Improve Application Availability by Configuring Kubernetes Liveness & Readiness Probe

Pallavi Priya Patharlagadda

United States of America
Email: *pallavipriya527.p[at]gmail.com*

Abstract: ***Because of Kubernetes' extraordinary flexibility, enterprises can simply install, scale, and manage production-grade, containerized workloads. Teams must navigate a lot of foreign terminology and technology in addition to tremendous complexity as a result of this flexibility. When deploying mature applications to Kubernetes, there are two phrases you should be aware of: readiness probe and liveness probe. To find out if a container is still running and responsive, we will cover using a liveness probe in Kubernetes clusters in this guide.***

Keywords: Kubernetes, containerized workloads, readiness probe, liveness probe, application deployment

1. Problem Statement

Kubernetes is an opensource system that helps to manage, scale, and deploy containerized applications. To achieve availability and scalability, Kubernetes need to know the application status. Imagine a situation where an application is stuck in a deadlock state and is nonresponsive. Similarly, imagine an application is still booting and Kubernetes starts sending the traffic. In both cases, we can see there is a drop in the traffic. How does Kubernetes know about the application readiness and liveness? Kubernetes provides probing to know the readiness and liveness of the application. we can discuss how to configure these in the subsequent sections.

2. Introduction

Developed in the Go program language, Kubernetes is an open-source container orchestration system that was first made available in 2014 under the terms of the Apache License 2.0. Although Google established it initially, the Cloud Native Computing Foundation (CNCF) is now responsible for its maintenance.

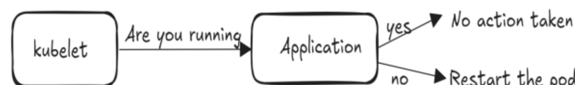
Congratulations on creating and deploying a fantastic app to Kubernetes! But how can you ensure that your application continues to function as intended? Liveness and readiness probes can help with that. By using these probes, Kubernetes can keep an eye on your application and make sure it's stable, responsive, and prepared to receive requests. Without them, Kubernetes wouldn't be able to detect whether your application has crashed or isn't operating correctly.

This post will explain liveness and readiness probes, their purpose, their significance, and how to set them up for Kubernetes deployments. You will learn how to configure both basic HTTP checks and more intricate exec probes. By the conclusion, you will understand how to maintain the health and functionality of your Kubernetes apps. Sounds excellent? Then let's get started!

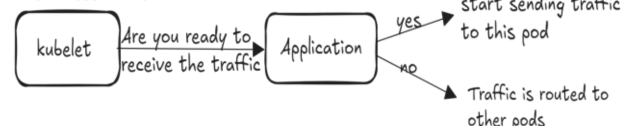
1) What are Liveness and Readiness Probes?

In Kubernetes, liveness probes are vital indicators for preserving the functionality and well-being of applications operating in containers. These probes are intended to identify and manage situations in which an application may be running but is not functioning because of problems like memory leaks, deadlocks, or other circumstances that cause it to become unresponsive. Kubernetes may ensure that services recover from failure without manual intervention by automatically restarting containers that fail the check by implementing a liveness probe. In production situations, where downtime can have serious consequences, this automatic recovery method is extremely important for ensuring service availability and continuity.

1. Liveness Probe



2. Readiness Probe



On the other hand, Readiness Probes concentrate on an application's operational preparedness to handle traffic. Kubernetes only forwards traffic to pods that are ready to process it, thanks to their assurance. This implies that Kubernetes waits to deliver requests to these pods when an application is starting up, loading big datasets, or going through initializations. The pod does not receive traffic until the readiness probe indicates that it is ready. This approach is essential to load balancing and the seamless running of services because it shields inactive services from incoming traffic, which could cause mistakes or lower end-user performance.

2) Why is it Important to Use Liveness and Readiness Probes?

The significance of Liveness and Readiness Probes increases in the context of cloud-native development when apps are distributed and operated on dynamic, scalable infrastructures like Kubernetes. Here's a closer examination of their relevance:

- With the use of liveness probes, Kubernetes can initiate automated self-healing activities by restarting containers that exhibit non-functionality. When dealing with

Paper ID: SR24820065625 DOI: https://dx.doi.org/10.21275/SR24820065625

problems that require a restart, this is very helpful in making sure that programs self-recover with the least amount of downtime.

- A key component of Kubernetes' load-balancing and scalability capabilities is readiness probes. Kubernetes may efficiently distribute requests among numerous instances, improving the overall responsiveness and dependability of applications, by guaranteeing that only containers that are ready get traffic. Additionally, this stops the cascade failure effect, which happens when overloaded or failed services affect other services' performance.
- Proximity and preparedness probes work together to maximize the efficiency of processing power. Kubernetes may guarantee effective resource allocation by restarting or stopping non-functional containers, giving priority to instances that are healthy and prepared for use.
- Maintaining a superior user experience is ultimately these probes' main objective. Kubernetes assists in providing constant, dependable service performance by protecting against routing traffic to unavailable or malfunctioning services. Consumer satisfaction is greater overall, with fewer errors and quicker response times.

Five options are available in Kubernetes to regulate these probes:

Name	Mandatory	Description	Default Value
`initialDelaySeconds`	Yes	Determines how long to wait after the container starts before beginning the probe.	0
`timeoutSeconds`	Yes	Determines how long to wait for the probe to finish. If this time is exceeded, then Kubernetes assumes that the probe failed.	1
`periodSeconds`	No	Specifies the frequency of the checks.	10
`successThreshold`	No	Specifies the minimum consecutive successes for the probe to be considered successful after it has failed.	1
`failureThreshold`	No	Specifies the minimum consecutive failures for the probe to be considered failed after it has succeeded.	3

3) Implementing Liveness and Readiness Probes in Kubernetes

We'll go in-depth into using Liveness and Readiness Probes in Kubernetes to simulate a real-world situation in the current section. We aim to guarantee the continuous high availability and effective traffic handling of our Kubernetes-managed application. We'll utilize a straightforward web application as our example service for this reason.

a) Liveness Probe:

Assume that our program is being run by a Pod inside a container, but for whatever reason—for example, a memory leak, excessive CPU consumption, an application deadlock, etc.—the application is not responding to our requests and is stuck in error mode. The liveness probe performs the tasks we instruct it to do, monitoring the health of the container and restarting it if it malfunctions. Three definitions of liveness probe exist:

Liveness command:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-pod
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 1200
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 30
      periodSeconds: 20
```

We are naming our container "liveness," and we are using the following command to initialize it:

- touch /tmp/healthy; sleep 60; rm -rf /tmp/healthy; sleep 1200

to create a file healthy at path /tmp/healthy, and delete it after 60 seconds.

```
exec:
  command:
  - cat
  - /tmp/healthy
```

With this command, the liveness probe is instructed to open the file at path /tmp/healthy; if it is unable to do so, the liveness probe will fail and the container will restart.

initialDelaySeconds: 30

This delay instructs Kubelet to hold off on running the first probe for thirty seconds.

periodSeconds: 20

This parameter tells Kubelet to run a probe once every twenty seconds.

Thus, based on the example above, our container will start and function properly for the first sixty seconds, at which point the liveness probe will fail and the container will restart.

Liveness HTTP request:

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 10
    periodSeconds: 5
```

In this instance, the application executing within the container will get an HTTP GET request from the kubelet addressed to the /healthz endpoint at port 8080. The Liveness probe will fail if the answer is an error. If not, it will regard the application as active.

TCP Liveness probe:

```
livenessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 20
     periodSeconds: 15
```

The application-running container's port 8080 will be attempted to be opened by the kubelet in this instance. The

application will be deemed healthy if it is successful; if not, the probe will fail and the container will restart.

b) Readiness Probe:

Sometimes we want our application to run, but we don't want it to serve traffic until certain requirements are satisfied, like filling a dataset or waiting for another service to start up, among other things. We employ readiness probes in these situations. Only after the readiness probe's condition is satisfied can our application handle traffic. The three definitions of the readiness probe are the same as those of the liveness probe.

4) When to use readiness probes?

When you want to make sure a container is ready to handle incoming network traffic before it begins taking requests, readiness probes come in handy.

Generally speaking, they are employed in situations when your application or container has to do lengthy startup operations before handling requests, such as loading configuration, creating database connections, connecting to message brokers and other microservices, or preheating caches.

Proactive readiness probes may also assist in coordinating the release order of your stateful applications. In some cases, you may need to make sure that specific services or components are completely functional before allowing them to communicate with other areas of the application.

They may also be involved in deploying upgrades and making sure pods scale smoothly. Through the use of readiness probes, you may hold off on transmitting traffic to newly created pods until they are prepared, as well as to pods that may be having problems as a result of an upgrade.

It's important to take into account any resource limitations in your cluster, since certain containers may require more time to process requests during periods of heavy demand or until they recover from resource depletion. One way to wait for the container to recover enough is to use readiness probes to postpone routing traffic.

1. HTTP Readiness Probe:

```
apiVersion: v1
kind: Pod
metadata:
  name: readiness-pod
spec:
  containers:
  - name: readiness-container
    image: readiness-image
    ports:
    - containerPort: 8080
    readinessProbe:
      httpGet:
        path: /somepath
        port: 8080
      initialDelaySeconds: 20
      periodSeconds: 15
```

2. TCP readiness probe example:

```
apiVersion: v1
kind: Pod
metadata:
  name: readiness-pod
spec:
  containers:
  - name: readiness-container
    image: readiness-image
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 20
      periodSeconds: 15
```

3. Command readiness probe example:

```
apiVersion: v1
kind: Pod
metadata:
  name: readiness-pod
spec:
  containers:
  - name: readiness-container
    image: readiness-image
    ports:
    - containerPort: 80
    readinessProbe:
      exec:
        command:
        - /bin/sh
        - -c
        - check-readiness.sh
      initialDelaySeconds: 20
      periodSeconds: 15
```

5) Best Practices for Using Probes in Kubernetes

It takes more than simply setting up to implement Liveness, Readiness, and Startup Probes in your Kubernetes deployments. It necessitates carefully analyzing the unique behaviors, requirements, and operational settings of your application. These enhanced best practices will help you maximize the use of probes in your cloud-native apps.

a) Select the Appropriate Probe Type

Every kind of probe has a unique function and is appropriate in a variety of situations:

HTTP GET Probes: Perfect for online applications where the health or readiness condition of the service can be simply represented via an HTTP endpoint. It is appropriate for services that can use HTTP status codes to report their status because it is simple and less intrusive.

TCP Socket Probes: Beneficial for applications that don't always require an HTTP server to be operational but where establishing a TCP connection is a dependable sign of health or preparedness. Databases and other backend services that listen on a certain port frequently experience this.

Exec Command Probes: With the ability to run custom instructions to assess the condition or preparedness of your

service, these probes provide the greatest flexibility. When checking numerous internal states or when the rationale for the health check is complex, this is especially helpful.

b) Set Up Timeouts and Delays Suitably

Initial Delay Seconds: With this configuration, your application has enough time to launch before the kubelet starts running the probes. A miscalculation of this latency may cause the kubelet to begin checking before the application is ready, which could result in premature probe failures.

Period Seconds: Establishes the frequency of probe execution. If you set this number too high, it may take longer to discover problems, and if you set it too low, it may overwhelm your application with health checks.

Timeout Seconds: The duration in seconds that the probe will time out. To prevent false negatives, it's important to give your application adequate time to reply, especially when it's under a lot of pressure.

c) Use Startup Probes to Help Containers That Start Slowly

Startup probes are an excellent complement to your Kubernetes health check plan, especially for apps that take longer to start up. They guard against the danger associated with aggressive liveness probe setups, which is the kubelet destroying the container before it has finished starting.

Use Case: If your service must import substantial datasets, gather data, or carry out any extended initialization processes, a startup probe makes sure Kubernetes does not see the application as unsuccessful during this beginning stage.

Configuration Tips: Determine the failure Period and threshold the startup probe's seconds to numbers that correspond to the longest possible starting time for your program. The liveness and readiness probes take over to control the program's lifecycle once the startup probe completes its initial run, signaling that the application is ready.

d) Track and Modify in Light of Observations

Logging and Monitoring: To gain insight into the behavior of your probes in production, provide comprehensive logging for the health endpoints in your application and keep an eye on these logs. This understanding will assist you in optimizing probe settings.

Iterative Refinement: The behavior of your application may alter over time as a result of demand fluctuations, dependency updates, or code changes. To ensure that your probes continue to be accurate and effective in the face of these changes, periodically evaluate and modify the setups.

e) Consider the Side Effects of the Probe

Performance Impact: Probes can add to the strain on your application when they are required. Take into account the performance effect of the probe execution, especially if it occurs frequently, especially when using exec and HTTP probes.

Idempotency: Make sure that when you run your readiness and liveness probes, your application remains in the same state. This is known as idempotence. This is essential to prevent unexpected side effects that can change the behavior of the program.

3. Conclusion

This concludes our review of Kubernetes liveness and readiness probes and how they support the seamless operation of your installations and apps. You can make sure your pods stay healthy and that users can still use your services by setting up these health checks. To keep problems from affecting your users, Kubernetes will automatically restart unhealthy pods or remove them from load balancers. Determining health checks may look like additional labor at the front, but the time savings from automated self-healing and less troubleshooting make it worthwhile. You'll have more time to concentrate on creating fantastic applications because your Kubernetes cluster will be operating smoothly.

References

[1] https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
[2] https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
[3] https://medium.com/@AADota/kubernetes-liveness-and-readiness-probes-difference-1b659c369e17
[4] https://medium.com/@sachinadi424/keeping-kubernetes-healthy-liveness-and-readiness-probe-explained-661757b825ff
[5] https://kubebyexample.com/learning-paths/application-development-kubernetes/lesson-4-customize-deployments-application-2
[6] https://spacelift.io/blog/kubernetes-readiness-probe