-with-kubernetes/

Controller-runtime

- The Kubernetes controller-runtime Project is a set of go libraries for building Controllers. It is leveraged by Kubebuilder and Operator SDK.
- Controller
- webhook

References

- https://github.com/kubernetes-sigs/controller-runtime
- https://godoc.org/github.com/kubernetes-sigs/controller-runtime/pkg

Admission webhook

- Admission webhooks are HTTP callbacks that receive admission requests and do something with them. You can define two types of admission webhooks, validating admission webhook and mutating admission webhook. Mutating admission webhooks are invoked first, and can modify objects sent to the API server to enforce custom defaults. After all object modifications are complete, and after the incoming object is validated by the API server, validating admission webhooks are invoked and can reject requests to enforce custom policies.

References

- https://www.qikqiak.com/post/k8s-admission-webhook/
- https://kubernetes.io/blog/2019/03/21/a-guide-to-kubernetes-admission-controllers/

Thanks!