K8S - Creating a kube-scheduler plugin

Julio Renner · Follow

11 min read · Jul 26, 2021

Saying it in a few words, the K8S scheduler is responsible for assigning *Pods* to *Nodes.* Once a new pod is created it gets in the scheduling queue. The attempt to schedule a pod is split in two phases: the *Scheduling* and the *Binding cycle.*

In the *Scheduling cycle* the nodes are filtered, removing those that don't meet the pod requirements. Next, the *feasible nodes* (the remaining ones)*,* are ranked based on a given score. Finally, the node with highest score is chosen. These steps are called *Filtering* and *Scoring [1].*

Once a node is chosen, the scheduler needs to make sure *kubelet* knows it needs to start the pod (containers) in the selected node. The step related to starting the pod into the selected node is called *Binding Cycle [2].*

The *Scheduling* and *Binding cycle* are composed by stages that are executed sequentially to calculate the pod placement. These stages are called extension points and can be used to shape the placement behavior. *Scheduling Cycles* for different pods are run sequentially, meaning that the *Scheduling Cycle* steps will be executed for one pod at a time, whereas *Binding Cycles* for different pods may be executed concurrently.

The components that implement the extension points of kubernetes scheduler are called *Plugins.* The native scheduling behavior is implemented using the *Plugin* pattern as well, in the same way that custom extensions, making the core of the kube-scheduler lightweight as the main scheduling logic is placed in the plugins.

The extension points where the plugins can be applied are shown in *Figure 1*. A plugin can implement one or more of the extension points and a detailed description of each can be found in [4] (I won't be copying stuff here, check it there before continuing 😃).

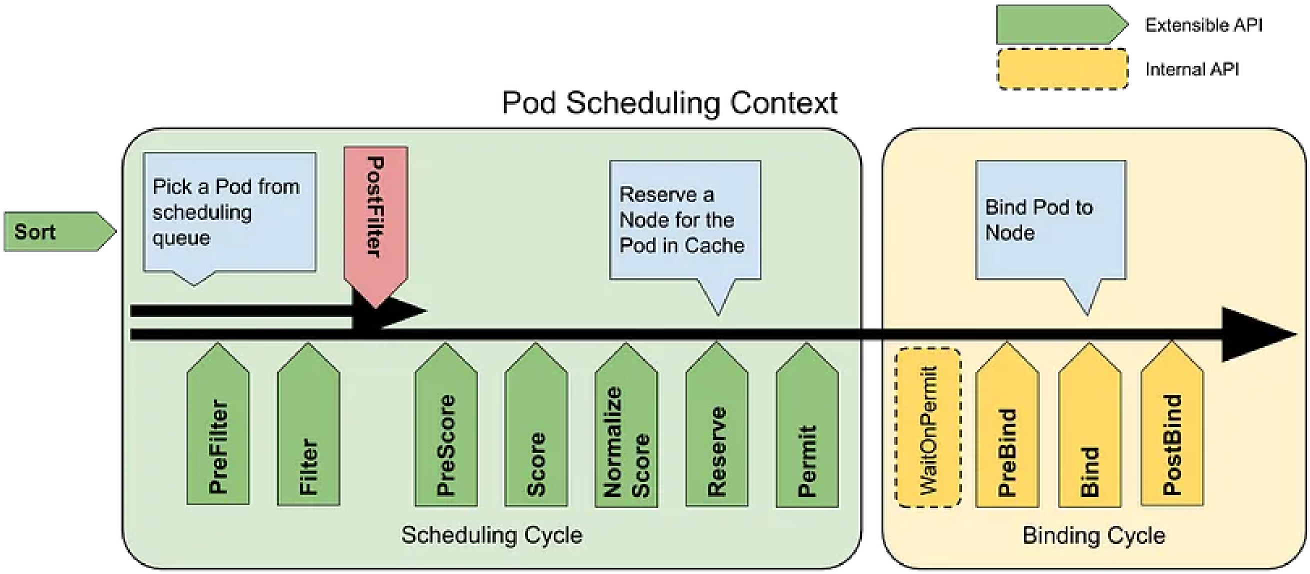


Figure 1

To configure the *Plugins* that should be executed in each extension point, and then change the scheduling behavior, kube-scheduler provides *Profiles* [3]. A scheduling *Profile* describes which plugins should be executed on each stage mentioned in [4]. It is possible to provide multiple profiles, which means that there's no need to deploy multiple schedulers to have different scheduling behaviors [5].

kube-scheduler

The kube-scheduler is implemented in Golang and *Plugins* are included to it in compilation time. Therefore, if you want to have your own plugin, you will need to have your own scheduler image.

A new plugin needs to be registered and get configured to the plugin API. Also, it needs to implement the extension points interfaces that are defined in the [kubernetes scheduler framework package](). Check out how it looks:

```go
// Plugin is the parent type for all the scheduling framework plugins.
type Plugin interface {
   Name() string
}

type QueueSortPlugin interface {
  Plugin
  Less(*QueuedPodInfo, *QueuedPodInfo) bool
}

type PreFilterPlugin interface {
  Plugin
  PreFilter(CycleState, *v1.Pod) *Status
  PreFilterExtensions() PreFilterExtensions
}
```

plugin_pkg_interfaces_example.go hosted with ❤ by GitHub view raw

The scheduler's code allows to add new plugins without having to fork it. For that, developers just need to write their own `main()` wrapper around the scheduler. As plugins must be compiled with the scheduler, writing a wrapper allows to re-use the scheduler's code in a clean way [7].

To do that, the main function will import the `k8s.io/kubernetes/cmd/kube-scheduler/app` and use the `NewSchedulerCommand` to register the custom plugins, providing the respective name and the constructor function:

```go
import (
    "k8s.io/kubernetes/cmd/kube-scheduler/app"
)

func main() {
    command := app.NewSchedulerCommand(
      app.WithPlugin("example-plugin1", ExamplePlugin1.New),
      app.WithPlugin("example-plugin2", ExamplePlugin2.New))
    if err := command.Execute(); err != nil {
        fmt.Fprintf(os.Stderr, "%v\n", err)
        os.Exit(1)
    }
}
```

scheduler_main.go hosted with ❤ by GitHub view raw

Configuration

The kube-scheduler configuration is where the profiles can be configured. Each profile allows plugins to be enabled, disabled and configured according to the configurations parameters defined by the plugin. Each profile configuration is separated into two parts [9]:

1. A list of enabled plugins for each extension point and the order they should run. If one of the extension points list is omitted, the default list will be used.
2. An optional set of custom plugin arguments for each plugin. Omitting config args for a plugin is equivalent to using the default config for that plugin.

Plugins that are enabled in different extension points must be configured explicitly in each of them.

The configuration is provided through the `KubeSchedulerConfiguration` struct. To enable it, it needs to be written to a configuration file and its path provided as a command line argument to kube-scheduler. E.g.:

```
kube-scheduler --config=/etc/kubernetes/networktraffic-config.yaml
```

Below you can see an example configuration of the `NetworkTraffic` plugin. In the example, the `clientConnection.kubeconfig` points to the kubeconfig path used by the kube-scheduler, with its defined authorizations in the control plane nodes. The `profiles` section overwrites the `default-scheduler` score phase enabling the `NetworkTraffic` plugin and disabling the others defined by default. The `pluginConfig` sets the configuration of the plugin, that will be provided during its initialization [8].

```yaml
apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration
clientConnection:
  kubeconfig: "/etc/kubernetes/scheduler.conf"
profiles:
- schedulerName: default-scheduler
  plugins:
    score:
      enabled:
      - name: NetworkTraffic
      disabled:
      - name: "*"
  pluginConfig:
  - name: NetworkTraffic
    args:
      prometheusAddress: "http://prometheus-1616380099-server.monitor"
      networkInterface: "ens192"
      timeRangeInMinutes: 3
```

networktraffic-config.yaml hosted with ❤ by GitHub view raw

PS: If you have HA with multiple control plane nodes, the configuration needs to be applied for each of them.

Creating a custom plugin

Now that we understand the basics of kube-scheduler, we can do what we came here for. As we've seen previously, adding a custom plugin requires to include our code during compilation time and we don't need to fork the scheduler code for that.

To proceed, we could create an empty repository and wrap the scheduler as described before, however, the project scheduler-plugins already does that and provides some custom plugins that are good examples to follow. So, we will just start from there.

Fork the scheduler-plugins repository and pull it into `$GOPATH/src/sigs.k8s.io`. With that done, we can start :)

To keep following the next steps, you need to:

1. Have a K8S cluster (I am using a cluster created with kubespray).
2. Have prometheus configured with node-exporter. Check kube-prometheus-stack.

NetworkTraffic Plugin

For this example, we are going to build a Score Plugin named "NetworkTraffic" that favors nodes with lower network traffic. To gather that information we will query prometheus.

To start, create the folder `pkg/networktraffic` and the files `networktraffic.go` and `prometheus.go` inside your fork of scheduler-plugins. The structure should look like this:

```
|- pkg
|-- networktraffic
|--- networktraffic.go
|--- prometheus.go
```

In the `networktraffic.go` we are going to have the implementation of the ScorePlugin interface and in the `prometheus.go` we will keep the logic to interact with prometheus.

Prometheus communication

In the `prometheus.go` we will start by declaring the struct used to interact with Prometheus. It will have the fields `networkInterface` and `timeRange`, which can be used to configure the query we will be executing. The field `address` points to the prometheus service on K8S and can also be configured. The field `api` will be used to store the prometheus client, which is created based on the `address` provided.

```
type PrometheusHandle struct {
  networkInterface string
  timeRange        time.Duration
  address          string
  api              v1.API
}
```

Now that we have the basic structure we can also implement the querying. We will be using the sum of the received bytes in a time range per node in a specific network interface. The `kubernetes_node` filter will query the metrics for the node provided, as described by the query below. The `device` filter will query the metrics on the provided network interface, and the last value between `[%s]` defines the time range taken into account. `sum_over_time` will sum all the values in the provided time range.

```
sum_over_time(node_network_receive_bytes_total{kubernetes_node=\"%s\
",device=\"%s\"}[%s])
```

At the end, the `prometheus.go` file will look like this:

```go
package networktraffic

import (
	"context"
	"fmt"
	"time"

	"github.com/prometheus/client_golang/api"
	v1 "github.com/prometheus/client_golang/api/prometheus/v1"
	"github.com/prometheus/common/model"
	"k8s.io/klog/v2"
)

const (
	// nodeMeasureQueryTemplate is the template string to get the query for the node used b
	nodeMeasureQueryTemplate = "sum_over_time(node_network_receive_bytes_total{kubernetes_n
)

// Handles the interaction of the networkplugin with Prometheus
type PrometheusHandle struct {
	networkInterface string
	timeRange        time.Duration
	address          string
	api              v1.API
}

func NewPrometheus(address, networkInterface string, timeRange time.Duration) *PrometheusHandle
	client, err := api.NewClient(api.Config{
		Address: address,
	})
	if err != nil {
		klog.Fatalf("[NetworkTraffic] Error creating prometheus client: %s", err.Error(
	}

	return &PrometheusHandle{
		networkInterface: networkInterface,
		timeRange:        timeRange,
		address:          address,
		api:              v1.NewAPI(client),
	}
}

func (p *PrometheusHandle) GetNodeBandwidthMeasure(node string) (*model.Sample, error) {
	query := getNodeBandwidthQuery(node, p.networkInterface, p.timeRange)
	res, err := p.query(query)
	if err != nil {
		return nil, fmt.Errorf("[NetworkTraffic] Error querying prometheus: %w", err)
	}
```

```
	}

	nodeMeasure := res.(model.Vector)
	if len(nodeMeasure) != 1 {
		return nil, fmt.Errorf("[NetworkTraffic] Invalid response, expected 1 value, go
	}

	return nodeMeasure[0], nil
}

func getNodeBandwidthQuery(node, networkInterface string, timeRange time.Duration) string {
	return fmt.Sprintf(nodeMeasureQueryTemplate, node, networkInterface, timeRange)
}

func (p *PrometheusHandle) query(query string) (model.Value, error) {
	results, warnings, err := p.api.Query(context.Background(), query, time.Now())

	if len(warnings) > 0 {
		klog.Warningf("[NetworkTraffic] Warnings: %v\n", warnings)
	}

	return results, err
}
```

ScorePlugin interface

Having the interaction with Prometheus done, we can move to the implementation of the Score Plugin. As mentioned, we will need to implement the Score Plugin Interface from the scheduler framework:

```go
// ScorePlugin is an interface that must be implemented by "Score"
// plugins to rank nodes that passed the filtering phase.
type ScorePlugin interface {
  Plugin

  // Score is called on each filtered node. It must return success
  // and an integer indicating the rank of the node. All scoring
  // plugins must return success or the pod will be rejected.
  Score(ctx context.Context, state *CycleState, p *v1.Pod, nodeName string) (int64, *Status)

  // ScoreExtensions returns a ScoreExtensions interface if it
  // implements one, or nil if does not.
  ScoreExtensions() ScoreExtensions
}

// ScoreExtensions is an interface for Score extended functionality.
type ScoreExtensions interface {
  // NormalizeScore is called for all node scores produced by the
  // same plugin's "Score" method. A successful run of
  // NormalizeScore will update the scores list and return a success
  // status.
  NormalizeScore(ctx context.Context, state *CycleState, p *v1.Pod, scores NodeScoreList) *Stat
}
```


The `Score` function is called for each node and returns whether it was successful and an integer indicating the rank of the node. At the end of the Score plugin execution, we should have a Score value in the range from 0 to 100. In some cases it could be difficult to have a value within that range without knowing the score of other nodes, for example. For those scenarios, we can use the `NormalizeScore` function implemented in the `ScoreExtensions` interface. The `NormalizeScore` function receives the result of all nodes and allows them to be changed.

Moreover, the ScorePlugin interface also have the `Plugin` interface as an <u>embedded</u> field. So, we must implement its `Name() string` function.

Now that we understand the ScorePlugin interface, let's go to the `networktraffic.go` file. We will start by defining the `NetworkTraffic` struct:

```go
// NetworkTraffic is a score plugin that favors nodes based on their
// network traffic amount. Nodes with less traffic are favored.
// Implements framework.ScorePlugin
```

```
type NetworkTraffic struct {
  handle     framework.FrameworkHandle
  prometheus *PrometheusHandle
}
```

With the structure defined, we can proceed with the `Score` function implementation. It will be straightforward. We will only call the `GetNodeBandwidthMeasure` function from our Prometheus structure providing the node name. The call will return a `Sample` which holds the value in the `Value` field. We will basically return it for each node.

```go
func (n *NetworkTraffic) Score(ctx context.Context, state *framework.CycleState, p *v1.Pod, node
	nodeBandwidth, err := n.prometheus.GetNodeBandwidthMeasure(nodeName)
	if err != nil {
		return 0, framework.NewStatus(framework.Error, fmt.Sprintf("error getting node b
	}

	klog.Infof("[NetworkTraffic] node '%s' bandwidth: %s", nodeName, nodeBandwidth.Value)
	return int64(nodeBandwidth.Value), nil
}
```

score.go hosted with ❤ by GitHub view raw

Network Traffic plugin Score function

Next, we will have returned the total bytes received by each node in a determined period of time. However, the scheduler framework expects a value from 0 to 100, thus, we still need to normalize the values to fulfill this requirement.

To do the normalization, we will implement the `ScoreExtensions` interface mentioned before. We will implement the interface embedded in the `NetworkTraffic` struct. In the `ScoreExtensions` function we will simply return the struct which implements the interface. The logic is placed under the `NormalizeScore` function.

```go
func (n *NetworkTraffic) ScoreExtensions() framework.ScoreExtensions {
	return n
}

func (n *NetworkTraffic) NormalizeScore(ctx context.Context, state *framework.CycleState, pod *
	var higherScore int64
	for _, node := range scores {
		if higherScore < node.Score {
			higherScore = node.Score
		}
	}

	for i, node := range scores {
		scores[i].Score = framework.MaxNodeScore - (node.Score * framework.MaxNodeScore
	}

	klog.Infof("[NetworkTraffic] Nodes final score: %v", scores)
	return nil
}
```


The `NormalizeScore` basically will take the highest value returned by prometheus and use it as the highest possible value, corresponding to the `framework.MaxNodeScore` (100). The other values will be calculated relatively to the highest score using the rule of three.

Finally, we will have a list where the nodes with more network traffic have a greater score in the range of [0,100]. If we use it like it is, we would favor nodes that have higher traffic, so, we need to reverse the values. For that, we will simply replace the node score with the result of the rule of three, subtracted by the max score.

An example of the calculation which take as an example three nodes (*a, b* and *c*), the values are in bytes, is given below:

```
a => 1000000    # 1MB
b => 1200000    # 1,2MB
c => 1400000    # 1,4MB

higherScore = 1400000

Y = (node.Score * framework.MaxNodeScore) / higherScore

Ya = 1000000 * 100 / 1400000
Yb = 1200000 * 100 / 1400000
```

```
Yc = 1400000 * 100 / 1400000

Ya = 71,42
Yb = 85,71
Yc = 100

Xa = 100 - Ya
Xb = 100 - Yb
Xc = 100 - Yc

Xa = 28,58
Xb = 14,29
Xc = 0
```

With that explained, we have the main pieces of our plugin. However, that's not all. As mentioned before, the scheduler plugins can be configured, and there are three configurations we will allow in our Network Traffic plugin, which were already mentioned:

- Prometheus address
- Prometheus query time range
- Prometheus query node network interface

Those values will be provided during the instantiation of the `NetworkTraffic` plugin by the scheduler framework, and we will need to declare a new struct called `NetworkTrafficArgs` that will be used to parse the configuration provided in the `KubeSchedulerConfiguration`. For that, we need to add a new function with the logic to create an instance of the `NetworkTraffic` plugin, described below:

```go
// New initializes a new plugin and returns it.
func New(obj runtime.Object, h framework.FrameworkHandle) (framework.Plugin, error) {
	args, ok := obj.(*config.NetworkTrafficArgs)
	if !ok {
		return nil, fmt.Errorf("want args to be of type NetworkTrafficArgs, got %T", ob

	}

	return &NetworkTraffic{
		handle:     h,
		prometheus: NewPrometheus(args.Address, args.NetworkInterface, time.Minute*time
	}, nil
}
```

score_new.go hosted with ❤ by GitHub view raw

The `New` function follows the scheduler framework `PluginFactory` interface.

We still haven't declared the `NetworkTrafficArgs` struct, and that will come next. However, we have (almost) all we need for `networktraffic.go`:

```go
package networktraffic

import (
        "context"
        "fmt"
        "time"

        v1 "k8s.io/api/core/v1"
        "k8s.io/apimachinery/pkg/runtime"
        "k8s.io/klog/v2"
        framework "k8s.io/kubernetes/pkg/scheduler/framework/v1alpha1"
        "sigs.k8s.io/scheduler-plugins/pkg/apis/config"
)

// NetworkTraffic is a score plugin that favors nodes based on their
// network traffic amount. Nodes with less traffic are favored.
// Implements framework.ScorePlugin
type NetworkTraffic struct {
        handle     framework.FrameworkHandle
        prometheus *PrometheusHandle
}

// Name is the name of the plugin used in the Registry and configurations.
const Name = "NetworkTraffic"

var _ = framework.ScorePlugin(&NetworkTraffic{})

// New initializes a new plugin and returns it.
func New(obj runtime.Object, h framework.FrameworkHandle) (framework.Plugin, error) {
        args, ok := obj.(*config.NetworkTrafficArgs)
        if !ok {
                return nil, fmt.Errorf("[NetworkTraffic] want args to be of type NetworkTraffic
        }

        klog.Infof("[NetworkTraffic] args received. NetworkInterface: %s; TimeRangeInMinutes: %

        return &NetworkTraffic{
                handle:     h,
                prometheus: NewPrometheus(args.Address, args.NetworkInterface, time.Minute*time
        }, nil
}

// Name returns name of the plugin. It is used in logs, etc.
func (n *NetworkTraffic) Name() string {
        return Name
}

func (n *NetworkTraffic) Score(ctx context.Context, state *framework.CycleState, p *v1.Pod, nod
```

```
func (n *NetworkTraffic) Score(ctx context.Context, state *framework.CycleState, p *v1.Pod, noc
	nodeBandwidth, err := n.prometheus.GetNodeBandwidthMeasure(nodeName)
	if err != nil {
		return 0, framework.NewStatus(framework.Error, fmt.Sprintf("error getting node
	}

	klog.Infof("[NetworkTraffic] node '%s' bandwidth: %s", nodeName, nodeBandwidth.Value)
	return int64(nodeBandwidth.Value), nil
}

func (n *NetworkTraffic) ScoreExtensions() framework.ScoreExtensions {
	return n
}

func (n *NetworkTraffic) NormalizeScore(ctx context.Context, state *framework.CycleState, pod *
	var higherScore int64
	for _, node := range scores {
		if higherScore < node.Score {
			higherScore = node.Score
		}
	}

	for i, node := range scores {
		scores[i].Score = framework.MaxNodeScore - (node.Score * framework.MaxNodeScore
	}

	klog.Infof("[NetworkTraffic] Nodes final score: %v", scores)
	return nil
}
```

Configuration

The scheduler-plugins project holds the configurations under `pkg/apis` folder. So, we will have ours plugin config there as well.

We will add the configuration in two places: `pkg/apis/config/types.go` and `pkg/apis/config/v1beta1/types.go`. The `config/types.go` holds the struct we will use in the `New` function, while the `v1beta1/types.go` holds the struct used to parse the information from the `KubeSchedulerConfiguration`.

Also, the config struct must follow the name pattern <Plugin Name>Args, otherwise, it won't be properly decoded and you will face issues.

```go
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

// NetworkTrafficArgs holds arguments used to configure NetworkTraffic plugin.
type NetworkTrafficArgs struct {
	metav1.TypeMeta

	// Address of the Prometheus Server
	Address string
	// NetworkInterface to be monitored, assume that nodes OS is homogeneous
	NetworkInterface string
	// TimeRangeInMinutes used to aggregate the network metrics
	TimeRangeInMinutes int64
}
```

config_types.go hosted with ❤ by GitHub view raw

config/types.go

```go
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object
// +k8s:defaulter-gen=true

// NetworkTrafficArgs holds arguments used to configure NetworkTraffic plugin.
type NetworkTrafficArgs struct {
	metav1.TypeMeta `json:",inline"`

	// Address of the Prometheus Server
	Address *string `json:"prometheusAddress,omitempty"`
	// NetworkInterface to be monitored, assume that nodes OS is homogeneous
	NetworkInterface *string `json:"networkInterface,omitempty"`
	// TimeRangeInMinutes used to aggregate the network metrics
	TimeRangeInMinutes *int64 `json:"timeRangeInMinutes,omitempty"`
}
```

v1beta1_types.go hosted with ❤ by GitHub view raw

config/v1beta1/types.go

With the structs added, we need to execute the `hack/update-codegen.sh` script. It will update the generated files with functions as `DeepCopy` for the added structures.

Furthermore, we will add a new function `SetDefaultNetworkTrafficArgs` in the `config/v1beta1/defaults.go`. The function will set the default values for the

`NetworkInterface` and `TimeRangeInMinutes` values, but `Address` still needs to be provided.

```go
// SetDefaultNetworkTrafficArgs sets the default parameters for the NetworkTraffic plugin
func SetDefaultNetworkTrafficArgs(args *NetworkTrafficArgs) {
	if args.TimeRangeInMinutes == nil {
		defaultTime := int64(5)
		args.TimeRangeInMinutes = &defaultTime
	}

	if args.NetworkInterface == nil || *args.NetworkInterface == "" {
		netInterface := "ens192"
		args.NetworkInterface = &netInterface
	}
}
```

defaults.go hosted with ❤ by GitHub view raw

Default values for plugin arguments

To finish the default values configuration, we need to make sure the function above is registered in the v1beta1 schema. Thus, make sure that it is registered in the file `pkg/apis/config/v1beta1/zz_generated.defaults.go`.

```go
package v1beta1

import (
	runtime "k8s.io/apimachinery/pkg/runtime"
)

// RegisterDefaults adds defaulters functions to the given scheme.
// Public to allow building arbitrary schemes.
// All generated defaulters are covering - they call all nested defaulters.
func RegisterDefaults(scheme *runtime.Scheme) error {
	scheme.AddTypeDefaultingFunc(&NetworkTrafficArgs{}, func(obj interface{}) {
		SetObjectDefaultNetworkTrafficArgs(obj.(*NetworkTrafficArgs))
	})

	return nil
}

func SetObjectDefaultNetworkTrafficArgs(in *NetworkTrafficArgs) {
	SetDefaultNetworkTrafficArgs(in)
}
```

zz_generated.default.go hosted with ❤ by GitHub view raw

Registering Plugin and Configuration

Now that the arguments structure is defined, our Plugin is ready. However, we still need to register the plugin and the configuration in the scheduler framework.

The scheduler-plugins project already has a couple plugins registered which makes things a bit easier as we have examples. The registration for the plugin configuration is placed under `pkg/apis/config`. In the file `register.go` we need to add the `NetworkTrafficArgs` in the call to the `AddKnownTypes` function. The same needs to be done in the `pkg/apis/config/v1beta1/register.go` file. With both files changed, the configuration registration is done.

Next, we move to the plugin registration, which is done in the `cmd/scheduler/main.go` file. In the *main* function, the `NetworkTraffic` plugin name and constructor need to be provided as arguments to the `NewSchedulerCommand`. It should look like this:

```
command := app.NewSchedulerCommand(
  app.WithPlugin(networktraffic.Name, networktraffic.New),
)
```

Also, notice that in the in the `main.go` file we have the import of `sigs.k8s.io/scheduler-plugins/pkg/apis/config/scheme`, which initializes the scheme with all configurations we have introduced in the `pkg/apis/config` files.

With that we are done from a code perspective. The full implementation can be found here, it also includes a couple of unit tests, so check it out!

Deploying and using the Plugin

Now that we have the plugin done, we can deploy it in our K8S cluster and start using it. In the scheduler-plugins repository, there is a documentation on how to do it, check it here. We would basically need to adapt those steps with the Plugin we just implemented.

Nonetheless, before applying the changes to the cluster, make sure that you have build the scheduler container image and pushed it to a container registry which is accessible from your kubernetes. I won't go into the details as it will differ based to the environment used. You can check the Makefile as well, as there are some commands to build and push the image and also this development doc may help you.

As our plugin doesn't introduce any CRD, a couple steps in the scheduler-plugins install doc can be skipped. As I mentioned, I am using a cluster created with kubespray with HA. Therefore, I will need to repeat the following steps on each control plane node.

1. Log into the control plane node.

2. Backup `kube-scheduler.yaml`

```
cp /etc/kubernetes/manifests/kube-scheduler.yaml
/etc/kubernetes/kube-scheduler.yaml
```

3. Create `/etc/kubernetes/networktraffic-config.yaml` and change the values according to your environment.

```yaml
apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration
clientConnection:
  kubeconfig: "/etc/kubernetes/scheduler.conf"
profiles:
- schedulerName: default-scheduler
  plugins:
    score:
      enabled:
      - name: NetworkTraffic
      disabled:
      - name: "*"
  pluginConfig:
  - name: NetworkTraffic
    args:
      prometheusAddress: "http://prometheus-1616380099-server.monitor"
      networkInterface: "ens192"
      timeRangeInMinutes: 3
```


4. Modify `/etc/kubernetes/manifests/kube-scheduler.yaml` to run scheduler-plugins with Network Traffic. The changes we have made are:

- Add the command arg `--config=/etc/kubernetes/networktraffic-config.yaml`.
- Change the `image` name.

- Add a `volume` pointing to the configuration absolute path.
- Add a `volumeMount` to make the configuration available to the scheduler pod.

Check the example below:

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    component: kube-scheduler
    tier: control-plane
  name: kube-scheduler
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-scheduler
    - --authentication-kubeconfig=/etc/kubernetes/scheduler.conf
    - --authorization-kubeconfig=/etc/kubernetes/scheduler.conf
    - --bind-address=0.0.0.0
    - --kubeconfig=/etc/kubernetes/scheduler.conf
    - --leader-elect=true
    - --leader-elect-lease-duration=15s
    - --leader-elect-renew-deadline=10s
    - --port=0
    - --config=/etc/kubernetes/networktraffic-config.yaml
    image: <YOUR_CONTAINER_REGISTRY>/scheduler-plugins/kube-scheduler:<YOUR_TAG>
    imagePullPolicy: Always
    livenessProbe:
      failureThreshold: 8
      httpGet:
        path: /healthz
        port: 10259
        scheme: HTTPS
      initialDelaySeconds: 10
      periodSeconds: 10
      timeoutSeconds: 15
    name: kube-scheduler
    resources:
      requests:
        cpu: 100m
    startupProbe:
      failureThreshold: 30
      httpGet:
        path: /healthz
        port: 10259
        scheme: HTTPS
      initialDelaySeconds: 10
      periodSeconds: 10
      timeoutSeconds: 15
    volumeMounts:
    - mountPath: /etc/kubernetes/scheduler.conf
      name: kubeconfig
```

```yaml
        name: kubeconfig
        readOnly: true
      - mountPath: /etc/kubernetes/networktraffic-config.yaml
        name: networktraffic-config
        readOnly: true
    hostNetwork: true
    priorityClassName: system-node-critical
    volumes:
    - hostPath:
        path: /etc/kubernetes/scheduler.conf
        type: FileOrCreate
      name: kubeconfig
    - hostPath:
        path: /etc/kubernetes/networktraffic-config.yaml
        type: File
      name: networktraffic-config
```

Now, we can start taking advantage of our custom plugin. Once you check the logs of the running pod, you should see lines with the node bandwidth returned from prometheus and you can make sure the behavior is as expected. Below, we can see that `node4` correctly has the higher score, as it is the node with less network traffic:

```
networktraffic.go:54] [NetworkTraffic] node 'node5' bandwidth: 312528236767
networktraffic.go:54] [NetworkTraffic] node 'node3' bandwidth: 327333347411
networktraffic.go:54] [NetworkTraffic] node 'node1' bandwidth: 321718404122
networktraffic.go:54] [NetworkTraffic] node 'node4' bandwidth: 224935743744
networktraffic.go:54] [NetworkTraffic] node 'node6' bandwidth: 415270171795
networktraffic.go:54] [NetworkTraffic] node 'node2' bandwidth: 270270915134
networktraffic.go:74] [NetworkTraffic] Nodes final score: [{node5 25} {node6 0} {node1 23} {node3 22} {node4 46} {node2 35}]
```

Hope this post is useful to you and feel free to give feedbacks on the comments, they are very appreciated!

References

1: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

2: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

3: https://kubernetes.io/docs/reference/scheduling/config/#profiles

4: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/#extension-points

5: https://kubernetes.io/docs/reference/scheduling/config/#multiple-profiles

6: https://github.com/kubernetes/enhancements/blob/master/keps/sig-scheduling/624-scheduling-framework/README.md

7: https://github.com/kubernetes/enhancements/blob/master/keps/sig-scheduling/624-scheduling-framework/README.md#custom-scheduler-plugins-out-of-tree

8: https://github.com/kubernetes/enhancements/blob/master/keps/sig-scheduling/624-scheduling-framework/README.md#optional-args

9: https://github.com/kubernetes/enhancements/blob/master/keps/sig-scheduling/624-scheduling-framework/README.md#configuring-plugins

Kubernetes | Kube Scheduler | K8s | Kubernetes Administration | Golang

Written by Julio Renner

More from Julio Renner

Julio Renner

Teoria da Janela Quebrada Desenvolvimento de Software

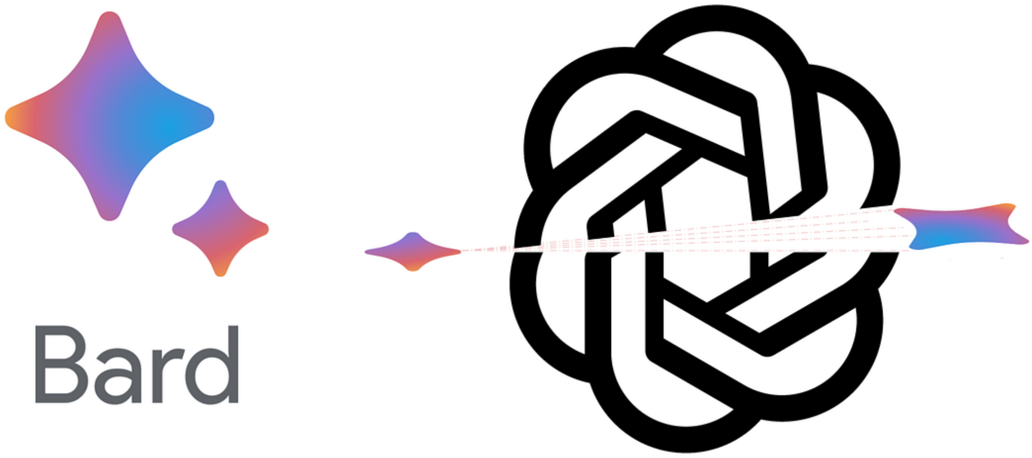
A ideia deste post é relacionar a Teoria da Janela Quebrada com práticas do desenvolvimento de software. Onde, muitas vezes, a...

3 min read · Jun 30, 2018

8 2

See all from Julio Renner

Recommended from Medium



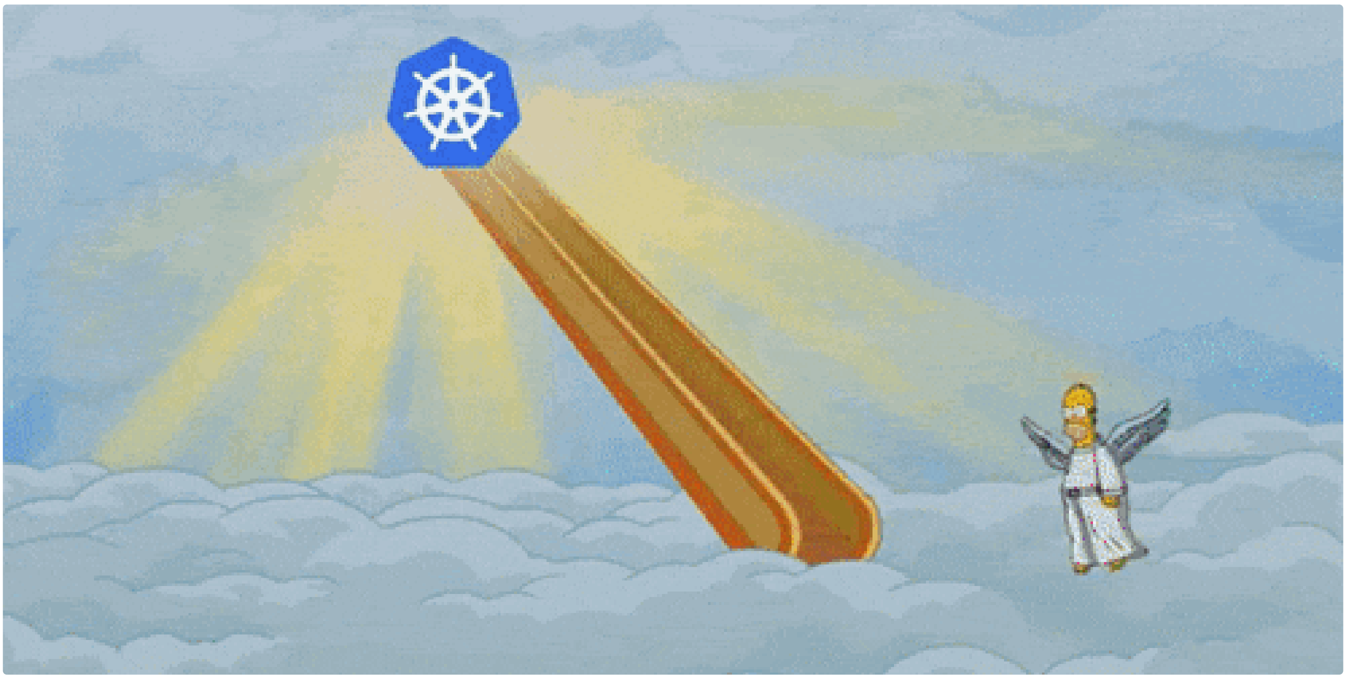
AL Anany

The ChatGPT Hype Is Over — Now Watch How Google Will Kill ChatGPT.

It never happens instantly. The business game is longer than you know.

· 6 min read · Sep 2

13.3K 405



Matteo Bianchi

2023 DevOps is terrible.

My analysis of modern DevOps evolution into Platform Engineering. Just a new trend or a revolution in the IT industry?

7 min read · Sep 21

Lists

General Coding Knowledge

20 stories · 431 saves

Natural Language Processing

698 stories · 309 saves

Wenqi Glantz in Better Programming

How to Use GitHub Packages as Artifacts Registry

Publish artifacts to and install artifacts from GitHub Packages for both Node.js and Java apps

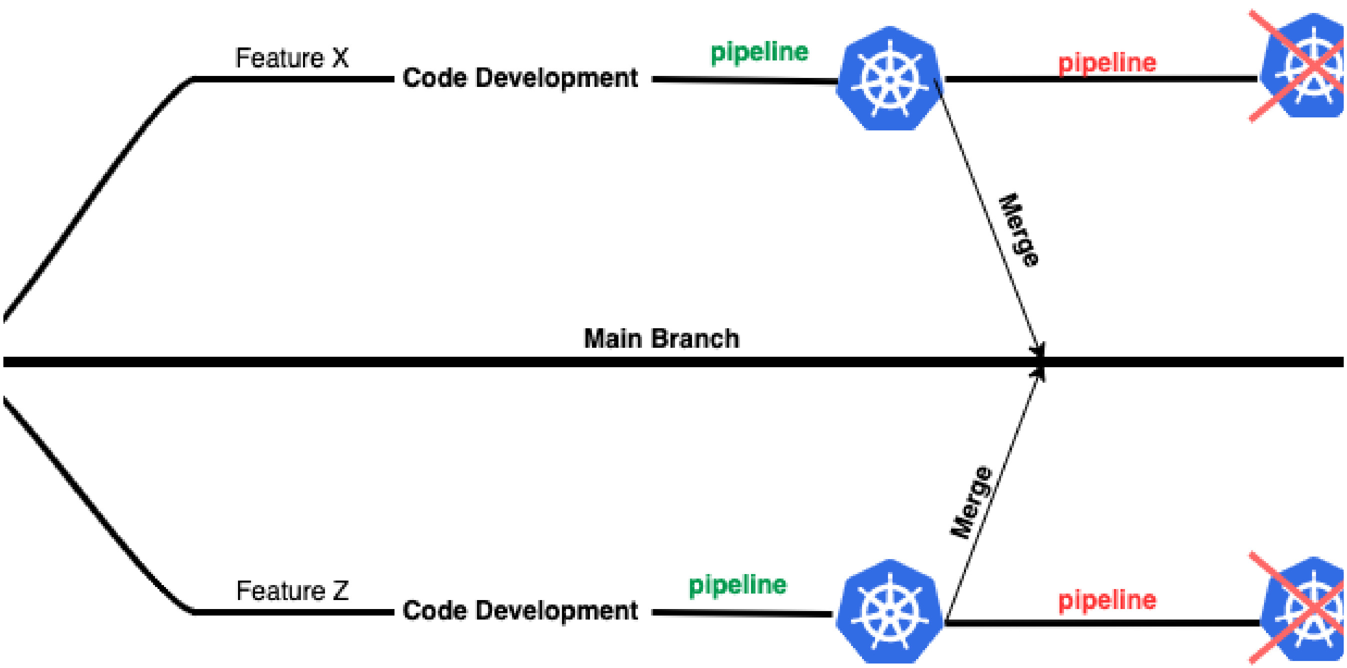
· 11 min read · Apr 26

Bacchus Jackson (BJax)

Helix: Setup for Markdown

Helix is fantastic, there's almost no need to sing it's praises these days. However, with all new tools, there are growing pains. It's...

5 min read · Sep 18



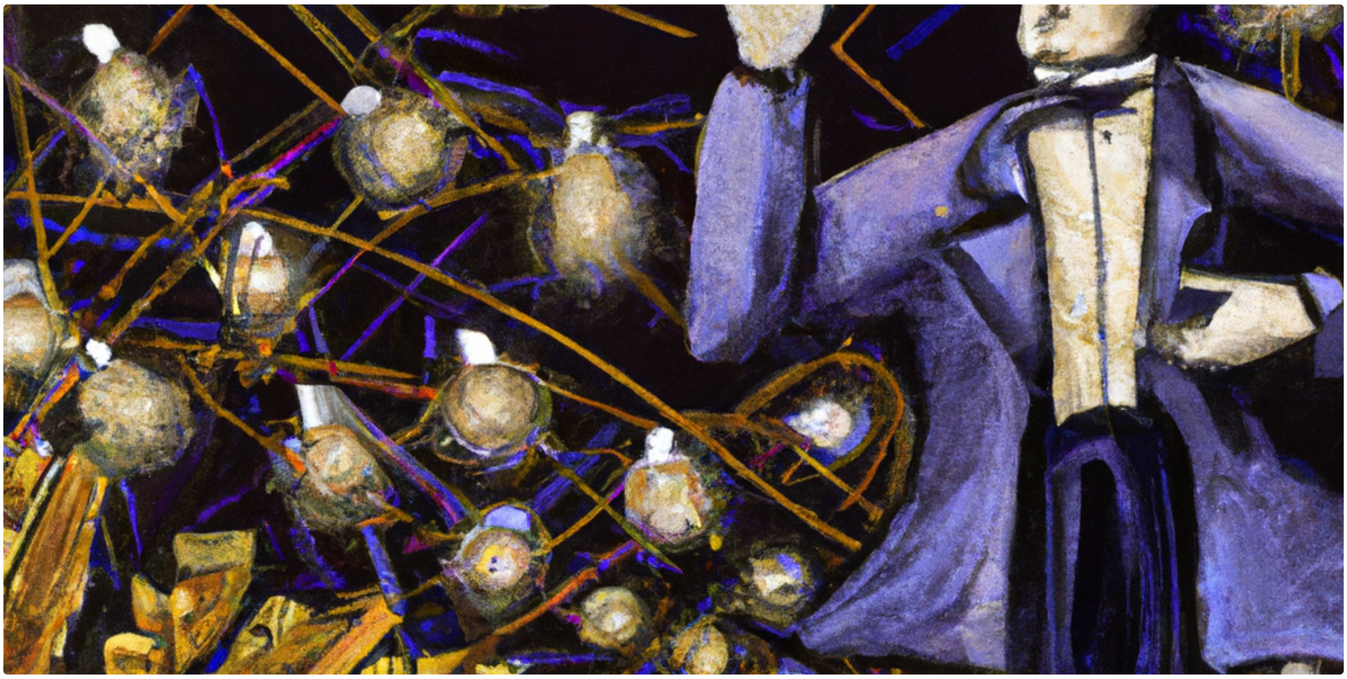
AmrAlaaYassen

Ephemeral Kubernetes Environments: A Cost-Effective Solution for Streamlining Minor Environments...

Introduction

14 min read · May 2

81 1



Daniel Specht in hurb.engineering

Building a Task Orchestrator with Python and Graph Algorithms: A Fun and Practical Guide

Understanding how tools work under the hood is not just useful from a particle/professional perspective, but quite interesting and fun as...

9 min read · May 12

26

See more recommendations