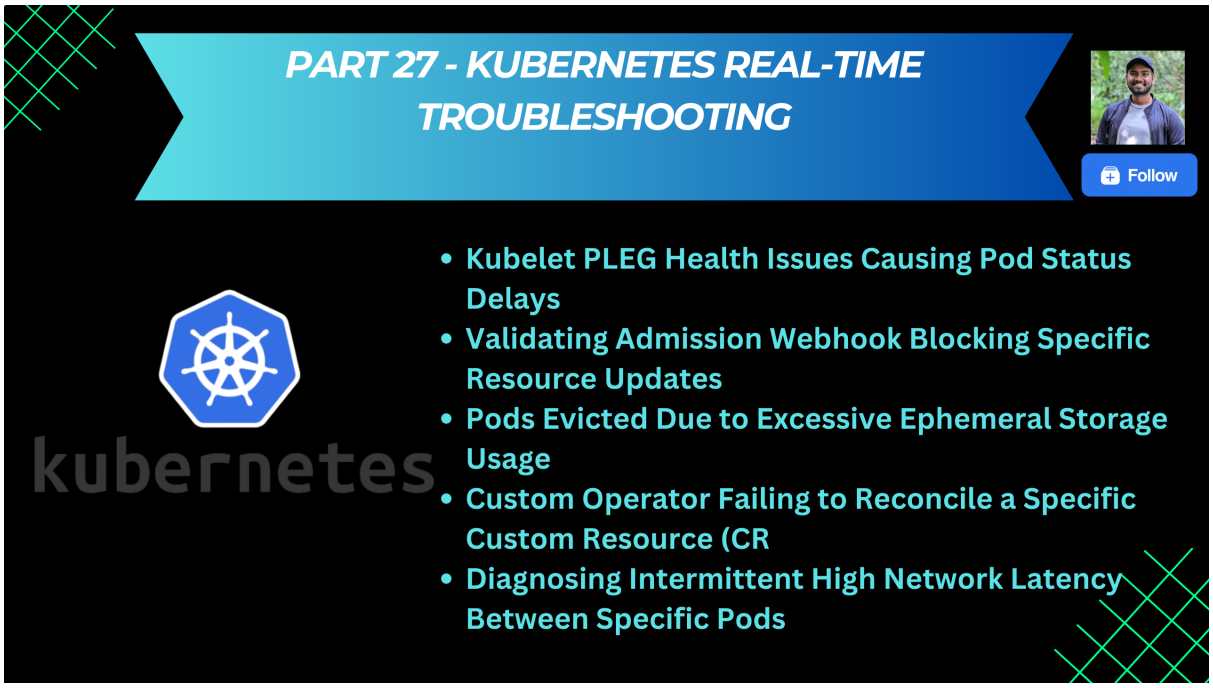
Part 27: Kubernetes Real-Time Troubleshooting

Introduction 🌐

Welcome to the world of Kubernetes troubleshooting, where every challenge is an opportunity to sharpen your skills and emerge victorious. Join us as we embark on a journey through common real-time scenarios, unraveling mysteries, and uncovering solutions along the way.



Scenario 131: Kubelet PLEG Health Issues Causing Pod Status Delays

Scenario

On a specific worker node, `cluster7-worker2`, pods are experiencing significant delays (minutes) in reflecting their true status (e.g., transitioning from `ContainerCreating` to `Running`, or showing readiness updates). `kubectl get pods` shows outdated information for pods on this node. Node status itself (`kubectl get node cluster7-worker2`) appears `Ready`, but application probes dependent on timely pod status fail intermittently. Logs within the containers show the application started much earlier than Kubernetes reported.

Solution - Context:

kubectl config use-context k8s-c22-prod

Steps:

1. **Confirm Node and Pod Status Discrepancy:**

   Observe pod status delays specifically on `cluster7-worker2`:

   kubectl get pods -A -o wide --field-selector spec.nodeName=cluster7-worker2

   # Compare status timestamps with actual container start times if possible (e.g., via logs)

   Verify the node itself is still marked `Ready`:

   kubectl get node cluster7-worker2

2. **Inspect Kubelet Logs on the Affected Node:**

   SSH into `cluster7-worker2`:

   ssh cluster7-worker2

   Check kubelet service logs, specifically looking for "PLEG" (Pod Lifecycle Event Generator) warnings or errors:

   # Using journalctl (common on systemd systems)

   sudo journalctl -u kubelet -f | grep -i pleg

   # Or check log files directly if configured differently

   # Look for messages like:

   # "PLEG is not healthy: pleg was last seen active ..."

   # "container runtime status check returned error..."

   Frequent "PLEG is not healthy" messages indicate Kubelet is struggling to get container status updates from the container runtime.

3. **Check Container Runtime Status:**

   Verify the container runtime (e.g., Docker, containerd, CRI-O) is running and responsive:

   # Example for containerd

   sudo systemctl status containerd

sudo crictl ps # Check if runtime API responds quickly

sudo crictl pods # Check if runtime lists pods correctly

Example for Docker

sudo systemctl status docker

sudo docker ps

Look for errors in the container runtime's logs (`journalctl -u containerd` or `journalctl -u docker`).

4. Investigate Node Resource Pressure:

High CPU, memory, or disk I/O pressure on the node can slow down both Kubelet and the container runtime, impacting PLEG.

Check node resource usage:

top # Check CPU/Memory

iostat -xz 1 5 # Check disk I/O wait times

Examine Kubelet metrics (if metrics-server is installed):

Might require port-forwarding or direct access to Kubelet's /metrics endpoint

Look for metrics related to PLEG latency:
kubelet_pleg_relist_duration_seconds_bucket

5. Look for Excessive Pod/Container Churn:

A very high rate of pod/container creation and deletion on the node can overload PLEG.

Check recent events on the node:

kubectl get events --field-selector
involvedObject.kind=Node,involvedObject.name=cluster7-worker2 --sort-
by='.metadata.creationTimestamp'

6. Restart Kubelet and Container Runtime:

As a potential remediation step, carefully restart the container runtime first, followed by Kubelet:

sudo systemctl restart containerd # Or docker

sleep 10 # Give runtime time to come up

sudo systemctl restart kubelet

Monitor Kubelet logs again (Step 2) to see if PLEG health recovers.

7. Analyze Underlying Cause (Example: Disk I/O Contention):

Assume `iostat` showed high `%iowait` and Kubelet logs indicated slow runtime responses correlating with disk activity.

Action: Investigate the cause of high disk I/O (e.g., a specific pod writing heavily, slow underlying storage). Mitigate the I/O load (e.g., move the problematic pod, optimize disk usage, upgrade storage).

8. Monitor Pod Status Updates:

After remediation, deploy a test pod to `cluster7-worker2` or observe existing pods to confirm status updates are now timely.

kubectl get pods -A -o wide --field-selector spec.nodeName=cluster7-worker2 -w

Outcome

The Kubelet PLEG health issue on `cluster7-worker2` is identified as the cause of delayed pod status updates.v The underlying reason for PLEG instability (e.g., container runtime issues, resource pressure like high disk I/O) is diagnosed and addressed. Kubelet PLEG becomes healthy, and pod status updates on the node become timely and accurate.

Scenario 132: Validating Admission Webhook Blocking Specific Resource Updates

Scenario

Attempts to update a specific Deployment (`payment-processor`) in the `finance` namespace are consistently failing with a generic error message like: `admission webhook "policy.example.com" denied the request: Forbidden by policy XYZ`. Creating new Deployments or updating other Deployments works fine. The `policy.example.com` webhook is intended to enforce label standards, but the target Deployment seems compliant.

Solution - Context:

kubectl config use-context k8s-c23-sec

Steps:

1. **Identify the Failing Request and Webhook:**

   Perform the failing update operation and capture the exact error message:

   kubectl apply -f payment-processor-update.yaml

   # Error: Internal error occurred: failed calling webhook "policy.example.com": Post "https://policy-webhook-svc.security.svc:443/validate?timeout=10s": context deadline exceeded OR admission webhook "policy.example.com" denied the request: Forbidden by policy XYZ.

   Note the specific webhook name (`policy.example.com`) and the error type (denial message or timeout).

2. **Inspect the Admission Webhook Configuration:**

   Get the `ValidatingWebhookConfiguration` definition:

   kubectl get validatingwebhookconfiguration policy.example.com -o yaml

   Check:

   `rules`: Do the rules correctly target `UPDATE` operations on `deployments` in the `apps` API group? Is there a `namespaceSelector` or `objectSelector` that might unexpectedly exclude/include this specific update?

   `clientConfig`: Verify the `service` reference (`policy-webhook-svc.security.svc`) points to the correct webhook backend. Check the `caBundle` if applicable.

   `failurePolicy`: Is it `Fail` (blocks request on error/timeout) or `Ignore`? The error message might differ based on this.

   `timeoutSeconds`: Is the timeout reasonable (e.g., 10s)?

3. **Examine the Webhook Service and Pods:**

   Verify the webhook service exists and points to the correct pods:

   kubectl get svc policy-webhook-svc -n security -o wide

   kubectl get endpoints policy-webhook-svc -n security

   Check the status and logs of the pods backing the webhook service:

kubectl get pods -n security -l app=policy-webhook

kubectl logs -n security <policy-webhook-pod-name> -f

Look for log entries corresponding to the denied request. The webhook logs should contain the exact reason for the denial (e.g., "Label 'cost-center' missing", "Image tag 'latest' is forbidden").

4. Test Webhook Connectivity:

Ensure the API server can reach the webhook service endpoint. Check for NetworkPolicies that might block egress from `kube-system` (where API server runs) to the `security` namespace on the webhook's port (usually 443).

Check NetworkPolicies in 'security' and 'kube-system' namespaces

kubectl get networkpolicy -n security

kubectl get networkpolicy -n kube-system

5. Simulate the AdmissionReview Request (Advanced):

Craft an `AdmissionReview` object mimicking the update request and send it directly to the webhook service endpoint (e.g., using `curl` from a pod with network access) to bypass the API server and isolate webhook logic. This requires understanding the AdmissionReview structure.

6. Analyze the Policy Logic Discrepancy:

Based on webhook logs (Step 3), understand why the policy is denying the request. Compare the Deployment manifest (`payment-processor-update.yaml`) against the policy rules described in the logs.

Example Cause: The policy requires a `team` label, but the update attempts to remove it, or the policy logic has a bug related to interpreting specific annotations on this particular Deployment.

7. Correct the Resource or Policy:

Option A (Fix Resource): Modify the Deployment manifest (`payment-processor-update.yaml`) to comply with the policy revealed in the logs (e.g., add the missing `team` label).

payment-processor-update.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: payment-processor

namespace: finance

labels:

app: payment-processor

team: core-finance # Added missing label

... rest of spec ...

Option B (Fix Policy): If the policy logic itself is flawed or too restrictive for this specific case, update the webhook's code or configuration and redeploy it.

8. Retry the Update Operation:

Apply the corrected Deployment manifest (if Option A was chosen):

kubectl apply -f payment-processor-update.yaml

If the policy was fixed (Option B), simply retry the original update. The request should now pass the webhook validation.

Outcome:

The specific rule or condition within the `policy.example.com` validating admission webhook that blocked the `payment-processor` Deployment update is identified.

The root cause (non-compliant resource manifest, buggy webhook logic, webhook availability issue) is determined.

The resource manifest or the webhook policy/deployment is corrected.

The Deployment update operation now succeeds without being blocked by the admission webhook.

Scenario 133: Pods Evicted Due to Excessive Ephemeral Storage Usage

Scenario

Multiple pods belonging to a data processing application (`log-aggregator`) in the `pipeline` namespace are being frequently evicted by Kubelet. Checking the pod status shows the reason `Evicted` and the message often indicates "Pod ephemeral local storage usage exceeds the total limit". The application processes large amounts of temporary data, but the usage seems higher than expected.

Solution - Context:

kubectl config use-context k8s-c24-data

Steps:

1. **Confirm Eviction Reason:**

   Get details of recently evicted pods:

   kubectl get pods -n pipeline -l app=log-aggregator --show-labels

   kubectl describe pod <evicted-pod-name> -n pipeline

   Verify the `Reason` is `Evicted` and the `Message` clearly states ephemeral storage limits were exceeded. Note the node the pod was running on.

2. **Check Pod Ephemeral Storage Limits:**

   Examine the Deployment or Pod spec for the `log-aggregator` application:

   kubectl get deployment log-aggregator -n pipeline -o yaml

   Look for `resources.limits.ephemeral-storage` defined for the container(s). If not defined, the pod shares the node's allocatable ephemeral storage, making it harder to pinpoint limits.

3. **Investigate Storage Usage on the Affected Node:**

   SSH into the node where the eviction occurred (obtained from Step 1).

   Check overall disk usage for partitions Kubelet uses for ephemeral storage (usually `/var/lib/kubelet` and `/var/lib/containerd` or `/var/lib/docker`):

   df -h

   Use `du` (disk usage) to find large directories within Kubelet's working area, focusing on pod/container volumes and logs:

   sudo du -sh /var/lib/kubelet/pods/ | sort -rh | head -n 10 # Check volumes

sudo du -sh /var/log/pods/ | sort -rh | head -n 10 # Check container logs (if default logging)

sudo du -sh /var/lib/containerd/io.containerd.snapshotter.v1.overlayfs/snapshots/ # Check container writable layers (path varies by runtime/driver)

Identify if specific directories associated with the `log-aggregator` pods (find their UID in `/var/lib/kubelet/pods/`) are consuming excessive space.

4. Analyze Container-Level Usage:

If a pod is still running (before eviction) or if you can redeploy it:

Exec into the container:

kubectl exec -it <log-aggregator-pod-name> -n pipeline -- /bin/bash

Check usage within the container's filesystem, focusing on temporary directories (`/tmp`), log directories, or application-specific cache/data directories:

du -sh /tmp

du -sh /var/log

du -sh /app/data # Example application data path

Common Culprits: Unmanaged log rotation within the container, large temporary files not being cleaned up, application cache growth.

5. Review Application Behavior:

Examine the `log-aggregator` application's code or configuration. How does it handle temporary data? Does it clean up files? Does it have internal log rotation configured? Could a bug be causing excessive file generation?

6. Implement Mitigation Strategies:

A) Set Explicit Limits: Define `ephemeral-storage` requests and limits in the Deployment spec to isolate its usage and potentially trigger earlier, more predictable eviction if limits are truly exceeded.

```
spec:
  containers:
  - name: log-aggregator
    image: ...
    resources:
```

```
    requests:
      ephemeral-storage: "1Gi"
    limits:
      ephemeral-storage: "2Gi" # Set a reasonable limit
```

B) Mount an `emptyDir` with SizeLimit: If temporary storage needs are known, use an `emptyDir` volume with a `sizeLimit`. This provides dedicated ephemeral storage separate from the container's writable layer and logs.

```
spec:
  containers:
  - name: log-aggregator
    # ...
    volumeMounts:
    - name: scratch-data
      mountPath: /app/temp
  volumes:
  - name: scratch-data
    emptyDir:
      sizeLimit: 1Gi # Limit the emptyDir volume
```

C) Fix Application Behavior: Modify the application to implement proper log rotation, clean up temporary files, or use storage more efficiently.

D) Configure Node Allocatable: Ensure Node Allocatable for ephemeral storage is configured correctly (`--kubelet-extra-args '--kube-reserved=ephemeral-storage=...' --system-reserved=ephemeral-storage=... --eviction-hard=nodefs.available<...'`) to protect node stability, although this doesn't fix the pod's usage.

7. Apply Changes and Monitor:

Apply the chosen mitigation (e.g., update Deployment with limits/emptyDir, deploy fixed application).

Monitor pods for evictions:

```
kubectl get pods -n pipeline -l app=log-aggregator -w
```

Monitor node disk usage: `df -h` on affected nodes.

Outcome:

The root cause of excessive ephemeral storage consumption by `log-aggregator` pods (e.g., unbounded logs, temp files, no limits set) is identified.

Appropriate mitigation strategies are implemented (setting limits, using size-limited `emptyDir`, fixing application logic).

Pods are no longer evicted due to ephemeral storage pressure, leading to stable application performance.

Scenario 134: Custom Operator Failing to Reconcile a Specific Custom Resource (CR)

Scenario

You have deployed a custom operator (`widget-operator`) managing `Widget` Custom Resources (CRs). Most `Widget` CRs are processed correctly, resulting in the creation of associated Deployments and Services. However, one specific CR, `special-widget` in the `factory` namespace, remains stuck. The operator logs show attempts to reconcile it but repeatedly fail with errors like "failed to update status" or timeout errors when trying to create associated resources, without clear reasons.

Solution - Context:

kubectl config use-context k8s-c25-op

Steps:

1. **Examine the Problematic Custom Resource (CR):**

Get the full YAML definition of the stuck CR:

kubectl get widget special-widget -n factory -o yaml

Check its `spec` for any unusual or potentially invalid configuration compared to working `Widget` CRs. Pay attention to syntax, data types, and values.

Check the `status` subresource. Does it contain any condition messages indicating partial progress or specific failures?

2. Deep Dive into Operator Logs:

Find the operator pod(s):

kubectl get pods -n widget-operator-system # Or the namespace where the operator runs

Increase log verbosity if possible (often configurable via operator Deployment args or ConfigMap).

Tail the operator logs, filtering for the specific CR name and namespace:

kubectl logs -n widget-operator-system <operator-pod-name> -f | grep 'special-widget\|factory'

Look beyond generic errors ("failed to update status"). Search for:

Errors trying to `GET`, `CREATE`, or `UPDATE` specific resources (Deployments, Services) owned by the CR.

Permission errors (Does the operator's ServiceAccount have RBAC rights in the `factory` namespace?).

API server errors (throttling, network issues).

Internal operator logic errors or panics related to the `special-widget` spec.

Timeout errors – what operation timed out?

3. Verify Operator RBAC Permissions:

Identify the ServiceAccount used by the operator Deployment.

Check the Roles/ClusterRoles and Bindings associated with that ServiceAccount:

kubectl get clusterrolebinding -o wide | grep <operator-serviceaccount-name>

kubectl get rolebinding -n factory -o wide | grep <operator-serviceaccount-name>

Describe the relevant Role/ClusterRole to see permissions

kubectl describe clusterrole <operator-clusterrole-name>

Ensure the operator has `CREATE`, `GET`, `LIST`, `WATCH`, `UPDATE`, `PATCH`, `DELETE` permissions for Deployments, Services (or whatever resources it manages) in the `factory` namespace. Also check permissions for the `Widget` CRD itself (especially `update` on the `status` subresource).

4. Check for Resource Conflicts or Quotas:

Could the operator be trying to create a Deployment or Service that conflicts with an existing resource name in the `factory` namespace?

Are there ResourceQuotas in the `factory` namespace that might prevent the operator from creating new Deployments, Services, or Pods?

kubectl get resourcequota -n factory

kubectl describe resourcequota <quota-name> -n factory

5. Inspect Finalizers:

Does the `special-widget` CR have a `metadata.finalizers` list?

kubectl get widget special-widget -n factory -o jsonpath='{.metadata.finalizers}'

Finalizers prevent deletion until the controller removes them. While usually related to deletion issues, a misbehaving operator might fail to manage them correctly even during reconciliation, potentially leading to status update failures if it tries to add/remove one inappropriately.

6. Test API Server Connectivity from Operator Pod:

Exec into the operator pod:

kubectl exec -it -n widget-operator-system <operator-pod-name> -- /bin/sh

Try basic `kubectl` (if installed) or `curl` commands against the Kubernetes API server from within the pod to rule out fundamental connectivity issues:

Assuming kubectl is available and configured

kubectl get deployment -n factory

Using curl (requires finding token and CA cert paths)

curl --cacert /path/to/ca.crt --header "Authorization: Bearer $(cat /path/to/token)" https://$KUBERNETES_SERVICE_HOST:$KUBERNETES_SERVICE_PORT/apis/apps/v1/namespaces/factory/deployments

7. Address the Root Cause (Example: Status Update Conflict):

Assume operator logs reveal repeated "the object has been modified; please apply your changes to the latest version and try again" errors specifically when trying to update the `special-widget` status. This often indicates another controller (or manual edit) modified the CR between the operator's read and write, or the operator's caching mechanism is stale.

Action: Review the operator's reconciliation logic. Ensure it uses `RetryOnConflict` error handling when updating status or spec. Check if any other controllers might be unexpectedly interacting with `Widget` CRs. Ensure the operator's client/cache is configured correctly. Restarting the operator pod might temporarily resolve cache issues but doesn't fix the underlying logic flaw. Fix the operator code to handle conflicts gracefully.

8. Observe Reconciliation:

After applying fixes (to RBAC, quotas, CR spec, or operator code/deployment), monitor the operator logs and the `special-widget` CR status:

kubectl logs -n widget-operator-system <operator-pod-name> -f | grep 'special-widget'

kubectl get widget special-widget -n factory -w

The operator should now successfully reconcile the CR, create the associated resources, and update its status.

Outcome:

The specific reason why the `widget-operator` failed to reconcile the `special-widget` CR (e.g., invalid spec, insufficient RBAC, resource quota, API conflict error, operator bug) is identified.

The configuration (CR, RBAC, Quota) or the operator itself is corrected.

The operator successfully reconciles the `special-widget` CR, creating/updating its associated resources and setting the appropriate status conditions.

Scenario 135: Diagnosing Intermittent High Network Latency Between Specific Pods

Scenario

Two services, `frontend` in the `web` namespace and `backend` in the `api` namespace, communicate frequently. Users report sporadic slowdowns. Monitoring shows occasional high latency (hundreds of ms) specifically for requests from `frontend` pods to the `backend` service ClusterIP. Direct pod-to-pod communication tests also show intermittent high latency, but it doesn't affect all nodes or pods equally. DNS resolution is fast, and NetworkPolicies are permissive between these namespaces.

Solution - Context:

kubectl config use-context k8s-c26-net

Steps:

1. Baseline and Characterize Latency:

Establish normal latency vs. high latency periods.

Use tools like `ping` and `curl` (with timing) from multiple `frontend` pods to the `backend` Service ClusterIP and directly to individual `backend` pod IPs.

From a frontend pod

apk add curl iputils # Or apt-get update && apt-get install curl iputils-ping

Test Service IP

ping -c 10 <backend-service-cluster-ip>

curl -o /dev/null -s -w 'Connect: %{time_connect}s\nTTFB: %{time_starttransfer}s\nTotal: %{time_total}s\n' http://<backend-service-cluster-ip>:<port>/health

Test direct Pod IP (find IPs via kubectl get pods -n api -o wide)

ping -c 10 <backend-pod-ip-1>

curl -o /dev/null -s -w '...' http://<backend-pod-ip-1>:<target-port>/health

ping -c 10 <backend-pod-ip-2>

curl -o /dev/null -s -w '...' http://<backend-pod-ip-2>:<target-port>/health

Try to correlate high latency with specific source `frontend` pods/nodes or specific destination `backend` pods/nodes.

2. Inspect CNI Plugin Logs:

Check logs for the CNI plugin daemonset pods (e.g., Calico, Cilium, Flannel) on the nodes hosting the affected `frontend` and `backend` pods.

Find CNI pods (often in kube-system)

kubectl get pods -n kube-system -l k8s-app=<cni-label> # e.g., k8s-app=calico-node

Check logs on relevant nodes

kubectl logs -n kube-system <cni-pod-on-frontend-node> -f

kubectl logs -n kube-system <cni-pod-on-backend-node> -f

Look for errors related to IP address allocation/management (IPAM), network interface setup, policy enforcement (even if permissive), or underlying network device issues.

3. Examine `kube-proxy` Logs and Configuration:

`kube-proxy` manages Service routing (iptables, ipvs). Check its logs on affected nodes:

kubectl get pods -n kube-system -l k8s-app=kube-proxy

kubectl logs -n kube-system <kube-proxy-pod-on-frontend-node> -f

kubectl logs -n kube-system <kube-proxy-pod-on-backend-node> -f

Look for errors related to updating routing rules or connection tracking.

Check if `kube-proxy` is using `iptables` or `ipvs` mode (`kubectl describe configmap kube-proxy -n kube-system`). `iptables` mode can suffer performance degradation with very large numbers of services/rules.

4. Check Node-Level Network Performance:

SSH into the nodes hosting the affected source `frontend` pod and destination `backend` pod. Test node-to-node network latency and bandwidth using tools like `ping` and `iperf3`:

On backend node:

sudo apt-get update && sudo apt-get install iperf3 # Or yum install iperf3

iperf3 -s

On frontend node:

sudo apt-get update && sudo apt-get install iperf3

ping -c 10 <backend-node-ip>

iperf3 -c <backend-node-ip> -t 10

High node-to-node latency points towards issues in the underlying physical/virtual network fabric or node network interfaces.

5. Investigate Network Interface Errors/Drops:

On the affected nodes, check network interface statistics for errors, drops, or overruns:

ip -s link show eth0 # Or the primary node interface

netstat -i # Check RX-DRP/TX-DRP, RX-ERR/TX-ERR

ethtool -S eth0 | grep -i 'error\|drop' # Detailed driver statistics

Investigate any increasing error/drop counters.

6. Consider MTU Mismatches:

Ensure consistent MTU (Maximum Transmission Unit) settings across nodes, pod networks (CNI configuration), and the underlying network fabric, especially if using overlays (VXLAN, Geneve). Mismatches can cause fragmentation or packet loss.

ip link show # Check MTU on node interfaces (eth0, tunl0, vxlan.calico etc.)

Check CNI config for MTU settings

Check pod interface MTU (exec into pod, run 'ip link show eth0')

7. Analyze Potential "Noisy Neighbor" Pods:

Could another pod on the same node as either the `frontend` or `backend` pod be saturating the node's network interface or CPU, impacting the performance of other pods?

Use node-level monitoring (`iftop`, `nethogs`) or CNI-specific tooling (if available) to check per-pod network usage on the affected nodes during high-latency periods.

8. Address the Root Cause (Example: CNI Overlay Issue on One Node):

Assume investigation (CNI logs, node-to-node iperf) reveals that traffic specifically involving `node-X` (hosting a backend pod) experiences high latency when using the CNI overlay (e.g., VXLAN), but node-to-node raw IP traffic is fine. CNI logs on `node-X` show occasional errors related to the overlay interface.

Action: Restart the CNI agent pod on `node-X`. If the issue persists, investigate deeper into `node-X`'s kernel/driver interaction with the overlay, consider draining/rebooting the node, or check for known CNI bugs related to the specific kernel version/setup.

9. Verify Latency Improvement:

Repeat the latency tests from Step 1 after implementing the fix. Confirm that the intermittent high latency between `frontend` and `backend` pods is resolved.

Outcome:

The source of intermittent high network latency (e.g., CNI issue on a specific node, `kube-proxy` bottleneck, underlying network fabric problem, MTU mismatch, noisy neighbor pod) is identified.

Corrective actions are taken at the appropriate level (CNI config/restart, node maintenance, underlying network fix, MTU alignment).

Network latency between `frontend` and `backend` pods becomes consistently low and predictable.

In the up-coming parts, we will discussion on more troubleshooting steps for the different Kubernetes based scenarios. So, stay tuned for the and follow @Prasad Suman Mohan for more such posts.