Troubleshooting Clusters

Debugging common cluster issues.

This doc is about cluster troubleshooting; we assume you have already ruled out your application as the root cause of the problem you are experiencing. See the application troubleshooting guide for tips on application debugging. You may also visit the troubleshooting overview document for more information.

Listing your cluster

The first thing to debug in your cluster is if your nodes are all registered correctly.

Run the following command:

```
kubectl get nodes
```

And verify that all of the nodes you expect to see are present and that they are all in the `Ready` state.

To get detailed information about the overall health of your cluster, you can run:

```
kubectl cluster-info dump
```

Example: debugging a down/unreachable node

Sometimes when debugging it can be useful to look at the status of a node -- for example, because you've noticed strange behavior of a Pod that's running on the node, or to find out why a Pod won't schedule onto the node. As with Pods, you can use `kubectl describe node` and `kubectl get node -o yaml` to retrieve detailed information about nodes. For example, here's what you'll see if a node is down (disconnected from the network, or kubelet dies and won't restart, etc.). Notice the events that show the node is NotReady, and also notice that the pods are no longer running (they are evicted after five minutes of NotReady status).

```
kubectl get nodes
```

```
NAME                     STATUS       ROLES     AGE     VERSION
kube-worker-1            NotReady     <none>    1h      v1.23.3
kubernetes-node-bols     Ready        <none>    1h      v1.23.3
kubernetes-node-st6x     Ready        <none>    1h      v1.23.3
kubernetes-node-unaj     Ready        <none>    1h      v1.23.3
```

```
kubectl describe node kube-worker-1
```

```
Name:               kube-worker-1
Roles:              <none>
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=kube-worker-1
                    kubernetes.io/os=linux
Annotations:        kubeadm.alpha.kubernetes.io/cri-socket: /run/containerd/containerd.sock
                    node.alpha.kubernetes.io/ttl: 0
                    volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp:  Thu, 17 Feb 2022 16:46:30 -0500
Taints:             node.kubernetes.io/unreachable:NoExecute
                    node.kubernetes.io/unreachable:NoSchedule
Unschedulable:      false
Lease:
  HolderIdentity:  kube-worker-1
  AcquireTime:     <unset>
  RenewTime:       Thu, 17 Feb 2022 17:13:09 -0500
Conditions:
  Type                 Status    LastHeartbeatTime                 LastTransitionTime                Reason
  ----                 ------    -----------------                 ------------------                ------
  NetworkUnavailable   False     Thu, 17 Feb 2022 17:09:13 -0500   Thu, 17 Feb 2022 17:09:13 -0500   WeaveIsU
  MemoryPressure       Unknown   Thu, 17 Feb 2022 17:12:40 -0500   Thu, 17 Feb 2022 17:13:52 -0500   NodeStat
  DiskPressure         Unknown   Thu, 17 Feb 2022 17:12:40 -0500   Thu, 17 Feb 2022 17:13:52 -0500   NodeStat
  PIDPressure          Unknown   Thu, 17 Feb 2022 17:12:40 -0500   Thu, 17 Feb 2022 17:13:52 -0500   NodeStat
  Ready                Unknown   Thu, 17 Feb 2022 17:12:40 -0500   Thu, 17 Feb 2022 17:13:52 -0500   NodeStat
Addresses:
  InternalIP:  192.168.0.113
  Hostname:    kube-worker-1
Capacity:
  cpu:                2
  ephemeral-storage:  15372232Ki
  hugepages-2Mi:      0
  memory:             2025188Ki
  pods:               110
Allocatable:
  cpu:                2
  ephemeral-storage:  14167048988
  hugepages-2Mi:      0
  memory:             1922788Ki
  pods:               110
System Info:
  Machine ID:                 9384e2927f544209b5d7b67474bbf92b
  System UUID:                aa829ca9-73d7-064d-9019-df07404ad448
  Boot ID:                    5a295a03-aaca-4340-af20-1327fa5dab5c
  Kernel Version:             5.13.0-28-generic
  OS Image:                   Ubuntu 21.10
  Operating System:           linux
  Architecture:               amd64
  Container Runtime Version:  containerd://1.5.9
  Kubelet Version:            v1.23.3
  Kube-Proxy Version:         v1.23.3
Non-terminated Pods:          (4 in total)
  Namespace                   Name                                 CPU Requests  CPU Limits  Memory Requests
  ---------                   ----                                 ------------  ----------  ---------------
  default                     nginx-deployment-67d4bdd6f5-cx2nz    500m (25%)    500m (25%)  128Mi (6%)
  default                     nginx-deployment-67d4bdd6f5-w6kd7    500m (25%)    500m (25%)  128Mi (6%)
  kube-system                 kube-proxy-dnxbz                     0 (0%)        0 (0%)      0 (0%)
  kube-system                 weave-net-gjxxp                      100m (5%)     0 (0%)      200Mi (10%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests     Limits
  --------           --------     ------
  cpu                1100m (55%)  1 (50%)
  memory             456Mi (24%)  256Mi (13%)
  ephemeral-storage  0 (0%)       0 (0%)
  hugepages-2Mi      0 (0%)       0 (0%)
Events:
```

```
...
```

```
kubectl get node kube-worker-1 -o yaml
```

```yaml
apiVersion: v1
kind: Node
metadata:
  annotations:
    kubeadm.alpha.kubernetes.io/cri-socket: /run/containerd/containerd.sock
    node.alpha.kubernetes.io/ttl: "0"
    volumes.kubernetes.io/controller-managed-attach-detach: "true"
  creationTimestamp: "2022-02-17T21:46:30Z"
  labels:
    beta.kubernetes.io/arch: amd64
    beta.kubernetes.io/os: linux
    kubernetes.io/arch: amd64
    kubernetes.io/hostname: kube-worker-1
    kubernetes.io/os: linux
  name: kube-worker-1
  resourceVersion: "4026"
  uid: 98efe7cb-2978-4a0b-842a-1a7bf12c05f8
spec: {}
status:
  addresses:
  - address: 192.168.0.113
    type: InternalIP
  - address: kube-worker-1
    type: Hostname
  allocatable:
    cpu: "2"
    ephemeral-storage: "14167048988"
    hugepages-2Mi: "0"
    memory: 1922788Ki
    pods: "110"
  capacity:
    cpu: "2"
    ephemeral-storage: 15372232Ki
    hugepages-2Mi: "0"
    memory: 2025188Ki
    pods: "110"
  conditions:
  - lastHeartbeatTime: "2022-02-17T22:20:32Z"
    lastTransitionTime: "2022-02-17T22:20:32Z"
    message: Weave pod has set this
    reason: WeaveIsUp
    status: "False"
    type: NetworkUnavailable
  - lastHeartbeatTime: "2022-02-17T22:20:15Z"
    lastTransitionTime: "2022-02-17T22:13:25Z"
    message: kubelet has sufficient memory available
    reason: KubeletHasSufficientMemory
    status: "False"
    type: MemoryPressure
  - lastHeartbeatTime: "2022-02-17T22:20:15Z"
    lastTransitionTime: "2022-02-17T22:13:25Z"
    message: kubelet has no disk pressure
    reason: KubeletHasNoDiskPressure
    status: "False"
    type: DiskPressure
  - lastHeartbeatTime: "2022-02-17T22:20:15Z"
    lastTransitionTime: "2022-02-17T22:13:25Z"
    message: kubelet has sufficient PID available
    reason: KubeletHasSufficientPID
    status: "False"
    type: PIDPressure
  - lastHeartbeatTime: "2022-02-17T22:20:15Z"
    lastTransitionTime: "2022-02-17T22:15:15Z"
    message: kubelet is posting ready status. AppArmor enabled
    reason: KubeletReady
    status: "True"
    type: Ready
  daemonEndpoints:
```

```yaml
    kubeletEndpoint:
      Port: 10250
  nodeInfo:
    architecture: amd64
    bootID: 22333234-7a6b-44d4-9ce1-67e31dc7e369
    containerRuntimeVersion: containerd://1.5.9
    kernelVersion: 5.13.0-28-generic
    kubeProxyVersion: v1.23.3
    kubeletVersion: v1.23.3
    machineID: 9384e2927f544209b5d7b67474bbf92b
    operatingSystem: linux
    osImage: Ubuntu 21.10
    systemUUID: aa829ca9-73d7-064d-9019-df07404ad448
```

Looking at logs

For now, digging deeper into the cluster requires logging into the relevant machines. Here are the locations of the relevant log files. On systemd-based systems, you may need to use `journalctl` instead of examining log files.

Control Plane nodes

- `/var/log/kube-apiserver.log` - API Server, responsible for serving the API
- `/var/log/kube-scheduler.log` - Scheduler, responsible for making scheduling decisions
- `/var/log/kube-controller-manager.log` - a component that runs most Kubernetes built-in controllers, with the notable exception of scheduling (the kube-scheduler handles scheduling).

Worker Nodes

- `/var/log/kubelet.log` - logs from the kubelet, responsible for running containers on the node
- `/var/log/kube-proxy.log` - logs from `kube-proxy`, which is responsible for directing traffic to Service endpoints

Cluster failure modes

This is an incomplete list of things that could go wrong, and how to adjust your cluster setup to mitigate the problems.

Contributing causes

- VM(s) shutdown
- Network partition within cluster, or between cluster and users
- Crashes in Kubernetes software
- Data loss or unavailability of persistent storage (e.g. GCE PD or AWS EBS volume)
- Operator error, for example misconfigured Kubernetes software or application software

Specific scenarios

- API server VM shutdown or apiserver crashing
  - Results
    - unable to stop, update, or start new pods, services, replication controller
    - existing pods and services should continue to work normally, unless they depend on the Kubernetes API
- API server backing storage lost
  - Results
    - the kube-apiserver component fails to start successfully and become healthy
    - kubelets will not be able to reach it but will continue to run the same pods and provide the same service proxying
    - manual recovery or recreation of apiserver state necessary before apiserver is restarted
- Supporting services (node controller, replication controller manager, scheduler, etc) VM shutdown or crashes
  - currently those are colocated with the apiserver, and their unavailability has similar consequences as apiserver
  - in future, these will be replicated as well and may not be co-located
  - they do not have their own persistent state
- Individual node (VM or physical machine) shuts down
  - Results
    - pods on that Node stop running
- Network partition
  - Results

        - partition A thinks the nodes in partition B are down; partition B thinks the apiserver is down. (Assuming the master VM ends up in partition A.)
- Kubelet software fault
    - Results
        - crashing kubelet cannot start new pods on the node
        - kubelet might delete the pods or not
        - node marked unhealthy
        - replication controllers start new pods elsewhere
- Cluster operator error
    - Results
        - loss of pods, services, etc
        - lost of apiserver backing store
        - users unable to read API
        - etc.

Mitigations

- Action: Use IaaS provider's automatic VM restarting feature for IaaS VMs
    - Mitigates: Apiserver VM shutdown or apiserver crashing
    - Mitigates: Supporting services VM shutdown or crashes
- Action: Use IaaS providers reliable storage (e.g. GCE PD or AWS EBS volume) for VMs with apiserver+etcd
    - Mitigates: Apiserver backing storage lost
- Action: Use [high-availability](#) configuration
    - Mitigates: Control plane node shutdown or control plane components (scheduler, API server, controller-manager) crashing
        - Will tolerate one or more simultaneous node or component failures
    - Mitigates: API server backing storage (i.e., etcd's data directory) lost
        - Assumes HA (highly-available) etcd configuration
- Action: Snapshot apiserver PDs/EBS-volumes periodically
    - Mitigates: Apiserver backing storage lost
    - Mitigates: Some cases of operator error
    - Mitigates: Some cases of Kubernetes software fault
- Action: use replication controller and services in front of pods
    - Mitigates: Node shutdown
    - Mitigates: Kubelet software fault
- Action: applications (containers) designed to tolerate unexpected restarts
    - Mitigates: Node shutdown
    - Mitigates: Kubelet software fault

What's next

- Learn about the metrics available in the Resource Metrics Pipeline
- Discover additional tools for monitoring resource usage
- Use Node Problem Detector to monitor node health
- Use `crictl` to debug Kubernetes nodes
- Get more information about Kubernetes auditing
- Use `telepresence` to develop and debug services locally

1 - Resource metrics pipeline

For Kubernetes, the *Metrics API* offers a basic set of metrics to support automatic scaling and similar use cases. This API makes information available about resource usage for node and pod, including metrics for CPU and memory. If you deploy the Metrics API into your cluster, clients of the Kubernetes API can then query for this information, and you can use Kubernetes' access control mechanisms to manage permissions to do so.

The HorizontalPodAutoscaler (HPA) and VerticalPodAutoscaler (VPA) use data from the metrics API to adjust workload replicas and resources to meet customer demand.

You can also view the resource metrics using the `kubectl top` command.

> **Note:** The Metrics API, and the metrics pipeline that it enables, only offers the minimum CPU and memory metrics to enable automatic scaling using HPA and / or VPA. If you would like to provide a more complete set of metrics, you can complement the simpler Metrics API by deploying a second metrics pipeline that uses the *Custom Metrics API*.

Figure 1 illustrates the architecture of the resource metrics pipeline.

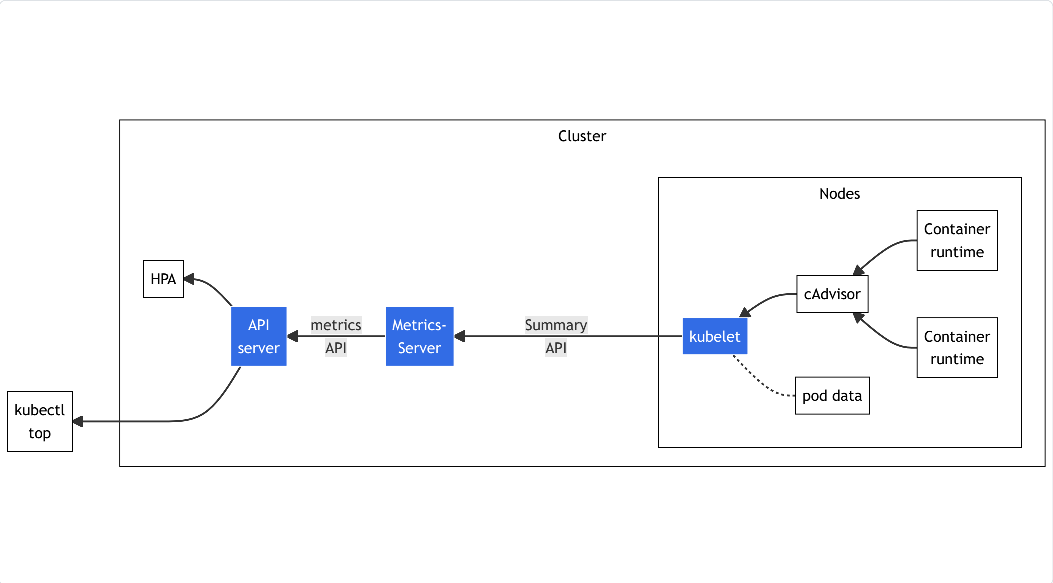

Figure 1. Resource Metrics Pipeline

The architecture components, from right to left in the figure, consist of the following:

- cAdvisor: Daemon for collecting, aggregating and exposing container metrics included in Kubelet.
- kubelet: Node agent for managing container resources. Resource metrics are accessible using the `/metrics/resource` and `/stats` kubelet API endpoints.
- Summary API: API provided by the kubelet for discovering and retrieving per-node summarized stats available through the `/stats` endpoint.
- metrics-server: Cluster addon component that collects and aggregates resource metrics pulled from each kubelet. The API server serves Metrics API for use by HPA, VPA, and by the `kubectl top` command. Metrics Server is a reference implementation of the Metrics API.
- Metrics API: Kubernetes API supporting access to CPU and memory used for workload autoscaling. To make this work in your cluster, you need an API extension server that provides the Metrics API.

  > **Note:** cAdvisor supports reading metrics from cgroups, which works with typical container runtimes on Linux. If you use a container runtime that uses another resource isolation mechanism, for example virtualization, then that container runtime must support CRI Container Metrics in order for metrics to be available to the kubelet.

Metrics API

FEATURE STATE: `Kubernetes 1.8 [beta]`

The metrics-server implements the Metrics API. This API allows you to access CPU and memory usage for the nodes and pods in your cluster. Its primary role is to feed resource usage metrics to K8s autoscaler components.

Here is an example of the Metrics API request for a `minikube` node piped through `jq` for easier reading:

```
kubectl get --raw "/apis/metrics.k8s.io/v1beta1/nodes/minikube" | jq '.'
```

Here is the same API call using `curl`:

```
curl http://localhost:8080/apis/metrics.k8s.io/v1beta1/nodes/minikube
```

Sample response:


```
{
  "kind": "NodeMetrics",
  "apiVersion": "metrics.k8s.io/v1beta1",
  "metadata": {
    "name": "minikube",
    "selfLink": "/apis/metrics.k8s.io/v1beta1/nodes/minikube",
    "creationTimestamp": "2022-01-27T18:48:43Z"
  },
  "timestamp": "2022-01-27T18:48:33Z",
  "window": "30s",
  "usage": {
    "cpu": "487558164n",
    "memory": "732212Ki"
  }
}
```


Here is an example of the Metrics API request for a `kube-scheduler-minikube` pod contained in the `kube-system` namespace and piped through `jq` for easier reading:

```
kubectl get --raw "/apis/metrics.k8s.io/v1beta1/namespaces/kube-system/pods/kube-scheduler-minikube" | jq '.'
```

Here is the same API call using `curl`:

```
curl http://localhost:8080/apis/metrics.k8s.io/v1beta1/namespaces/kube-system/pods/kube-scheduler-minikube
```

Sample response:

```
{
  "kind": "PodMetrics",
  "apiVersion": "metrics.k8s.io/v1beta1",
  "metadata": {
    "name": "kube-scheduler-minikube",
    "namespace": "kube-system",
    "selfLink": "/apis/metrics.k8s.io/v1beta1/namespaces/kube-system/pods/kube-scheduler-minikube",
    "creationTimestamp": "2022-01-27T19:25:00Z"
  },
  "timestamp": "2022-01-27T19:24:31Z",
  "window": "30s",
  "containers": [
    {
      "name": "kube-scheduler",
      "usage": {
        "cpu": "9559630n",
        "memory": "22244Ki"
      }
    }
  ]
}
```


The Metrics API is defined in the k8s.io/metrics repository. You must enable the API aggregation layer and register an APIService for the `metrics.k8s.io` API.

To learn more about the Metrics API, see resource metrics API design, the metrics-server repository and the resource metrics API.

> **Note:** You must deploy the metrics-server or alternative adapter that serves the Metrics API to be able to access it.

Measuring resource usage

CPU

CPU is reported as the average core usage measured in cpu units. One cpu, in Kubernetes, is equivalent to 1 vCPU/Core for cloud providers, and 1 hyper-thread on bare-metal Intel processors.

This value is derived by taking a rate over a cumulative CPU counter provided by the kernel (in both Linux and Windows kernels). The time window used to calculate CPU is shown under window field in Metrics API.

To learn more about how Kubernetes allocates and measures CPU resources, see meaning of CPU.

Memory

Memory is reported as the working set, measured in bytes, at the instant the metric was collected.

In an ideal world, the "working set" is the amount of memory in-use that cannot be freed under memory pressure. However, calculation of the working set varies by host OS, and generally makes heavy use of heuristics to produce an estimate.

The Kubernetes model for a container's working set expects that the container runtime counts anonymous memory associated with the container in question. The working set metric typically also includes some cached (file-backed) memory, because the host OS cannot always reclaim pages.

To learn more about how Kubernetes allocates and measures memory resources, see meaning of memory.

Metrics Server

The metrics-server fetches resource metrics from the kubelets and exposes them in the Kubernetes API server through the Metrics API for use by the HPA and VPA. You can also view these metrics using the `kubectl top` command.

The metrics-server uses the Kubernetes API to track nodes and pods in your cluster. The metrics-server queries each node over HTTP to fetch metrics. The metrics-server also builds an internal view of pod metadata, and keeps a cache of pod health. That cached pod health information is available via the extension API that the metrics-server makes available.

For example with an HPA query, the metrics-server needs to identify which pods fulfill the label selectors in the deployment.

The metrics-server calls the kubelet API to collect metrics from each node. Depending on the metrics-server version it uses:

- Metrics resource endpoint `/metrics/resource` in version v0.6.0+ or
- Summary API endpoint `/stats/summary` in older versions

To learn more about the metrics-server, see the metrics-server repository.

You can also check out the following:

- metrics-server design
- metrics-server FAQ
- metrics-server known issues
- metrics-server releases
- Horizontal Pod Autoscaling

Summary API source

The kubelet gathers stats at the node, volume, pod and container level, and emits this information in the Summary API for consumers to read.

Here is an example of a Summary API request for a `minikube` node:

```
kubectl get --raw "/api/v1/nodes/minikube/proxy/stats/summary"
```

Here is the same API call using `curl`:

```
curl http://localhost:8080/api/v1/nodes/minikube/proxy/stats/summary
```

> **Note:** The summary API `/stats/summary` endpoint will be replaced by the `/metrics/resource` endpoint beginning with metrics-server 0.6.x.

2 - Tools for Monitoring Resources

To scale an application and provide a reliable service, you need to understand how the application behaves when it is deployed. You can examine application performance in a Kubernetes cluster by examining the containers, pods, services, and the characteristics of the overall cluster. Kubernetes provides detailed information about an application's resource usage at each of these levels. This information allows you to evaluate your application's performance and where bottlenecks can be removed to improve overall performance.

In Kubernetes, application monitoring does not depend on a single monitoring solution. On new clusters, you can use resource metrics or full metrics pipelines to collect monitoring statistics.

Resource metrics pipeline

The resource metrics pipeline provides a limited set of metrics related to cluster components such as the Horizontal Pod Autoscaler controller, as well as the `kubectl top` utility. These metrics are collected by the lightweight, short-term, in-memory metrics-server and are exposed via the `metrics.k8s.io` API.

metrics-server discovers all nodes on the cluster and queries each node's kubelet for CPU and memory usage. The kubelet acts as a bridge between the Kubernetes master and the nodes, managing the pods and containers running on a machine. The kubelet translates each pod into its constituent containers and fetches individual container usage statistics from the container runtime through the container runtime interface. If you use a container runtime that uses Linux cgroups and namespaces to implement containers, and the container runtime does not publish usage statistics, then the kubelet can look up those statistics directly (using code from cAdvisor). No matter how those statistics arrive, the kubelet then exposes the aggregated pod resource usage statistics through the metrics-server Resource Metrics API. This API is served at `/metrics/resource/v1beta1` on the kubelet's authenticated and read-only ports.

Full metrics pipeline

A full metrics pipeline gives you access to richer metrics. Kubernetes can respond to these metrics by automatically scaling or adapting the cluster based on its current state, using mechanisms such as the Horizontal Pod Autoscaler. The monitoring pipeline fetches metrics from the kubelet and then exposes them to Kubernetes via an adapter by implementing either the `custom.metrics.k8s.io` or `external.metrics.k8s.io` API.

Prometheus, a CNCF project, can natively monitor Kubernetes, nodes, and Prometheus itself. Full metrics pipeline projects that are not part of the CNCF are outside the scope of Kubernetes documentation.

What's next

Learn about additional debugging tools, including:

- Logging
- Monitoring
- Getting into containers via `exec`
- Connecting to containers via proxies
- Connecting to containers via port forwarding
- Inspect Kubernetes node with crictl

3 - Monitor Node Health

Node Problem Detector is a daemon for monitoring and reporting about a node's health. You can run Node Problem Detector as a `DaemonSet` or as a standalone daemon. Node Problem Detector collects information about node problems from various daemons and reports these conditions to the API server as NodeCondition and Event.

To learn how to install and use Node Problem Detector, see Node Problem Detector project documentation.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Killercoda
- Play with Kubernetes

Limitations

- Node Problem Detector only supports file based kernel log. Log tools such as `journald` are not supported.
- Node Problem Detector uses the kernel log format for reporting kernel issues. To learn how to extend the kernel log format, see Add support for another log format.

Enabling Node Problem Detector

Some cloud providers enable Node Problem Detector as an Addon. You can also enable Node Problem Detector with `kubectl` or by creating an Addon pod.

Using kubectl to enable Node Problem Detector

`kubectl` provides the most flexible management of Node Problem Detector. You can overwrite the default configuration to fit it into your environment or to detect customized node problems. For example:

1. Create a Node Problem Detector configuration similar to `node-problem-detector.yaml`:

[debug/node-problem-detector.yaml]

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-problem-detector-v0.1
  namespace: kube-system
  labels:
    k8s-app: node-problem-detector
    version: v0.1
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      k8s-app: node-problem-detector
      version: v0.1
      kubernetes.io/cluster-service: "true"
  template:
    metadata:
      labels:
        k8s-app: node-problem-detector
        version: v0.1
        kubernetes.io/cluster-service: "true"
    spec:
      hostNetwork: true
      containers:
      - name: node-problem-detector
        image: registry.k8s.io/node-problem-detector:v0.1
        securityContext:
          privileged: true
        resources:
          limits:
            cpu: "200m"
            memory: "100Mi"
          requests:
```

```yaml
            cpu: "20m"
            memory: "20Mi"
        volumeMounts:
        - name: log
          mountPath: /log
          readOnly: true
      volumes:
      - name: log
        hostPath:
          path: /var/log/
```

Note: You should verify that the system log directory is right for your operating system distribution.

2. Start node problem detector with `kubectl`:

```
kubectl apply -f https://k8s.io/examples/debug/node-problem-detector.yaml
```

Using an Addon pod to enable Node Problem Detector

If you are using a custom cluster bootstrap solution and don't need to overwrite the default configuration, you can leverage the Addon pod to further automate the deployment.

Create `node-problem-detector.yaml`, and save the configuration in the Addon pod's directory `/etc/kubernetes/addons/node-problem-detector` on a control plane node.

Overwrite the configuration

The default configuration is embedded when building the Docker image of Node Problem Detector.

However, you can use a `ConfigMap` to overwrite the configuration:

1. Change the configuration files in `config/`
2. Create the `ConfigMap` `node-problem-detector-config`:

```
kubectl create configmap node-problem-detector-config --from-file=config/
```

3. Change the `node-problem-detector.yaml` to use the `ConfigMap`:

debug/node-problem-detector-configmap.yaml

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-problem-detector-v0.1
  namespace: kube-system
  labels:
    k8s-app: node-problem-detector
    version: v0.1
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      k8s-app: node-problem-detector
      version: v0.1
      kubernetes.io/cluster-service: "true"
  template:
    metadata:
      labels:
        k8s-app: node-problem-detector
        version: v0.1
        kubernetes.io/cluster-service: "true"
    spec:
      hostNetwork: true
      containers:
      - name: node-problem-detector
        image: registry.k8s.io/node-problem-detector:v0.1
        securityContext:
          privileged: true
        resources:
          limits:
            cpu: "200m"
            memory: "100Mi"
          requests:
```

```yaml
            cpu: "20m"
            memory: "20Mi"
        volumeMounts:
        - name: log
          mountPath: /log
          readOnly: true
        - name: config # Overwrite the config/ directory with ConfigMap volume
          mountPath: /config
          readOnly: true
      volumes:
      - name: log
        hostPath:
          path: /var/log/
      - name: config # Define ConfigMap volume
        configMap:
          name: node-problem-detector-config
```

4. Recreate the Node Problem Detector with the new configuration file:

```
# If you have a node-problem-detector running, delete before recreating
kubectl delete -f https://k8s.io/examples/debug/node-problem-detector.yaml
kubectl apply -f https://k8s.io/examples/debug/node-problem-detector-configmap.yaml
```

> **Note:** This approach only applies to a Node Problem Detector started with `kubectl`.

Overwriting a configuration is not supported if a Node Problem Detector runs as a cluster Addon. The Addon manager does not support `ConfigMap`.

Kernel Monitor

Kernel Monitor is a system log monitor daemon supported in the Node Problem Detector. Kernel monitor watches the kernel log and detects known kernel issues following predefined rules.

The Kernel Monitor matches kernel issues according to a set of predefined rule list in [`config/kernel-monitor.json`](). The rule list is extensible. You can expand the rule list by overwriting the configuration.

Add new NodeConditions

To support a new `NodeCondition`, create a condition definition within the `conditions` field in `config/kernel-monitor.json`, for example:

```json
{
  "type": "NodeConditionType",
  "reason": "CamelCaseDefaultNodeConditionReason",
  "message": "arbitrary default node condition message"
}
```

Detect new problems

To detect new problems, you can extend the `rules` field in `config/kernel-monitor.json` with a new rule definition:

```json
{
  "type": "temporary/permanent",
  "condition": "NodeConditionOfPermanentIssue",
  "reason": "CamelCaseShortReason",
  "message": "regexp matching the issue in the kernel log"
}
```

Configure path for the kernel log device

Check your kernel log path location in your operating system (OS) distribution. The Linux kernel [log device](#) is usually presented as `/dev/kmsg`. However, the log path location varies by OS distribution. The `log` field in `config/kernel-monitor.json` represents the log path inside the container. You can configure the `log` field to match the device path as seen by the Node Problem Detector.

Add support for another log format

Kernel monitor uses the `Translator` plugin to translate the internal data structure of the kernel log. You can implement a new translator for a new log format.

Recommendations and restrictions

It is recommended to run the Node Problem Detector in your cluster to monitor node health. When running the Node Problem Detector, you can expect extra resource overhead on each node. Usually this is fine, because:

- The kernel log grows relatively slowly.
- A resource limit is set for the Node Problem Detector.
- Even under high load, the resource usage is acceptable. For more information, see the Node Problem Detector benchmark result.

4 - Debugging Kubernetes nodes with crictl

FEATURE STATE: `Kubernetes v1.11 [stable]`

`crictl` is a command-line interface for CRI-compatible container runtimes. You can use it to inspect and debug container runtimes and applications on a Kubernetes node. `crictl` and its source are hosted in the cri-tools repository.

Before you begin

`crictl` requires a Linux operating system with a CRI runtime.

Installing crictl

You can download a compressed archive `crictl` from the cri-tools release page, for several different architectures. Download the version that corresponds to your version of Kubernetes. Extract it and move it to a location on your system path, such as `/usr/local/bin/`.

General usage

The `crictl` command has several subcommands and runtime flags. Use `crictl help` or `crictl <subcommand> help` for more details.

You can set the endpoint for `crictl` by doing one of the following:

- Set the `--runtime-endpoint` and `--image-endpoint` flags.
- Set the `CONTAINER_RUNTIME_ENDPOINT` and `IMAGE_SERVICE_ENDPOINT` environment variables.
- Set the endpoint in the configuration file `/etc/crictl.yaml`. To specify a different file, use the `--config=PATH_TO_FILE` flag when you run `crictl`.

> **Note:** If you don't set an endpoint, `crictl` attempts to connect to a list of known endpoints, which might result in an impact to performance.

You can also specify timeout values when connecting to the server and enable or disable debugging, by specifying `timeout` or `debug` values in the configuration file or using the `--timeout` and `--debug` command-line flags.

To view or edit the current configuration, view or edit the contents of `/etc/crictl.yaml`. For example, the configuration when using the `containerd` container runtime would be similar to this:

```
runtime-endpoint: unix:///var/run/containerd/containerd.sock
image-endpoint: unix:///var/run/containerd/containerd.sock
timeout: 10
debug: true
```

To learn more about `crictl`, refer to the `crictl` documentation.

Example crictl commands

The following examples show some `crictl` commands and example output.

> **Warning:** If you use `crictl` to create pod sandboxes or containers on a running Kubernetes cluster, the Kubelet will eventually delete them. `crictl` is not a general purpose workflow tool, but a tool that is useful for debugging.

List pods

List all pods:

```
crictl pods
```

The output is similar to this:

```
POD ID              CREATED              STATE               NAME                         NAMESPACE
926f1b5a1d33a       About a minute ago   Ready               sh-84d7dcf559-4r2gq          default
4dccb216c4adb       About a minute ago   Ready               nginx-65899c769f-wv2gp       default
a86316e96fa89       17 hours ago         Ready               kube-proxy-gblk4             kube-system
919630b8f81f1       17 hours ago         Ready               nvidia-device-plugin-zgbbv   kube-system
```

List pods by name:

```
crictl pods --name nginx-65899c769f-wv2gp
```

The output is similar to this:

```
POD ID              CREATED             STATE               NAME                     NAMESPACE           ATTE
4dccb216c4adb       2 minutes ago       Ready               nginx-65899c769f-wv2gp   default             0
```

List pods by label:

```
crictl pods --label run=nginx
```

The output is similar to this:

```
POD ID              CREATED             STATE               NAME                     NAMESPACE           ATTE
4dccb216c4adb       2 minutes ago       Ready               nginx-65899c769f-wv2gp   default             0
```

List images

List all images:

```
crictl images
```

The output is similar to this:

```
IMAGE                                     TAG                 IMAGE ID            SIZE
busybox                                   latest              8c811b4aec35f       1.15MB
k8s-gcrio.azureedge.net/hyperkube-amd64   v1.10.3             e179bbfe5d238       665MB
k8s-gcrio.azureedge.net/pause-amd64       3.1                 da86e6ba6ca19       742kB
nginx                                     latest              cd5239a0906a6       109MB
```

List images by repository:

```
crictl images nginx
```

The output is similar to this:

```
IMAGE               TAG                 IMAGE ID            SIZE
nginx               latest              cd5239a0906a6       109MB
```

Only list image IDs:

```
crictl images -q
```

The output is similar to this:

```
sha256:8c811b4aec35f259572d0f79207bc0678df4c736eeec50bc9fec37ed936a472a
sha256:e179bbfe5d238de6069f3b03fccbecc3fb4f2019af741bfff1233c4d7b2970c5
sha256:da86e6ba6ca197bf6bc5e9d900febd906b133eaa4750e6bed647b0fbe50ed43e
sha256:cd5239a0906a6ccf0562354852fae04bc5b52d72a2aff9a871ddb6bd57553569
```

List containers

List all containers:

```
crictl ps -a
```

The output is similar to this:

```
CONTAINER ID        IMAGE
1f73f2d81bf98       busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea4f41046e0f37d47
9c5951df22c78       busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea4f41046e0f37d47
87d3992f84f74       nginx@sha256:d0a8828cccb73397acb0073bf34f4d7d8aa315263f1e7806bf8c55d8ac139d5f
1941fb4da154f       k8s-gcrio.azureedge.net/hyperkube-amd64@sha256:00d814b1f7763f4ab5be80c58e98140dfc69df107f
```

List running containers:

```
crictl ps
```

The output is similar to this:

```
CONTAINER ID        IMAGE
1f73f2d81bf98       busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea4f41046e0f37d47
87d3992f84f74       nginx@sha256:d0a8828cccb73397acb0073bf34f4d7d8aa315263f1e7806bf8c55d8ac139d5f
1941fb4da154f       k8s-gcrio.azureedge.net/hyperkube-amd64@sha256:00d814b1f7763f4ab5be80c58e98140dfc69df107f
```

Execute a command in a running container

```
crictl exec -i -t 1f73f2d81bf98 ls
```

The output is similar to this:

```
bin   dev   etc   home  proc  root  sys   tmp   usr   var
```

Get a container's logs

Get all container logs:

```
crictl logs 87d3992f84f74
```

The output is similar to this:

```
10.240.0.96 - - [06/Jun/2018:02:45:49 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"
10.240.0.96 - - [06/Jun/2018:02:45:50 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"
10.240.0.96 - - [06/Jun/2018:02:45:51 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"
```

Get only the latest `N` lines of logs:

```
crictl logs --tail=1 87d3992f84f74
```

The output is similar to this:

```
10.240.0.96 - - [06/Jun/2018:02:45:51 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"
```

Run a pod sandbox

Using `crictl` to run a pod sandbox is useful for debugging container runtimes. On a running Kubernetes cluster, the sandbox will eventually be stopped and deleted by the Kubelet.

1. Create a JSON file like the following:

   ```
   {
     "metadata": {
       "name": "nginx-sandbox",
       "namespace": "default",
       "attempt": 1,
       "uid": "hdishd83djaidwnduwk28bcsb"
     },
     "log_directory": "/tmp",
     "linux": {
     }
   }
   ```

2. Use the `crictl runp` command to apply the JSON and run the sandbox.

   ```
   crictl runp pod-config.json
   ```

   The ID of the sandbox is returned.

Create a container

Using `crictl` to create a container is useful for debugging container runtimes. On a running Kubernetes cluster, the sandbox will eventually be stopped and deleted by the Kubelet.

1. Pull a busybox image

```
crictl pull busybox
```

```
Image is up to date for busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea4f41046e0f37d47
```

2. Create configs for the pod and the container:

   Pod config:

```
{
  "metadata": {
    "name": "busybox-sandbox",
    "namespace": "default",
    "attempt": 1,
    "uid": "aewi4aeThua7ooShohbo1phoj"
  },
  "log_directory": "/tmp",
  "linux": {
  }
}
```

   Container config:

```
{
  "metadata": {
    "name": "busybox"
  },
  "image":{
    "image": "busybox"
  },
  "command": [
    "top"
  ],
  "log_path":"busybox.log",
  "linux": {
  }
}
```

3. Create the container, passing the ID of the previously-created pod, the container config file, and the pod config file. The ID of the container is returned.

```
crictl create f84dd361f8dc51518ed291fbadd6db537b0496536c1d2d6c05ff943ce8c9a54f container-config.json pod
```

4. List all containers and verify that the newly-created container has its state set to `Created`.

```
crictl ps -a
```

   The output is similar to this:

```
CONTAINER ID        IMAGE               CREATED             STATE               NAME                ATTE
3e025dd50a72d       busybox             32 seconds ago      Created             busybox             0
```

Start a container

To start a container, pass its ID to `crictl start`:

```
crictl start 3e025dd50a72d956c4f14881fbb5b1080c9275674e95fb67f965f6478a957d60
```

The output is similar to this:

```
3e025dd50a72d956c4f14881fbb5b1080c9275674e95fb67f965f6478a957d60
```

Check the container has its state set to `Running`.

```
crictl ps
```

The output is similar to this:

```
CONTAINER ID   IMAGE    CREATED            STATE    NAME     ATTEMPT
3e025dd50a72d  busybox  About a minute ago  Running  busybox  0
```

What's next

- [Learn more about `crictl`](#).
- [Map docker CLI commands to `crictl`](#).

5 - Auditing

Kubernetes *auditing* provides a security-relevant, chronological set of records documenting the sequence of actions in a cluster. The cluster audits the activities generated by users, by applications that use the Kubernetes API, and by the control plane itself.

Auditing allows cluster administrators to answer the following questions:

- what happened?
- when did it happen?
- who initiated it?
- on what did it happen?
- where was it observed?
- from where was it initiated?
- to where was it going?

Audit records begin their lifecycle inside the kube-apiserver component. Each request on each stage of its execution generates an audit event, which is then pre-processed according to a certain policy and written to a backend. The policy determines what's recorded and the backends persist the records. The current backend implementations include logs files and webhooks.

Each request can be recorded with an associated *stage*. The defined stages are:

- `RequestReceived` - The stage for events generated as soon as the audit handler receives the request, and before it is delegated down the handler chain.
- `ResponseStarted` - Once the response headers are sent, but before the response body is sent. This stage is only generated for long-running requests (e.g. watch).
- `ResponseComplete` - The response body has been completed and no more bytes will be sent.
- `Panic` - Events generated when a panic occurred.

> **Note:** The configuration of an Audit Event configuration is different from the Event API object.

The audit logging feature increases the memory consumption of the API server because some context required for auditing is stored for each request. Memory consumption depends on the audit logging configuration.

Audit policy

Audit policy defines rules about what events should be recorded and what data they should include. The audit policy object structure is defined in the `audit.k8s.io` API group. When an event is processed, it's compared against the list of rules in order. The first matching rule sets the *audit level* of the event. The defined audit levels are:

- `None` - don't log events that match this rule.
- `Metadata` - log request metadata (requesting user, timestamp, resource, verb, etc.) but not request or response body.
- `Request` - log event metadata and request body but not response body. This does not apply for non-resource requests.
- `RequestResponse` - log event metadata, request and response bodies. This does not apply for non-resource requests.

You can pass a file with the policy to `kube-apiserver` using the `--audit-policy-file` flag. If the flag is omitted, no events are logged. Note that the `rules` field **must** be provided in the audit policy file. A policy with no (0) rules is treated as illegal.

Below is an example audit policy file:

audit/audit-policy.yaml

```yaml
apiVersion: audit.k8s.io/v1 # This is required.
kind: Policy
# Don't generate audit events for all requests in RequestReceived stage.
omitStages:
  - "RequestReceived"
rules:
  # Log pod changes at RequestResponse level
  - level: RequestResponse
    resources:
    - group: ""
      # Resource "pods" doesn't match requests to any subresource of pods,
      # which is consistent with the RBAC policy.
      resources: ["pods"]
  # Log "pods/log", "pods/status" at Metadata level
  - level: Metadata
    resources:
    - group: ""
      resources: ["pods/log", "pods/status"]

  # Don't log requests to a configmap called "controller-leader"
  - level: None
    resources:
    - group: ""
      resources: ["configmaps"]
      resourceNames: ["controller-leader"]

  # Don't log watch requests by the "system:kube-proxy" on endpoints or services
  - level: None
    users: ["system:kube-proxy"]
    verbs: ["watch"]
    resources:
    - group: "" # core API group
      resources: ["endpoints", "services"]

  # Don't log authenticated requests to certain non-resource URL paths.
  - level: None
    userGroups: ["system:authenticated"]
    nonResourceURLs:
    - "/api*" # Wildcard matching.
    - "/version"

  # Log the request body of configmap changes in kube-system.
  - level: Request
    resources:
    - group: "" # core API group
      resources: ["configmaps"]
    # This rule only applies to resources in the "kube-system" namespace.
    # The empty string "" can be used to select non-namespaced resources.
    namespaces: ["kube-system"]

  # Log configmap and secret changes in all other namespaces at the Metadata level.
  - level: Metadata
    resources:
    - group: "" # core API group
      resources: ["secrets", "configmaps"]

  # Log all other resources in core and extensions at the Request level.
  - level: Request
    resources:
    - group: "" # core API group
    - group: "extensions" # Version of group should NOT be included.

  # A catch-all rule to log all other requests at the Metadata level.
  - level: Metadata
    # Long-running requests like watches that fall under this rule will not
```

```
    # generate an audit event in RequestReceived.
    omitStages:
      - "RequestReceived"
```

You can use a minimal audit policy file to log all requests at the `Metadata` level:

```
# Log all requests at the Metadata level.
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
- level: Metadata
```

If you're crafting your own audit profile, you can use the audit profile for Google Container-Optimized OS as a starting point. You can check the configure-helper.sh script, which generates an audit policy file. You can see most of the audit policy file by looking directly at the script.

You can also refer to the `Policy` configuration reference for details about the fields defined.

Audit backends

Audit backends persist audit events to an external storage. Out of the box, the kube-apiserver provides two backends:

- Log backend, which writes events into the filesystem
- Webhook backend, which sends events to an external HTTP API

In all cases, audit events follow a structure defined by the Kubernetes API in the `audit.k8s.io` API group.

Note:
In case of patches, request body is a JSON array with patch operations, not a JSON object with an appropriate Kubernetes API object. For example, the following request body is a valid patch request to `/apis/batch/v1/namespaces/some-namespace/jobs/some-job-name`:

```
[
  {
    "op": "replace",
    "path": "/spec/parallelism",
    "value": 0
  },
  {
    "op": "remove",
    "path": "/spec/template/spec/containers/0/terminationMessagePolicy"
  }
]
```

Log backend

The log backend writes audit events to a file in JSONlines format. You can configure the log audit backend using the following `kube-apiserver` flags:

- `--audit-log-path` specifies the log file path that log backend uses to write audit events. Not specifying this flag disables log backend. `-` means standard out
- `--audit-log-maxage` defined the maximum number of days to retain old audit log files
- `--audit-log-maxbackup` defines the maximum number of audit log files to retain

- `--audit-log-maxsize` defines the maximum size in megabytes of the audit log file before it gets rotated

If your cluster's control plane runs the kube-apiserver as a Pod, remember to mount the `hostPath` to the location of the policy file and log file, so that audit records are persisted. For example:

```
    --audit-policy-file=/etc/kubernetes/audit-policy.yaml \
    --audit-log-path=/var/log/kubernetes/audit/audit.log
```

then mount the volumes:

```
...
volumeMounts:
  - mountPath: /etc/kubernetes/audit-policy.yaml
    name: audit
    readOnly: true
  - mountPath: /var/log/kubernetes/audit/
    name: audit-log
    readOnly: false
```

and finally configure the `hostPath`:

```
...
volumes:
- name: audit
  hostPath:
    path: /etc/kubernetes/audit-policy.yaml
    type: File

- name: audit-log
  hostPath:
    path: /var/log/kubernetes/audit/
    type: DirectoryOrCreate
```

Webhook backend

The webhook audit backend sends audit events to a remote web API, which is assumed to be a form of the Kubernetes API, including means of authentication. You can configure a webhook audit backend using the following kube-apiserver flags:

- `--audit-webhook-config-file` specifies the path to a file with a webhook configuration. The webhook configuration is effectively a specialized kubeconfig.
- `--audit-webhook-initial-backoff` specifies the amount of time to wait after the first failed request before retrying. Subsequent requests are retried with exponential backoff.

The webhook config file uses the kubeconfig format to specify the remote address of the service and credentials used to connect to it.

Event batching

Both log and webhook backends support batching. Using webhook as an example, here's the list of available flags. To get the same flag for log backend, replace `webhook` with `log` in the flag name. By default, batching is enabled in `webhook` and disabled in `log`. Similarly, by default throttling is enabled in `webhook` and disabled in `log`.

- `--audit-webhook-mode` defines the buffering strategy. One of the following:
    - `batch` - buffer events and asynchronously process them in batches. This is the default.
    - `blocking` - block API server responses on processing each individual event.
    - `blocking-strict` - Same as blocking, but when there is a failure during audit logging at the RequestReceived

stage, the whole request to the kube-apiserver fails.

The following flags are used only in the `batch` mode:

- `--audit-webhook-batch-buffer-size` defines the number of events to buffer before batching. If the rate of incoming events overflows the buffer, events are dropped.
- `--audit-webhook-batch-max-size` defines the maximum number of events in one batch.
- `--audit-webhook-batch-max-wait` defines the maximum amount of time to wait before unconditionally batching events in the queue.
- `--audit-webhook-batch-throttle-qps` defines the maximum average number of batches generated per second.
- `--audit-webhook-batch-throttle-burst` defines the maximum number of batches generated at the same moment if the allowed QPS was underutilized previously.

Parameter tuning

Parameters should be set to accommodate the load on the API server.

For example, if kube-apiserver receives 100 requests each second, and each request is audited only on `ResponseStarted` and `ResponseComplete` stages, you should account for ≅200 audit events being generated each second. Assuming that there are up to 100 events in a batch, you should set throttling level at least 2 queries per second. Assuming that the backend can take up to 5 seconds to write events, you should set the buffer size to hold up to 5 seconds of events; that is: 10 batches, or 1000 events.

In most cases however, the default parameters should be sufficient and you don't have to worry about setting them manually. You can look at the following Prometheus metrics exposed by kube-apiserver and in the logs to monitor the state of the auditing subsystem.

- `apiserver_audit_event_total` metric contains the total number of audit events exported.
- `apiserver_audit_error_total` metric contains the total number of events dropped due to an error during exporting.

Log entry truncation

Both log and webhook backends support limiting the size of events that are logged. As an example, the following is the list of flags available for the log backend:

- `audit-log-truncate-enabled` whether event and batch truncating is enabled.
- `audit-log-truncate-max-batch-size` maximum size in bytes of the batch sent to the underlying backend.
- `audit-log-truncate-max-event-size` maximum size in bytes of the audit event sent to the underlying backend.

By default truncate is disabled in both `webhook` and `log`, a cluster administrator should set `audit-log-truncate-enabled` or `audit-webhook-truncate-enabled` to enable the feature.

What's next

- Learn about Mutating webhook auditing annotations.
- Learn more about `Event` and the `Policy` resource types by reading the Audit configuration reference.

6 - Developing and debugging services locally using telepresence

> **Note:** This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

Kubernetes applications usually consist of multiple, separate services, each running in its own container. Developing and debugging these services on a remote Kubernetes cluster can be cumbersome, requiring you to get a shell on a running container in order to run debugging tools.

`telepresence` is a tool to ease the process of developing and debugging services locally while proxying the service to a remote Kubernetes cluster. Using `telepresence` allows you to use custom tools, such as a debugger and IDE, for a local service and provides the service full access to ConfigMap, secrets, and the services running on the remote cluster.

This document describes using `telepresence` to develop and debug services running on a remote cluster locally.

Before you begin

- Kubernetes cluster is installed
- `kubectl` is configured to communicate with the cluster
- Telepresence is installed

Connecting your local machine to a remote Kubernetes cluster

After installing `telepresence`, run `telepresence connect` to launch its Daemon and connect your local workstation to the cluster.

```
$ telepresence connect

Launching Telepresence Daemon
...
Connected to context default (https://<cluster public IP>)
```

You can curl services using the Kubernetes syntax e.g. `curl -ik https://kubernetes.default`

Developing or debugging an existing service

When developing an application on Kubernetes, you typically program or debug a single service. The service might require access to other services for testing and debugging. One option is to use the continuous deployment pipeline, but even the fastest deployment pipeline introduces a delay in the program or debug cycle.

Use the `telepresence intercept $SERVICE_NAME --port $LOCAL_PORT:$REMOTE_PORT` command to create an "intercept" for rerouting remote service traffic.

Where:

- `$SERVICE_NAME` is the name of your local service
- `$LOCAL_PORT` is the port that your service is running on your local workstation
- And `$REMOTE_PORT` is the port your service listens to in the cluster

Running this command tells Telepresence to send remote traffic to your local service instead of the service in the remote Kubernetes cluster. Make edits to your service source code locally, save, and see the corresponding changes when accessing your remote application take effect immediately. You can also run your local service using a debugger or any other local development tool.

How does Telepresence work?

Telepresence installs a traffic-agent sidecar next to your existing application's container running in the remote cluster. It then captures all traffic requests going into the Pod, and instead of forwarding this to the application in the remote cluster, it routes all traffic (when you create a global intercept) or a subset of the traffic (when you create a personal intercept) to your local development environment.

What's next

If you're interested in a hands-on tutorial, check out this tutorial that walks through locally developing the Guestbook application on Google Kubernetes Engine.

For further reading, visit the Telepresence website.

7 - Windows debugging tips

Node-level troubleshooting

1. My Pods are stuck at "Container Creating" or restarting over and over

   Ensure that your pause image is compatible with your Windows OS version. See Pause container to see the latest / recommended pause image and/or get more information.

   > **Note:** If using containerd as your container runtime the pause image is specified in the `plugins.plugins.cri.sandbox_image` field of the of config.toml configration file.

2. My pods show status as `ErrImgPull` or `ImagePullBackOff`

   Ensure that your Pod is getting scheduled to a compatible Windows Node.

   More information on how to specify a compatible node for your Pod can be found in this guide.

Network troubleshooting

1. My Windows Pods do not have network connectivity

   If you are using virtual machines, ensure that MAC spoofing is **enabled** on all the VM network adapter(s).

2. My Windows Pods cannot ping external resources

   Windows Pods do not have outbound rules programmed for the ICMP protocol. However, TCP/UDP is supported. When trying to demonstrate connectivity to resources outside of the cluster, substitute `ping <IP>` with corresponding `curl <IP>` commands.

   If you are still facing problems, most likely your network configuration in cni.conf deserves some extra attention. You can always edit this static file. The configuration update will apply to any new Kubernetes resources.

   One of the Kubernetes networking requirements (see Kubernetes model) is for cluster communication to occur without NAT internally. To honor this requirement, there is an ExceptionList for all the communication where you do not want outbound NAT to occur. However, this also means that you need to exclude the external IP you are trying to query from the `ExceptionList`. Only then will the traffic originating from your Windows pods be SNAT'ed correctly to receive a response from the outside world. In this regard, your `ExceptionList` in `cni.conf` should look as follows:

   ```
   "ExceptionList": [
                   "10.244.0.0/16",  # Cluster subnet
                   "10.96.0.0/12",   # Service subnet
                   "10.127.130.0/24" # Management (host) subnet
               ]
   ```

3. My Windows node cannot access `NodePort` type Services

   Local NodePort access from the node itself fails. This is a known limitation. NodePort access works from other nodes or external clients.

4. vNICs and HNS endpoints of containers are being deleted

   This issue can be caused when the `hostname-override` parameter is not passed to kube-proxy. To resolve it, users need to pass the hostname to kube-proxy as follows:

   ```
   C:\k\kube-proxy.exe --hostname-override=$(hostname)
   ```

5. My Windows node cannot access my services using the service IP

   This is a known limitation of the networking stack on Windows. However, Windows Pods can access the Service IP.

6. No network adapter is found when starting the kubelet

   The Windows networking stack needs a virtual adapter for Kubernetes networking to work. If the following commands return no results (in an admin shell), virtual network creation — a necessary prerequisite for the kubelet to work — has failed:

   ```
   Get-HnsNetwork | ? Name -ieq "cbr0"
   Get-NetAdapter | ? Name -Like "vEthernet (Ethernet*"
   ```

   Often it is worthwhile to modify the InterfaceName parameter of the `start.ps1` script, in cases where the host's network adapter isn't "Ethernet". Otherwise, consult the output of the `start-kubelet.ps1` script to see if there are errors during virtual network creation.

7. DNS resolution is not properly working

   Check the DNS limitations for Windows in this section.

8. `kubectl port-forward` fails with "unable to do port forwarding: wincat not found"

   This was implemented in Kubernetes 1.15 by including `wincat.exe` in the pause infrastructure container `mcr.microsoft.com/oss/kubernetes/pause:3.6`. Be sure to use a supported version of Kubernetes. If you would like to build your own pause infrastructure container be sure to include wincat.

9. My Kubernetes installation is failing because my Windows Server node is behind a proxy

   If you are behind a proxy, the following PowerShell environment variables must be defined:

   ```
   [Environment]::SetEnvironmentVariable("HTTP_PROXY", "http://proxy.example.com:80/", [EnvironmentVariable
   [Environment]::SetEnvironmentVariable("HTTPS_PROXY", "http://proxy.example.com:443/", [EnvironmentVariab
   ```

Flannel troubleshooting

1. With Flannel, my nodes are having issues after rejoining a cluster

   Whenever a previously deleted node is being re-joined to the cluster, flannelD tries to assign a new pod subnet to the node. Users should remove the old pod subnet configuration files in the following paths:

   ```
   Remove-Item C:\k\SourceVip.json
   Remove-Item C:\k\SourceVipRequest.json
   ```

2. Flanneld is stuck in "Waiting for the Network to be created"

   There are numerous reports of this issue; most likely it is a timing issue for when the management IP of the flannel network is set. A workaround is to relaunch `start.ps1` or relaunch it manually as follows:

   ```
   [Environment]::SetEnvironmentVariable("NODE_NAME", "<Windows_Worker_Hostname>")
   C:\flannel\flanneld.exe --kubeconfig-file=c:\k\config --iface=<Windows_Worker_Node_IP> --ip-masq=1 --kub
   ```

3. My Windows Pods cannot launch because of missing `/run/flannel/subnet.env`

   This indicates that Flannel didn't launch correctly. You can either try to restart `flanneld.exe` or you can copy the files over manually from `/run/flannel/subnet.env` on the Kubernetes master to `C:\run\flannel\subnet.env` on the Windows worker node and modify the `FLANNEL_SUBNET` row to a different number. For example, if node subnet

10.244.4.1/24 is desired:

```
FLANNEL_NETWORK=10.244.0.0/16
FLANNEL_SUBNET=10.244.4.1/24
FLANNEL_MTU=1500
FLANNEL_IPMASQ=true
```

Further investigation

If these steps don't resolve your problem, you can get help running Windows containers on Windows nodes in Kubernetes through:

- StackOverflow [Windows Server Container] topic
- Kubernetes Official Forum [discuss.kubernetes.io]
- Kubernetes Slack [#SIG-Windows Channel]