# Practice M6: Observability and Troubleshooting

For the purpose of this practice, we will assume that we are working on a machine with either a Windows 10/11 or any recent Linux distribution and there is a local virtualization solution (like VirtualBox, Hyper-V, VMware Workstation, etc.) installed with Kubernetes cluster created on top of it

***Please note that long commands may be hard to read here. To handle this, you can copy them to a plain text editor first. This will allow you to see them correctly. Then you can use them as intended***

*For this practice we will assume that we are working on a three-node cluster (one control plane node and two worker nodes) with an arbitrary pod network plugin installed*

## Part 1: Health and Status Checks

We will explore the three types of checks (probes). Let’s start with the liveness probe

Samples are either inspired and/or taken directly from here:

<https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>

### Liveness Probes

#### Exec Probe

We will test the command/exec type of liveness probe first

Check this (**part1/1-liveness-cmd.yaml**) manifest file

apiVersion: v1

kind: Pod

metadata:

  labels:

    test: liveness

  name: liveness-cmd

spec:

  containers:

  - name: liveness

    image: busybox

    args:

    - /bin/sh

    - -c

    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600

    livenessProbe:

      exec:

        command:

        - cat

        - /tmp/healthy

      initialDelaySeconds: 5

      periodSeconds: 10

Now create the pod

**kubectl apply -f part1/1-liveness-cmd.yaml**

And get information about it within the first 60 seconds

**kubectl describe pod liveness-cmd**

You can see that the liveness check has been assigned, and currently everything seems to be okay

If you ask again for the information at least 65 seconds after the pod creation

**kubectl describe pod liveness-cmd**

Now we see that the check has failed and after another 30 seconds, if we ask for the pods

**kubectl get pod liveness-cmd**

We can see that the pod has been restarted

But why after 30 seconds? *Perhaps because we have three failure checks with 10 second interval*

Let’s clean up

**kubectl delete -f part1/1-liveness-cmd.yaml**

#### HTTP Probe

Now, let’s see how we can work with **HTTP** probes

Check this (**part1/2-liveness-http.yaml**) manifest file

apiVersion: v1

kind: Pod

metadata:

  labels:

    test: liveness

  name: liveness-http

spec:

  containers:

  - name: liveness

    image: k8s.gcr.io/liveness

    args:

    - /server

    livenessProbe:

      httpGet:

        path: /healthz

        port: 8080

        httpHeaders:

        - name: Custom-Header

          value: Awesome

      initialDelaySeconds: 3

      periodSeconds: 3

Now create the pod

**kubectl apply -f part1/2-liveness-http.yaml**

And get information about it within the first 10 seconds

**kubectl describe pod liveness-http**

You can see that the liveness check has been assigned, and currently everything seems to be okay

If you ask again for the information at least 15 seconds after the pod creation

**kubectl describe pod liveness-http**

Now we see that the check has failed and after another 10 seconds, if we ask for the pods

**kubectl get pod liveness-http**

We can see that the pod has been restarted

But why after 10 seconds? *Perhaps because we have three failure checks with 3 second interval*

If we wait a bit and check again, after a few cycles we will notice that the process is pausing for extended period of time with the **CrashLoopBackOff** status

It is quite normal. It is to prevent the cluster. We can check more here:

<https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy>

Let’s clean up

**kubectl delete -f part1/2-liveness-http.yaml**

### Readiness Probes

We will simulate a situation in which we have an application that needs some time to get ready and then after a while it breaks

#### Exec Probe

Check this (**part1/3-readiness-cmd.yaml**) manifest file

apiVersion: v1

kind: Pod

metadata:

  labels:

    app: readiness-cmd

  name: readiness-cmd

spec:

  initContainers:

  - name: init-data

    image: alpine

    command: ["/bin/sh", "-c"]

    args:

      - echo '(Almost) Always Ready to Serve ;)' > /data/index.html

    volumeMounts:

    - name: data

      mountPath: /data

  containers:

  - name: cont-main

    image: nginx

    volumeMounts:

    - name: data

      mountPath: /usr/share/nginx/html

    - name: check

      mountPath: /check

    readinessProbe:

      exec:

        command:

        - cat

        - /check/healthy

      initialDelaySeconds: 5

      periodSeconds: 5

  - name: cont-sidecar-postpone

    image: alpine

    command: ["/bin/sh", "-c"]

    args:

      - while true; do

          sleep 20;

          touch /check/healthy;

          sleep 60;

        done

    volumeMounts:

    - name: check

      mountPath: /check

  - name: cont-sidecar-break

    image: alpine

    command: ["/bin/sh", "-c"]

    args:

      - while true; do

          sleep 60;

          rm /check/healthy;

          sleep 20;

        done

    volumeMounts:

    - name: check

      mountPath: /check

  volumes:

  - name: data

    emptyDir: {}

  - name: check

    emptyDir: {}

---

apiVersion: v1

kind: Service

metadata:

  name: readiness-cmd

  labels:

    app: readiness-cmd

spec:

  type: NodePort

  ports:

  - port: 80

    nodePort: 30001

    protocol: TCP

  selector:

    app: readiness-cmd

Now create the resources

**kubectl apply -f part1/3-readiness-cmd.yaml**

And get information about it within the first 20 seconds

**kubectl describe pod readiness-cmd**

You can see that the readiness probe is assigned and failed

After another 10 seconds it will succeed

Then, we can check the service

**kubectl describe svc readiness-cmd**

There will be one endpoint

We can open a browser tab and navigate to **http://<control-plane-ip>:30001**

Our application should be there

After another 50 seconds the main container in the pod will be marked as not ready

If check again the service, we will notice that there aren’t any endpoints

**kubectl describe svc readiness-cmd**

The cycle will repeat

It would be easier to combine the above commands with watch of the resources in another terminal

**watch -n 2 kubectl get pods,svc -o wide**

Finally, let’s clean up

**kubectl delete -f part1/3-readiness-cmd.yaml**

#### HTTP Probe

Now, let’s see a variant of the earlier sample but with HTTP probe

Check this (**part1/4-readiness-http.yaml**) manifest file

apiVersion: v1

kind: Pod

metadata:

  labels:

    app: readiness-http

  name: readiness-http

spec:

  initContainers:

  - name: init-data

    image: alpine

    command: ["/bin/sh", "-c"]

    args:

      - echo '(Almost) Always Ready to Serve ;)' > /data/index.html

    volumeMounts:

    - name: data

      mountPath: /data

  containers:

  - name: cont-main

    image: nginx

    volumeMounts:

    - name: data

      mountPath: /usr/share/nginx/html

    readinessProbe:

      httpGet:

        path: /healthy.html

        port: 80

      initialDelaySeconds: 5

      periodSeconds: 5

  - name: cont-sidecar-postpone

    image: alpine

    command: ["/bin/sh", "-c"]

    args:

      - while true; do

          sleep 20;

          echo 'WORKING' > /check/healthy.html;

          sleep 60;

        done

    volumeMounts:

    - name: data

      mountPath: /check

  - name: cont-sidecar-break

    image: alpine

    command: ["/bin/sh", "-c"]

    args:

      - while true; do

          sleep 60;

          rm /check/healthy.html;

          sleep 20;

        done

    volumeMounts:

    - name: data

      mountPath: /check

  volumes:

  - name: data

    emptyDir: {}

---

apiVersion: v1

kind: Service

metadata:

  name: readiness-http

  labels:

    app: readiness-http

spec:

  type: NodePort

  ports:

  - port: 80

    nodePort: 30001

    protocol: TCP

  selector:

    app: readiness-http

Now create the resources

**kubectl apply -f part1/4-readiness-http.yaml**

And get information about it within the first 20 seconds

**kubectl describe pod readiness-http**

You can see that the readiness probe is assigned and failed

After another 10 seconds it will succeed

Then, we can check the service

**kubectl describe svc readiness-http**

There will be one endpoint

We can open a browser tab and navigate to **http://<control-plane-ip>:30001**

Our application should be there

After another 50 seconds the main container in the pod will be marked as not ready

If check again the service, we will notice that there aren’t any endpoints

**kubectl describe svc readiness-http**

The cycle will repeat

It would be easier to combine the above commands with watch of the resources in another terminal

**watch -n 2 kubectl get pods,svc -o wide**

Finally, let’s clean up

**kubectl delete -f part1/4-readiness-http.yaml**

### Startup Probes

Usually, we combine startup probes with the other two – liveness and readiness

Startup probes are used to give time for the application to start and during this period all other probes are paused

#### Startup and Liveness Same Type

Let’s check this (**part1/5-startup-liveness-same.yaml**) manifest file

apiVersion: v1

kind: Pod

metadata:

  name: startup-same

spec:

  containers:

  - name: startup

    image: alpine

    command: ["/bin/sh", "-c"]

    args:

    - t=$(( 10 + $RANDOM % 100 )); echo 'Sleep for '$t; sleep $t; touch /tmp/healthy; sleep 60; rm -rf /tmp/healthy; sleep 600

    livenessProbe:

      exec:

        command:

        - cat

        - /tmp/healthy

      initialDelaySeconds: 5

      periodSeconds: 5

    startupProbe:

      exec:

        command:

        - cat

        - /tmp/healthy

      failureThreshold: 22

      periodSeconds: 5

Now create the resources

**kubectl apply -f part1/5-startup-liveness-same.yaml**

And check how much time it will need to “initialize”

**kubectl logs startup-same**

Okay, now we can ask for additional information about the pod

**kubectl describe pod startup-same**

Wait a while and check again

After a while the initialization will finish, and startup probe will detect this

Then the liveness probe will be allowed to check periodically

In the next seconds, we must refresh a few times to catch the moment the liveness probe will kick the container

Finally, let’s clean up

**kubectl delete -f part1/5-startup-liveness-same.yaml**

#### Startup and Liveness Mixed Type

We are not obliged to match the types of the checks for different probes

Let’s test a mixed case. Check the following (**part1/6-startup-liveness-mixed.yaml**) manifest

apiVersion: v1

kind: Pod

metadata:

  labels:

    app: startup-mixed

  name: startup-mixed

spec:

  initContainers:

  - name: init-data

    image: alpine

    command: ["/bin/sh", "-c"]

    args:

      - echo '(Almost) Always Ready to Serve ;)' > /data/index.html

    volumeMounts:

    - name: data

      mountPath: /data

  containers:

  - name: cont-main

    image: nginx

    volumeMounts:

    - name: data

      mountPath: /usr/share/nginx/html

    livenessProbe:

      httpGet:

        path: /healthy.html

        port: 80

      initialDelaySeconds: 5

      periodSeconds: 5

    startupProbe:

      exec:

        command:

        - cat

        - /usr/share/nginx/html/healthy.html

      failureThreshold: 10

      periodSeconds: 5

  - name: cont-sidecar-postpone

    image: alpine

    command: ["/bin/sh", "-c"]

    args:

      - while true; do

          sleep 20;

          echo 'WORKING' > /check/healthy.html;

          sleep 60;

        done

    volumeMounts:

    - name: data

      mountPath: /check

  - name: cont-sidecar-break

    image: alpine

    command: ["/bin/sh", "-c"]

    args:

      - while true; do

          sleep 60;

          rm /check/healthy.html;

          sleep 20;

        done

    volumeMounts:

    - name: data

      mountPath: /check

  volumes:

  - name: data

    emptyDir: {}

---

apiVersion: v1

kind: Service

metadata:

  name: startup-mixed

  labels:

    app: startup-mixed

spec:

  type: NodePort

  ports:

  - port: 80

    nodePort: 30001

    protocol: TCP

  selector:

    app: startup-mixed

Now create the resources

**kubectl apply -f part1/6-startup-liveness-mixed.yaml**

And get information about it within the first 20 seconds

**kubectl describe pod startup-mixed**

You can see that the startup probe is assigned and failed

After another 10 seconds it will succeed

Then, we can check the service

**kubectl describe svc startup-mixed**

There will be one endpoint

We can open a browser tab and navigate to **http://<control-plane-ip>:30001**

Our application should be there

After another 50 seconds the main container in the pod will be marked as not ready because the liveness probe will catch that the container is not live

After a certain tries the container will be restarted

The cycle will repeat

It would be easier to combine the above commands with watch of the resources in another terminal

**watch -n 2 kubectl get pods,svc -o wide**

Finally, let’s clean up

**kubectl delete -f part1/6-startup-liveness-mixed.yaml**

## Part 2: Logging and Auditing

Let’s test the auding feature first

### Auditing

Samples are either inspired and/or taken directly from here:

<https://kubernetes.io/docs/tasks/debug-application-cluster/debug-application-introspection/>

We will start with something simple

For this, we must create a simple audit policy and then give instructions to the **API Server** where it is and where to store the logs

Check this (**part2/1-audit-simple.yaml**) manifest

# Log all requests at the Metadata level.

apiVersion: audit.k8s.io/v1

kind: Policy

rules:

- level: Metadata

It will log all requests on metadata level. This will include information like when, who, what, etc. but not the request or the response body

Then, we must copy it to the control plane node

We will create a folder **/var/lib/k8s-audit** and copy it there

**sudo mkdir /var/lib/k8s-audit**

If we are working on the control plane node, then we can use this

**sudo cp part2/1-audit-simple.yaml /var/lib/k8s-audit/**

Otherwise, we must use other means to copy the file in the target folder

Next, we must prepare a folder to store the logs

Let’s create **/var/log/k8s-audit**

**sudo mkdir /var/log/k8s-audit**

Now, we must change the configuration (**/etc/kubernetes/manifests/kube-apiserver.yaml**) of the **API Server**

**sudo vi /etc/kubernetes/manifests/kube-apiserver.yaml**

*Making a backup of the manifest before editing is always a good idea*

Navigate to the volume mounts section and add

    - mountPath: /var/lib/k8s-audit/1-audit-simple.yaml

      name: audit-policy

      readOnly: true

    - mountPath: /var/log/k8s-audit/audit.log

      name: audit-log

      readOnly: false

Then in the volumes section add

  - hostPath:

      path: /var/lib/k8s-audit/1-audit-simple.yaml

      type: File

    name: audit-policy

  - hostPath:

      path: /var/log/k8s-audit/audit.log

      type: FileOrCreate

    name: audit-log

Then return to the command section and add the following two lines

    - --audit-policy-file=/var/lib/k8s-audit/1-audit-simple.yaml

    - --audit-log-path=/var/log/k8s-audit/audit.log

Save and close the file

*All the above are available in the* ***part2/1-additions.yaml*** *file*

Sit back and watch when the **API Server** will restart

**kubectl get pods -n kube-system -w**

It may return an error a few times, but it will start working eventually 😉

Once everything is up and running, let’s create a pod and see what happens

Start it with

**kubectl run logtest1 --image=alpine -- sleep 1d**

Now, check if the pod is running

**kubectl get pods**

And then check the audit log

**cat /var/log/k8s-audit/audit.log | grep logtest1**

Wow, plenty of events for such a simple task

The output is not quite readable

We can install utility like **jq** and use it to get a better look at what happened

**cat /var/log/k8s-audit/audit.log | grep logtest1 | jq**

Still too much information but at least it is more readable 😉

We can check how many records we have with

**cat /var/log/k8s-audit/audit.log | grep logtest1 | jq -s length**

More than 20 (24 to be exact), wow

Let’s clean up a bit and then check another sample policy

Stop and remove the pod

**kubectl delete pod logtest1**

Now, check this (**part2/2-audit.yaml**) manifest

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"

Then, we must copy it to the control plane node (folder **/var/lib/k8s-audit**)

**sudo cp part2/2-audit.yaml /var/lib/k8s-audit/**

Now, we must change the configuration (**/etc/kubernetes/manifests/kube-apiserver.yaml**) of the **API Server**

**sudo vi /etc/kubernetes/manifests/kube-apiserver.yaml**

Navigate to the volume mounts section and change the audit policy mount to

    - mountPath: /var/lib/k8s-audit/2-audit.yaml

      name: audit-policy

      readOnly: true

Then in the volumes section modify the audit policy volume to

  - hostPath:

      path: /var/lib/k8s-audit/2-audit.yaml

      type: File

    name: audit-policy

Then return to the command section and modify the following line

    - --audit-policy-file=/var/lib/k8s-audit/2-audit.yaml

Save and close the file

Sit back and watch when the **API Server** will restart

**kubectl get pods -n kube-system -w**

It may return an error a few times, but it will start working eventually 😉

Once everything is up and running, let’s create a pod and see what happens

Start it with

**kubectl run logtest2 --image=alpine -- sleep 1d**

Now, check if the pod is running

**kubectl get pods**

And then check the audit log

**cat /var/log/k8s-audit/audit.log | grep logtest2 | jq**

We can check how many records we have with

**cat /var/log/k8s-audit/audit.log | grep logtest2 | jq -s length**

Now, they are less than 20 (15 to be exact). Great improvement 😉

Let’s clean up a bit and then refine our simple policy

**kubectl delete pod logtest2**

Now, we must change the configuration (**/etc/kubernetes/manifests/kube-apiserver.yaml**) of the **API Server**

And remove those blocks that we added earlier

Save and close the file

*Or use the backup created before the changes*

### Logging

Samples are either inspired and/or taken directly from here:

<https://kubernetes.io/docs/concepts/cluster-administration/logging/>

Let’s explore a few logging techniques and scenarios

#### Basic Logging

The most basic form of logging is for containers to emit messages on their **stdout**/**stderr**

Check the following manifest (**part2/3-basic-logging.yaml**)

apiVersion: v1

kind: Pod

metadata:

  name: counter

spec:

  containers:

  - name: count

    image: alpine

    args: [/bin/sh, -c,

            'i=0; while true; do echo "$i: $(date)"; i=$((i+1)); if [ -f /stop.file ]; then exit; fi; sleep 5; done']

This pod has one container which publishes a message every five seconds on its **stdout**

Start the pod

**kubectl apply -f part2/3-basic-logging.yaml**

Now check its logs with

**kubectl logs counter**

We can use the above command in follow mode as well

**kubectl logs counter --follow**

Press **Ctrl+C** to stop following

Now, what will happen if we delete the pod? Let’s see

**kubectl delete pod counter**

Ask for the logs again

**kubectl logs counter**

No luck ☹

But wait, there is still some hope. We said that **Kubernetes** keeps the last/previous instance

So, we can modify our command to

**kubectl logs counter --previous**

Still, no luck ☹ Why? *Perhaps due to that when we delete a pod, everything goes with it including the logs. But then, what about the previous instance? It is it previous instance of a container in case it gets restarted*

Let’s rerun the sample

**kubectl apply -f part2/3-basic-logging.yaml**

Ask for the logs again

**kubectl logs counter**

Now, restart the container

**kubectl exec counter -- touch /stop.file**

And check again for the logs

**kubectl logs counter**

This will return the logs of the current instance

And now for the previous

**kubectl logs counter --previous**

Now, let’s clean up

**kubectl delete pod counter**

#### Streaming Sidecar

Let’s try another scenario – one application container with two different log files and two streaming sidecar containers – one for each log file

Check this (**part2/4-streaming-sidecar.yaml**) manifest

apiVersion: v1

kind: Pod

metadata:

  name: counter

spec:

  containers:

  - name: main

    image: busybox

    args:

    - /bin/sh

    - -c

    - >

      i=0;

      while true;

      do

        echo "$i: $(date)" >> /var/log/1.log;

        echo "$(date) INFO $i" >> /var/log/2.log;

        i=$((i+1));

        if [ -f /stop.file ]; then exit; fi;

        sleep 5;

      done

    volumeMounts:

    - name: varlog

      mountPath: /var/log

  - name: sidecar-1

    image: busybox

    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/1.log']

    volumeMounts:

    - name: varlog

      mountPath: /var/log

  - name: sidecar-2

    image: busybox

    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/2.log']

    volumeMounts:

    - name: varlog

      mountPath: /var/log

  volumes:

  - name: varlog

    emptyDir: {}

Send it to the cluster

**kubectl apply -f part2/4-streaming-sidecar.yaml**

And check if started

**kubectl get pods**

Then, after a while, ask for the contents of each of the two files

**kubectl exec counter -c main -- cat /var/log/1.log**

**kubectl exec counter -c main -- cat /var/log/2.log**

Now, we can see them using the **kubectl log** command and the appropriate sidecar container

**kubectl logs counter -c sidecar-1**

**kubectl logs counter -c sidecar-2**

In the same manner each sidecar container may stream the logs to an external solution

Let’s clean up

**kubectl delete -f part2/4-streaming-sidecar.yaml**

## Part 3: Troubleshooting

Let’s check just two cluster-related scenarios and then a few application-related scenarios

### Cluster

Let’s imagine that we must finish a cluster creation but someone else has prepared the machines for us

We are assured that everything is ready and is a matter of just to execute **kubeadm**

So, we buy this and start the procedure

*Note: you should prefix the* ***kubeadm*** *commands with* ***sudo*** *if working with a regular user*

Log on the **VM1** and execute the following to initialize the cluster

**kubeadm init --apiserver-advertise-address=192.168.99.231 --pod-network-cidr 10.244.0.0/16**

Installation should finish relatively quickly

Hm, it appears that something is wrong

Someone forgot to turn off the **SWAP**

Let’s correct this

Once done, reboot and try again

This time it succeeds

Now, let’s join the first worker node

Log on to **VM2**

Let’s first check the **SWAP**

Okey, it is set up as required

Let’s join the node

**kubeadm join 192.168.99.231:6443 --token <token> --discovery-token-ca-cert-hash <hash>**

Okay, we are done here

Log on to **VM3**

Let’s first check the **SWAP**

Hm, here it is enabled as well

Let’s disable it and reboot

Now, try to join the machine to the cluster

**kubeadm join 192.168.99.231:6443 --token <token> --discovery-token-ca-cert-hash <hash>**

Ha, another error. Something about **iptables**

Let’s check the **/etc/sysctl.conf** file and **/etc/sysctl.d/** folder

Aha, here is the issue

Correct the values and reboot

Now, try to join the machine to the cluster

**kubeadm join 192.168.99.231:6443 --token <token> --discovery-token-ca-cert-hash <hash>**

Again error. Now, try to reset the procedure with

**kubeadm reset**

And repeat again the join command

Finally

Return to **VM1**

Copy the configuration

Check the list of nodes

**kubectl get nodes**

Note that they appear as **not ready**

Check the pods as well

**kubectl get pods -n kube-system**

Hm, most of the pods are operational, but there is one pair that is not (**CoreDNS**)

Let's check why the node is not ready

**kubectl describe node k8s1**

Scroll to top and look for **Ready** and **KubeletNotReady** words

It appears that there isn't any (POD) network plugin installed

Let’s do it

We will install **Flannel**

First, we download it

**wget -q https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml -O /tmp/kube-flannel.yaml**

Then adjust it if needed (for example, change the interface)

And finally, deploy it

**kubectl apply -f /tmp/kube-flannel.yaml**

We can watch the progress with

**kubectl get pods --all-namespaces -w**

After a while both **Flannel** and **CoreDNS** will be fully operational

Press **Ctrl + C** to stop the monitoring

Check again the status of the node

**kubectl get nodes**

It should be operational and ready as well

Okay, it seems that we are done

Let’s reboot all nodes and see if they will work

Everything seems to be working 😊

*Should you want to reproduce the scenarios, you must prepare*

* *Three identical Debian based VMs with the appropriate repositories (Docker + Kubernetes) and packages installed (Docker + kubeadm + kubectl + kubelet)*
* *Swap should be turned off just for VM2*
* *The* ***/etc/sysctl.d/k8s.conf*** *file should be with wrong content (missing settings and/or wrong values) for VM3*

### Application

Check the included scenarios (**scenario1**) in folder **part3/**

We will observe the application’s behavior and try to make sure it is working as it should be

The rest (scenario2, 3 and 4) are for homework