Queuing and scaling in Kubernetes infrastructure

PS6 - Computer Science

Report

Fribourg, January 2021

Romain Agostinelli

romain.agostinelli@edu.hefr.ch

Supervisors:

Mäder Michael

michael.maeder@hefr.ch

Supcik Jacques

jacques.supcik@hefr.ch

Version: 4.0

Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Contents

1 Document's history

Sous cette section se trouve l'historique du document avec ce qui a été ajouté, supprimé ou modifier en fonction des versions.

Version	Modifications
v1.0	Document creation Introduction Project's environment
v1.1	Technical research Scaling K8s HPA
v1.2	Custom Metrics Server K8s Jobs Sidecar pattern Forwarding - Receiving
v.1.3	Passing K8s env. var. Queuin with KubeMQ
v.2	Monitor KubeMQ
v.2.1	Finding solution iterations Kueuer - Conception design Kueuer - Proxy Kueuer - QAdapter
v.2.2	Kueuer - Responder
v.3	Kueuer - Queues/Monitoring HPA Kueuer Evolutions
v.3.1	Tests
v.3.2	Refactoring Research - Reverse Proxy
v.4	BibTex Glossary Final check

2 Introduction

This project consists in developing a system that provide mechanisms to handle a high rate of request/second to a service considering that this service can take up to 30sec to answer. The goal is to limit the "waiting" time of each requests and do not create a bottleneck as the requests arrives.

For example, let's consider a movie theater with theses constraints:

- A new film has just been released and everyone wants to see it.
- Before entering the movie theater, people must give their ticket to a ticket booth.
- The duration of the film is 30sec.
- It can only have one person in the movie theater room to see the film.
- People don't want to wait more than 30sec before being redirected to a room.
- A cinema room costs money and we try to minimize the total cost.

In the example, the service is the movie theater that offers different path (films) and the people coming to the cinema are the requests. The duration of the film refers to the computation time of the service for this path.

The enumerated constraints above describe the problem we must resolve. How to make all the people happy? How to satisfy all these constraints?

In this project, we'll investigate how to solve this problem by using queues and (auto-) scaling.

Queuing in IT is the same idea of queuing in real life. If we take back the described situation with the cinema, queuing is setting a queue before the ticket booth so people can wait here and be taken in the order they arrived.

Scaling is the concept of increasing the number of cinema room regarding the number of people waiting and decreasing this number if nobody is here (at midnight for example) because an opened cinema room costs money.

3 Project's environment

This section describes in which environment the project is done, for which use case and so on. It contains two subsections referring to the main infrastructure aspects: first the infrastructure in which the project takes place and then the technologies used for external components.

3.1 Infrastructure

The project takes place in a Kubernetes (K8s) infrastructure. The services are in a K8s environment and the requests are directed to the K8s cluster.

For containerization technology, Docker is used. Be careful because K8s just announce (at the time this project has begun) that Docker will be deprecated in the future[1].

3.1.1 Kubernetes

On K8s' website[2] we found a sentence that describe really well what Kubernetes is: "Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling,

[1] https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/
[2] http://kubernetes.io/

and management of containerized applications".

K8s was initially developed by Google to manage a lot of services deployed in container. K8s is not a container platform as Docker, but it uses it.

Kubernetes allows you to manage your container by dispatching them on different machines, by creating namespaces, clusters and so on. Bellow, we discuss some features or terms related to K8s and necessary to understand for the project. Please refer to the graphic on figure 1 :

- Cluster: A cluster is a logic component representing a whole infrastructure. The cluster contains all the nodes of the infrastructure. It is the biggest component.
- Node: A node represents a machine. This machine can be a physical machine or a virtual machine that runs different pods and/or service, etc. It must have a container run-time.
- Pod: A pod is the smallest object of K8s. It's a logical object that contains containers. Generally, there is one container per pod but for some use cases, a pod can contains multiple containers. When a pod contains multiple container, we can call them "side-cars" container. There is generally one primary container and multiple sidecars that can run in the same pod to help/interact with the primary container.
- Service: A service is component running on the node (same logical level than a pod) and creates a gateway between the Web and the cluster for one pod (generally). A service is most of the time linked to only one pod and provides access to the pod via web. If a pod is replicated (two or more same pods running in the same time), the service is capable of load balancing the traffic between all the replicas of the pod.

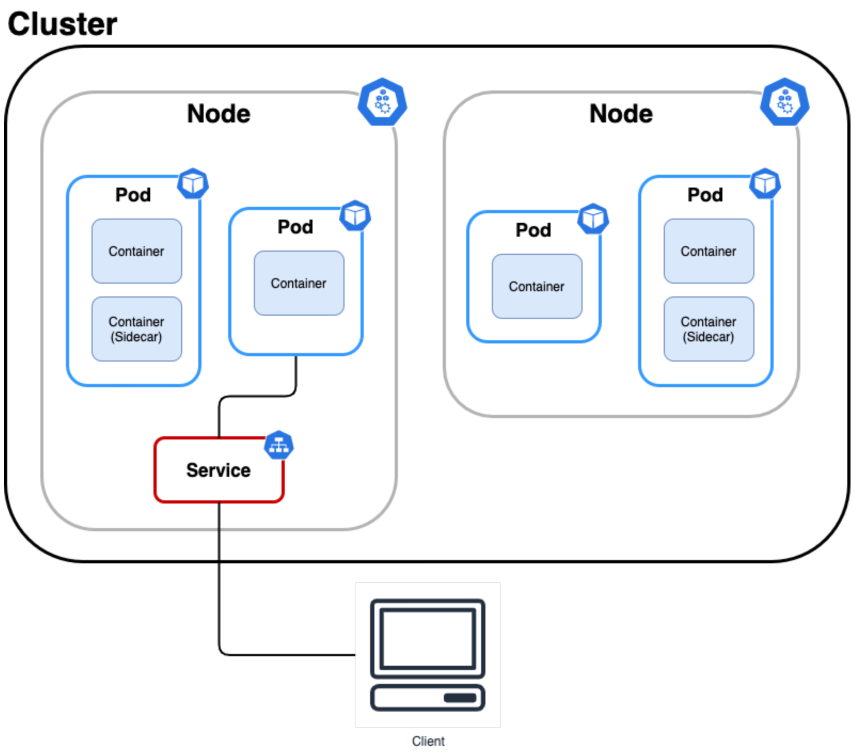


Figure 1: Basic K8s objects

3.1.2 Minikube

Minikube is a small local K8s that can be easily installed on different OS like Windows, macOS and Linux. It is used to deploy a Kubernetes cluster with one node, which is the machine on which Minikube is launched. Minikube provides a fast installed Kubernetes development environment and allows you to choose between different container providers. It is adapted for development and not for production.

More can be seen on Minikube's website[3].

As an alternative for Minikube, there is Kind[1] that provides a quick way to setup a running cluster on our machine. This solution is also proposed by Kubernetes.

For this project, Minikube is used because at the beginning of the project I wasn't familiar with Kubernetes and all the demonstration on Kubernetes' website are with Minikube. So I decided to use it for simplicity.

3.2 Technologies

This chapter discusses different technologies that is used in the project, like programming language, framework and queuing solution.

3.2.1 Programming Language

In this project, Java is used to develop different solutions or tests. The Java language has the advantages to be highly known and fast for API.

Kubernetes is developed in Go and all that is developed for is in Go. Unfortunately, I (Romain Agostinelli) don't know Go and even if it would have been the right choice, I decided to not use Go as it can slow me down during the project.

Java seems a good alternative, not as fast as Go but more known and easy to setup.

3.2.2 Frameworks

As Java is used as programming language, the fastest way to setup Rest application is to use Spring Boot. Spring Boot provides a fast way to setup Java application and it has a big community with lot of tutorials and solutions.

3.2.3 Queuing

For the queuing part, there are multiple technologies that can be used. Just to cite the most well known and used:

- Apache Kafka
- RabbitMQ
- Redis

Instead of the three cited queuing solutions, there is one that is really recent (2018-2019) and seems amazing on the paper. It is a Kubernetes Native Queuing System named KubeMQ. Compared to the others queuing solutions, it has a feature that is important in this project: Exactly one Delivery. This feature that the others don't have means that each message will be delivered exactly once so it guarantees us that each message putted in a queue will be given to a service only once. So, if multiple services/threads are connected at the same time to a message queue, we are sure that only one will received the message and then it will be next processed only once too.

[3]https://minikube.sigs.k8s.io/docs/

For more comparisons between KubeMQ and other solutions as Kafka, Redis and RabbitMQ, there is a page dedicated to it on KubeMQ's website[4].

For this project, KubeMQ will be used as queuing solution.

4 Technical research

This section discusses all the researches and tests done with different technology before finding a concrete solution.

4.1 Scaling

There are two types of scaling: Vertical scaling and Horizontal scaling.

Vertical scaling consists of adding resources (such as RAM size or CPU core). Horizontal scaling consists in increasing the number of total machines by duplicating the existing ones and distributing the load between the replicas. We can see a graphical representation on figure 2

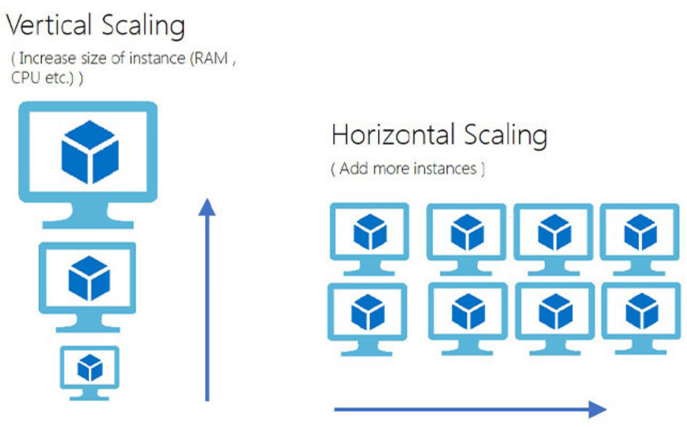


Figure 2: Horizontal vs Vertical scaling

Source: webairy.com

The vertical scaling is not appropriate for our problem because the issue is not that there aren't enough resources for computing but the computation just take times and there is to much requests.

4.2 K8s HPA

In Kubernetes environment, HPA stands for "Horizontal Pod Autoscaler". It is a system that can scale up (or down) the number of a pod regarding different metrics. A HPA is configured for only one pod, so, if there are several pods (not replicas of pod but clearly different pods who run various containers) that must be auto-scaled, it will have several HPA.

[4]https://kubemq.io/compare-kubemq/

4.2.1 How it works

There is multiple Kubernetes components involved into the HPA working flow (figure 3):

- CAdvisor: this is a lightweight component that run on each Node by default. It is capable of retrieving the CPU and memory usage of each pod inside its node.
- Kubelet: running on each Node, it offers the possibility to communicate with its node via API.
- Metric-Server: cluster-wide component (aggregator)[2] that is not installed by default. Its job is to collect and aggregate all metrics (memory and CPU) from all nodes (via Kubelet) and provide these information through the Kubernetes API : "/apis/metrics.k8s.io/".
- HPA: component working as controller (more info on controller pattern here[3]) that use the Kubernetes API to know the metrics for the pod it is linked to.
- Control Plane: logical view of various components which control the Kubernetes architecture. More info here[4].

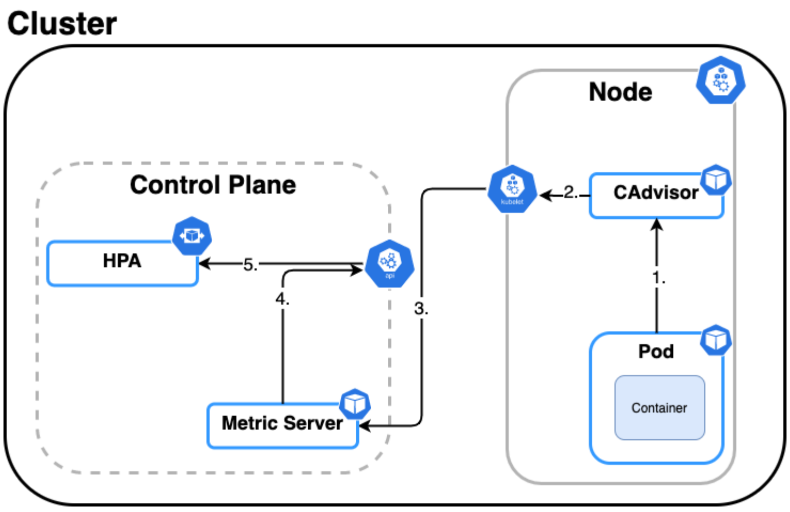


Figure 3: HPA working flow

On the figure 3 above, there is multiple steps for a HPA to works properly. Here their description:

1. CAdvisor collect information for each pod in its node.
2. CAdvisor provide collected information for its node via the kubelet component.
3. Metric-Server aggregate the metrics for each node in the cluster using the kubelet of each node.
4. Metric-Servere provide metric for the whole cluster (each node/container of the cluster) via Kubernetes API.
5. HPA retrieves information about the pod that it must scale via the Kubernetes API.

By default, the CAdvisor and Metric-Server can retrieve only two types of metrics:

- Memory
- CPU

So the HPA, in its base configuration, can only be configured on these two metrics.

When setting up a HPA for a pod, the K8s infrastructure must know how many resources a container is allowed to use (limit) and how many it requests to run. If a node has enough resources, a container can use more than it requests, but not more than its limit. Resources can be CPU consumption and/or memory consumption. It is mandatory to setup the limit of each container in the pod deployment to use HPA on this pod. This way, the HPA can know when to scale up/down a pod. For example, if a container inside the pod begins to use more CPU than its limit, the HPA will scale up the pod to distribute the load between multiple replicas of the same pod to maintain a CPU consumption between request and limit for each pod. The full documentation is available here[5].

The next chapter (ch. 4.3) will discussed about the possibility to setup a custom metric server that can allow to provide more than these two type of metrics and maybe offers the HPA to be based on different custom metrics.

4.2.2 Installation and tests

The documentation about the installation is available here[6].

As seen above, the K8s infrastructure needs a metric server to aggregate all the metrics from all the nodes. The metric server deployment file can be found on the official repository. Warning: If you are running it on Minikube or Kind, please add two lines as follow in the metrics-server Deployment under the container spec (inside the template):

```
command:
- /metrics-server
- --kubelet-insecure-tls
```

There is an example on the project repository.

When the metric server is running, it's time to configure the application we want to automatically scale. For this, the application needs to tells its limits. To do so, a simple array can be added to the Deployment file of the pod under each container presents inside the template:

```
resources:
  limits:
    cpu: 100m
  requests:
    cpu: 100m
```

A full example of a Deployment file is available here[5] and can be used as it is.

Finally, when everything is up deployed, it is time to setup a HPA. For this, a simple command can be used which contains the metric to look at for autoscaling, the minimum replicas of the pod and the maximum:

```
#Generic command:
kubectl autoscale deployment <name_of_deployment> <metric> --min=<min>
 ↪  --max=<max>
#Example for the deployment cited above
kubectl autoscale deployment nodejs --cpu-percent=20 --min=1 --max=10
```

[5]https://gitlab.forge.hefr.ch/ps6-2021-kubemq/deployment/-/blob/cbae9c6728c859e68891d41214121b4468749291/simple-rest-deployment.yaml

A HPA can also be configured to use multiple metrics and it can be done using a deployment file like this:

```yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

More informations available on the HPA setup walkthrough available here[6].

4.3 Custom Metrics Server

As discussed in the previous chapter (ch. 4.2), a HPA is based on metrics that the metric-server provide. By default, the metric server offers 2 types of metrics:

- CPU consumption
- Memory consumption

Theoretically, there is the possibility to create a custom metric server that provide information for a pod based on a custom worker. This way, it will become possible to deploy a custom metric server and base the HPA on this server and do not change the way all works.

The current version (v1beta2) of what a metric server must serve through its API for custom metrics is available here[6] and this uses the global "k8s.io/apimachinery/pkg/apis/meta/v1" for common (to all APIs in K8s) response information. This second file is available here[7]. These two files are written in Go.

After compiling and aggregating information, it is possible to create a Json structure of what a custom metric server must serve:

```json
{
    "kind": "MetricValueList",
    "apiVersion": "custom.metrics.k8s.io/v1beta2",
    "items": [
        {
            "metric": {
                "name": "<name_of_custom_metric>"
```

[6]https://github.com/kubernetes/metrics/blob/master/pkg/apis/custom_metrics/v1beta2/types.go
[7]https://github.com/kubernetes/apimachinery/blob/master/pkg/apis/meta/v1/types.go

```
            },
            "describedObject": {
                "kind": "<type_of_object>",
                "apiVersion": "/v1beta2",
                "name": "<name_of_object>",
                "namespace": "<namespace_of_object>"
            },
            "timestamp": <timestamp_in_ISO_form>,
            "value": "<value_of_metric>"
        }
    ]
}
```

The server must also serve this information under a defined path. There is no concrete list of path that a custom metric server must serve. There is old proposal[8] that is not up to date but on which it is possible to extract some information.

After investigation and tries, the custom metric server must serve this kind of path:

```
# The name in <> are correlated to the previous json description.
"/apis/custom.metrics.k8s.io/v1beta2/namespaces/<namespace_of_custom_metric_service>
    /services/<name_of_custom_metric_service>/<name_of_custom_metric>"
```

With those information it is possible to create a custom metric server and compute our own metrics.

4.3.1 Installation and tests

As discussed before, it is possible to setup a custom metric server than can be helpful if there is need to scale a kubernetes component regarding a queue length or load computed from something else than memory or cpu.

There is multiple steps to setup a custom metrics server:

First, create the server. It is possible to create a custom metrics server easily regarding what the research give us about what it must serve and on which path it must serve it. But, there is another thing: a service linked to the Kubernetes APIService (see ch. 4.2 to know how metric server works) must offer a certificate and TLS connection. It is possible to create self-signed certificate to give to our custom-metrics server.

Then, a service must be deployed to provide access to the custom-metrics server pods. The service must offer a https path.

After having a custom server running and a service that provide access to it, we need to register this service as provider for the K8s API: v1beta2.custom.metrics.k8s.io. This is done via deployment of an object of kind: APIService.

```
apiVersion: apiregistration.k8s.io/v1
kind: APIService
metadata:
  name: v1beta2.custom.metrics.k8s.io
spec:
```

[8] https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/resource-metrics-api.md

```
  insecureSkipTLSVerify: true # Needed if using Minikube
  group: custom.metrics.k8s.io
  groupPriorityMinimum: 1000
  versionPriority: 5
  service:
    name: custom-metrics-service
    namespace: default
  version: v1beta2
```

Finally, it is the time to setup a HPA based on our custom metric. To do this, instead of using the metric type "Resource" like in the chapter 4.2.2, we must use the type "Object" to configure exactly what we want. The "Object" kind contains multiple entries but the most interesting in a custom metrics server is the field named "describedObject". This field is the description of the custom metrics service (warning: service not server).

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: nodejs-hpa
spec:
  scaleTargetRef: # Object on which the HPA will have effect on
    apiVersion: apps/v1
    kind: Deployment
    name: nodejs
  minReplicas: 1
  maxReplicas: 30
  metrics:
  - type: Object
    object:
      target:
        type: Value # use metric value, not average value
        value: 12 # here the target value of the metric
      describedObject: # description of the custom metrics service
        apiVersion: v1
        kind: Service
        name: custom-metrics-service
      metric: # metric to base the HPA on
        name: kubemq-length
```

A custom metric server is available here[9] and a complete deployment file for this custom-metrics server and HPA is available here[10].

If you want to setup the complete environment, you can execute these both commands:

```
# Pull the deployment repository of this project:
git clone git@gitlab.forge.hefr.ch:ps6-2021-kubemq/deployment.git

# Go to cloned project
cd deployment
```

[9] https://gitlab.forge.hefr.ch/ps6-2021-kubemq/services/custom-metrics-server/-/tree/master
[10] https://gitlab.forge.hefr.ch/ps6-2021-kubemq/deployment/-/blob/e7a6c78602de8edece0b87f688cf28a5f8f24119/researches/custom-metrics-server.yaml

```

# Deploy simple nodejs rest server+service (server available here:
 ↪  https://gitlab.forge.hefr.ch/ps6-2021-kubemq/services/nodejs-simple):
kubectl apply -f simple-rest-deployment.yaml

# Deploy custom metrics server+service, APIService registration and HPA
 ↪  for nodejs rest server:
kubectl apply -f custom-metrics-server.yaml
```

4.4 Kubernetes Jobs

Kubernetes' Jobs[7] is a task that creates various number of pods. The job tracks the number of pods successful completions and ends when a certain number of completion is reached.

So, the idea is that a job is created only when there is job to do or something to compute/process. It can be possible to link a job to a message queue and when a message appears on the queue, a job is triggered.

To do so, it is possible to setup a CronJob[8] that check for queue content and triggers Jobs when there is data inside it. This solution is proposed here[11].

There is an example of implementation on the K8s' website[9] using a Redis queue.

K8s' Jobs might be a solution but required to transform the way an existing service works. This architecture is more suitable for parallels processing of a same data or processing chain of the data. More over, it takes each time a job is launched a bit of time to have it running. This can cause performance issue.

4.5 Sidecar pattern

After some research on Istio[12], a service mesh for K8s, it is quasi impossible to not be confronted to Sidecar pattern as Istio used it to works.

Sidecar pattern consists of running multiple (generally lightweight and fast) containers along a main container inside a Pod. These sidecars are here to help the main container.

In the case of Istio, sidecars are injected automatically inside each pods along main container (remember: a Pod contains generally only one container). The injected sidecars work as proxy and intercept requests and redirect them to other proxies inside other pods to create a service mesh.

The sidecar pattern can be really useful in multitude of situation:

- Setting proxy in front of container.
  - For communication (as Istio do)
  - For logging (logging incomming request or outgoing response from a container)
  - For authentication and verification
  - For incoming data verification/cleanup/escaping
  - For caching

[11] https://stackoverflow.com/a/54045849
[12] https://istio.io/

  - etc.
- Making an adapter for specific communication protocol
- etc.

4.6 Implementation and tests

A story on the website "Medium" has been written by "Bhargav Bachina" and is very useful as tutorial. This story is available here[10].

To create sidecars inside a deployment is really easy, it only needs to add more containers under the spec field of the template inside a deployment.

Below there is a deployment example of a main container named "nodejs" and its sidecar named "sidecar-container".

```
kind: Deployment
metadata:
  name: nodejs
  labels:
    app: nodejs
spec:
  selector:
    matchLabels:
      app: nodejs
  replicas: 1
  template:
    metadata:
      labels:
        app: nodejs
    spec:
      containers:
      - name: nodejs
        image: agost/nodejs-starter:1.2
        imagePullPolicy: Always
      - name: sidecar-container
        image: busybox
        command: ["/bin/sh"]
        args: ["-c", "while true; do echo echo \$(date -u) 'Hi I am from
         ↪ Sidecar container' >> /var/log/index.html; sleep 5;done"]
```

4.7 Forwarding / Redirecting requests

During the solution finding iteration process of the project, it comes the need of redirecting a request to another service for response. A service A (Forwarder) receive a request and answer directly by telling the client to go on another service (other URL pointing on service B: Receiver). See figure 4.

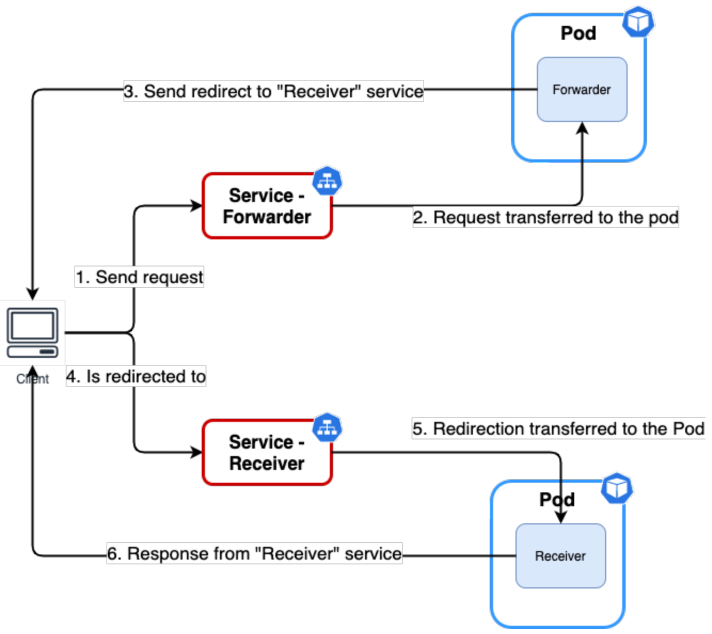

Pod
Forwarder
3. Send redirect to "Receiver" service
Service - Forwarder
2. Request transferred to the pod
1. Send request
Client
4. Is redirected to
Service - Receiver
5. Redirection transferred to the Pod
Pod
Receiver
6. Response from "Receiver" service

Figure 4: Redirecting workflow in K8s infrastructure

Additionally, the service "Receiver" must be capable of processing a lot of requests simultaneously (in parallel), so it requests to not have a "blocking API" that takes requests sequentially.

Implementation of a solution and tests:

For reminder: the project uses Java and Spring Boot.

For forwarding a request from a RestController in Spring Boot, it is possible to use the object "RedirectView" with the option "contextRelative" set to "false" because like this, the url put through the RedirectView is used as complete url and not additional path to the current relative address.

```java
@GetMapping("/")
public RedirectView forward(HttpServletRequest req, HttpServletResponse
 ↪  resp) {
    UUID requestUUID = UUID.randomUUID();
    log.info("Forward request [" + requestUUID + "]");
    return new RedirectView("http://localhost:8085/" + requestUUID,
     ↪  false);
}
```

Now, as described above, the receiver must put each request in a separate thread. By putting the treatment each request into separated threads, we lose the symetrical property of the communication. So it is mandatory to use a "DeferredResult" as result of the request. A "DeferredResult" object will trigger the ResponseServlet only when its method "setResult" is called. When this is done, the ReponseServlet will be notified and will respond to the request.

For putting the computation into a thread, instead of creating a Java Thread by using *new Thread()*, it is possible to use the ForkJoinPool used by Spring Boot. A ForkJoinPool acts like a big pool of jobs to run (normally used to fork and join threads). As Spring Boot use the ForkJoinPool internally, when we put something in it, it will be executed when it is the best moment for Spring Boot. More information on ForkJoinPool here[11].

On the Java code bellow, we can see an implementation of the use of the ForkJoinPool along side the use of DefferedResult:

```java
@GetMapping("/{id}")
public DeferredResult<String> getMethod(@PathVariable String id) {
    DeferredResult<String> output = new DeferredResult<>();
    log.info("Received request [" + id + "]");
    ForkJoinPool.commonPool().submit(() -> {
        log.info("Execution");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        output.setResult("voilà");
    });
    output.onCompletion(() -> log.info("Computation done"));
    return output;
}
```

The both project (Receiver and Forwarder) used as test are available here[13].

To test it with "curl", do not forget to use the "-L" parameter for following the redirect as written bellow:

```
curl --request GET -L http://localhost:8086/
```

4.8 Passing K8s environment variable to Spring Boot container

This chapter discusses how the environment variables are shared and how they coexist. Finally, a test project with a simple implementation is presented.

4.8.1 K8s vs Docker vs Spring environment variables

In Kubernetes, there is the possibility to setup environment variables to a container and with Docker, it is possible to use environment variable when launching a container as the same as Spring Boot also support environment variables. Now the question is: How to pass environment variables from Kubernetes to Spring Boot via Docker?

The answer is: there is no such big deal.

As Spring Boot will use environment variables present on the machine it runs on, it will use the environment variables of the container.

When creating a Dockerfile, we have the possibility to setup environment variables that will be present on the container and set default values for them.

For each container inside a Pod, K8s allows to set environment variables that will be passed to the container. It works as the same way as if we launch the container on Docker using the "–env" parameter like this:

[13]https://gitlab.forge.hefr.ch/ps6-2021-kubemq/services/forwarder-receiver

```
# see:
#https://docs.docker.com/engine/reference/commandline
    # /run/#set-environment-variables--e---env---env-file
docker run -e MYVAR1 --env MYVAR2=foo --env-file ./env.list ubuntu bash
```

The official documentation[12] of Docker says this: "Use the -e, –env, and –env-file flags to set simple (non-array) environment variables in the container you're running, or overwrite variables that are defined in the Dockerfile of the image you're running."

So it is confirmed that the environment variables created by Dockerfile will be overwritten by the environment variable declared in the K8s configuration.

On the figure 5 we have a simple representation of the environment variables and application.

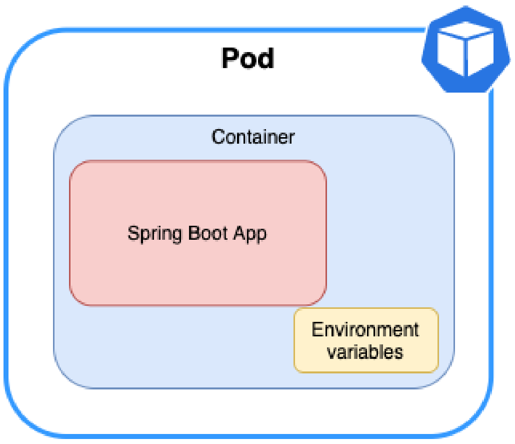


Figure 5: Spring Boot App and environment variables of the container

4.8.2 Installation and tests

To transfert data from Kubernetes yaml file to a Spring Boot application, the app must succeed to retrieve the environment variables presents on the machine and use it in the code. To do so with Spring Boot, we can use Application Properties. Application Properties are the properties that we give to the application like the server port, the database url and so on. It is used to configure the application. There is a file named "application.properties" in each Spring Boot application under "/resources". Custom properties can be created here.

First, create custom properties inside the "application.properties" file and set these custom properties to environment variable name using the "$<env_var>:<default_value>" form.

```
kubemq.queue=${queue:null}
kubemq.address=${queue_addr:null}
```

Now a configuration Spring component must be created to configure and interact with these properties inside our code. It is a class that is used only to store the value of these properties. It must have the same field names as the properties. Here a valid class for the both props declared in the "application.properties" file:

```java
import lombok.Value;
import
 ↪  org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.ConstructorBinding;

@ConfigurationProperties("kubemq")
@ConstructorBinding
@Value
public class KubeMQProps {
    String address;
    String queue;
}
```

Then, the custom properties configuration must be enable on the application to allow the configuration class to configure and access the properties. This is done by adding the annotation "@EnableConfigurationProperties" on the "Application class" and putting the configuration files for properties as parameter for the annotation. Here an coherant example for the both properties declared above:

```java
@SpringBootApplication
@EnableConfigurationProperties({KubeMQProps.class})
public class PropertiesApplication {
    public static void main(String[] args) {
        SpringApplication.run(PropertiesApplication.class, args);
    }
}
```

Now the Spring Boot application can interact with environment variables but there is not a single one in the container. This is time for adding environment variable via the Dockerfile. This can be done like this:

```dockerfile
FROM openjdk:16-jdk-alpine
VOLUME /tmp
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar

ENV queue "default_queue_name"
ENV queue_addr "default_queue_addr"
ENV port 8080

ENTRYPOINT ["java","-jar", "/app.jar"]
```

Finally, when everything is set up, it only remains to set the environment variable in the K8s configuration file of the Pod/Deployment. The document on how to add environment variables to a container is provided at this address[13]. The simple way is to add the field "env" to container configuration and list under this field all the environment variables like this:

```yaml
- name: env-var-spring-boot
  image: agost/io.kueuer.proxy:latest
  imagePullPolicy: Always
  ports:
  - containerPort: 3001
  env:
  - name: port
    value: "3001"
  - name: queue_addr
    value: "kubemq-cluster-grpc.kubemq.svc.cluster.local:50000"
  - name: queue
    value: "s1-request"
```

4.9 Queuing with KubeMQ

KubeMQ documentation is available here[14].

KubeMQ is a native Message Broker for Kubernetes. It is quite new with a lot of features. The big advantage of KubeMQ over other message broker is that it is very simple to use, it is fast and lightweight and native for Kubernetes. It also provides a complete for CLI to help testing and developing.

Various SDK are available like .Net, Go, Node, Java, Python and Rest. More than queuing features, it allow to as connectors with KubeMQ Targets, Bridges and Sources. These functionalities are not useful for this project but are very interesting. More on that here[15].

KubeMQ provide three different queuing patterns:

- Queues : Simple message queues (FIFO) with extended features.
- Pub/Sub : Publisher/Subscriber with event through channels.
- RPC : Remote Procedure Call pattern with commands and queries.

4.9.1 Queues

KubeMQ queues offers various useful features for a publisher or a consumer. For a publisher, he has the possibility to publish a single or batch messages to a queue. He can also setup Message expiration and Message delay (scheduled delay).

A consumer has the possibilities to:

- Make long polling (wait a certain amount of time on a message)
- Peek messages in the queue (without consuming them)
- Change the visibility of a message (consuming it, ack it or failed it)
- Ack all the messages in the queue
- Resend them

An interesting consumer feature is the Message Visibility. A consumer can set a visibility time to message and during this time the message will not be available for other consumers connected to the queue. The consumer can ACK the message during the visibility time and then the message will disappear definitely from the queue. If the consumer does not send an ACK during this period, the message is considered as "Failed" and it appears again for other consumers. The consumer can extend the visibility period at any time. This feature

[14]https://docs.kubemq.io/
[15]https://docs.kubemq.io/learn/kubemq-connectors

works like re-injection of the message on failure to process it. The figures 6 and 7 shows a graphical representation of this feature.

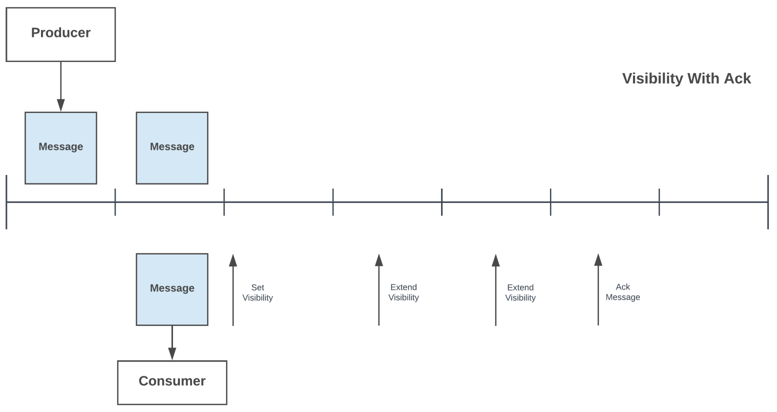


Figure 6: KubeMQ Visibility with Ack

Source: KubeMQ.io

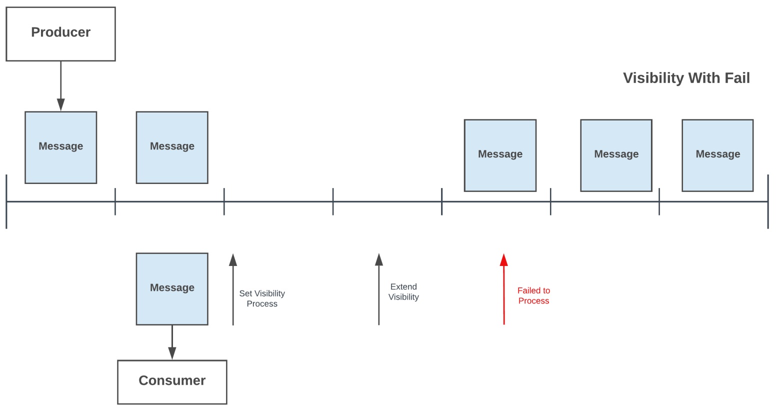


Figure 7: KubeMQ Visibility with Failure

Source: KubeMQ.io

4.9.2 Publisher / Subscriber

KubeMQ can be used to make a pub/sub pattern (figure 8). In KubeMQ, the messages are called "Events" and the queues are called "Channel" when it is related to pub/sub pattern.

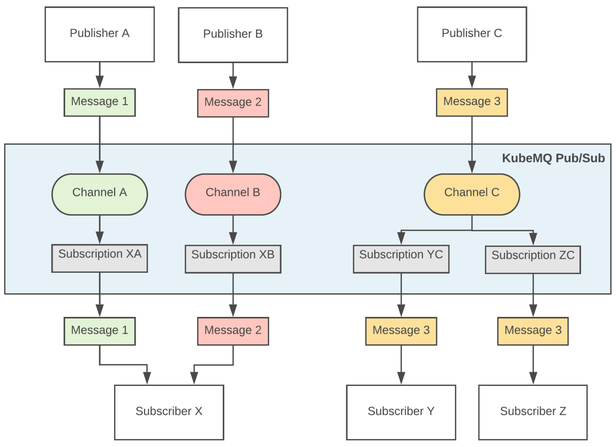


Figure 8: KubeMQ Basic Pub/Sub pattern

Source: KubeMQ.io

Additionally, KubeMQ has another interesting feature that is the Event stores with Pub/Sub Pattern. This feature provides the possibility to store all the events into a channel dedicated database (figure 9). By storing all the events that are published into an Event Store, it allows a subscriber to read events that where published before the subscriber subscribe to the corresponding channel. With event stores, a subscriber has the possibility to read the events when it subscribes to the channel:

- From new events only.
- From the first event that were published into the channel.
- From the last event that were published into the channel.
- From sequence (from specific sequence)
- From Time (time in the past)
- From Time Delta (e.g. 30 sec back)

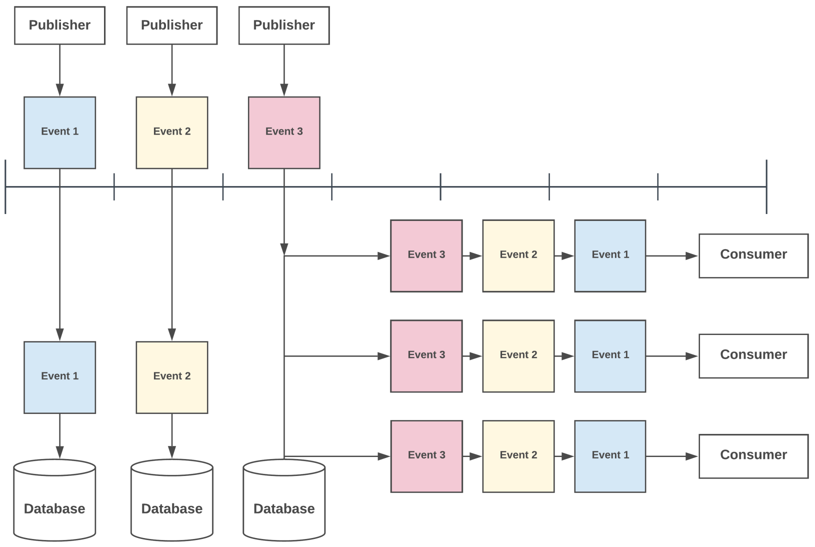


Figure 9: KubeMQ Pub/Sub with Event Stores

Source: KubeMQ.io

4.10 Monitoring KubeMQ

KubeMQ provide "KubeMQ Dashboard" which is a pre-configured Grafana/Prometheus with all the metrics KubeMQ exports.

This deployment is available on GitHub[16].

There is the possibility of deploying Grafana/Prometheus simultaneously or either one or the other.

To deploy the whole package, kubemqctl can be used like this:

```
kubemqctl get dashboard
```

To deploy only Prometheus:

```
# Clone the project and move inside the directory
git clone https://github.com/kubemq-io/kubemq-dashboard.git
cd kubemq-dashboard

# Apply the deployment file to the kubernetes environment
# Warning, there is a typo in the repository
kubectl apply -f promethues.yaml
```

The Prometheus configuration file with all the exported metrics is available in this same repository[17].

[16]https://github.com/kubemq-io/kubemq-dashboard
[17]https://github.com/kubemq-io/kubemq-dashboard/blob/master/prometheus/kubemq.rule.yaml

4.11 Reverse Proxies

This section discusses existing solution for queuing incoming requests.

It is important during the project to take a step back, see what we are doing and does it makes sense. The project is about queuing incoming requests to an application running in a Kubernetes infrastructure and scaling regarding metrics based on the queue.

A reverse proxy is a server placed before the application and used to route the requests from the client to the right application. There is several existing reverse proxies and some offers queuing features.

In the list of reverse proxies below, every proxy is capable of queuing incoming requests. They may be a potential solution for the project instead of creating a whole new system.

- NGINX is a well known reverse proxy, now part of the F5 company. It is capable of queuing incoming request to prevent requests from beeing dropped (see : Advanced configuration for deploying in Kubernetes[14]).
- F5 BIG-IP is used worldwide and is very powerful. It is also capable of queuing incoming requests as the manual documentation says[15].
- HAProxy is a load balancer/reverse proxy capable of high performance. HAProxy procedure for queuing incoming requests is presented in this publication [16]. It offers the possibility to set maximum requests accepted before queuing others.

In conclusion, solution already exists for queuing incoming requests which can be suitable for the project. Now, it remains to test in details and see if it offers the possibility to collect metrics based on their internal queues.

This check of existing solutions was made to late during the project and these solution have not been tested. It can be the next step if continuing this project.

5 Finding solution iterations

This section presents all the different iterations and researches/proposals of solutions made during the project. All of these proposals are linked to several tests described in chapter 4 to determine if it can be a suitable solution.

None of the solutions in this chapter is a suitable solution. To see why, check out the disadvantage listing of each solution.

5.1 Jobs

This is the first conception that was proposed. This solution (figure 10) works with Jobs and CronJobs (see ch. 4.4).

To queue incoming requests, it uses the idea of a "Dispatcher" that will works as proxy for the whole cluster. It intercepts incoming requests and regarding the service it was destined to, it will put the request into the queue corresponding to this service. There is one queue per service. The Dispatcher has a HPA that is based on the number of incoming requests per second to help handle a lot of incoming requests to the cluster.

For each queue, it will have a CronJob that will periodically check the length of the queue and if there is more than one element inside the queue, it will launch Jobs that will pull what's inside the queue, process it and put back the result. This solution does not use auto-scaler because it is the job of the CronJob to determine how many Jobs it must run.

With this solution, we say goodbye to the old implementation of existing service, and we convert them into executable jobs.

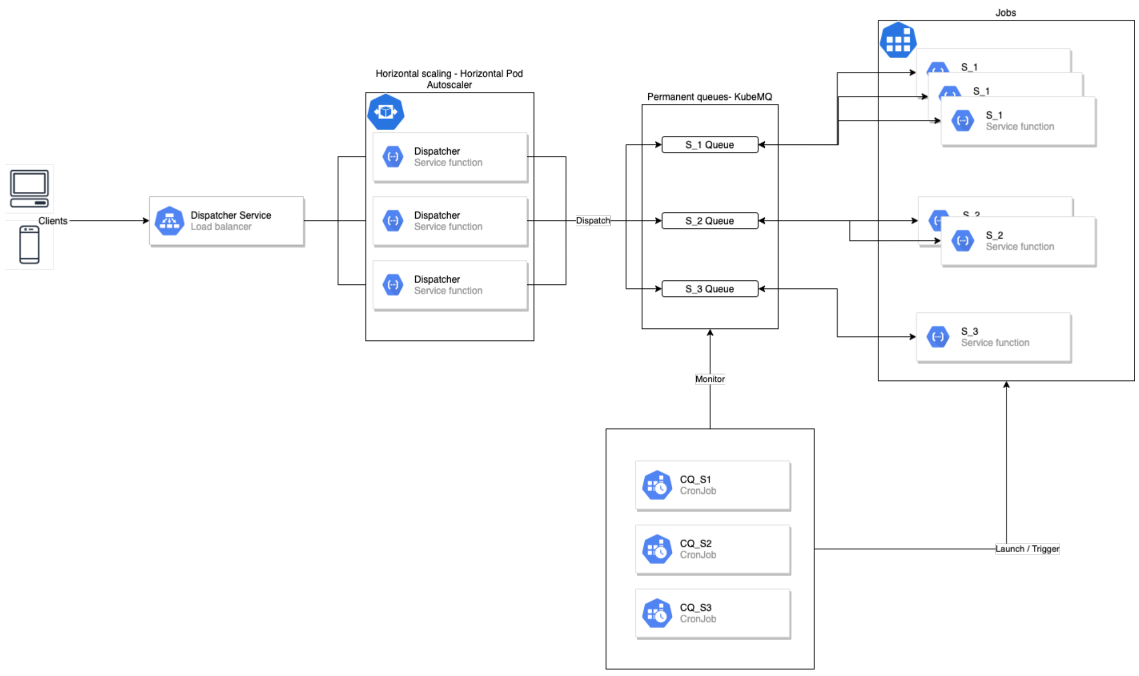


Figure 10: Jobs concept

There are disadvantages of using a "Dispatcher" pattern like this:

- How to intercept requests?
- There is no way of setting a proxy like this without changing the URL of each service. It requires not to access the service via its address but by the address of the dispatcher and so, modify each client to not call the address of the services anymore but the dispatcher instead.

Using Jobs have a lot of disadvantages:

- KubeMQ doesn't offer a way to get the queue length directly.
- All the existing services must be refactored to work as Job and not as basic containers anymore.
- CronJob can involve latency (not often enough checks) or load (too often checks) if they are not well tuned.

5.2 Custom metrics server

This other conception forgets the idea of using jobs because it is not suitable for the use case of the project.

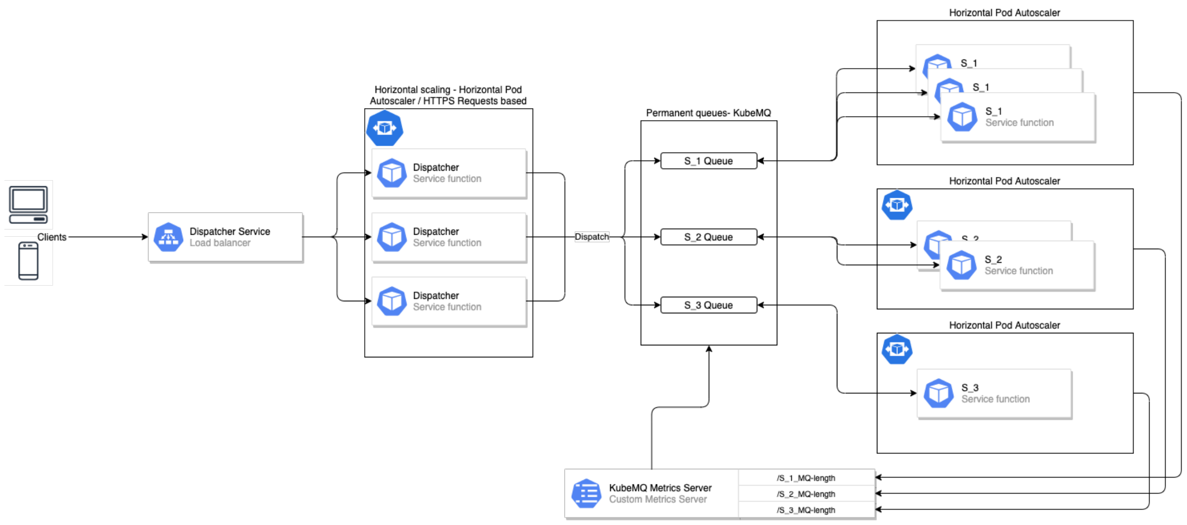


Figure 11: Custom metrics server concept

This concept (figure 11) still has the idea of a general "Dispatcher" which intercepts the request then dispatch them inside the corresponding queues of the service for which they were targeted.

The concept of this solution is to use a custom metrics server (see chap. 4.3) which is then used by all the HPA to scale up or down a service. For each queue, the custom metrics server serves a path that is used by the corresponding HPA. The custom metric served is named "kubemq-length" and it corresponds to the length of the queue.

In addition of the disadvantages and interrogations of the "Dispatcher" concept, this solutions have a lot of other disadvantages:

- KubeMQ doesn't offer a way to get the queue length directly.
- When a new service is added, the custom metrics server must be modified to serve a new path.
- Having a custom metrics server is not maintainable comparing of a Prometheus server.
- How existing services interact with the queues? They must be modified to work with queues.

5.3 Proxy and Adapter

This penultimate iteration is completely different form the previous ones.

This solution (figure 12) addresses the response confusion by creating a second queue dedicated to the responses.

It gets rid of the "Dispatcher" to replace it by a Proxy for each service. This way, no more needs to change the service path or clients to redirect to a dispatcher. The proxy must be invisible for the client. This proxy will intercept requests, put them in request queue and wait on the corresponding response appears in the response queue.

The most important thing is that this solution implements a sidecar (see ch. 4.5) named "QAdapter" that will communicate with the service and make the bridge between the service and the queues to avoid to change an existing service.

Finally, there is no more custom metrics server that are not maintainable and will use Prometheus (see ch. 4.10) for monitoring instead.

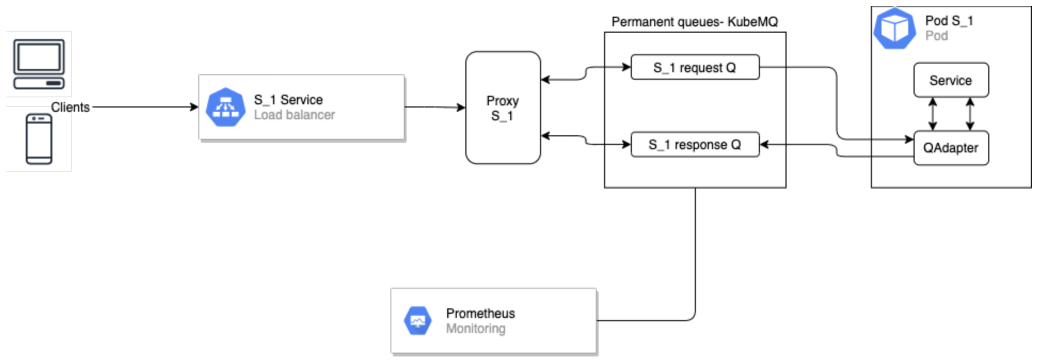


Figure 12: Proxy and Adapter concept

There are still some disadvantages to this solution:

- The Proxy must remember a list of requests in a table to know what to respond to whom.
- The Proxy works as two applications: one putting the data in the request queue and another one fetching continuously the data inside the response queue. This is not a good separation of work.

6 Kueuer

Kueuer is the name of the final solution proposal for this project. This final proposal is very similar to the one proposed in chapter 5.3(Proxy - Adapter).

This last proposed solution removes the disadvantages of the previous proposal (Proxy - Adapter) by dividing the Proxy into two different components but keeps the same idea.

6.1 Conception - Design

This chapter explains the general idea and the general concept behind the Kueuer system. It will not dive into the details for each parts. They all have their own chapter.

6.1.1 Architecture

This solution is constituted by five parts:

- Proxy : intercept the request and put them into a request queue.
- QAdapter: adapter between the both queues and an existing application.
- Responder: Wait on responses published into the response queue to return them to the client.
- Queues: one queue for request (incoming data) and one channel for response (outgoing response).
- Monitoring and Scaling: Prometheus server for monitoring queues and HPA for scaling application (Pods).

All these parts will be discussed in detail in the following chapters.

The idea is that each existing application inside the Kubernetes environment will have its own Proxy, its own request and response queues. Additionally, each container will have its own QAdapter. The Responder can be used in two different configurations: 1. as application dependents (means one Responder per application like the Proxy) or 2. as cluster wide (it can be connected to several response Q of different applications)

The QAdapter is linked to a container (Sidecar pattern see ch. 4.5) unlike the Proxy which is linked to the K8s' service of the application.

On the figure 13 we can see the architecture in place for a single application K8s environment. This single application is named "S_1".

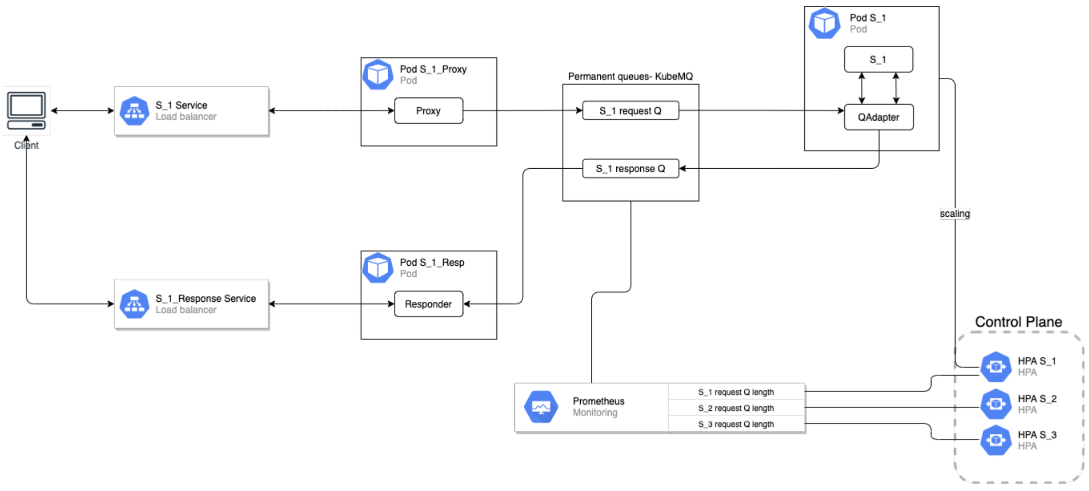


Figure 13: Kueuer architecture for one service S_1

Each application has its own HPA (see chap. 4.2) that will automatically, horizontally (vertically vs horizontally: see ch. 4.1) scale up/down the application based on a shared Prometheus metrics server that collect metrics for each queues.

6.1.2 Communication

The easiest way to understand the communication between the components is to refers to the architecture schema above (figure 13) and imagining that the client wants to make a request to the "S_1" application:

1. The client make a request as normal to the K8s' service of the application S_1. No changes for him.
2. The K8s' service "S_1 Service" redirect the request to the Proxy.
3. The Proxy "intercept" the request and put it in the queue "S_1 request Q".
4. The Proxy send back a redirection link to the client. This redirection link points to the "S_1 Responder Service".
5. The client follow the redirection to the Responder.
6. The Responder, which is subscribed to the "S_1 response Q" wait until the corresponding response for the client's request appears, and so, the client wait with him.
7. Meanwhile, the QAdapter consume the request putted into the "S_1 request Q" by the Proxy (3)

8. The QAdapter make the call to the application "S_1" and wait for the response.
9. When the QAdapter receives the response from the application, it put it in the "S_1 response Q" and go back waiting on the "S_1 request Q" to be fulfilled.
10. The Responder receives the response putted by the QAdapter and send it back to client.

On the more detailed sequence diagram below (figure 14), we can see that to make a relation between a request that were putted into the request queue and a response published into the response queue, the system use a unique identifier: one UUID. The Proxy will create a UUID for each request it intercepts and redirects the client on a path containing this generated UUID. This generated UUID acts like a token for the client, this token give him the right to access to its response via the Responder.

The UUID follow the request through all its processing, when the QAdapter put the response in the response queue, it uses the same UUID as the request.

Another important point, is that the Responder must be capable of handling numerous requests in parallel and must not have a "blocking API" (explained and tested in chapter 4.7) so it creates a thread each time it receives a request and this thread will just subscribe himself the internal observer and wait to be notified to send back the response to the client.

The Responder uses an internal machinery with internal observer to avoid subscribing multiple times to the same response queue. This observer is used to store all the response received (pub/sub pattern) and notify response threads waiting on a response (identification with UUID).

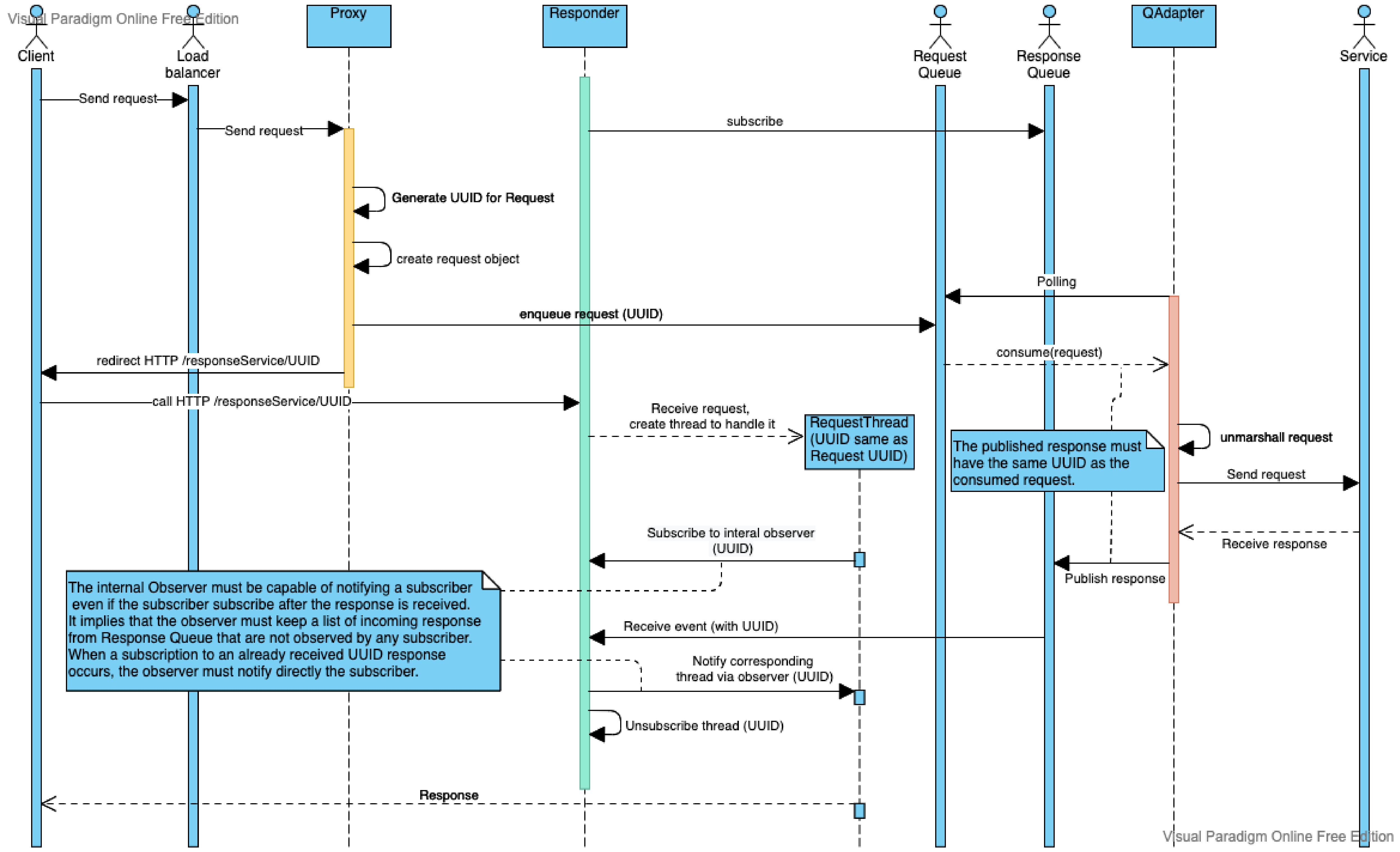

Visual Paradigm Online Free Edition
Client
Load balancer
Proxy
Responder
Request Queue
Response Queue
QAdapter
Service
Send request
Send request
subscribe
Generate UUID for Request
create request object
Polling
enqueue request (UUID)
consume(request)
redirect HTTP /responseService/UUID
call HTTP /responseService/UUID
Receive request, create thread to handle it
RequestThread (UUID same as Request UUID)
The published response must have the same UUID as the consumed request.
unmarshall request
Send request
Subscribe to interal observer (UUID)
Receive response
Publish response
The internal Observer must be capable of notifying a subscriber even if the subscriber subscribe after the response is received. It implies that the observer must keep a list of incoming response from Response Queue that are not observed by any subscriber. When a subscription to an already received UUID response occurs, the observer must notify directly the subscriber.
Receive event (with UUID)
Notify corresponding thread via observer (UUID)
Unsubscribe thread (UUID)
Response
Visual Paradigm Online Free Edition

Figure 14: Sequence diagram of a client call on application named Service (actor on the right)

6.2 Proxy

The proxy component is used only for intercepting requests, converting them, put them into the request queue and redirecting the client to the Responder. It is a simple component having only one RestController.

6.2.1 Design

As said before, the Proxy component is a little component. It must:

1. Accept (Intercept) all kind of request.
2. Create a message corresponding to the request to put into the request queue.
3. Redirect to the Responder with the generated UUID.

As we can see on the class diagram of the application (figure 15), there is few component that are explained below:

- AppConfig + ResponderProps + KubeMQProps are used to configure environment variable to configure the proxy. The system used to retrieve environment variables from K8s config file to Spring Boot application is explained at chapter 4.8.
- ProxyInterceptor : RestController of the application that have only one method: "intercept". This method can handle all kind of request and return a RedirectView to redirect the user to the Responder. Same principle as explained in the chapter 4.7.
- Request : This pojo represent a request. It is the payload used to put into the request queue. It contains the uuid of the request (generated by the intercept method), the path, payload, incoming ip address and the HTTP Method of the request.

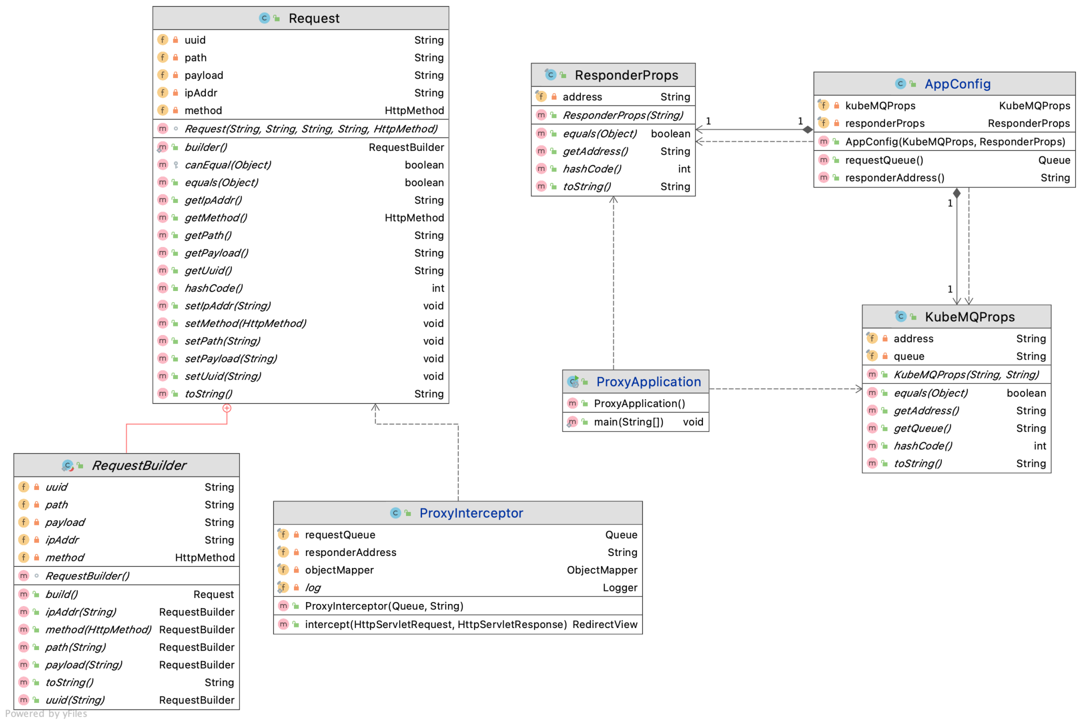


Figure 15: Proxy class diagram

6.2.2 Implementation

The implementation is not difficult, it is a mix of several technical tests described in these chapters:

- Forwarding-Redirect : ch. 4.7
- Environment Variables: ch. 4.8

The only interesting part of the implementation is how to put a message into a KubeMQ queue. All the documentation is available on the KubeMQ Java SDK on GitHub[17].

First a "Queue" object is needed. A the implementation is with Spring Boot, it is possible to create a single instance of a Queue and share it across the whole project using a Bean. As we can see on the class diagram (figure 15), the Bean is created inside the AppConfig class. When creating a Queue, it requires some parameters as the queue name, the clientID and the address of the queue:

```java
@Bean
public Queue requestQueue() throws ServerAddressNotSuppliedException,
 ↪  SSLException {
    return new Queue(kubeMQProps.getQueue(), "proxy",
     ↪  kubeMQProps.getAddress());
}
```

Then it is possible to create a message payload and push it into the queue using the "Message" object:

```java
Request request = Request.builder()
        .path(req.getRequestURI())
        .ipAddr(req.getRemoteAddr())
        .method(HttpMethod.valueOf(req.getMethod()))
        .payload("test")
        .uuid(UUID.randomUUID().toString())
        .build();
Message msg = new Message()

         ↪  .setBody(Converter.ToByteArray(objectMapper.writeValueAsString(request)))
        .setMetadata("empty");
SendMessageResult res = requestQueue.SendQueueMessage(msg);
```

A Jackson Object mapper is used to convert the Request object into String. This provides a way to not have compatibility issues when un-marshalling the object on the QAdapter that uses the same object mapper.

6.2.3 Deployment

The proxy need to be configured when deployed. The service of the application it is related to must be changed to redirect requests to the proxy instead of the application.

Configuration files are available in appendix ch. 11.2.3

The Proxy component has three environment variables. Each environment variable annotated with "-" in the default value column are mandatory.

Name	Description	Example	default value
port	The port on which the Spring Boot application runs	3001	8080
queue_addr	The address KubeMQ grpc server. Use Kubernetes DNS.	kubemq-cluster-grpc.kubemq.svc.cluster.local:50000	-
responder_addr	The outside address of the responder	192.168.99.104:30001	-

6.3 QAdapter

The QAdapter is the only component connected to both queues simultaneously. It is a consumer and a publisher. The idea of the QAdapter is to avoid modifying existing application to run with queues. The adapter will pull the queued requests present into the request queue and make the corresponding HTTP request to the application running in the same Pod. When it receives the response for the HTTP request, it publishes the response into the response queue of the application. The QAdapter use the Sidecar pattern (ch. 4.5) so it runs along side an application.

6.3.1 Design

The QAdapter works like this:

1. It pulls a request from the request queue.
2. It "process" the pulled request by creating a HTTP request to the application on the same Pod. This step is blocking, it means that the QAdapter cannot pull another request before the current one have not finished to process.
3. When it receives the response to its HTTP request, it publishes the response into the response queue.
4. After publishing the response into the response queue, it is again free to take another request from the request queue.

Below the class diagram of the QAdapter (figure 16):

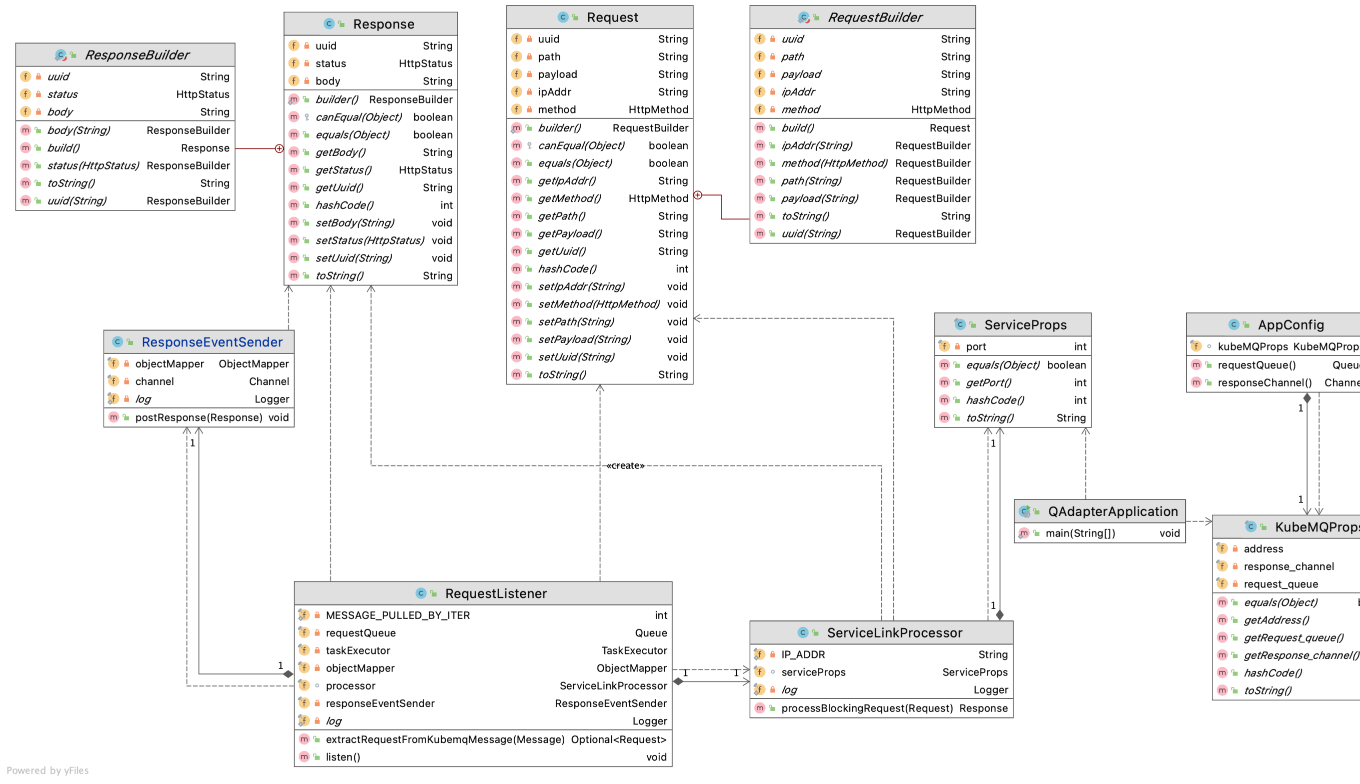


Figure 16: Proxy class diagram

On the class diagram we can see these classes:

- ServiceProps + AppConfig + KubeMQProps are used to configure and use environment variables given through K8s configuration of the container. (see ch. 4.8).
- RequestListener: this Spring Component is the core of the QAdapter. It is the listener connected to the request queue. It contains the method "listen" which is called automatically after initialization and contains the "while(true)" loop that check if something appears into the request queue and processes it.
- ServiceLinkProcessor: this Spring Component is used to process a request pulled from the request queue. It has only one method that process a Request object and return a Response. This method makes the call to the application the QAdapter is linked to. Its only one method is blocking and called by the method "listen" of the RequestListener when it finds something in the request queue. The fact this method is blocking forces the "while(true)" loop to stop during the processing and it avoids pulling other requests from the queue while already processing one.
- ResponseEventSender: this Spring Component is used to publish a Response object into the response queue. It is called by the method "listen()" of the RequestListener component after finishing processing a request.
- Request + Response are two POJO to store the data in objects.

The sequence diagram below (figure 17) shows graphically what is explained above:

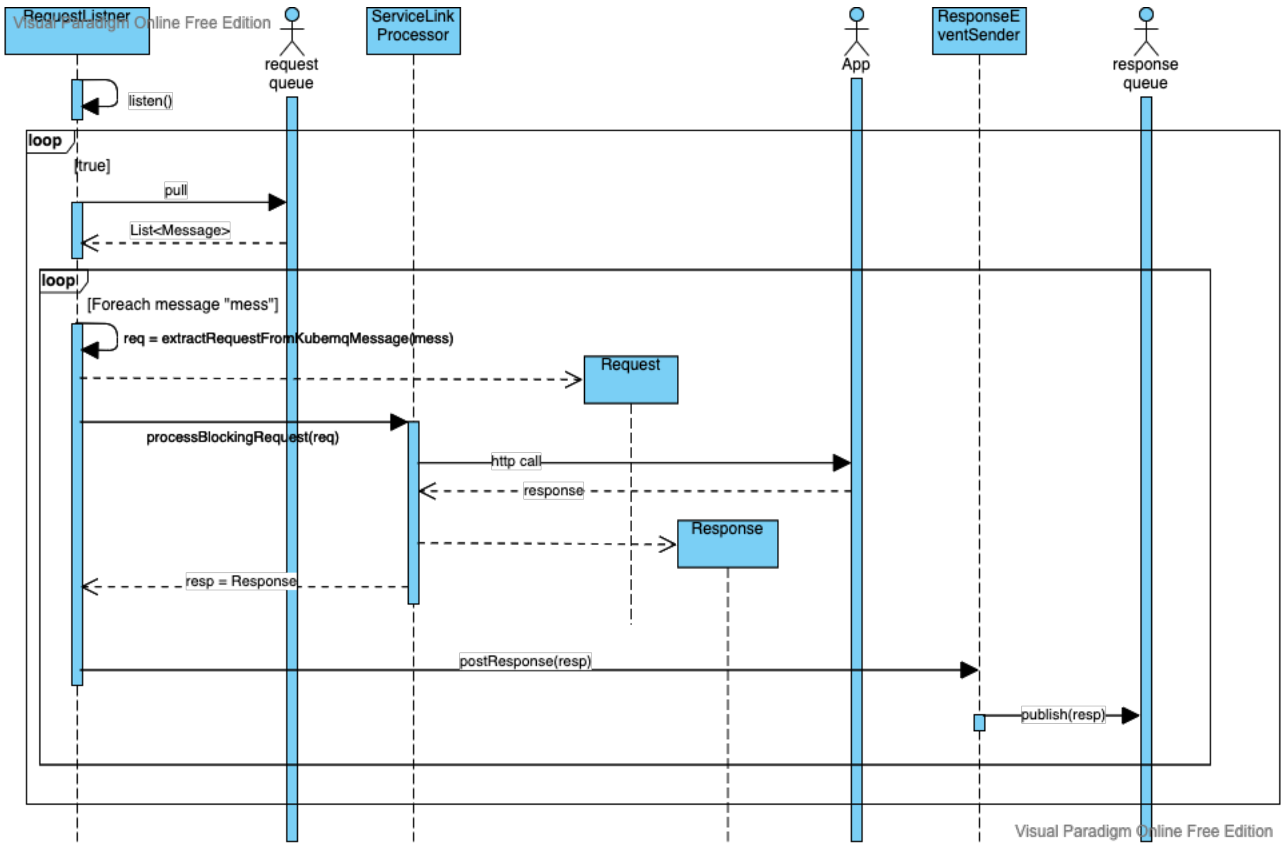


Figure 17: QAdapter sequence diagram

6.3.2 Implementation

There is no very complex implementation. There is just some point that might be important to understand:

First, how the method "ServiceLinkProcessor:processBlockingRequest" if effectively blocking: this method uses the Spring WebClient Object to send HTTP requests. There is multiple part of configuring the WebClient (see documentation[18] or tutorials[18]) and after configuring a request and the client, it's time to ask for the response. Spring WebClient is by default an async client, but it is the possibility to ask to wait on the response using the method "block()":

```
// Declare the response here to be able to put the status code in it
// in the Mono parameterization
Response responseEvent = new Response();

Mono<String> response = req.exchangeToMono(clientResponse -> {
    // put the status code in the response
    responseEvent.setStatus(clientResponse.statusCode());
    if (clientResponse.statusCode().is2xxSuccessful()) {
        return clientResponse.bodyToMono(String.class);
    } else if (clientResponse.statusCode()
```

[18]https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/reactive/function/client/WebClient.html

```java
            .is4xxClientError()) {
        return Mono.just("Error response");
    } else {
        return clientResponse.createException()
                .flatMap(Mono::error);
    }
});
String payload = response.block(); // <----- HERE
```

Also, as the QAdapter works as a sidecar, there is no need to know the address of the application is linked to because all application running in the same pods runs on the same machine. So the application linked to the QAdapter is reachable via the loopback address (127.0.0.1).

Then, it's also interesting to see how to consume message from a queue and how to publish an event to a channel. To consume a message from a queue using the KubeMQ Java SDK, the program needs to know the address of KubeMQ and the name of the queue. These information are given through the environment variables. After creating the Queue object as Spring Bean (see ch. 6.2.2), it is possible to pull message using the "ReceiveQueueMessages(numberOfPulled, wait_time)" like this (the code above is inside the while(true) loop of the "RequestListner:listen" method):

```java
ReceiveMessagesResponse res =
 ↪  requestQueue.ReceiveQueueMessages(MESSAGE_PULLED_BY_ITER, null); //null =
 ↪  default time
if (res.getIsError()) {
    log.error("Cannot get a message in queue " + requestQueue.getQueueName() +
     ↪  " Error: " + res.getError());
    continue;
}
```

For publishing event into a Channel, it just has to create an Event object, put body payload in it and send it with the "Channel:SendEvent" method. Example:

```java
Event event = new Event();
event.setEventId(response.getUuid());
try {

     ↪  event.setBody(Converter.ToByteArray(objectMapper.writeValueAsString(response)));
} catch (IOException ex) {

     ↪  log.error("Cannot convert response object to byte[] for publishing event in channel {}",
     ↪  channel.getChannelName());
    return; // TODO: Handle better than this
}
try {
    channel.SendEvent(event);
} catch (ServerAddressNotSuppliedException | SSLException e) {
    log.error("Error when trying to send event in channel {}",
     ↪  channel.getChannelName());
}
```

6.3.3 Deployment

As described before, the QAdapter works as sidecar. It must be integrated to the deployment of the application (explained in chapter 4.5).

Configuration file for deployment of an App with QAdapter attached to it is available in appendix ch. 11.2.1.

The QAdapter component has 5 environment variables. Each environment variable annotated with "-" in the default value column are mandatory.

Name	Description	Example	default value
port	The port on which the Spring Boot application runs	3001	8080
queue_addr	The address KubeMQ grpc server. Use Kubernetes DNS.	kubemq-cluster-grpc.kubemq.svc.cluster.local:50000	-
request_queue	The name of the request queue	s1-request	-
response_channel	The name of the response queue (=channel)	s1-response	-
service_port	The port of the main container of the pod.	8080	3000

6.4 Responder

The Responder component is used for two things. First it is used to handle a lot of pending requests from clients who wait on their response. Then, it is use to retrieve all the response from the response queues. We can compare it to a waiting room at the Post. You took a ticket and you received a number (here a UUID) and now you must go to the waiting room and wait until we call your number. The terminal on which you take the ticket is the Proxy component and the waiting room is the Responder component.

The Responder Component is the only component that can be cluster wide. You can imagine having only one Responder for the whole cluster and this Responder is connected to every response queues. If we take back the comparison with the Post waiting room, imagine having multiple Post offices with each their own terminal where you can take a ticket. All these Post offices share the same waiting room. This is possible because of the definition of the UUID: "A UUID is 128 bits long, and can guarantee uniqueness across space and time."[19].

6.4.1 Design

The design of the Responder is done a way to limit the number of subscriptions to KubeMQ channels and to be capable of handling a lot of requests without blocking.

For the first constraints (limit the number of subscriptions), it has an internally observer pattern. This internal observer pattern allows the Responder to subscribe only once to each channel it must be connected to. The subject of the internal observer pattern is based on events coming from all the channels. This subject is a datastore of all events retrieved across the channels. Observers subscribe to this subject by giving the UUID they are waiting for. When a new event arrives into the subject, it checks if one observer is waiting on it, if yes, it gives the event to

[19]https://www.ietf.org/rfc/rfc4122.txt

the corresponding observer. Regarding the uniqueness of a UUID, when an observer receives an event, it is automatically unsubscribed from the subject because it will never have another event with the same UUID. If there is no observer waiting on a event when the event arrives, the subject must be capable of storing the event until the corresponding observer comes. It can happen that an event arrives to the subject before the corresponding observer tries to subscribe because of communication delay or client error. For each event, it must have a corresponding observer.

For the second constraint (handling a lot of request without blocking), it uses the Java ForkJoinPool object to create a job each time a request arrives to the rest controller. These job work with DefferedResult to send the result back to client on a certain event. This implementation and concept are described in the chapter 4.7. To make the link with the internal observer pattern, the jobs created at each incoming request are the observers.

Finally, the Responder offers only one route and it is "/<UUID>". The client must have a correct UUID to access to the server and a used UUID to access to data.

Warning, be aware that UUID do not protect against attacks or brute forcing (see this article[19]).

The Responder contains multiple classes (figure 18):

- ResponseEventReceiver: This class is used to automatically subscribe to the channels. When subscribing to a channel, it is mandatory to give a grpc StreamObserver which will consume the incomming events.
- ResponseStreamObserver: extends grpc StreamObserver and is used to consume events coming from the queues. When an event is received and has been correctly unmarshalled, it sends it to the EventSubect.
- EventSubject: The subject of the internal observer pattern.
- ResponderController: Rest controller serving the "/<UUID>" path. Add a job to the ForkJoinPool.commonPool when it receives a request. Its "getResponse(String):DefferedResult<String>" is the method which serves the "/<UUID>" path.
- Response: POJO for convertir an Event's body to a "Response".
- KubeMQProps + AppConfig are configuration for using environment variables.
- CannotSubscribeException: Java exception thrown when the ResponseEvenReceiver cannot connect to a channel.

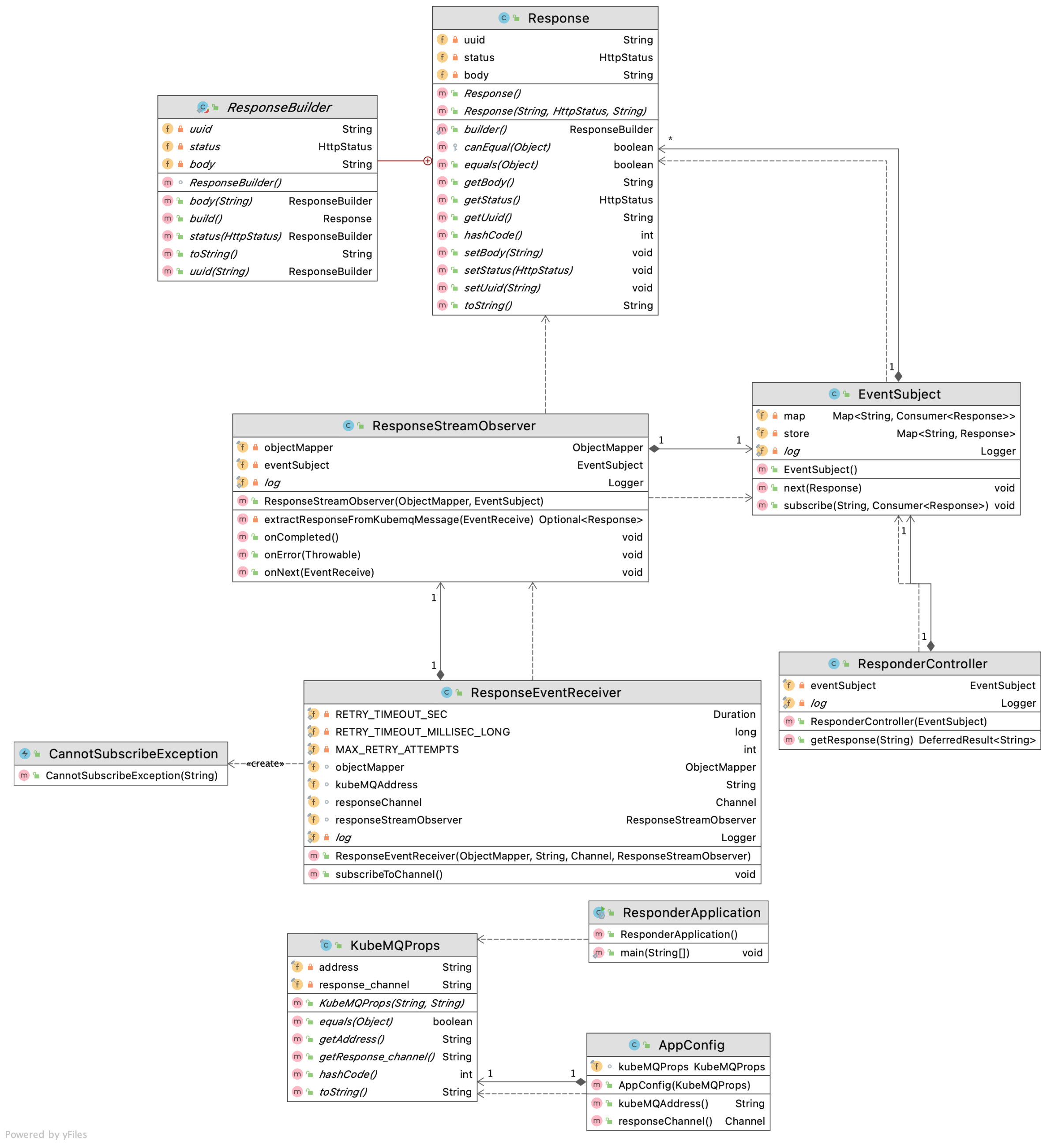

Response
uuid String
status HttpStatus
body String
Response()
Response(String, HttpStatus, String)
builder() ResponseBuilder
canEqual(Object) boolean
equals(Object) boolean
getBody() String
getStatus() HttpStatus
getUuid() String
hashCode() int
setBody(String) void
setStatus(HttpStatus) void
setUuid(String) void
toString() String
ResponseBuilder
uuid String
status HttpStatus
body String
ResponseBuilder()
body(String) ResponseBuilder
build() Response
status(HttpStatus) ResponseBuilder
toString() String
uuid(String) ResponseBuilder
EventSubject
map Map<String, Consumer<Response>>
store Map<String, Response>
log Logger
EventSubject()
next(Response) void
subscribe(String, Consumer<Response>) void
ResponseStreamObserver
objectMapper ObjectMapper
eventSubject EventSubject
log Logger
ResponseStreamObserver(ObjectMapper, EventSubject)
extractResponseFromKubemqMessage(EventReceive) Optional<Response>
onCompleted() void
onError(Throwable) void
onNext(EventReceive) void
ResponderController
eventSubject EventSubject
log Logger
ResponderController(EventSubject)
getResponse(String) DeferredResult<String>
ResponseEventReceiver
RETRY_TIMEOUT_SEC Duration
RETRY_TIMEOUT_MILLISEC_LONG long
MAX_RETRY_ATTEMPTS int
objectMapper ObjectMapper
kubeMQAddress String
responseChannel Channel
responseStreamObserver ResponseStreamObserver
log Logger
ResponseEventReceiver(ObjectMapper, String, Channel, ResponseStreamObserver)
subscribeToChannel() void
CannotSubscribeException
CannotSubscribeException(String)
«create»
ResponderApplication
ResponderApplication()
main(String[]) void
KubeMQProps
address String
response_channel String
KubeMQProps(String, String)
equals(Object) boolean
getAddress() String
getResponse_channel() String
hashCode() int
toString() String
AppConfig
kubeMQProps KubeMQProps
AppConfig(KubeMQProps)
kubeMQAddress() String
responseChannel() Channel
1
*
Powered by yFiles

Figure 18: Responder class diagram

6.4.2 Implementation

For the controller, this is very similar to what is explained in the chapter about forwarding/redirecting requests (ch. 4.7). The creation of job is the same. The only difference is that a job consists of subscribing to the EventSubject and waiting for the response:

```
ForkJoinPool.commonPool().submit(() -> {
    eventSubject.subscribe(uuid, response -> {
```

```java
        log.info("Putting response into deferred result");
        deferredResult.setResult(response.getBody());
    });
});
```

To implement the EventSubject, HashMap are used to store the observer and the event without observers. It uses HashMap because of the complexity to retrieve an information into Map that is generally O(1) and in the worst cases O(n). This is important if we want to store and retrieve a lot of incoming requests.

The subject contains two HashMap: one for storing observers waiting on event and another one for storing events that have not yet an observer. You cannot unsubscribe from the subject because it is done automatically.

```java
// Two maps:
private final Map<String, Consumer<Response>> map;
private final Map<String, Response> store;

public void subscribe(String uuid, Consumer<Response> responseConsumer) {
    if (store.containsKey(uuid)) { // event already received and present in
    ↪   store
        log.info("Event already in store, executing..");
        responseConsumer.accept(store.get(uuid));
        log.info("Finish executing");
        store.remove(uuid);
    } else {
        map.put(uuid, responseConsumer);
    }
}

public void next(Response response) {
    if (!map.containsKey(response.getUuid())) {
        store.put(response.getUuid(), response);
    } else {
        map.get(response.getUuid()).accept(response);
        map.remove(response.getUuid());
    }
}
```

6.4.3 Deployment

The Responder can be deployed cluster wide because it can subscribe to multiple response queues simultaneously.

A configuration file for deploying the Responder component is available in the appendix chapter 11.2.4.

The Responder has 3 environment variables. Each environment variable annotated with "-" in the default value column are mandatory.

Name	Description	Example	default value
port	The port on which the Spring Boot application runs	3001	8080
queue_addr	The address KubeMQ grpc server. Use Kubernetes DNS.	kubemq-cluster-grpc.kubemq.svc.cluster.local:50000	-
response_channels	List of channel names separated by a comma ","	Only one: "s1-response" / two: "s1-response,s2-response"	-

The responder also need a service of type NodePort to be accessible from the outside. For example:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: responder-service
spec:
  selector:
    app: responder
  type: NodePort
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
    nodePort: 30001
```

6.5 Queues

In the Kueuer system, queues are the central point. For queueing, KubeMQ is used.

The idea is simple: each application receives one queue and one channel:

- Request queue
- Response channel

For the request queue, it contains message transformed by the proxy. The producer is the proxy, who transform incoming request into message to put in queue and the consumer is a QAdapter connected to this queue (it may have more than one QAdapter if the POD is scaled up).

The response channel works with the publisher/subscriber pattern. The publisher is a QAdapter that finished to process a request and put the response in the response channel, and the subscribers are the multiple instances of the Responder component.

6.6 Monitoring and HPA

KubeMQ exposes metrics that can be used by a metric server as Prometheus.

The Kueuer system uses a custom Prometheus configuration to setup a Prometheus server. Bellow, a sample of the routes defined for the Prometheus server:

```yaml
- name: queues
    rules:
      - record: kubemq:messages:count:queues:send:sum
        expr: sum(kubemq_messages_count{type="queues",side="send"}) OR on()
         ↪  vector(0)
      - record: kubemq:messages:count:queues:receive:sum
        expr: sum(kubemq_messages_count{type="queues",side="receive"}) OR on()
         ↪  vector(0)
      - record: kubemq:messages:count:queues:sum
```

Theses routes allow to use pre-configured question to ask the Prometheus server.

6.6.1 Prometheus Adapter

Prometheus is a server capable of retrieving metrics from other services but it is not able to serve them the way a HPA configuration can use.

A HPA (Horizontal Pod Autoscaler) need to interact with the basic metrics server, the custom metrics server or the external metrics server defined in the K8s' API.

Prometheus is not suitable to be configured directly as custom or external metrics server.

This is why we use a component named "Prometheus-Adapter". This Prometheus adapter acts as external or custom metrics server (it implements the specification of the K8s custom/external metrics server) and is also capable of asking a Prometheus server to retrieve metrics.

In the case of the Kueuer system, the Prometheus Adapter will act as **external metrics** server because we want to base a HPA on metrics coming from another pod (metrics don't come from the pod the HPA is the controller but from another pod, here KubeMQ).

6.6.2 HPA

With the Prometheus Adapter, it is possible now to set up a HPA based on external metrics. A HPA can be based on multiple metrics at the same time and so, it is possible to declare several metrics in the Prometheus Adapter and base the HPA on all theses metrics.

6.6.3 Implementation

The most difficult part of the implementation of working HPA based on KubeMQ metrics is the configuration. The most time consuming to understand is the Prometheus adapter configuration. Later, to install Prometheus Adapter, the package manager "HELM" is used.

The configuration discussed here is the helm configuration of the community version of the Prometheus Adapter available here[20].

A whole explanation (see Configuration Walkthrough[20]) about the configuration's options of the Prometheus Adapter is available on the official repository[21].

```yaml
prometheus:
  url: http://prometheus-service.kubemq.svc
  port: 8080
rules:
```

[20] https://artifacthub.io/packages/helm/prometheus-community/prometheus-adapter
[21] https://github.com/kubernetes-sigs/prometheus-adapter

```
  external:
  - seriesQuery: '{__name__=~"kubemq_messages_count.*"}'
    resources:
      overrides:
        namespace:
          resource: namespace
        pod:
          resource: pod
    name:
      matches: ^(.*)
      as: "queue-length-increase-rate"
    metricsQuery: rate(sum(kubemq_messages_count{<< range $key, $value :=
    ↪  .LabelValuesByName >><< if ne $key "namespace" >><< $key >>="<< $value
    ↪  >>",<< end >><< end >>})[10m:])
```

First thing to see in this configuration is the Prometheus server address.

Then we have a "rules" section containing several rules. Each rule is divided in 4 parts:

1. The Metric discovery (discover the metric available on the Prometheus server. Here it will check for all metrics available on the Prometheus server beginning with "kubemq_message_count. prefix). SeriesQuery
2. Association. Used to decide to which element in the Kubernetes environment this metric is available. Here, the metric will be available for POD elements and Namespaces. Resources section
3. Naming (define with which pattern the metrics must be served). Here the metric is served with the name "queue-length-increase-rate".
4. Querying. The query to execute on the Prometheus server to retrieve the metric.

On the first implementation of the configuration, I created a list of exact same queries with each time another property set in the kubemq_messages_count(<here>). This property changed regarding the queue we are looking for. This method was not good and another solution has been implemented: let the HPA select with a field on which queue it wants to retrieve metrics.

6.6.4 Deployment

First, the Prometheus server must be deployed. To do so, a already existing configuration to retrieve the metrics from KubeMQ has been created and is available here[22].

In the current state of the project, it is needed to pull the "Deployment" repository and apply the file "Promethues.yaml" chap. 11.3.1

```
git clone git@gitlab.forge.hefr.ch:ps6-2021-kubemq/deployment.git
cd deployment
kubectl apply -f kubemq-dashboard/promethues.yaml
```

Now that Prometheus is running, it is time to setup the Prometheus adapter. To do so, it is more convenient to use "helm" as the official repository[23] describes. A configuration file has already

[22] https://gitlab.forge.hefr.ch/ps6-2021-kubemq/deployment/-/blob/29be92af3ed8f789406e5b3b37ca2b1ca5cbdcee/kubemq-dashboard/promethues.yaml

[23] https://github.com/kubernetes-sigs/prometheus-adapter

been implemented (chap. 11.3.2) in the deployment/kubemq-dashboard folder can be used like this:

```
cd deployment
# Ajout du repo helm
helm repo add prometheus-community
 ↪  https://prometheus-community.github.io/helm-charts
# Création
helm install your_release --namespace=<namespace> -f
 ↪  kubemq-dashboard/prometheus-adapter-config.yaml
 ↪  prometheus-community/prometheus-adapter
```

A HPA is based on External metrics and each external metrics must contain in their "selector" properties these information: "channel=s1-request,type=queues,side=send"

Name	Description	Example	default value
channel	the KubeMQ queue on which the HPA must be based	s1-request	-

in addition, the HPA can set every KubeMQ properties for selectioin of metrics it wants. All the properties are listed on KubeMQ documentation[21].

Here an example of a configured HPA to use two different metrics exported by the Prometheus Adapter:

```yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: s1-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: s1
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: External
    external:
      metric:
        name: queue-length-increase-rate
        selector: "channel=s1-request,type=queues,side=send"
      target:
        type: Value
        value: 0.2
```

The final HPA used is available in the appendix chapter 11.2.2.

Warning: When setting HPA, do not forget to set limits in the concerned pod.

6.7 Evolution

There is a lot of possible evolution for this project, but two are the most important. The following sections describe the both main evolution/improvements to make to this system.

6.7.1 Automation / Injection

One of the main goals to a project like this is to remove all the configuration and make it a lot more simpler to deploy.

Istio (Service Mesh) can be taken as example. There is quasi no-need of heavy configuration to launch Istio. It automatically deploys proxy, sidecars and so on.

The first main evolution is to develop a system capable of injecting proxy component like Istio make and like this, the user will not have any configuration to do. Everything can be automated.

To follow this first possible evolution, here a list of maybe useful links:

- Explanation of the creation of a sidecar injector: https://engineering.salesforce.com/a-generic-sidecar-injector-for-kubernetes-c05eede1f6bb
- Open-source sidecar injector: https://github.com/tumblr/k8s-sidecar-injector
- Google explaining how to use Istio as sidecar injector: https://cloud.google.com/service-mesh/docs/proxy-injection
- Kubernetes Admission Controllers: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/
- Kubernetes Dynamic admission control: https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/

6.7.2 Language Refactoring

When this project began, I (Romain Agostinelli), didn't know Kubernetes and the language chosen for developing solution was Java as it is a well know language and typed language.

It is during the development/project that I figured out that the language used everywhere for Kubernetes is Go (it makes sense as Go was developed inside Google).

A big improvement would be to re-code the entire project with Go to make it compatible with other existing components for Kubernetes.

The Go language has been defined as the default language to works on Kubernetes by Google and the community.

The more early this refactor will be made, the more easily it will be made. For now, the applications are not complexes.

7 Tests

The final stage of the project consists of testing the solution developed.

This chapter discusses the tools used for testing, the two kinds of test done with results, what issues/bugs the the tests shows up and finally a synthesis/conclusion.

As a reminder, the goal of the project is queueing incoming requests to a service inside a Kubernetes infrastructure and being capable of scaling the service regarding queue metrics.

7.1 Tools

For testing the solution developed, a lot of existing tools exist and are very useful.

7.1.1 Siege

A first one, developed especially for testing REST services, is siege. Siege is an open-source utility that can perform load testing. It is used via a terminal, and configuration is done via command options. Here an example of the use of siege that generate 10 parallel threads during 10 min with other output configurations:

```
siege -q -v  -c 10 -b --log=./siege.log -j -t 10m http://127.0.0.1:62569
```

The official repository for siege is accessible here[24].

7.1.2 K6

K6 is well known tool for load testing. It is written in Go and allows to configure load tests with virtual users and complex user simulation.

It uses a NodeJS interface (facade) to setup tests but then use a core written in Go for performance.

It is really easy to use and offers a lot of functionalities, like virtual users, staging, checks and so on.

The official documentation is available here[25].

The two kinds of tests made at the end of the project are using K6 as testing tool, utility.

7.2 Tests and results

This chapter shows the two kinds of test that were made, with their configuration and procedure.

7.2.1 Environment simulation

For testing, a simulation environment has been setup and the tests are again one service randomly waiting between 0 and 30 seconds. There is no need to simulate CPU consumption because the goal is to scale based on queues metrics and not on resources used.

The queue metric used for auto-scaling is the queue incoming messages instant rate. This metric show a good status of number of incoming requests treated by the proxy.

The environment infrastructure is the same as presented on the figure 13 (chapter 6.1.1) and is based on one service.

The K8s cluster has these resources:

Resources	Amount
CPU	8 cores
Memory	10240 MB

[24] https://github.com/JoeDog/siege
[25] https://k6.io/docs/

To setup the environment, every "yaml" files are available on GitLab in the deployment project under the "kueuer" directory. The deployment project is available here[26].

Below, the directory tree with what to deploy first:

```
kueuer
|-- merged_configs
|    |-- app-kueuer-proxy-outside.yaml
|    |-- app-kueuer-proxy-sidecar.yaml
|-- metrics
|    |-- bck
|    |    |-- prometheus-adapter-config-backup.yaml
|    |-- prometheus-adapter-config.yaml
|    |-- prometheus.sh           #<----- 2nd, install metrics env. on the K8s
 ↪  cluster
|    |-- promethues.yaml
|-- minikube-setup.sh            #<----- 1st, install minikube env. and KubeMQ
|-- test_environment
|    |-- app-hpa.yaml
|    |-- app-service.yaml
|    |-- app-with-qadapter.yaml
|    |-- bck
|    |    |-- hpa-conf-backup.yaml
|    |-- deploy-sample-app.sh    #<----- 3rd, deploy a simple Kueuer
 ↪  infrastructure
|    |-- proxy-outside.yaml
|    |-- responder-hpa.yaml
|    |-- responder-service.yaml
|    |-- responder.yaml
|    |-- siege.log
|-- tests                        #<----- 4th, try launching a K6 test.
     |-- script-func.js
     |-- script-stress.js
```

For deployment, please open the ".sh" files instead of executing them, they contain instruction and listing of commands to help installing the environment.

7.2.2 Threshold

In the automated testing tool K6, there is the possibility to set threshold. Threshold are the minimum requirements for a test to pass. In the tests described in the sections bellow, there is one threshold set and it is based on the metric "http request failed".

For validating the test, the rate of http requests failed must be under 0.01.

```
thresholds: {
    http_req_failed: [{ threshold: 'rate<0.01', abortOnFail: false }],
},
```

[26] https://gitlab.forge.hefr.ch/ps6-2021-kubemq/deployment

7.2.3 Functional testing

First of all, the functional testing: does everything works?, is there issues / bugs?

This first part of the tests is checking main goals of the project as:

- Queueing incoming requests
- Auto-scaling based on metrics from queues

and also find and report possible bugs / issues.

To do so, the automated test uses 3 stages:

1. Normal amount of incoming requests -> Nothing to scale up/down
2. Increasing amount of incoming requests -> Must scale up regarding queues metrics
3. Bellow normal amount of incoming requests -> Must scale down regarding queues metrics

In K6 words, these three steps are translated to:

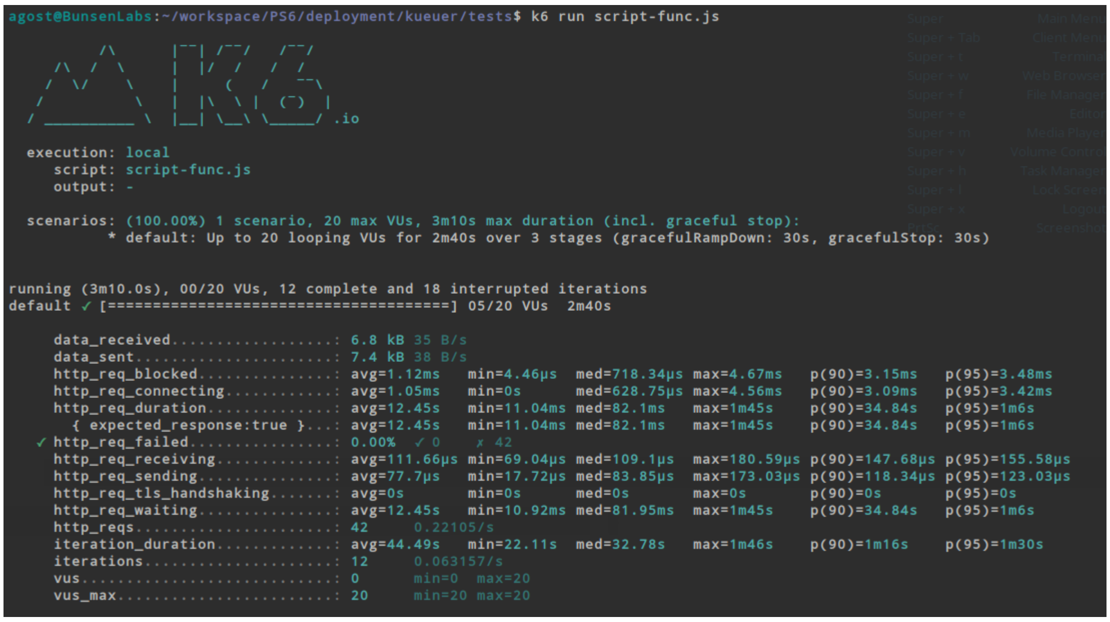
```
stages: [
    { duration: '1m', target: 5 }, // normal load
    { duration: '40s', target: 20 }, // autoscaler increase
    { duration: '1m', target: 5 }, // autoscaler down
],
```

The complete configuration used for the functional test is available at chapter 11.4.1.

The goal is to show if the environment works by using it normally, stressing a bit and then decrease the number of incoming requests. Here, the "target" value do not signify requests/seconds but number of concurrent virtual users making calls to the service.

Results:
The tests have been run several times and always pass, the pods are scaled up correctly and scaled down too.

Here an example of a K6 test result (figure 19):

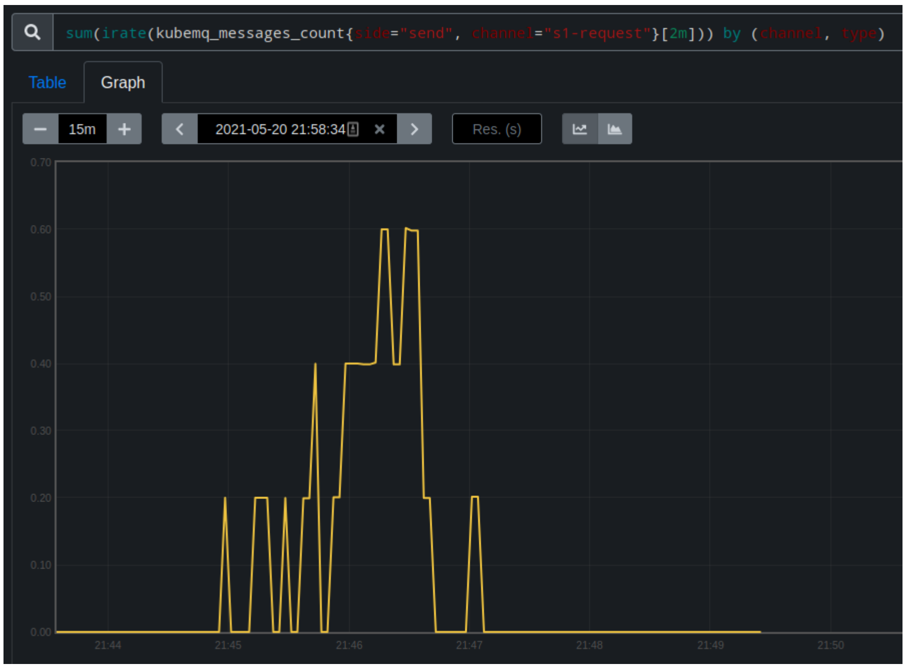


```
agost@BunsenLabs:~/workspace/PS6/deployment/kueuer/tests$ k6 run script-func.js

          /\      |‾‾| /‾‾/   /‾‾/
     /\  /  \     |  |/  /   /  /
    /  \/    \    |     (   /   ‾‾\
   /          \   |  |\  \ |  (‾)  |
  / __________ \  |__| \__\ \_____/ .io

  execution: local
     script: script-func.js
     output: -

  scenarios: (100.00%) 1 scenario, 20 max VUs, 3m10s max duration (incl. graceful stop):
           * default: Up to 20 looping VUs for 2m40s over 3 stages (gracefulRampDown: 30s, gracefulStop: 30s)


running (3m10.0s), 00/20 VUs, 12 complete and 18 interrupted iterations
default ✓ [======================================] 05/20 VUs  2m40s

     data_received..................: 6.8 kB 35 B/s
     data_sent......................: 7.4 kB 38 B/s
     http_req_blocked...............: avg=1.12ms   min=4.46µs  med=718.34µs max=4.67ms   p(90)=3.15ms   p(95)=3.48ms
     http_req_connecting............: avg=1.05ms   min=0s      med=628.75µs max=4.56ms   p(90)=3.09ms   p(95)=3.42ms
     http_req_duration..............: avg=12.45s   min=11.04ms med=82.1ms   max=1m45s    p(90)=34.84s   p(95)=1m6s
       { expected_response:true }...: avg=12.45s   min=11.04ms med=82.1ms   max=1m45s    p(90)=34.84s   p(95)=1m6s
   ✓ http_req_failed................: 0.00%  ✓ 0     ✗ 42
     http_req_receiving.............: avg=111.66µs min=69.04µs med=109.1µs  max=180.59µs p(90)=147.68µs p(95)=155.58µs
     http_req_sending...............: avg=77.7µs   min=17.72µs med=83.85µs  max=173.03µs p(90)=118.34µs p(95)=123.03µs
     http_req_tls_handshaking.......: avg=0s       min=0s      med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=12.45s   min=10.92ms med=81.95ms  max=1m45s    p(90)=34.84s   p(95)=1m6s
     http_reqs......................: 42     0.22105/s
     iteration_duration.............: avg=44.49s   min=22.11s  med=32.78s   max=1m46s    p(90)=1m16s    p(95)=1m30s
     iterations.....................: 12     0.063157/s
     vus............................: 0      min=0  max=20
     vus_max........................: 20     min=20 max=20
```

Figure 19: K6 final result for functional test

On the result showed above (figure 19), we can see that the average response time is 12.45s what is accurate and what we can expect for a service waiting between 0 and 30 seconds. Also, the maximum time is 1m45s, this happened when entering the stressing stage of the test, when they were not enough replicas to handle the request.

On the Prometheus server (figure 20), we can see that incoming messages instant rate metric reacts very well corresponding to the 3 defined stages and we can see each stages.



```
sum(irate(kubemq_messages_count{side="send", channel="s1-request"}[2m])) by (channel, type)
```

Figure 20: Prometheus metrics evolution for functional test

7.3 Stress test

The second kind of test made is a stress test. A stress test consists of testing the environment with a lot of incoming requests and check its behavior. For the stress test, a lower time consuming fake service is used due to resource limitation. It is possible to imagine scaling until 100 replicas of the same pod in big infrastructure with a lot of resource, but not with the environment used for these tests.

Here, the maximum amount of replicas allowed for the fake service is 30 and the threshold still the same. The goal of this stress test is to see if the queue can make a good buffer for the incoming requests rates while the auto-scaler increase the number of pods.

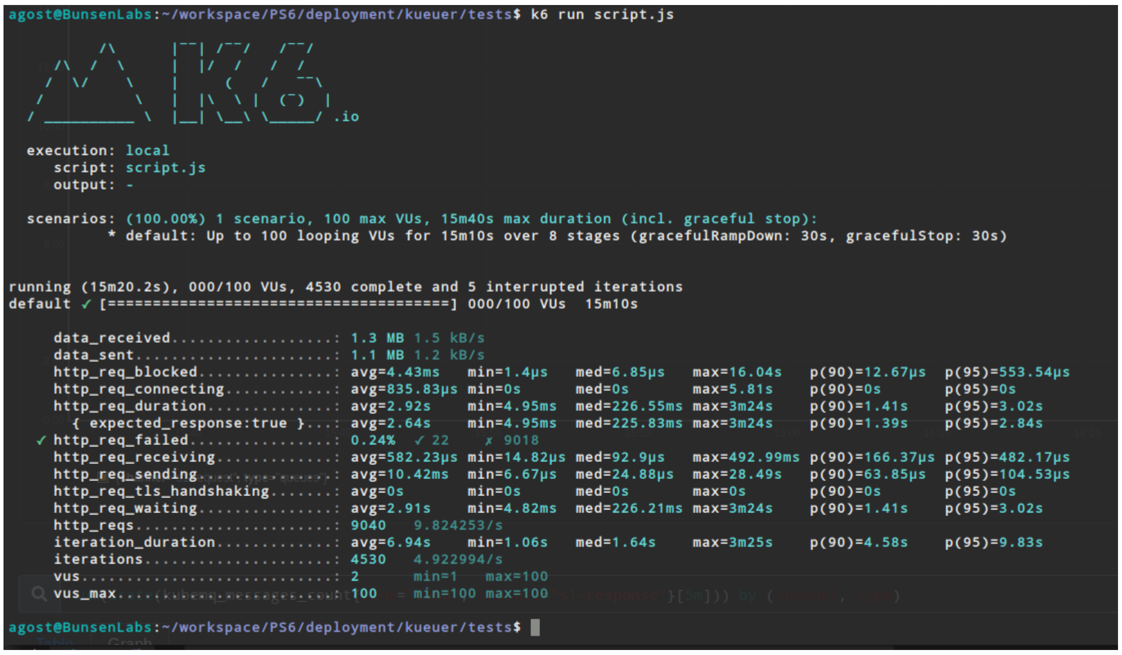
The test consists in 7 stages with a spike to 100 simultaneous virtual users trying to access the service. Below, the described stages in K6:

```
stages: [
    { duration: '10s', target: 10 }, // below normal load
    { duration: '1m', target: 10 },
    { duration: '10s', target: 100 }, // spike to 100 users
    { duration: '3m', target: 100 }, // stay at 100 for 3 minutes
    { duration: '3m', target: 20 },
    { duration: '7m', target: 10 }, // down sclaing time
    { duration: '40s', target: 50 }, // stress
],
```

The complete configuration used for the stress test is available at chapter 11.4.2.

Results:

On the figure 21 bellow, we can see the K6 result for this test:

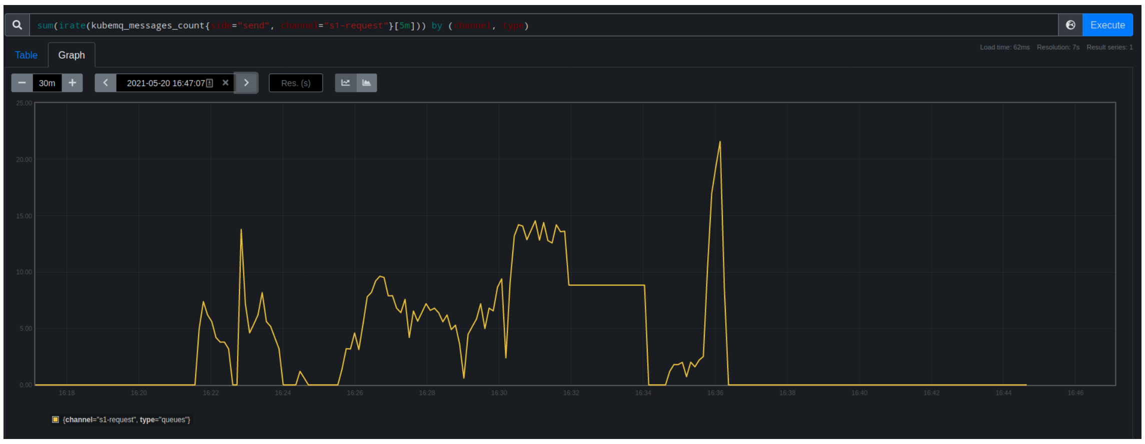


```
agost@BunsenLabs:~/workspace/PS6/deployment/kueuer/tests$ k6 run script.js

          /\      |‾‾| /‾‾/   /‾‾/
     /\  /  \     |  |/  /   /  /
    /  \/    \    |     (   /   ‾‾\
   /          \   |  |\  \ |  (‾)  |
  / __________ \  |__| \__\ \_____/ .io

  execution: local
     script: script.js
     output: -

  scenarios: (100.00%) 1 scenario, 100 max VUs, 15m40s max duration (incl. graceful stop):
           * default: Up to 100 looping VUs for 15m10s over 8 stages (gracefulRampDown: 30s, gracefulStop: 30s)


running (15m20.2s), 000/100 VUs, 4530 complete and 5 interrupted iterations
default ✓ [======================================] 000/100 VUs  15m10s

     data_received..................: 1.3 MB 1.5 kB/s
     data_sent......................: 1.1 MB 1.2 kB/s
     http_req_blocked...............: avg=4.43ms   min=1.4µs   med=6.85µs   max=16.04s   p(90)=12.67µs  p(95)=553.54µs
     http_req_connecting............: avg=835.83µs min=0s      med=0s       max=5.81s    p(90)=0s       p(95)=0s
     http_req_duration..............: avg=2.92s    min=4.95ms  med=226.55ms max=3m24s    p(90)=1.41s    p(95)=3.02s
       { expected_response:true }...: avg=2.64s    min=4.95ms  med=225.83ms max=3m24s    p(90)=1.39s    p(95)=2.84s
   ✓ http_req_failed................: 0.24%  ✓ 22     ✗ 9018
     http_req_receiving.............: avg=582.23µs min=14.82µs med=92.9µs   max=492.99ms p(90)=166.37µs p(95)=482.17µs
     http_req_sending...............: avg=10.42ms  min=6.67µs  med=24.88µs  max=28.49s   p(90)=63.85µs  p(95)=104.53µs
     http_req_tls_handshaking.......: avg=0s       min=0s      med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=2.91s    min=4.82ms  med=226.21ms max=3m24s    p(90)=1.41s    p(95)=3.02s
     http_reqs......................: 9040   9.824253/s
     iteration_duration.............: avg=6.94s    min=1.06s   med=1.64s    max=3m25s    p(90)=4.58s    p(95)=9.83s
     iterations.....................: 4530   4.922994/s
     vus............................: 2      min=1    max=100
     vus_max........................: 100    min=100 max=100

agost@BunsenLabs:~/workspace/PS6/deployment/kueuer/tests$
```

Figure 21: K6 final result for stress test

We can see that the system is capable of receiving a lot of incoming requests (regarding the environment's resources) and the threshold of requests failed rate passes. There is 0.24% of requests lost, due to different factors: environment limited resources, HPA scaling time, replicas creation time.

On the Prometheus server, the used metric is visible and pikes can be clearly distinguished (figure 22):



```
sum(irate(kubemq_messages_count{side="send", channel="s1-request"}[5m])) by (channel, type)
```

Figure 22: Prometheus metric evolution for final stress test

7.4 Bugs reported / fixed

The functional testing reveals 3 majors bugs, 2 on the Responder and one on the QAdapter:

1. Responder auto down scale closes pending requests
2. Responder stores too much events when there is multiple replicas
3. QAdapter auto down scale can cause loss of messages

Each of these three bugs are described in the next chapters.

7.4.1 Responder closes pending requests

During the stress test, a HPA has been setup for the Responder component. This HPA was also based (as the HPA for the service) on incoming messages to queue, but on the response queue (unlike the HPA for the service which is based on the request queue).

When the need of Reponder component decrease, the HPA will scale down the number of replicas for the Responder. The issue arrives when there is connected request on a Responder and this is shut down. All the requests are stopped for the client.

No concrete solution has been found, but a little improvement can be set up by configuring the scale down behavior of the Responder HPA. This improvement consists of expanding the stabilization window to prevent the down scaling too close in time with a request pike. Also, limiting the number of replicas that must be scaled down at the same time and setting a bigger time window between two down scale improved the tests results.

To do what described just above, these lines were added to the Responder HPA:

```yaml
behavior:
  scaleDown:
    stabilizationWindowSeconds: 600 # 10 minutes
    policies:
    - type: Pods
      value: 1
      periodSeconds: 600 # wait 10 minutes before scaling down another replica
    selectPolicy: Min
```

7.4.2 Responder stores too much events

As a reminder: The Responder component is based on a KubeMQ Events Store with Pub/Sub protocol on the response queue. The Pub/Sub protocol implies that the subscriber receives all the events published to the response queue. This means that if there are 3 subscribers to the response queue and a event "A" is published, each of the 3 subscribers will receive the event "A". This is not like a standard queue with publisher/consumer where only one of the consumers receive a message published in the queue.

The Responder must use the Pub/Sub protocol because when the Proxy component redirects to a Responder, it does not redirect to a Responder directly, but to its service/load balancer and it is impossible to predict to which Responder replica the load balancer will redirect the request. So, every Responder must receive all the events published in the response queue they are attached to.

As described in the chapter 6.4.2 (Responder implementation), the Responder implements a storage mechanism of events for all events not linked to an existing observer. The issue is that a responder does not receive all corresponding requests for all the events it receives because the requests are distributed accross the Responder replicas but the events are sent to all Responders.

This implies that the store of the Responder makes 3 times the number of requests it receives (if considering a equitable distribution done by the load balancer) when there are 3 replicas (same logic for other number of replica). This behavior can cause performance loss and high consumption of disk resource.

The solution found is to setup a cron job that will clean all outdated instances of events. To know which is outdated, it has to save the time they were added. Are considered outdated, all events which were added more time ago than the server timeout. This means that if there is a connection which requests an event, this connection will be closed before the event will be cleared from the event store.

To implement this logic, two classes were added: ResponseStore and StoreCleanerCron.

The StoreCleanerCron class implements a unique method that is launched every 10 minutes in another thread that the main one. Below, the implementation of the single method:

```java
@Scheduled(fixedRate = 1000*60*10)
public void cleanResponseStore() {
    log.warn("Clearing ResponseStore - Cron");
    ResponseStore.clearOldEvents(
    ↪   environment.getProperty("spring.mvc.async.request-timeout",Duration.class));
}
```

The response store is a synchronized implementation of a response store. It must be synchronized because the StoreCleanerCron runs in another thread a the map which represents the store is a shared resource.

On the figure 23 below, the class diagram of the implemented solution. 22):

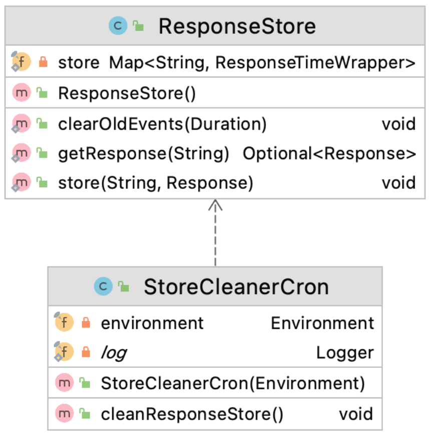

ResponseStore
store Map<String, ResponseTimeWrapper>
ResponseStore()
clearOldEvents(Duration) void
getResponse(String) Optional<Response>
store(String, Response) void
StoreCleanerCron
environment Environment
log Logger
StoreCleanerCron(Environment)
cleanResponseStore() void

Figure 23: Class diagram of implemented solution for responder storing too much events

7.4.3 QAdapter auto-scale loss of messages

When the service HPA shut down replicas for down scaling, requests on which the terminating replicas were working on (at the time of the shut down) are lost.

The same issue appears when shutting down simple Rest service which has pending requests.

This issue is not compatible with the project constraints where all messages must been processed.

The solution found to prevent loss of messages when a replica of a service is stopped, is to implement the use of Message visibility feature of KubeMQ. This feature (described in the chapter 4.9.1) allows to set a visibility to a message in queue without removing it from the queue. Visibility can be "visible" or "not visible". If a message is visible, every consumer see it like a normal message, but if it is not visible, it acts like there is no message. The visibility has a timer and if there is no ack for the message within the allotted time, the message visibility changes to "visible" and be accessed by other consumers.

When retrieving a message in the queue and setting visibility to it, a initial visibility duration must be set. The consumer who got the message and set the visibility time has the possibility to expend this time if he needs more time.

The implementation of the visibility logic requires some changes in the QAdapter component. Instead of blocking on the response of the service the QAdapter is linked to (as described in chapter 6.3.2), it will send the request asynchronously and during the time between the request sending and the response receiving, it will check for the response (polling). If the response is not present and no issues appeared, it extends the visibility time to ask for more time. Time between checks (polling) is constant.

Sending visibility extension to KubeMQ can overload traffic the QAdapter must send the less visibility extension as possible.

So, a visibility extension is only sent when the current time + time between response check appears later than the end of the current visibility time. To illustrate this strategy, if we take the figure 24, if the current time is $t_0 + 1$ and there is still no response from the service, we won't make a visibility extension because we know that at the next time the response will be checked

($t_0 + 2$), the visibility still valid. On the other hand, if the current time is $t_0 + 4$, we now that at our next response check ($t_0 + 5$) the visibility will not be valid anymore. This means that we need to extend the visibility before the next response check.

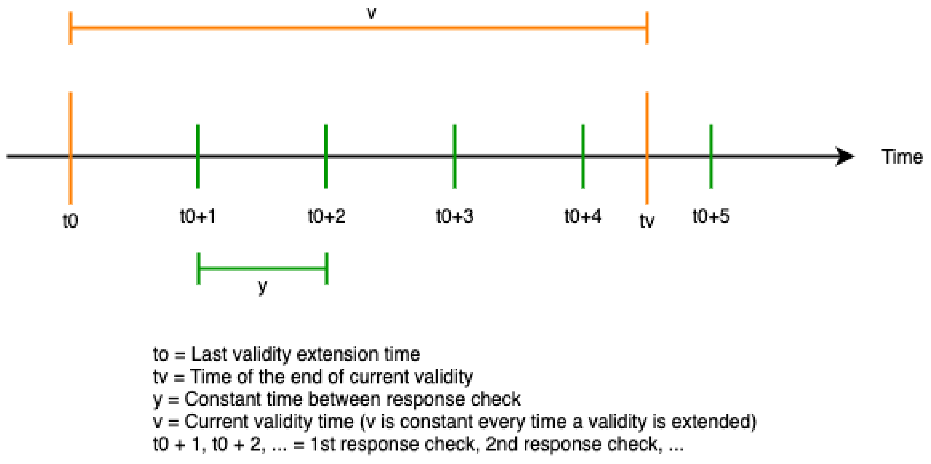

v
Time
t0
t0+1
t0+2
t0+3
t0+4
tv
t0+5
y
to = Last validity extension time
tv = Time of the end of current validity
y = Constant time between response check
v = Current validity time (v is constant every time a validity is extended)
t0 + 1, t0 + 2, ... = 1st response check, 2nd response check, ...

Figure 24: Graph representing validity check on QAdapter

Implementation of this strategy is done by 2 main changes: Adding a new class named "TransactionContextHolder" that is used as buffer class for response polling. When the response is received, it will push the response into an instance of this class and the polling loop will make the checks for response on the same instance. This class is synchronized to avoid conflict between the thread of the WebClient and the main train doing polling. This class also implement the possibility to save the last visibility extension time and is capable of return the duration between the last extension time and the current time. Adding a new method named "maintainConnectionUntilResponse" capable of polling and extension of the current visibility. This new method replaces the use of the method ".block()" on "Mono<?>" object used to send the request to the service.

7.5 Tests synthesis

The implemented solution works. It is not in production form but more like a proof of concept. There is still some little bugs related to KubeMQ Transaction that appears some times.

When comparing with a simple REST service without the Kueuer system we can find some advantages and some disadvantages of the Kueuer solution:

Disadvantages:

- Resource consumption. The need of deploying new objects in the Kubernetes environment, like the Proxy, the Responder, the Prometheus server and adapter and all the QAdapters makes the Kueuer solution not suitable for K8s clusters limited in resources.
- Scaling reactivity. With the QAdapter container in each service, it increases the time for Kubernetes to create a new replica of the service which slows down the scaling reactivity.
- Request duration. The request duration may be increased due to the complexity the Kueuer system adds.

Advantages:

- Distributed works. The Kueuer system works with queues and this offers the possibility to easily attach new worker on it.
- Less rejected requests. With the queue working as a buffer, the Kueuer system offers less rejections of requests, they will be processed and not returned to the client as rejected because of overloading.
- No message lost, less client re-call. When downscaling the amount of replicas of a service, no message will be lost, it will be automatically re-injected into the request queue.

8 Conclusion

This section discusses the conclusion of the project. The conclusion is divided in four different parts. The first part is the objectives of the project, then the second part, the state of the project: what was achieved. Next the third part is evolution of the project and the next steps and finally, the last part, a personal conclusion on the project by the writer (Romain Agostinelli).

8.1 Objectives and achievement

The goal of the project is to set a system capable of queuing and scaling (based on queue metrics) in a Kubernetes environment.

Multiple proposals have been made to solve this problem and one has been chosen. The name of the chosen solution is "Kueuer" (for Kubernetes and Queue). This solution is implemented and tested.

The main objective of the project has been achieved. It remains little bugs and improvements to do but a working solution solution can be easily deployed and tested.

Moreover, the developed system can be used in different environment and is easily configurable (do not depends on a specific Kubernetes environment).

8.2 State of the project

A system named "Kueuer" is developed, documented and tested. This system is actually based on KubeMQ and workers coded in Java.

The developed system is not a production system. It is more a "proof-of-concept" and must not be use as-is in a production environment. A complete deployment directory is available on the project's GitLab (see ch. 9).

8.3 Evolution of the project

There is two things to distinguish, the project and the developed solution.

For improvement for the developed solution, please refer to the chapter 6.7.

For the project, a good evolution would be to test other forms of solution, other system.

After testing the Kueuer system, the main problem detected is the consumption of resources. Maybe an alternative solution would be to decrease the number of new component in the architecture. Implementing a single proxy which does everything may be solution. It could improve reactivity by limiting the resources needed for launching new replica. I think it is a good way for next researches based on the same subject.

8.4 Personal conclusion

For me, it was the first time working with Kubernetes. It requires a bit of time to become familiar with Kubernetes environment but when I began to understand how everything works, how things communicate and what is possible, I immediately loved working with it. Today, I think I have a good comprehension of how Kubernetes works and it may help me developing project for Kubernetes in the future.

The subject is a real problem case and searching, experimenting and developing solutions was very exciting. During the whole project, I acquired a lot of knowledge, both in computer architecture and in development.

The only regret I have, is to not have more time to works on this project. I think I will continue to work on solutions on my side, testing things. It may be a good project for me to learn the Go language to move the created solution into a Go program.

I became really attached and interested in this topic, queuing and scaling. The project led me to something I never did: developing for IT infrastructure. I always developed solution for client requirement, not for architecture requirement and today, after this project, I think I found what I really want to do later: working on this kind of projects.

I made some mistakes during the project that I wanted to list here as a reminder for future projects:

- Plan more time for testing and begin testing at the beginning of the project
- Implement automated tests and deployment
- Plan development pause to check what already exists, to take step back
- Always make your estimated task time $*\pi$
- Keep it simple

9 Resources

All the developments, code and documentation is available on the project's GitLab at this address:

`https://gitlab.forge.hefr.ch/ps6-2021-kubemq`

Also, already compiled on images of the "Kueuer" system are available on the Docker Hub page of the writer:

`https://hub.docker.com/u/agost/` and begin with the prefix: "agost/io.kueuer.*"

10 Déclaration d'honneur - Declaration of honour

Je, soussigné, Romain Agostinelli, déclare sur l'honneur que le travail rendu est le fruit d'un travail personnel. Je certifie ne pas avoir eu recours au plagiat ou à toutes autres formes de fraudes. Toutes les sources d'information utilisées et les citations d'auteur ont été clairement mentionnées.

Romain Agostinelli:

[illegible]

Glossary

K8s Synonym for "Kubernetes". This mean "K" then 8 letters and finally "s" to form the word "Kubernetes". . 2, 3

OS Operating System. Most known are Windows (Microsoft), macOS (Apple) and Linux (multiple OS based on Unix kernel). 4

References

[1] Kubernetes. Kubernetes manage resources container.

[2] Kubernetes. Kubernetes api server aggregation.

[3] Kubernetes. Kubernetes controllers.

[4] Kubernetes. Kubernetes components.

[5] Kubernetes. Kubernetes manage resources container.

[6] Kubernetes. Kubernetes pod autoscal walkthrough.

[7] Kubernetes. Jobs.

[8] Kubernetes. Cron job.

[9] Kubernetes. Fine parallel processing work queue.

[10] Bhargav Bachina. Kubernetes — learn sidecar container pattern.

[11] Baeldung. Java fork join pool.

[12] Docker. Docker command run.

[13] Kubernetes. Define environment variable.

[14] NGINX. Advanced configuration with annotations.

[15] F5 Manual. Preventing tcp connection requests from being dropped.

[16] Nick Ramirez. Protect servers with haproxy connection limits and queues.

[17] KubeMQ. Kubemq java sdk.

[18] Baeldung. Spring 5 webclient.

[19] VERSPRITE. Spring 5 webclient.

[20] Prometheus Adapter. Configuration walkthrough.

[21] KubeMQ. Kubemq dashboard.

11 Appendixes

This section contains all the different appendixes of the project.

11.1 Simple App with HPA based on CPU consumption

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: s1
  labels:
    app: s1
spec:
  selector:
    matchLabels:
      app: s1
  replicas: 1 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: s1
    spec:
      containers:
      - name: s1
        image: agost/io.kueuer.timesim:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 3000
        resources:
          requests:
            cpu: "250m"
---
apiVersion: v1
kind: Service
metadata:
  name: s1-service
spec:
  selector:
    app: s1
  type: LoadBalancer
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 3000
    nodePort: 30000
---
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: s1-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: s1
  minReplicas: 1
  maxReplicas: 30
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 20
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 60 # 1 minutes
      policies:
      - type: Pods
        value: 1
        periodSeconds: 60 # wait 1 minutes before scaling down (1 by 1)
      selectPolicy: Min
```

11.2 Environment deployment configuration files

11.2.1 App with QAdapter Deployment file

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: s1
  labels:
    app: s1
spec:
  selector:
    matchLabels:
      app: s1
  replicas: 1 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: s1
    spec:
      containers:
      - name: s1
        image: agost/io.kueuer.timesim:latest
        imagePullPolicy: Always
      - name: qadapter
        image: agost/io.kueuer.qadapter:latest
        imagePullPolicy: Always
        env:
        - name: port
          value: "3003"
        - name: queue_addr
          value: "kubemq-cluster-grpc.kubemq.svc.cluster.local:50000"
        - name: request_queue
          value: "s1-request"
        - name: response_channel
          value: "s1-response"
        - name: service_port
          value: "3000"
```

11.2.2 App - HPA Based on Queue metrics

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: s1-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: s1
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: External
    external:
      metric:
        name: queue-incoming-message-per-second
        selector: {
          matchLabels: {
            channel: "s1-request",
            type: "queues"
          }
        }
      target:
        type: Value
        value: 0.05
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 30 # 30 secs
      policies:
      - type: Pods
        value: 2
        periodSeconds: 30 # wait 30 seconds before scaling down (2 by 2)
      selectPolicy: Min
```

11.2.3 Proxy Deployment file

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: s1-proxy
  labels:
    app: s1-proxy
spec:
  selector:
    matchLabels:
      app: s1-proxy
  replicas: 1 # set the number of replicas
  template:
    metadata:
      labels:
        app: s1-proxy
    spec:
      containers:
      - name: s1-proxy
        image: agost/io.kueuer.proxy:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 3001
        env:
        - name: port
          value: "3001"
        - name: queue_addr
          value: "kubemq-cluster-grpc.kubemq.svc.cluster.local:50000"
        - name: queue
          value: "s1-request"
        - name: responder_addr
          value: "192.168.99.111:30001"
          #value: "responder-service.default.svc.cluster.local:8080"
---
apiVersion: v1
kind: Service
metadata:
  name: s1-service
spec:
  selector:
    app: s1-proxy
  type: NodePort
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 3001
    nodePort: 30002
```

11.2.4 Reponder deployment configuration

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: responder
  labels:
    app: responder
spec:
  selector:
    matchLabels:
      app: responder
  replicas: 1 # set number of replicas here
  template:
    metadata:
      labels:
        app: responder
    spec:
      containers:
      - name: responder
        image: agost/io.kueuer.responder:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 8080 # Default app is 8080
        env:
        - name: queue_addr
          value: "kubemq-cluster-grpc.kubemq.svc.cluster.local:50000"
        - name: response_channels
          value: "s1-response"
---
apiVersion: v1
kind: Service
metadata:
  name: responder-service
spec:
  selector:
    app: responder
  type: NodePort
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
    nodePort: 30001
```

11.3 Metrics configurations files

11.3.1 Prometheus Configuration

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prometheus-deployment
  namespace: kubemq
  labels:
    app: prometheus-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: prometheus-server
  template:
    metadata:
      labels:
        app: prometheus-server
    spec:
      containers:
        - name: prometheus
          image: prom/prometheus
          args:
            - "--config.file=/etc/prometheus/prometheus.yml"
            - "--storage.tsdb.path=/prometheus/"
          ports:
            - containerPort: 9090
          volumeMounts:
            - name: prometheus-config-volume
              mountPath: /etc/prometheus/
            - name: prometheus-storage-volume
              mountPath: /prometheus/
      volumes:
        - name: prometheus-config-volume
          configMap:
            defaultMode: 420
            name: prometheus-server-conf
        - name: prometheus-storage-volume
          emptyDir: {}
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: prometheus
rules:
  - apiGroups: [""]
    resources:
      - nodes
      - nodes/proxy
      - services
      - endpoints
      - pods
    verbs: ["get", "list", "watch"]
  - apiGroups:
      - extensions
    resources:
      - ingresses
    verbs: ["get", "list", "watch"]
  - nonResourceURLs: ["/metrics"]
    verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
  - kind: ServiceAccount
    name: default
    namespace: kubemq
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-server-conf
  labels:
    name: prometheus-server-conf
  namespace: kubemq
data:
  prometheus.yml: |-
    global:
      scrape_interval: 5s
      evaluation_interval: 5s
    rule_files:
    - ./kubemq.rule.yaml
    scrape_configs:
    - job_name: scrape_kubemq
      kubernetes_sd_configs:
      - role: pod
        namespaces:
          names:
          - "kubemq"
      relabel_configs:
      - source_labels: [__meta_kubernetes_pod_container_port_number]
        action: keep
        regex: 8\d{3}
  kubemq.rule.yaml: |-
    groups:
      - name: events
        rules:
          - record: kubemq:messages:count:events:send:sum
            expr: sum(kubemq_messages_count{type="events",side="send"}) OR on()
             ↪  vector(0)
          - record: kubemq:messages:count:events:receive:sum
            expr: sum(kubemq_messages_count{type="events",side="receive"}) OR
             ↪  on() vector(0)
          - record: kubemq:messages:count:events:sum
            expr:
             ↪  kubemq:messages:count:events:send:sum+kubemq:messages:count:events:receive:sum
          - record: kubemq:messages:volume:events:send:sum
            expr: sum(kubemq_messages_volume{type="events",side="send"}) OR
             ↪  on() vector(0)
          - record: kubemq:messages:volume:events:receive:sum
            expr: sum(kubemq_messages_volume{type="events",side="receive"}) OR
             ↪  on() vector(0)
          - record: kubemq:messages:volume:events:sum
            expr:
             ↪  kubemq:messages:volume:events:send:sum+kubemq:messages:volume:events:receive:sum
          - record: kubemq:messages:errors:events:sum
            expr: sum(kubemq_errors_count{type="events"}) OR on() vector(0)
      - name: events_store
        rules:
          - record: kubemq:messages:count:events_store:send:sum
            expr: sum(kubemq_messages_count{type="events_store",side="send"})
             ↪  OR on() vector(0)
          - record: kubemq:messages:count:events_store:receive:sum
            expr:
             ↪  sum(kubemq_messages_count{type="events_store",side="receive"})
             ↪  OR on() vector(0)
          - record: kubemq:messages:count:events_store:sum
            expr:
             ↪  kubemq:messages:count:events_store:send:sum+kubemq:messages:count:events_store:re
          - record: kubemq:messages:volume:events_store:send:sum
            expr: sum(kubemq_messages_volume{type="events_store",side="send"})
             ↪  OR on() vector(0)
          - record: kubemq:messages:volume:events_store:receive:sum
            expr:
             ↪  sum(kubemq_messages_volume{type="events_store",side="receive"})
             ↪  OR on() vector(0)
          - record: kubemq:messages:volume:events_store:sum
            expr:
             ↪  kubemq:messages:volume:events_store:send:sum+kubemq:messages:volume:events_store:
          - record: kubemq:messages:errors:events_store:sum
            expr: sum(kubemq_errors_count{type="events_store"}) OR on()
             ↪  vector(0)
      - name: queues
        rules:
          - record: kubemq:messages:count:queues:send:sum
            expr: sum(kubemq_messages_count{type="queues",side="send"}) OR on()
             ↪  vector(0)
          - record: kubemq:messages:count:queues:receive:sum
            expr: sum(kubemq_messages_count{type="queues",side="receive"}) OR
             ↪  on() vector(0)
          - record: kubemq:messages:count:queues:sum
            expr:
             ↪  kubemq:messages:count:queues:send:sum+kubemq:messages:count:queues:receive:sum
          - record: kubemq:messages:volume:queues:send:sum
            expr: sum(kubemq_messages_volume{type="queues",side="send"}) OR on() vector(0)
          - record: kubemq:messages:volume:queues:receive:sum
            expr: sum(kubemq_messages_volume{type="queues",side="receive"}) OR on() vector(0)
          - record: kubemq:messages:volume:queues:sum
            expr: kubemq:messages:volume:queues:send:sum+kubemq:messages:volume:queues:receive:sum
          - record: kubemq:messages:errors:queues:sum
            expr: sum(kubemq_errors_count{type="queues"}) OR on() vector(0)
      - name: queries
        rules:
          - record: kubemq:messages:count:queries:send:sum
            expr: sum(kubemq_messages_count{type="queries",side="send"}) OR on() vector(0)
          - record: kubemq:messages:count:queries:receive:sum
            expr: sum(kubemq_messages_count{type="queries",side="receive"}) OR on() vector(0)
          - record: kubemq:messages:count:queries:sum
            expr: kubemq:messages:count:queries:send:sum+kubemq:messages:count:queries:receive:sum
          - record: kubemq:messages:volume:queries:send:sum
            expr: sum(kubemq_messages_volume{type="queries",side="send"}) OR on() vector(0)
          - record: kubemq:messages:volume:queries:receive:sum
            expr: sum(kubemq_messages_volume{type="queries",side="receive"}) OR on() vector(0)
          - record: kubemq:messages:volume:queries:sum
            expr: kubemq:messages:volume:queries:send:sum+kubemq:messages:volume:queries:receive:su
          - record: kubemq:messages:errors:queries:sum
            expr: sum(kubemq_errors_count{type="queries"}) OR on() vector(0)
      - name: commands
        rules:
          - record: kubemq:messages:count:commands:send:sum
            expr: sum(kubemq_messages_count{type="commands",side="send"}) OR on() vector(0)
          - record: kubemq:messages:count:commands:receive:sum
            expr: sum(kubemq_messages_count{type="commands",side="receive"}) OR on() vector(0)
          - record: kubemq:messages:count:commands:sum
            expr: kubemq:messages:count:commands:send:sum+kubemq:messages:count:commands:receive:su
          - record: kubemq:messages:volume:commands:send:sum
            expr: sum(kubemq_messages_volume{type="commands",side="send"}) OR on() vector(0)
          - record: kubemq:messages:volume:commands:receive:sum
            expr: sum(kubemq_messages_volume{type="commands",side="receive"}) OR on() vector(0)
          - record: kubemq:messages:volume:commands:sum
            expr: kubemq:messages:volume:commands:send:sum+kubemq:messages:volume:commands:receive:
```

```
          - record: kubemq:messages:errors:commands:sum
            expr: sum(kubemq_errors_count{type="commands"}) OR on() vector(0)
      - name: responses
        rules:
          - record: kubemq:messages:count:responses:sum
            expr: sum(kubemq_messages_count{type="responses"}) OR on()
             ↪  vector(0)
          - record: kubemq:messages:volume:responses:sum
            expr: sum(kubemq_messages_volume{type="responses"}) OR on()
             ↪  vector(0)
          - record: kubemq:messages:errors:responses:sum
            expr: sum(kubemq_errors_count{type="responses"}) OR on() vector(0)

---
apiVersion: v1
kind: Service
metadata:
  name: prometheus-service
  namespace: kubemq
  annotations:
    prometheus.io/scrape: 'true'
    prometheus.io/port:   '9090'
spec:
  selector:
    app: prometheus-server
  type: NodePort
  ports:
    - port: 8080
      targetPort: 9090
      nodePort: 30000
```

11.3.2 Prometheus adapter configuration

```
prometheus:
  url: http://prometheus-service.kubemq.svc
  port: 8080
rules:
  external:
  - seriesQuery: '{__name__=~"kubemq_messages_count.*"}'
    resources:
      overrides:
        namespace:
          resource: namespace
        pod:
          resource: pod
    name:
      matches: ^(.*)
      as: "queue-incoming-message-per-second"
    metricsQuery: sum(irate(kubemq_messages_count{<< range $key, $value :=
     ↪  .LabelValuesByName >><< if ne $key "namespace" >><< $key >>="<< $value
     ↪  >>",<< end >><< end >>side="send"}[5m])) by (channel, type)
```

```
 ######################## WARNING: queue-length is not guarantee
 - seriesQuery: '{__name__=~"kubemq_messages_count.*"}'
   resources:
     overrides:
       namespace:
         resource: namespace
       pod:
         resource: pod
   name:
     matches: ^(.*)
     as: "queue-length"
   metricsQuery: sum(kubemq_messages_count{<< range $key, $value :=
     ↪  .LabelValuesByName >>< if ne $key "namespace" >><< $key >>="<< $value
     ↪  >>",<< end >><< end >>side="send"})-sum(kubemq_messages_count{<< range
     ↪  $key, $value := .LabelValuesByName >><< if ne $key "namespace" >><<
     ↪  $key >>="<< $value >>",<< end >><< end >>side="receive"})

# For the metric queries function, this is a workaround to an issue on external
 ↪  metrics on prometheus-adaper:
# workaround proposition:
 ↪  https://github.com/kubernetes-sigs/prometheus-adapter/issues/255#issuecomment-601955919
# Issue: https://github.com/kubernetes-sigs/prometheus-adapter/issues/282
```

11.4 K6 Tests configurations

11.4.1 Functional testing K6 configuration

```javascript
import http from 'k6/http';
import { sleep } from 'k6';

export let options = {
    stages: [
        { duration: '1m', target: 5 }, // normal load
        { duration: '40s', target: 20 }, // autoscaler increase
        { duration: '1m', target: 5 }, // autoscaler down
    ],
    thresholds: {
        http_req_failed: [{ threshold: 'rate<0.01', abortOnFail: false }],
    },
}

export default function () {
    let params = {
        timeout: '300s', // chrome default timeout
        tags: { k6test: 'yes' },
    };
    http.get('http://192.168.99.111:30002', params);
    sleep(1);
}
```

11.4.2 K6 Stress test configuration

```
import http from 'k6/http';
import { sleep } from 'k6';

export let options = {
    stages: [
        { duration: '10s', target: 10 }, // below normal load
        { duration: '1m', target: 10 },
        { duration: '10s', target: 100 }, // spike to 100 users
        { duration: '3m', target: 100 }, // stay at 100 for 3 minutes
        { duration: '10s', target: 20 }, // scale down. Recovery stage.
        { duration: '3m', target: 20 },
        { duration: '7m', target: 10 }, // down sclaing time
        { duration: '40s', target: 50 }, // stress
    ],
    thresholds: {
        http_req_failed: [{ threshold: 'rate<0.01', abortOnFail: false }],
    },
    discardResponseBodies: true,
}

export default function () {
    let params = {
        timeout: '300s', // chrome default timeout
        tags: { k6test: 'yes' },
    };
    http.get('http://192.168.99.110:30002', params);
    sleep(1);
}
```

12 Appendix - Cahier des charges - Specification

12.1 Introduction

These days, cloud services are receiving thousands of requests per second. During a normal development, it is often considered to use request queuing systems (RabbitMQ, Apache Kafka) to process requests in their order of arrival. These systems can create long waits if the number of requests per second is very high or if the service in question takes a long time to respond.

Another way to manage this request flow is to duplicate services so that several requests can be handled in parallel and thus reduce the wait.

Kubernetes is a popular state-of-the-art open-source computing workload orchestration system. It provides means to automate application deployments and scaling capabilities depending on the load of service requests.

The aim of the project is to combine hyperscaling capabilities of Kubernetes and queuing mechanisms to create an infrastructure capable of handling thousands of requests per second.

12.2 Context

The company Swisscom SA intends to migrate some of its services in a K8s environment and thus have a large Kubernetes micro-services infrastructure that exposes these different services.

Some micro-services take longer to respond than others, sometimes due to a service in general or sub-services that individually take a lot of time and slow down the processing of requests.

The goal is to be able to set up a system to manage requests and automatically increase the number of replicas of a service, whether it is a service in its entirety or only sub-services.

12.3 Objective

Implementation of a solution based on different techniques (such as hyperscaling and queuing) to handle numerous queries in order to improve request processing performance in a test environment simulating a large-scale Kubernetes infrastructure. The main constraint is the implementation of the *exactly-one semantics* concept in the solution.

12.4 Infrastructure topologies

In such infrastructure, multiple types of micro-services topologies can be imagined. This project considers two types: a first one, very simple, that will be used and setup first and a second one, more complex on which it will be tried to extend the solution found for the first one to make it works on a more complex infrastructure.

The project can then be divided in two parts, each one based on different micro-service topologies.

The both presented infrastructure topologies in the chapters below (figures 25 & 26) represent base infrastructure topologies. They are not optimized and the project aim is to improve these topologies regarding different constraints.

12.4.1 First part

For the first part of the project, a simple topology (figure 25) will be considered. The project will be focused on improving this topology and implementing a solution regarding the objectives and constraints.

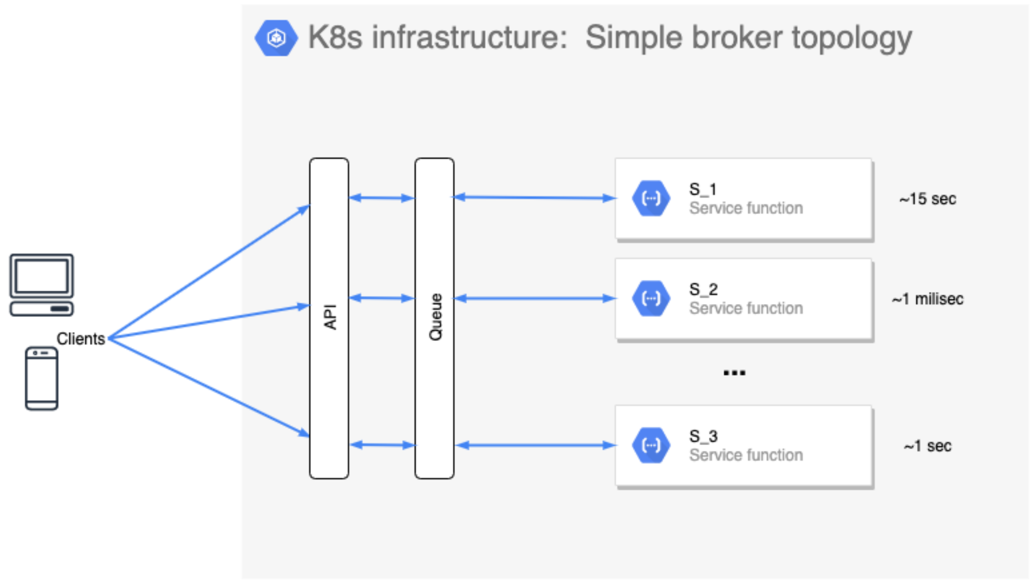


Figure 25: First simple topology

12.4.2 Second part

For the second part of the project, a more complex topology (figure 26) will be used and the solution found for the first part will be extended to fit this second topology and improve this one too.

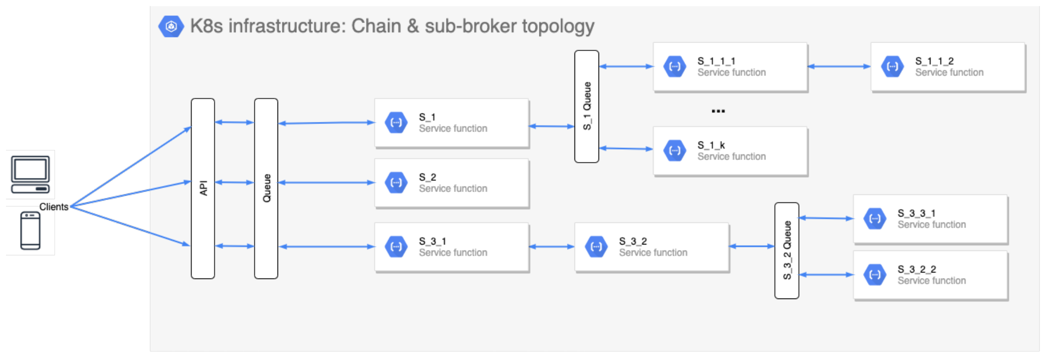


Figure 26: Second complex topology

12.5 Activities

In this section you will find the different activities present in the planning. Each sub-section represents a particular activity (step) of the project with a more precise definition of the deliverable as well as the tasks to be performed for the activity.

12.5.1 Kickoff and specs

Deliverable : Specification + planning

Tasks :

- Understand and getting familiar with K8s.
- Creation of the projects specifications and validate it.
- Creation of the planning using Agile method.
- Preliminary search:
  - Kubernetes
  - Overscaling
  - Queuing technique:
    - Apache Kafka
    - KubeMQ
    - RabbitMQ
  - Constraint:
    - Exactly-once semantics

12.5.2 Setting up a development environment

This second step of the project consists in setting up a development environment with Kubernetes using K3s or K8s solution.

Deliverable : Working development environment (K{3|8}s running).

Tasks :

- Compare and choose between K8s and K3s
- Setup K{3|8} environment
- Tests on infrastructure, learning technology
- Research about technology update

12.5.3 First micro-service

The third step of the project consists in the creation and integration in the K3|8s development environment of multiple micro-services which simulate short and long processing request time. This is a simple micro-service environment, a more complex will come later.

It also contains the first research and experimentation of solutions allowing to solve the objective.

Deliverable : Micro-services which simulate request processing with various amount of time.

Tasks :

- Setup first micro-service topology infrastructure
- Research about exactly-once semantics
- Research of existing solution
- Comparison of solutions
- Proposal of one solution

12.5.4 Solution

This fourth step consists of the implementation of the chosen solution in the previous step in the test environment. This contains multiple parts as analysis and conception of the chosen solution and then implementation.

Deliverable : A working implementation of the solution.

Tasks :

- Analysis and conception of the solution
- Implementation of the solution
- Testing of the solution

12.5.5 Complex infrastructure

This last step consists of extending the solution implemented in the previous step for a more complex micro-service topology.

Deliverable : Extension of the solution for complex topology.

Tasks :

- Extension of the topology infrastructure
- Extension of the solution for the new infrastructure
- Testing of the solution

12.6 Planning

This project follows an Agile planning in order to be able to adapt as well as possible to the progress of research. This planning (figure 27 p.72) will allow us to focus on a single problem per sprint.

The decision to use *GitLab* and its *issues* system for planning was made. The flexibility of the platform better corresponds to the needs of Agile planning and will allow teachers and externals to follow the progress of the project live.

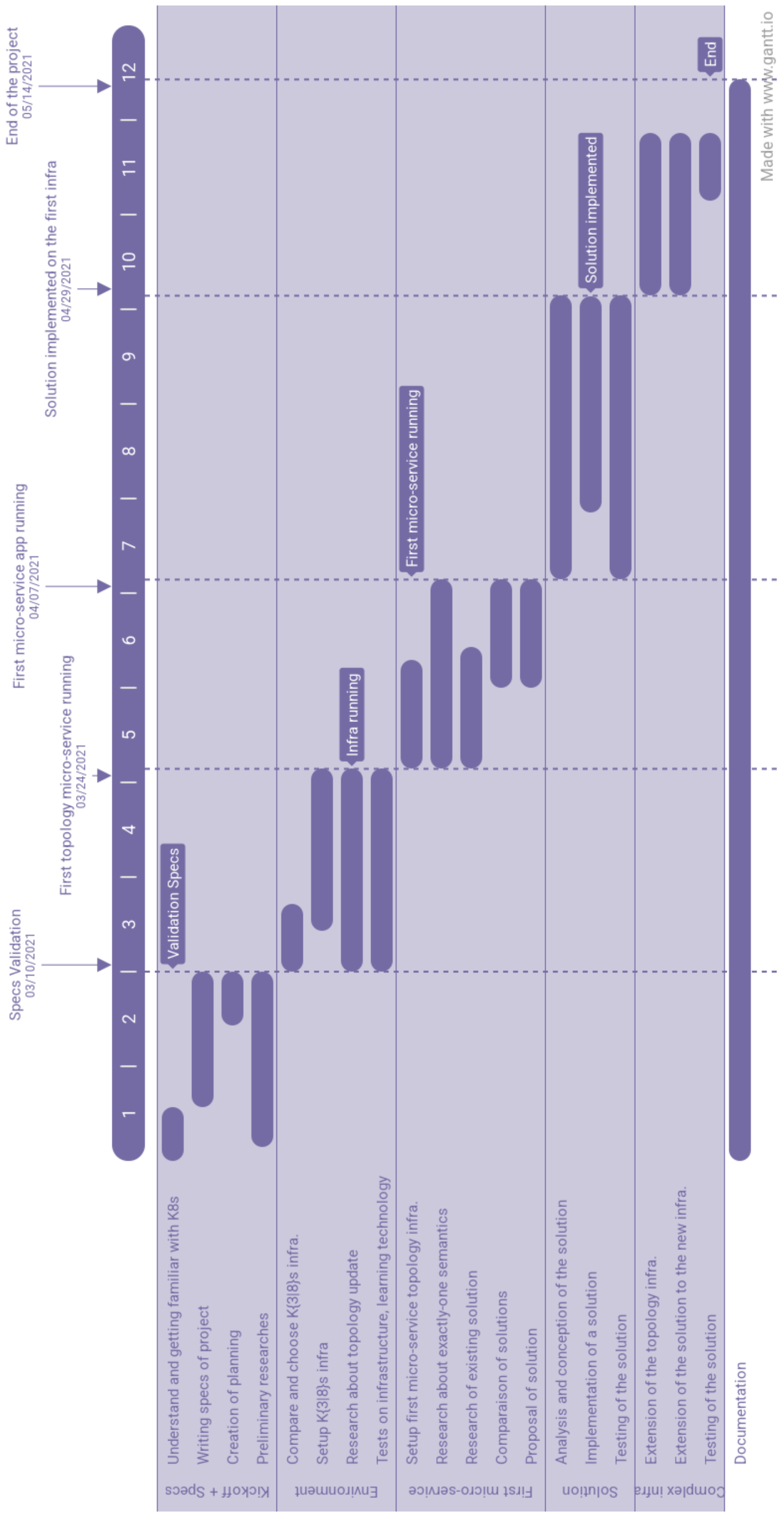


Figure 27: First Agile planning