API Priority And Fairness: Kube-APIServer Flow-control Protection

Min Jin, Ant Group

Topic

Kubernetes Feature-Gate: “APIPriorityAndFairness” (Since 1.18)

Kubernetes Blog: API Priority And Fairness Alpha

https://kubernetes.io/blog/2020/04/06/kubernetes-1-18-feature-api-priority-and-fairness-alpha/

Design / KEP:

https://github.com/kubernetes/enhancements/tree/master/keps/sig-api-machinery/1040-priority-and-fairness

About Me

Min Jin

Software Engineer, Ant Group

Kubernetes sub-project owners (apiserver-builder, apiserver-runtime, Java client library, etc.). SIG API-Machinery working member for about 3 years.

yue9944882

yue9944882@gmail.com

Team

Mike Spreitzer
MikeSpreitzer

@IBM

Daniel Smith
lavalamp

@Google

David Eads
deads2k

@RedHat

And the contributors so far: Aaron Prindle, Jonathan Tomer, Bruce Ma, Yu Liao, Mengyi Zhou!

Summary

- Background and motivation for kicking off this feature (8min)
- System design retrospection (8min)
- Introduction to our alpha stage implementation (4min)
- DEMO: Customize flow-control settings for your cluster (2min)
- Planned enhancements for beta stage (3min)

(Total Presentation ~25min)

Background and Motivation

Higher Goal

- **Self-Protection**

  - Prioritize cluster-critical requests for self-maintenance

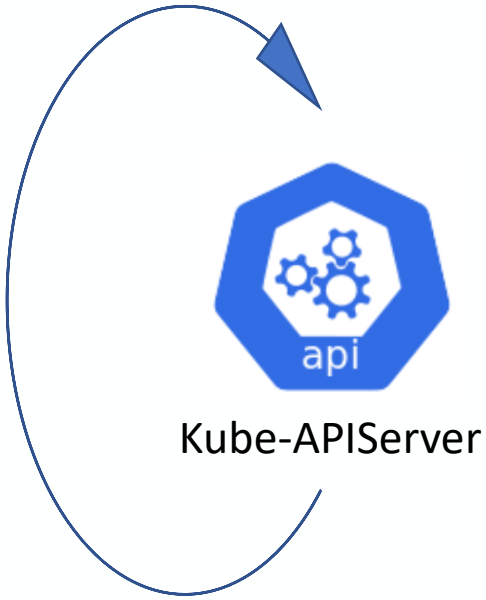
  - Prevent spammy clients or buggy controllers stunning the whole cluster

Self-Protection

In terms of “Protection”, we’re actually protecting Kube-APIServer from incoming requests. So we should start by understanding the different kinds of traffic being served by a typical Kube-APIServer:

- APIServer loopbacks
- Delegated requests from aggregated apiserver or admission webhooks
- Controllers: Deployment of Doom
- Daemons: Kubelet Amuck

Kube-APIServer Loopback



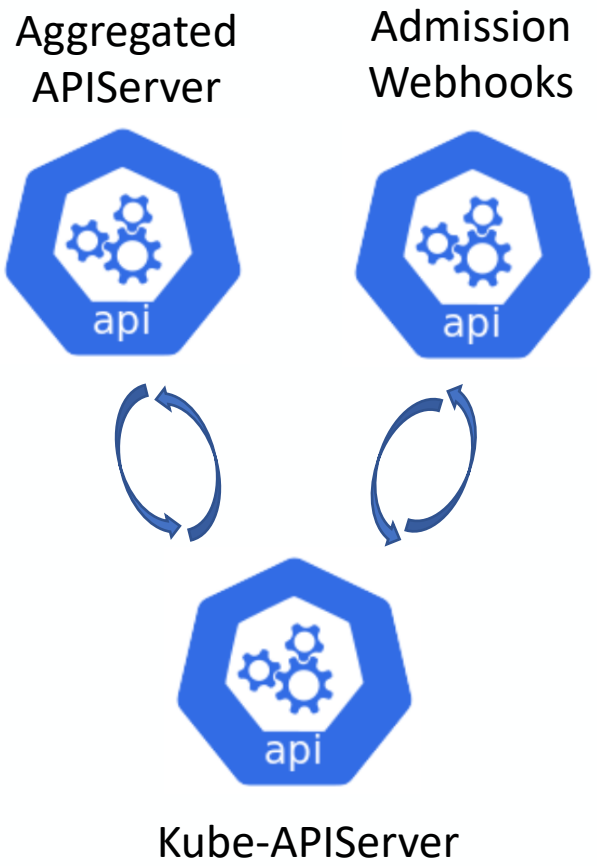
Kube-APIServer will be requesting against itself even if there're no client requests at all, such as:

- **Informer Factory**: Kube-APIServer acquires the status of the cluster by accessing the object cache provided by the informer factory. These informers will keep raising LIST/WATCH requests until the Kube-APIServer is down.

- **Embeded Controllers in Kube-APIServer**:
  - Cluster CA rotater
  - CRD related controllers
  - APIService (APIServer Aggregation) related controllers

All these requests are called "loopback" because they're raised and served within Kube-APIServer instance, and they are supposed to be "first-class citizens" in the kubernetes world because they are highly connected with the healthy status of the whole cluster.

Priority Inversion



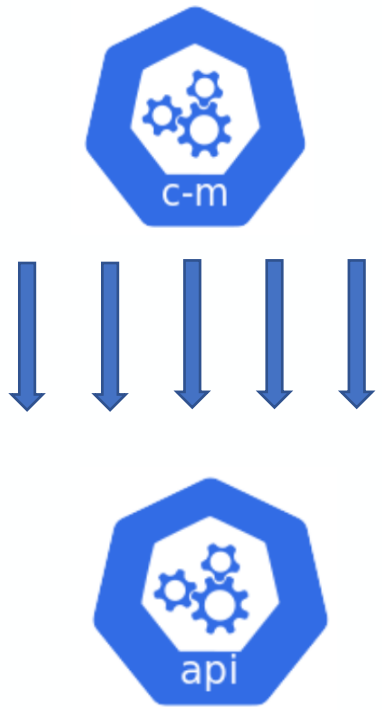
You're free to customzie your cluster by the extensibilities provided by Kube-APIServer. These extensions will be invoked during the time when the Kube-APIServer executing/serving an incoming request, and they can also spawn new requests back to the Kube-APIServer which results in **cyclic dependency** in the request chain.

- **APIService (Aggregated APIServer)**

- **{Mutating,Validating}WebhookConfiguration**

Ideally the spawned/child requests should always have a higher priority than its parent to avoid deadlocks in the requests chain.

Deployment of Doom



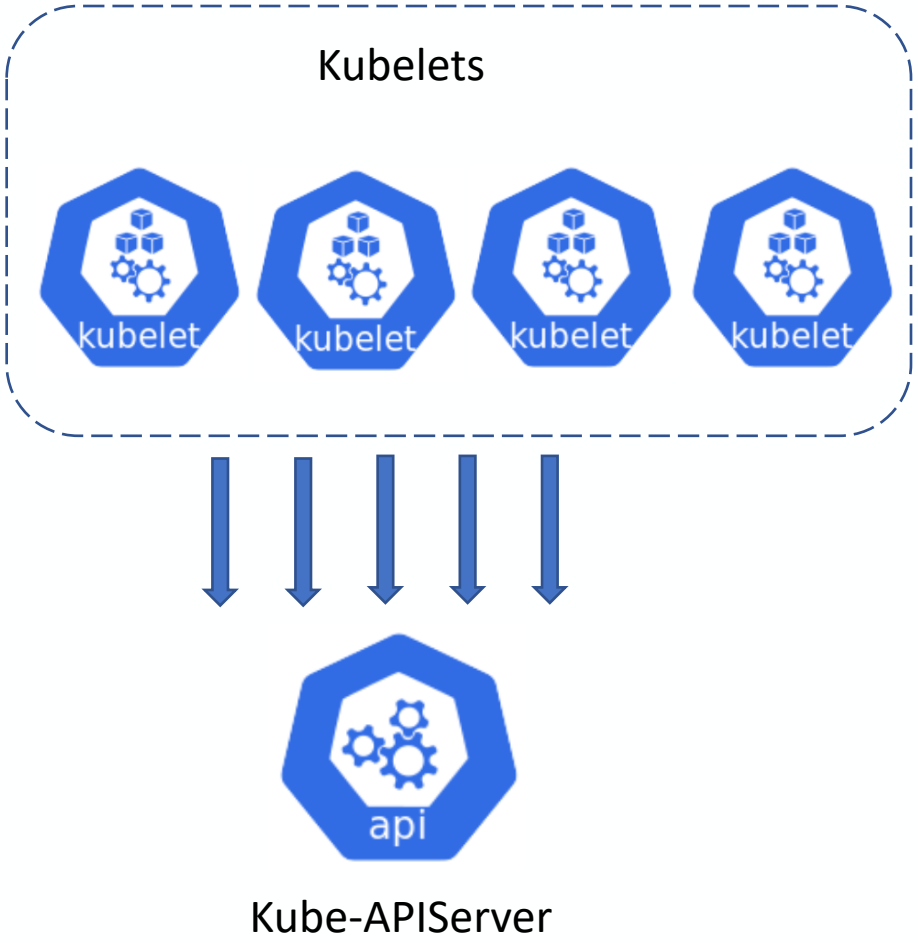
We had a situation where a bug in the Deployment controller caused it to run amuck under certain circumstances, issuing requests in a tight loop. We would like controller bugs to not take the whole system down.

The same issue also applies to those third-party controllers installed to the cluster for different purposes, a buggy controller may have ill behavior such as:

1. Frequently issuing heavy unpaginated LIST requests to the Kube-APIServer which should have been avoided by reading the cache or list resources with a pager.

2. Having too many failing task items in the queue that the controller keeps retrying inefficiently with many meaningless requests against the Kube-APIServer.

Kubelet Amuck



The controller that runs amuck might not be a central singleton, it could be a **kubelet**, **kube-proxy**, or **other per-node** or otherwise multiplied controller. In such a situation we would like only the guilty individual to suffer, not all its peers and the rest of the system.

While this particular issue isn't necessarily connected a bug, it can be the cluster reaching its scalability limit. E.g. the cluster can't bear more kubelet instances. But the fun fact is that usually we don't know the limit until we actually reach it. But we feel like to sense the limit by something like a mild warning instead of seeing the whole cluster burning in fire.

Higher Goal

- **Multi-Tenancy**

  - Provide guaranteed capacity for controllers that are “less important”

  - Tenants (in the same priority band) sharing the cluster should get an equal share of service.

Multi-Tenancy

Kubernetes cluster is designed to be shared by multiple tenants.

- Each tenant corresponds with a kube API namespace.
- Each tenant corresponds with a user name.
- Each tenant corresponds with a prefix of the user name.
- Each tenant corresponds with a user's group. Other groups may exist. There is a subset of the groups that serve to identify tenants. Each user belongs to exactly one of the tenant-identifying groups.
- K-sigs/multi-tenancy subproject has a brand new API definition of tenant which is basically consists of a group of namespaces.

Non Goal

- No coordination between apiservers nor with a load balancer is attempted. Each apiserver independently protects itself. We imagine that later developments may add support for informing load balancers about the load state of the apiservers.

- Will not attempt auto-tuning the capacity limit(s). Instead the administrator will configure each apiserver's capacity limit(s), analogously to how the max-in-flight limits are configured today.

- Will not attempt to reproduce the functionality of the existing event rate limiting admission plugin. Events are a somewhat special case. For now we intend to simply leave the existing admission plugin in place.

System Design

Retrospection

Flow-control Basics

There're two basic approaches of flow-control methods:

- **At the source / Client-side**

  * Kubernetes/client-go provides a Token-Bucket rate-limiter for muzzling the clients, and there was even a dedicated admission controller for rate-limit the events (now deprecated) but client-side rate-limiting always has a few defects:

    1. Users can opt-out from rate-limiting by granting the bucket a minus or infinite capacity.
    2. Tough to control the granularity if multiple controllers/components are built in a same go process.

- **At the gateway / Server-side**

Existing Server-side Limiter

1. Limiting the total number of executing requests

--max-mutating-requests-inflight int Default: 200	
	The maximum number of mutating requests in flight at a given time. When the server exceeds this, it rejects requests. Zero for no limit.
--max-requests-inflight int Default: 400	
	The maximum number of non-mutating requests in flight at a given time. When the server exceeds this, it rejects requests. Zero for no limit.

2. Apply a timeout for non-long-running requests

--min-request-timeout int Default: 1800	
	An optional field indicating the minimum number of seconds a handler must keep a request open before timing it out. Currently only honored by the watch request handler, which picks a randomized value above this number as the connection timeout, to spread out load.

Linux Qdisc

Classless Qdisc

* **RR**: Round Robin schedules packets without priority. Can't handle bursty traffic.

* **CoDel**: Controlled Delay aimed at overcoming bufferbloat. The algorithm controls delay for each single queue independently which cannot do a global optimized scheduling decision.

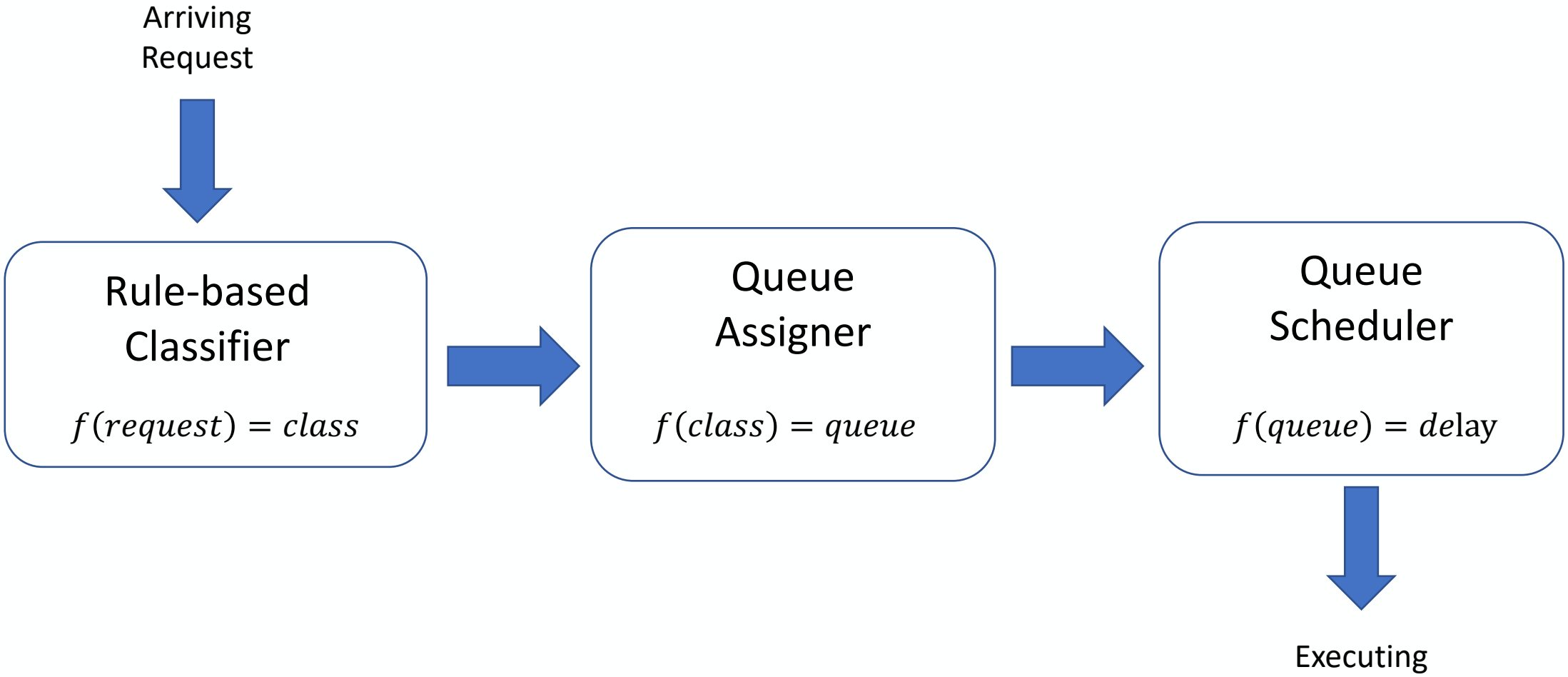
* **TBF**: Token Bucket Filter is good at shaping uneven traffic. Needs the users carefully tuning proper parameters {AverageRate, BurstCapacity} for the bucket.

Classful Qdisc

* **DRR**: Deficit Robin Robin is good at both traffic shaping and providing fairness. If we can properly abstract the cost into the "Quantum" in the algorithm.

* **HTB**: Hierarchical Token Bucket basically organizes multiple TBF into a tree structure and allows capacity borrowing between the leaves.

Classful Flow-Control System



Classifier

We extend the abstraction of Class from Linux TC system to Priority Level in the Kube-APIServer. In the new flow-control system, a Priority Level is:

- A priority band that requests in higher priorities should be executed in prior to lower priorities.
- A request class in which all its matching requests are handled equally.
- A request class where we applies the same rejection strategy (either reject immediately or waiting in-queue)

Classifier

To classify the requests into proper Priority Level, the input we can get from Kube-APIServer's requests context is:

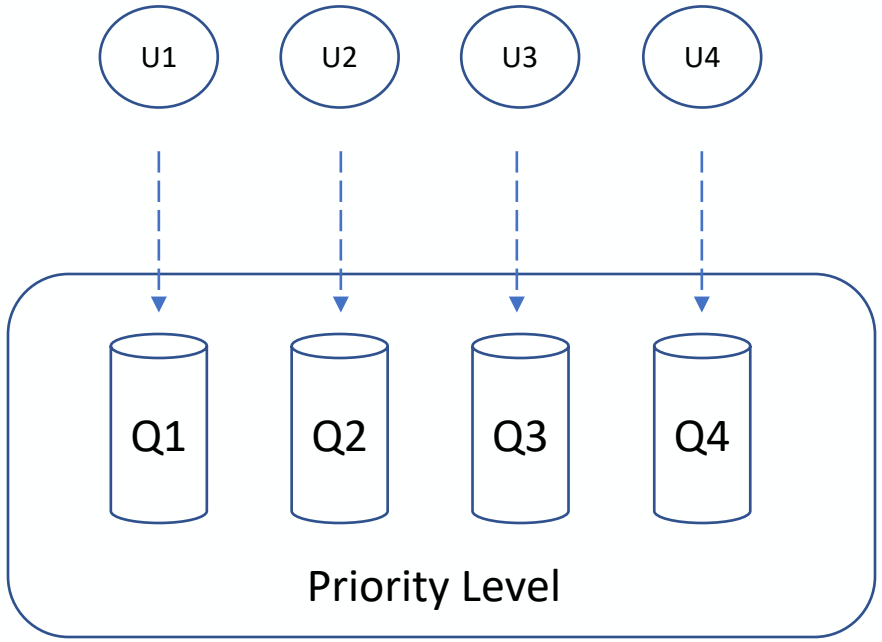
- **Client Identity**: These identity information is encrypted in the client-side X509 certificate hence reliable.
  - User (Name)
  - User Groups
- **Requesting Target**
  - Requesting Namespace (Empty "" for cluster-scoped)
  - Other request Metadata, such as verbs, target resource types, etc.

Queue Assigner

Each Priority Level contains a group of request-queues for scheduling. How to map a request to one of queues?

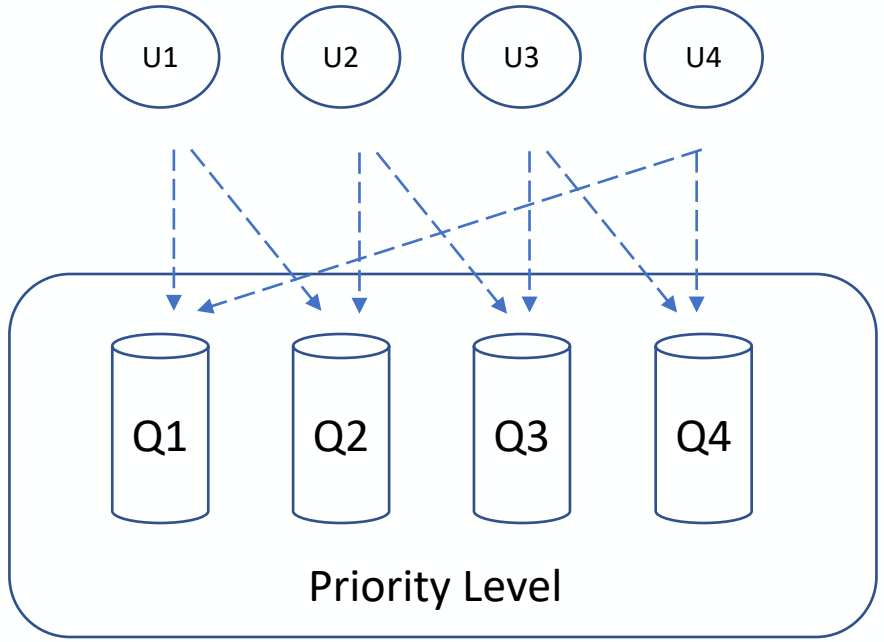
One Queue Per User/Tenant

#(users) == #(queues)



Shuffle Sharding

#(hand-size) == 2 && #(queues) == 4
requests from each user will be ramdomly queued into one of the 2 queue candidate

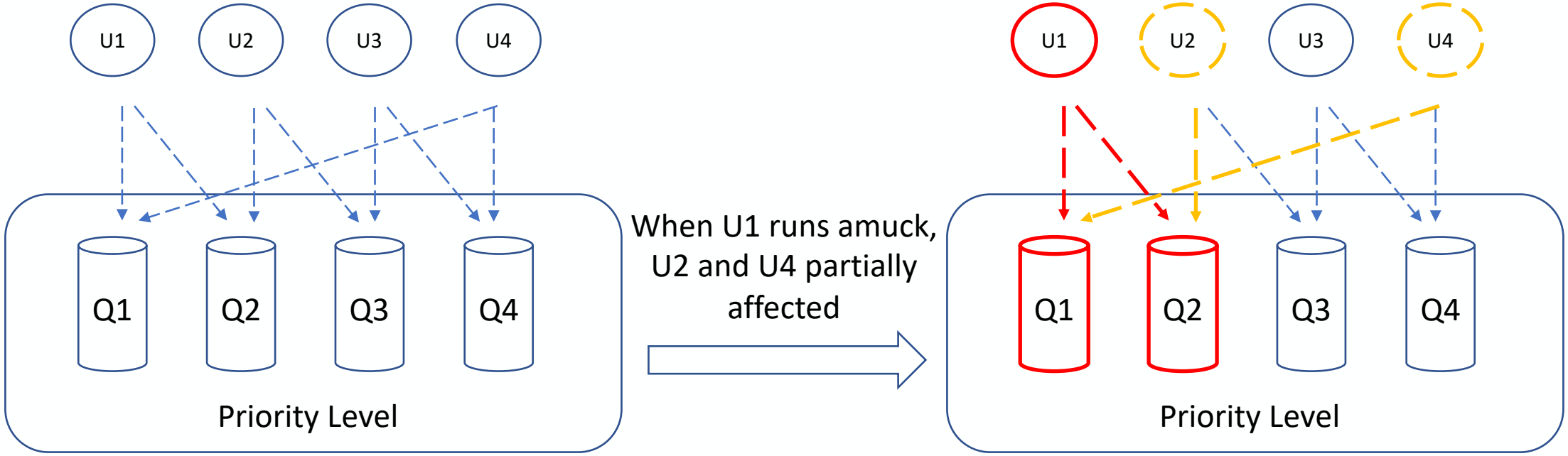


Shuffle Sharding

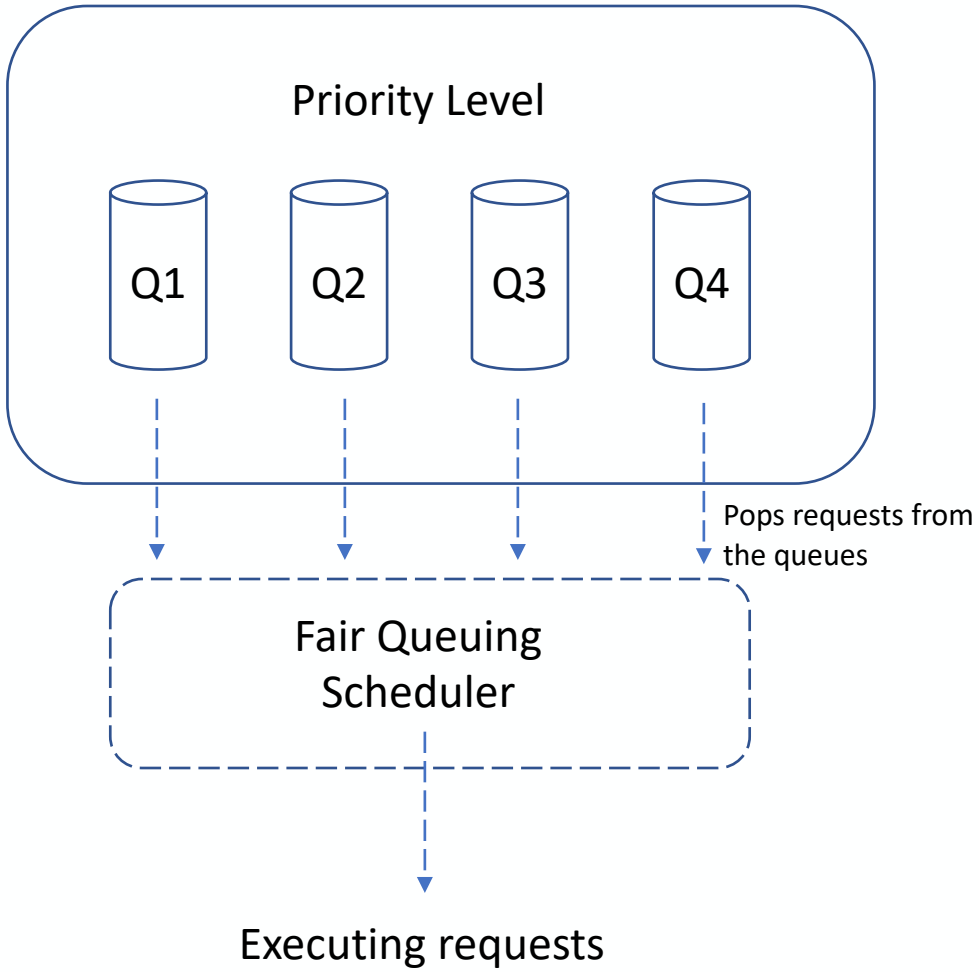
- Save users from adjusting/tuning number of queues each time when a new user get abroad the kubernetes cluster

- Bound the memory cost from the managed queues and the pending items in-queue.

- Distribute the disturbances from the noisy users to a limit number of other queues.



Queue Scheduler



We uses Fair Queuing algorithm which is aiming at the following goals upon scheduling requests from the queues:

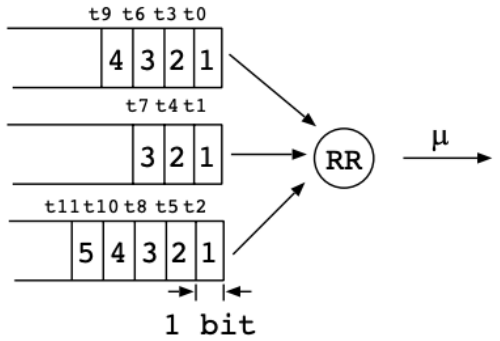
- Even distribution
- Max-Min Fairness

FQ: Fair Queuing

Bit-by-Bit Round Robin

1 round, $R(\ )$, is defined as all non-empty queues have been served 1 quantum

- $R(t_5) = 2$
- time at Round 3? Round 4?



BbB-RR achieves max-min fair share

Max-min fair-share isolates flows

BbB-RR protects against misbehaving flows

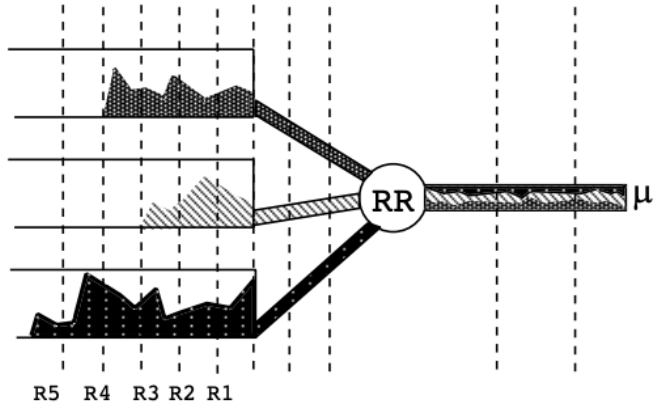
A.k.a. Generalized Processor Sharing (GPS)

Fluid-Flow Approximation

A continuous service model

- instead of thinking of each quantum as serving discrete bits in a given order
- think of each connection as a stream of fluid, described by the speed and volume of flow

At each quantum the same amount of fluid from each (non-empty) stream flows out concurrently



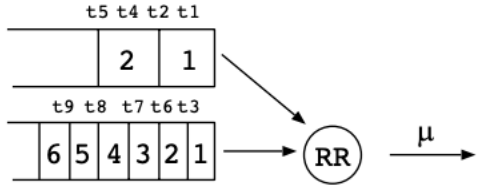
FQ: Fair Queuing

Packetized Scheduling

Packet-by-packet round robin:

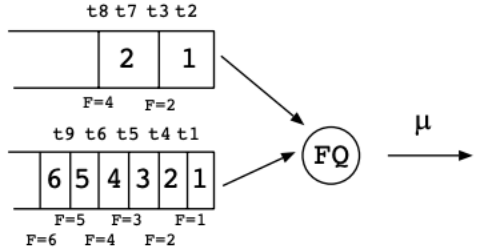
Problem:

Gives bigger share to flows with big packets



Packet-by-packet fair-queueing:

- F: finish round, the round a packet finishes service
- simulates RR in the computation of F's
- serve packets with the smallest F first



Variation: Credit Accumulation

Allow a flow to have a bigger share if it has been idle

Compute bid per packet:

$$B^{\alpha}_{i} = \mathrm{MAX}(B^{\alpha}_{i-1}, A^{\alpha}_{i} - \delta) + P^{\alpha}_{i}$$

- if $\delta = 0$, no credit accumulated, $B^{\alpha}_{i} = F^{\alpha}_{i}$
- if $\delta = \infty$, $B^{\alpha}_{i} = F^{\alpha}_{i-1} + P^{\alpha}_{i}$, regardless of packet i's arrival time

Credit accumulation is discouraged because it can be abused: accumulate credits for a long time, then send a big burst of data

FQ for Server Requests

Limitations for adapting FQ into Kube-APIServer:

1. Dispatching requests to be served rather than packets to be transmitted.
2. Multiple requests may be served at once.
3. The actual service time (i.e., duration) is not known until a request is done being served.

FQ for Server Requests

> 1. Dispatching requests to be served rather than packets to be transmitted.
> 2. Multiple requests may be served at once.

Adapting the original concept of virtual time R(t) with C --- the concurrentcy limit of the Kube-APIServer.

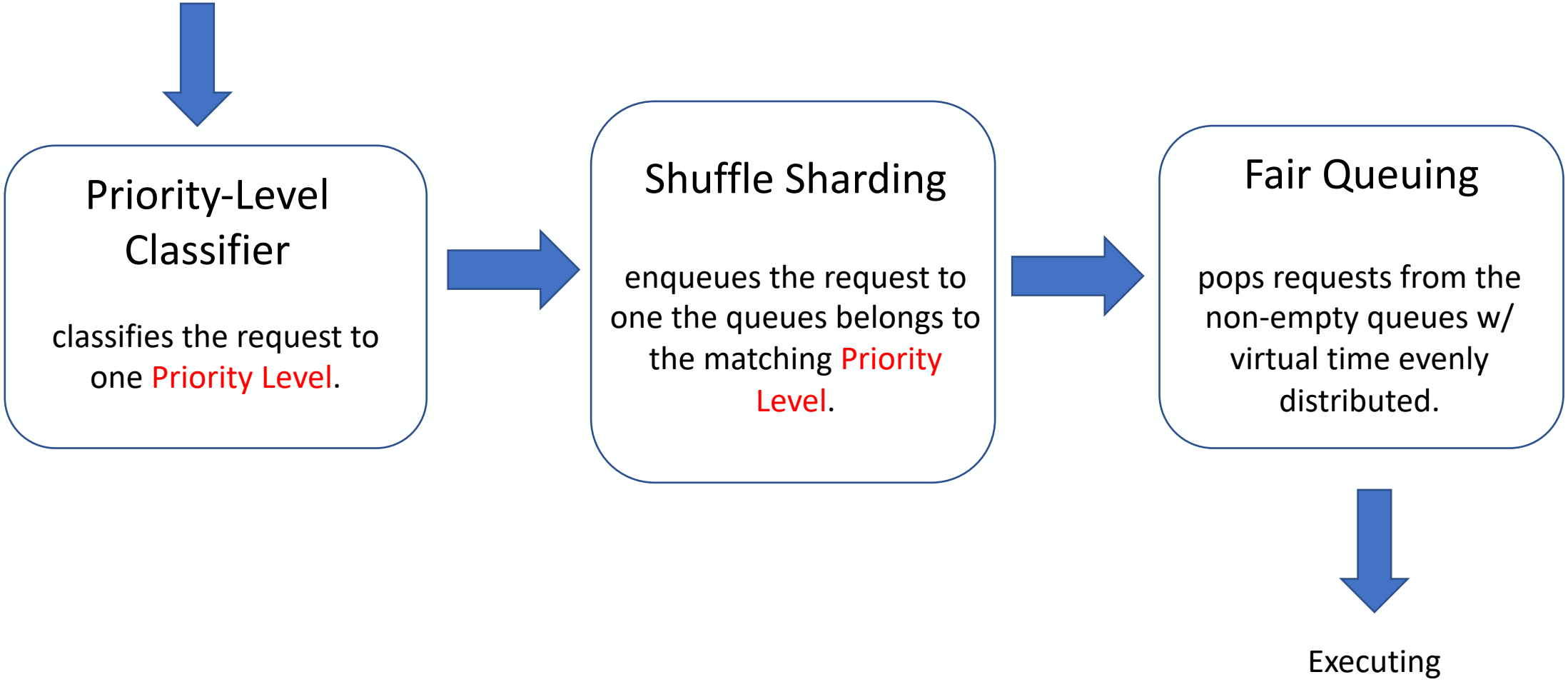
$$R(t) = \frac{1}{NEQ(t)} \longrightarrow R(t) = \frac{C}{NEQ(t)}$$

> 3. The actual service time (i.e., duration) is not known until a request is done being served.

Modifying the algorithm to dispatch based on an initial guess at the request's service time (duration) and then make the corresponding adjustments once the request's actual service time is known.

APF System

Arriving Request consists of user identiy, and request metadata.



Kubernetes (1.18+)

Alpha Implementation

Documentation

User-Facing Documentation:

https://kubernetes.io/docs/concepts/cluster-administration/flow-control/

Enabling the feature at Kube-APIServer by doing the following configurations:

- Enabling feature gate *APIPriorityAndFairness=true*
- Adding *--runtime-config=flowcontrol.apiserver.k8s.io/v1alpha1=true* to the Kube-APIServer's starting flags.

FlowSchema (Exempt)


```
apiVersion: flowcontrol.apiserver.k8s.io/v1alpha1
kind: FlowSchema
metadata:
  name: exempt
spec:
  matchingPrecedence: 1
  priorityLevelConfiguration:
    name: exempt
  rules:
  - nonResourceRules:
    - nonResourceURLs:
      - '*'
      verbs:
      - '*'
    resourceRules:
    - apiGroups:
      - '*'
      clusterScope: true
      namespaces:
      - '*'
      resources:
      - '*'
      verbs:
      - '*'
    subjects:
    - group:
        name: system:masters
      kind: Group
status:
  conditions:
  - lastTransitionTime: "2020-10-19T13:20:19Z"
    message: This FlowSchema references the
PriorityLevelConfiguration object named
      "exempt" and it exists
    reason: Found
    status: "False"
    type: Dangling
```

FlowSchema (Catch-All)


```
apiVersion: flowcontrol.apiserver.k8s.io/v1alpha1
kind: FlowSchema
metadata:
  name: catch-all
spec:
  distinguisherMethod:
    type: ByUser
  matchingPrecedence: 10000
  priorityLevelConfiguration:
    name: catch-all
  rules:
  - nonResourceRules:
    - nonResourceURLs:
      - '*'
      verbs:
      - '*'
    resourceRules:
    - apiGroups:
      - '*'
      clusterScope: true
      namespaces:
      - '*'
      resources:
      - '*'
      verbs:
      - '*'
    subjects:
    - group:
        name: system:unauthenticated
      kind: Group
    - group:
        name: system:authenticated
      kind: Group
status:
  conditions:
  - lastTransitionTime: "2020-10-19T13:20:19Z"
    message: This FlowSchema references the
PriorityLevelConfiguration object named
      "catch-all" and it exists
    reason: Found
    status: "False"
```

PriorityLevelConfiguration


```
apiVersion: flowcontrol.apiserver.k8s.io/v1alpha1
kind: PriorityLevelConfiguration
metadata:
  name: leader-election
spec:
  limited:
    assuredConcurrencyShares: 10
    limitResponse:
      queuing:
        handSize: 4
        queueLengthLimit: 50
        queues: 16
      type: Queue
  type: Limited
status: {}
```

```
apiVersion: flowcontrol.apiserver.k8s.io/v1alpha1
kind: PriorityLevelConfiguration
metadata:
  name: catch-all
spec:
  limited:
    assuredConcurrencyShares: 1
    limitResponse:
      type: Reject
  type: Limited
status: {}
```

DEMO:

Customize Configuration for a KinD cluster

Planned Enhancements for Beta Stage

Blocking Items

- **Improving observability and robustness**: Adding debug endpoint dumping fine-grained states of the queues for priority-levels.

- **Providing approaches to opt-out client-side rate-limiting**: Configurable client-side rate-limitting(QPS/Burst) via either kubeconfig or command-line flags

- **Necessary e2e tests**

Non-Blocking Items

- Supports concurrency limiting upon long-running requests.
- Allow constant concurrency/relative shares in the priority-level API model.
- Automatically manages versions of mandatory/suggested configuration.
- Discriminates paginated LIST requests.


KEEP CLOUD NATIVE
EVERYWHERE
KubeCon
CloudNativeCon
North America 2020
Virtual

Q&A