Università degli Studi di Bologna
L.M. in Artificial Intelligence

JADE Load Balancer

Matteo Milanese

A.A. 2021/2022

Abstract

This project aims at creating an agent-based load balancer by means of JADE and Kubernetes (abbr.: K8s). After an introduction in which the technologies used are presented, it will be described how the developed agents behave and how they interact with each other.
Next, it will be explained how the whole infrastructure has been containerized and deployed to K8s, and how it has been addressed the problem of distributing the work across agents through the use of Prometheus and K8s' Horizontal Pod Autoscaler thus, achieving an effective load balancer.

1 Introduction

In this section it will be briefly presented the two main used technologies, giving an overview of what they are and explaining the main underlying concepts behind them, to make the reader able to fully understand the rest of this paper.

1.1 JADE

JADE is a Java-based software framework that simplifies the implementation of multi-agent systems through a middleware that complies with the FIPA specifications. JADE offers a distributed agent platform, where distributed means that a single (logical) JADE system can be split among different networked hosts. Hence, Jade promotes a peer-to-peer interpretation of a MAS [1].

JADE platform is composed of containers, each of which can hosts one or more agents. Together, they represent the physical network distribution of a platform. Containers are the agents' runtime: the environments without which agents cannot exist. For every platform, there is one main container on which the peripheral ones auto registers upon creation [1].

The main container hosts:

- the *Agent Management System* (AMS) which provides a location-transparent naming service: the *white page service*
- the *Directory Facilitator* (DF) which provides the *yellow page service*: keeps track of all advertised services provided by all the agents in the same Jade platform
- optionally, the *Remote Management Agent* (RMA), a GUI aimed at controlling and/or inspecting the platform

1.2 Kubernetes

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation [11].

A container is an isolated user space instance [5], in which an application is packed along with all its dependencies. Container are immutable: it cannot be changed the code of a container that is already running [2]. They are also the basic building block of a *Pod*.

Pods are the smallest deployable units of computing that can be created and managed in Kubernetes. A Pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context [7].

2 Workers and Dispatchers

Now that the main technologies are defined, we can move to define the agents developed in this projects.

Since the goal is to emulate a load balancer, it has been created two types of agents: ***Workers*** and ***Dispatchers***.

2.1 Workers

Worker agents basically listen for incoming job requests and eventually accepts them, exposing also *Prometheus'* metrics of queued and running jobs to be used in K8s (see sections *Metrics* and *Autoscaling*). They are equipped with a queue where jobs can be scheduled based on their priority (for the purpose of this project, priority of jobs is always the same). They perform the following tasks:

- Listen for incoming job requests
- If the queue is full, signal to dispatcher that can't accept jobs, otherwise respond positively
- Accept jobs and
  - queue them based on their priority
  - run some of them in parallel

To perform an effective scheduling, jobs are first negotiated before being actually accepted, as described in the above bulleted list. This also solves some multi-dispatchers issue in which, if two dispatchers try to gain the only available slot in a queue, conflicts can happen.

2.2 Dispatchers

Dispatchers create and send jobs to *workers*. They perform the following tasks:

- Retrieve worker agents by name: only agent with name starting with "`worker`" (with any casing) are taken into account
- Ask the agents if they can perform job
- If the response is positive, dispatch the job to the first available agent, otherwise repeat the steps trying to re-dispatch the same job

It's worth remarking the fact that these agents negotiate the dispatch of jobs before actually sending them to workers for motivations explained in previous section.

3 Containerization and Deployment

Since JADE is a slightly old framework, a very important task consists in containerize it before deploying it on K8s.

This has been achieved using Gradle and Open JDK prebuilt images and Docker.

To go into details, it has been created a Dockerfile in order to build an image of the project containing all the necessary dependencies, i.e.: JADE, agents and all the jars needed to run the project:

```
# STAGE 1 - Gradle build
FROM gradle:7.3.3-jdk17-alpine AS builder
COPY ./LoadBalancer /root/LoadBalancer
WORKDIR /root/LoadBalancer
RUN gradle runnableJar


# STAGE 2 - Production image
FROM openjdk:17-alpine AS production
COPY --from=builder /root/LoadBalancer/build/libs /LoadBalancer
WORKDIR /LoadBalancer
EXPOSE 1099
ENTRYPOINT ["java"]
CMD ["--version"]
```

It's a two-stage build process in which in the first stage it's used a Gradle image that contains all the build tools needed to produce the final JARs, while the second stage builds the final image that will contain only Java and the built project from the first stage. This results in a smaller, production-ready image.
Please, refer to Docker documentation for more details, in particular to *Getting Started*, *Part 2: Sample application*, section *Build the app's container image*.

The image has then been used as the base for all the containers deployed on K8s.

Since K8s can only download images from a registry, it has been necessary to deploy a local registry in which to push images, and then made K8s pull from it.

Project deployment on K8s is made via so called *Deployments*: they represent a set of multiple, identical Pods with no unique identities. A Deployment runs multiple replicas of an application and automatically replaces any instances that fail or become unresponsive. In this way, Deployments help ensure that one or more instances of the application are available to serve user requests [3].

Another important aspect is Pod communication: JADE platform and main container need to communicate with peripheral containers and vice versa though, the use of K8s' *Services*: they are an abstract way to expose an application running on a set of Pods as a network service [10]. In other words, they make Pods able to connect each other by using a FQDN instead of their IP address which, incidentally, can change, since pods can be created or destroyed based on K8s and user needs.

4 Metrics

So far, the project has been set up and deployed. Now, metrics exposed by agents should be consumed by K8s. To make it aware of those metrics and make it able to read them, we need *Prometheus Adapter* and *Prometheus Operator*.

Let's introduce *Prometheus* first.

4.1 Prometheus

Prometheus is an open-source system monitoring and alerting toolkit originally built at SoundCloud. It is now a standalone open source project and maintained independently of any company. Prometheus collects and stores its metrics as time series data, i.e.: metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels [6].

It offers some instrumentation client libraries that, by writing some code, make the project able to expose custom metrics readable by Prometheus. This approach has been used to expose metrics as described in *Workers* section.

4.2 Prometheus Operator

Prometheus Operator provides Kubernetes native deployment and management of Prometheus and related monitoring components. Its purpose is to simplify and automate the configuration of a Prometheus based monitoring stack for Kubernetes clusters [9].

4.3 Prometheus Adapter

Prometheus Adapter provides a layer between Prometheus Operator and K8s. It contains an implementation of the Kubernetes resource metrics, custom metrics, and external metrics APIs. It gathers the names of available metrics from Prometheus at a regular interval and then only exposes metrics that follow specific forms [8].

4.4 Adding to Kubernetes

To add them both to K8s it has been used *Helm*, a package manager for Kubernetes. It provides a convenient and easy way to install packages and services in a K8s environment, as well as managing their configuration.

That said, configuration part is crucial to make them work together; the most important task is tell *Adapter* where to contact *Operator* and which metrics scrape from it. All these configurations are done declaratively via `yaml` files, as well as all other configurations for K8s.

Below a chunk of configuration of *Prometheus Adapter*:

```yaml
prometheus:
  # Base URL where to contact Prometheus and scrape metrics
  url: http://prom-stack-kube-prometheus-prometheus.default.svc

  # Port of Prometheus
  port: 9090

  # Path of Prometheus
  path: ""
```

```
# ...

rules:
  default: true
  custom:

  # Select metrics
  - seriesQuery: 'queued_jobs_total{kubernetes_namespace!="",kubernetes_pod_name!=""}'
    resources:
      overrides:
        # Set what kubernetes_namespace and kubernetes_pod_name targets
        kubernetes_namespace: {resource: "namespace"}
        kubernetes_pod_name: {resource: "pod"}

    # Rename metric to indicate what query (below) exposes
    name:
      matches: "^(.*)_total"
      as: "${1}_per_second"

    # Make a query exporting the original metric as queued_job_per_second (i.e.:
    # query aggregates data to have some meaningful information)
    metricsQuery: 'sum(rate(<<.Series>>{<<.LabelMatchers>>}[2m])) by (<<.GroupBy>>)'
```

5 Autoscaling

Now that project has been deployed and metrics are set, we can achieve an effective autoscaling through the use of *Horizontal Pod Autoscaler*.

Horizontal Pod Autoscaler automatically updates a workload resource (such as a *Deployment* or *StatefulSet*), with the aim of automatically scaling the workload to match demand [4].

It reads metrics exposed as described in the previous sections and replicates pods when needed, as specified in its configuration.

The motivation for making Pods scaling, and in particular, Pods hosting containers running *Worker* agents, is the following: the basic task of a load balancer is scale based on load demand thus, if the demand increase, the instances of the application increase, and viceversa.

Applying this rationale to the current project, it has been made the Pods hosting Workers scale based on the items per seconds in their queue. That is, a Pod hosting a *Worker* scales when the agent's queue is full for a long period of time, reducing in this way the "pressure" on it.

Here the configuration used in this project:

```
kind: HorizontalPodAutoscaler
apiVersion: autoscaling/v2beta1
metadata:
  name: worker-autoscaler
spec:
  scaleTargetRef:
    # point the HPA at the worker agents' deployments
    apiVersion: apps/v1
    kind: Deployment
    name: worker
  # autoscale between 1 and 10 replicas
  minReplicas: 1
  maxReplicas: 10
  metrics:
  # use a "Pods" metric, which takes the average of the
```

```
  # given metric across all pods controlled by the autoscaling target
  - type: Pods
    pods:
      metricName: queued_jobs
      # see https://kubernetes.io/docs/tasks/run-application/
      # horizontal-pod-autoscale-walkthrough/#quantities
      targetAverageValue: 10m
```

6 Conclusions

The project effectively demonstrates how an agent-based load balancer can be created and deployed, and how Kubernetes can be exploited to have an effective scaling and a convenient agent replication.

Moreover, it demonstrates how Kubernetes can be a great tool as the base infrastructure for multi-agent systems, since its architecture and its capabilities can support an agent-based framework such as JADE.

References

[1] R. Calegari and A. Omicini, *Agents in jade*, Multi-Agent Systems Course, Master in Artificial Intelligence, 2021.

[2] "Containers." (Feb. 21, 2022), [Online]. Available: https://kubernetes.io/docs/concepts/containers/.

[3] "Deployment." (Feb. 21, 2022), [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/deployment.

[4] "Horizontal pod autoscaling." (Feb. 21, 2022), [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale.

[5] "Os-level virtualization." (Feb. 21, 2022), [Online]. Available: https://en.wikipedia.org/wiki/OS-level_virtualization.

[6] "Overview." Prometheus. (Feb. 21, 2022), [Online]. Available: https://prometheus.io/docs/introduction/overview/.

[7] "Pods." (Feb. 21, 2022), [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/.

[8] "Prometheus adapter." (Feb. 21, 2022), [Online]. Available: https://github.com/kubernetes-sigs/prometheus-adapter.

[9] "Prometheus operator." Overview. (Feb. 21, 2022), [Online]. Available: https://github.com/prometheus-operator/prometheus-operator.

[10] "Service." (Feb. 21, 2022), [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/.

[11] "What is kubernetes?" (Feb. 21, 2022), [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.