POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Thesis

Dynamic Sharing of Computing and Network Resources between Different Clusters

Supervisor
prof. Fulvio Risso

Candidate
Francesco Borgogni

Company tutor
TOP-IX
Leonardo Camiciotti

Academic year 2019-2020

Acknowledgements

This is the end of five intense but beautiful years, in which I have learnt and received so much, technically speaking but especially on a personal level. In these occasions it is important to stop for a moment and reflect upon our lives, and thank for everything we have received.

"Nothing in this world that's worth having comes easy."

I thank my supervisor Prof. Fulvio Risso, who has always been kind and available during these months of thesis work and has always suggested great advice and ideas to improve. A big thank you also to Alex Palesandro, who has been an invaluable component of the project we are developing and an excellent software team leader, and who guided and assisted me in facing the initial impact with the huge world of Kubernetes. I also thank all the people in TOP-IX who have been extremely courteous and available, especially Leonardo Camiciotti for his patience and his brilliant ideas and suggestions.

Then, I want to express my deepest gratitude to my friends, my parish group and my family: I would not be the person I am without them, their teachings, encouragements and also their corrections. Obviously they could not help me with the technical aspects of this thesis, but without them I would never have made it, and I hope to be able to show every day my love for all of them. A special thank goes to Emanuele and Davide, true brothers in this journey, and to Matt Fradd too.

"Freedom from evil is not a destination we reach: it's a daily choice we make."

Contents

List of Figures 7

1 Introduction 11

1.1 Goal of the thesis 11

2 Background 13

2.1 Cluster federation 13

2.2 Kubefed 14

2.2.1 Kubefed limitations 16

3 Technologies 18

3.1 Kubernetes: a bit of history 18

3.2 Applications deployment evolution 19

3.3 Container orchestrators 20

3.4 Kubernetes architecture 22

3.4.1 Control plane components 22

3.4.2 Node components 24

3.5 Kubernetes objects 25

3.5.1 Label & Selector 27

3.5.2 Namespace 27

3.5.3 Pod 27

3.5.4 ReplicaSet 27

3.5.5 Deployment 28

3.5.6 Service 29

3.6 Virtual-Kubelet 30

3.7 Kubebuilder 31

4 Protocol 33
4.1 Point to point connections 33
4.1.1 Ad-hoc request 34
4.1.2 Operating and release phases 35
4.2 Messages 36
4.2.1 Advertisement 36
4.2.2 Request (user $\rightarrow$ cluster) 37
4.2.3 Request (cluster $\rightarrow$ cluster) 38
4.2.4 Acknowledge 39
4.2.5 Quote 39
4.2.6 Offer 39
4.3 Broker 40

5 Software architecture 42
5.1 Global architecture 42
5.2 Broadcaster 44
5.3 Controller 45
5.4 Policies 46

6 Implementation 48
6.1 Advertisement operator 48
6.1.1 Advertisement API definition 49
6.1.2 Broadcaster 52
6.1.3 Controller 54
6.2 Kubernetes provider 55
6.2.1 Pod lifecycle methods 57
6.2.2 Pod translation methods 58

7 Results 61
7.1 Test environment 61
7.2 Functional tests 62
7.3 Performance and scalability tests 64
7.3.1 Test results 65
7.3.2 Test limitations 67

8 Conclusions and future works 68

Bibliography 70

List of Figures

2.1 KubeFed architecture overview. 14

2.2 KubeFed main concepts [9]. 16

3.1 Evolution in applications deployment [10]. 19

3.2 Container orchestrators use [3]. 21

3.3 Kubernetes architecture [10]. 22

3.4 Kubernetes master and worker nodes [10]. 26

3.5 Kubernetes pods [10]. 28

3.6 Kubernetes Services [10]. 29

3.7 Virtual-Kubelet concept [22]. 31

4.1 *Advertisement* messages exchange. 34

4.2 *Request* submitted by a user and sent to another cluster. 34

4.3 Request served with an auction. 35

4.4 *Advertisement* messages exchange with broker. 41

4.5 *Request* submitted by a user and sent to another cluster through the broker. 41

5.1 Software components. 42

5.2 Architecture. 43

5.3 Broadcaster differentiation example. 44

5.4 Controller reconciliation from Advertisement to Virtual Node. . . . 45

5.5 OPA policy decoupling [13]. 46

6.1 Implementation architecture. 49

7.1 Observed times with 10 Advertisements. 66
7.2 Observed time trends with 100 Advertisements. 66

Listings

2.1 FederatedDeployment example [9] 15
3.1 Basic example of Kubernetes Deployment [10]. 28
3.2 Basic example of Kubernetes Service [10]. 30
4.1 *Advertisement* message. 36
4.2 *Request* message. 38
4.3 *Request* message sent by *home cluster* to others. 38
4.4 *Acknowledge* message. 39
4.5 *Quote* message. 39
4.6 *Offer* message. 39
6.1 Advertisement struct definition. 50
6.2 AdvertisementSpec struct definition. 51
6.3 AdvertisementStatus struct definition. 51
6.4 StartBroadcaster function. 52
6.5 GenerateAdvertisement function. 52
6.6 Reconcile method. 54
6.7 Virtual-Kubelet provider interface [22]. 55
6.8 KubernetesProvider struct definition. 56
6.9 KubernetesConfig struct definition. 56
6.10 CreatePod method. 57
6.11 GetPod method. 58
6.12 Home-to-Foreign function. 58
6.13 Foreign-to-Home function. 60
7.1 Virtual-Kubelet Deployment. 62

7.2 Advertisement operator Deployment. 63
7.3 Bash script to configure the cluster and create Advertisements. . . . 64
7.4 Sample Advertisement used in the tests. 65

Chapter 1

Introduction

In the last several years, ICT world has seen an incredible innovation with the introduction of virtualization first, then with containerization and finally with orchestrators. In this last field, one of the main actors is Kubernetes, an open source system for managing containerized applications in a clustered environment. The spread of Kubernetes is rapidly increasing; in cloud providers such as Google Cloud Platform and Microsoft Azure it is the most popular choice [1] and many companies and organizations have started to set up their own clusters in order to migrate their applications on it. With the advent of 5G and edge computing also telecommunications companies are moving towards a Kubernetes solution [4].

In a similar scenario, if we could share resources between clusters this would open many use cases:

- different users with their small clusters (for example Minikube [12]) can partially or totally offload their applications to others;
- different companies could interconnect and get payed for hosting others' applications;
- in an IoT scenario, edge nodes (which typically have limited resources) can send requests to more powerful ones;
- in an edge computing scenario, an application can be scheduled on the best cluster in order to reduce latency.

1.1 Goal of the thesis

Kubernetes does not support natively this sharing of resources among clusters: a concept of "Federation" has been defined and is being developed, but the project is relatively new and not very mature (it is still alpha [9]).

This work, carried out by the Computer Networks Group at Politecnico di Torino and developed with the collaboration of TOP-IX [18], proposes an alternative solution to this problem, focusing on a protocol that allows different clusters to know each other, exchange their information and send their jobs. The aim is to provide to a user who can use Kubernetes the possibility to exploit the proposed solution with no extra effort, interacting with the same interface and working with the same resources, but taking advantage of many other clusters on which he can deploy his applications.

The discussion is structured as follows:

- **Chapter 2** introduces some background aspects about the problem of cluster federation, the solution proposed by Kubernetes and why we propose a different approach.
- **Chapter 3** provides an extensive presentation of Kubernetes, its architecture and concepts, and an introduction to the tools used to develop the solution.
- **Chapter 4** describes the designed protocol, the messages exchanged and some possible scenarios.
- **Chapter 5** illustrates the architecture of the software that has been developed.
- **Chapter 6** shows the implementation of the designed solution in Kubernetes.
- **Chapter 7** explains how the implementation has been tested and reports some qualitative results.
- **Chapter 8** closes the thesis with final conclusions and possible future works.

Chapter 2

Background

This chapter provides an introduction to cluster federation, explaining what "federating some clusters" means and why it is relevant. An overview of Kubernetes proposal for managing this issue (Kubefed) is illustrated and analysed, showing some limitations that give a reason for the work done. If you are interested in deepening Kubernetes Federation, please refer to the official documentation, from which this chapter takes inspiration [9].

2.1 Cluster federation

Many organizations today deploy their applications on large clusters across hybrid (private and public) clouds. Cloud providers give to the user the feeling to dispose of infinite resources, but they need to predict the user requirements in order to provide services with high availability at minimal costs. In order to accomplish this goal, cloud providers can cooperate together to bring new business opportunities, such as expanding available resources, achieving cost-effective asset optimization and adopting power saving policies. They require simplified solutions to create and manage a flexible aggregation of all the clusters. Such aggregation is called **cloud federation** [2, 23].

Cloud federation requires one provider to wholesale or rent computing resources to another cloud provider. Those resources become a temporary or permanent extension of the buyer's cloud computing environment, depending on the specific federation agreement between providers [19]. This allows different cloud providers the opportunity to work collaboratively, offering better services to customers and at the same time increasing their productivity. Customers can advantage from cloud federation for a larger offer of available services, the capability of price comparison and the removal of vendor lock-in [23].

2.2 Kubefed

Kubernetes Cluster Federation (KubeFed) allows to coordinate the configuration of multiple Kubernetes clusters from a single set of APIs in a hosting cluster. Its aim is to provide mechanisms for expressing which clusters should have their configuration managed and what that configuration should be [9].

KubeFed extends Kubernetes APIs for Federated Resources through the CRD mechanism. It manages CRDs and implements functions such as synchronizing resources and cross-cluster scheduling. In KubeFed there are two types of cluster:

- **Host cluster**: a cluster which is used to expose the KubeFed API and run the KubeFed control plane.
- **Member cluster**: a cluster which is registered with the KubeFed API and that KubeFed controllers have authentication credentials for. The Host Cluster can also be a Member Cluster.

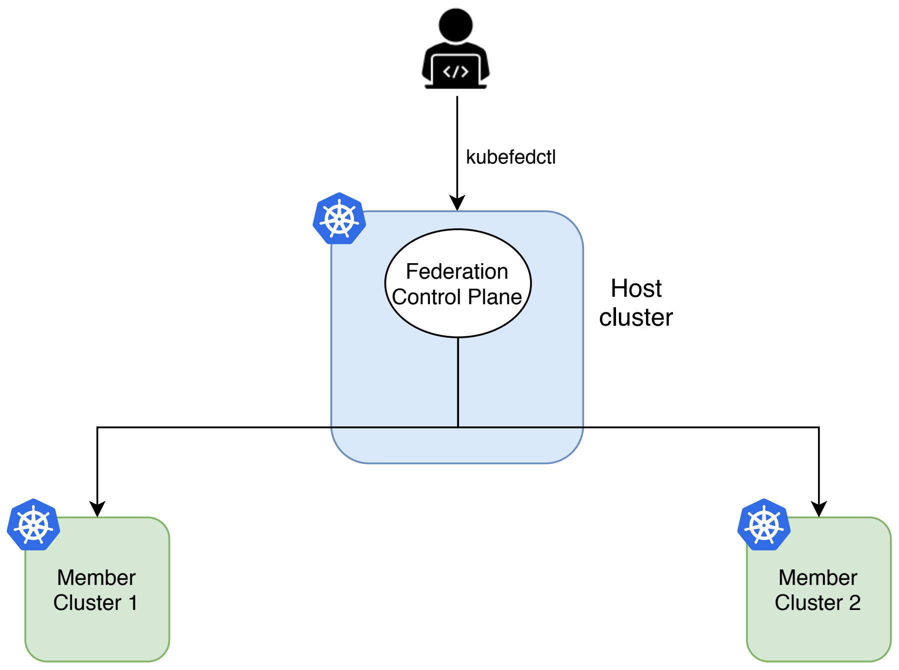

Figure 2.1. KubeFed architecture overview.

KubeFed is configured with two types of information:

- **Type configuration** declares which API types KubeFed should handle. It consists of three main concepts:

  - **Template**: defines the object representation, common to all federated clusters;
  - **Placement**: defines which clusters will be allowed to see the resource;
  - **Override**: defines variations to the template, which can be different on each cluster.

- **Cluster configuration** declares which clusters KubeFed should target. You can add/delete clusters with `kubefedctl join/unjoin`.

Propagation is the process that distributes resources to federated clusters.

```
apiVersion: types.kubefed.io/v1beta1
kind: FederatedDeployment
metadata:
  name: test-deployment
  namespace: test-namespace
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      replicas: 3
      selector:
        matchLabels:
          app: nginx
      template:
        metadata:
          labels:
            app: nginx
        spec:
          containers:
          - image: nginx
            name: nginx
  placement:
    clusters:
    - name: cluster2
    - name: cluster1
  overrides:
  - clusterName: cluster2
    clusterOverrides:
    - path: "/spec/replicas"
      value: 5
    - path: "/metadata/annotations/foo"
      op: "remove"
```

Listing 2.1. FederatedDeployment example [9]

With these three other concepts we have all the building blocks that can be used by higher-level APIs:

- **Status** collects the status of resources distributed by KubeFed across all federated clusters.
- **Policy** determines which subset of clusters a resource is allowed to be distributed to
- **Scheduling** refers to a decision-making capability that can decide how workloads should be spread across different clusters similar to how a human operator would

All these concepts are summarized in the scheme below.

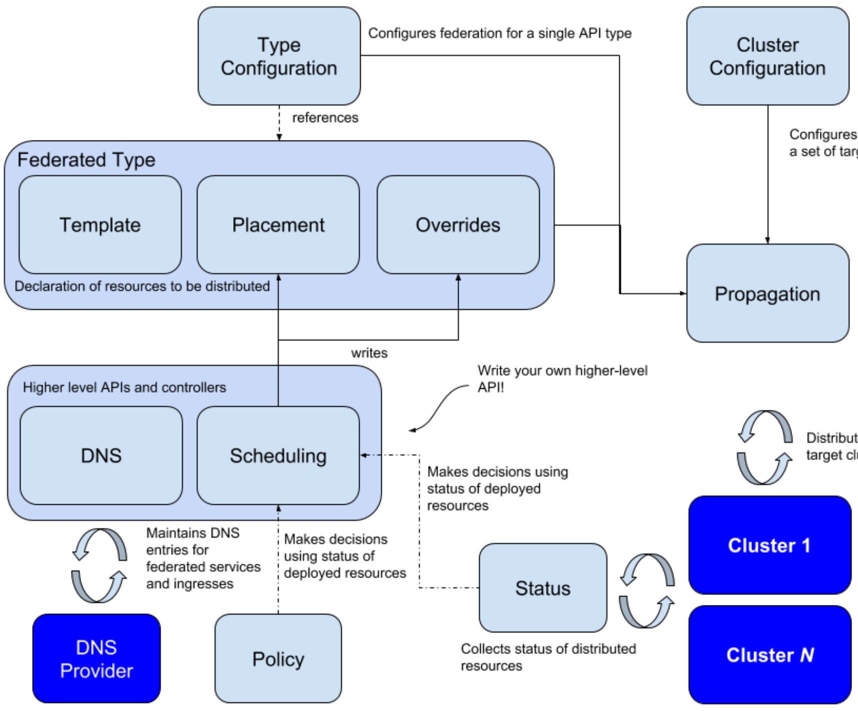
Type Configuration
Configures federation for a single API type
Cluster Configuration
references
Configures federation with a set of target clusters
Federated Type
Template
Placement
Overrides
Declaration of resources to be distributed
Propagation
writes
Higher level APIs and controllers
Write your own higher-level API!
DNS
Scheduling
Distributes resources to target clusters
Makes decisions using status of deployed resources
Cluster 1
Maintains DNS entries for federated services and ingresses
Makes decisions using status of deployed resources
Status
DNS Provider
Policy
Collects status of distributed resources
Cluster N

Figure 2.2. KubeFed main concepts [9].

2.2.1 Kubefed limitations

KubeFed is a centralized solution, where all the decisions are taken by the elected 'master' (i.e. the *Host cluster*). This is clearly impossible between clusters belonging to different organizations, because no one wants to leave the control to another entity. Another important limitation is that KubeFed is not transparent: to interact with the federation a specific interface is needed (`kubefedctl` CLI) and the API resources must explicitly specify that they can be executed on other clusters (for example the `FederatedDeployment` showed in listing 2.1).

These are just the main reasons for which we decided to implement a different solution, which could be

- **transparent**: a user who can use Kubernetes should be able to use the application with no extra knowledge required, using the same interfaces (e.g. `kubectl`) and the same API resources (e.g. `Deployment`); moreover, the offloaded resources should be seen as if they were on the user's cluster, so that he can use them without changing his interaction with Kubernetes;
- **autonomous**: the user can set some policies, but even without them the system should work as well, deciding if and where to offload a job; the user could even work in the cluster without knowing that his applications are executed somewhere else;
- **decentralized**: if a cluster is offloading its applications, it should have the complete control over them and it should be the one that takes scheduling decisions, not a centralized entity.

Chapter 3

Technologies

So far we have often named **Kubernetes** and described one of the tools it offers, Kubefed. In this chapter we will analyse Kubernetes architecture, showing also its history and evolution though time, in order to lay the foundations for all the work which will be exposed later on. Kubernetes (often shortened as K8s) is a huge framework and a deep examination of it would require much more time and discussion, so we will try to present its main concepts and components; if you are interested in deepening, please refer to the official documentation, from which this chapter takes inspiration [10].

The chapter continues with an introduction to other technologies and tools used to develop the solution, in particular **Virtual-Kubelet** [22], a project which allows to create virtual nodes with a particular behaviour, and **Kubebuilder** [7], a tool to build custom resources.

3.1 Kubernetes: a bit of history

Around 2004, Google created the **Borg** system, a small project, with less than 5 people initially working on it. The project was developed as a collaboration with a new version of Google's search engine. Borg was a large-scale internal cluster management system, which "ran hundreds of thousands of jobs, from many thousands of different applications, across many clusters, each with up to tens of thousands of machines" [21].

In 2013 Google announced **Omega**, a flexible and scalable scheduler for large compute clusters. Omega provided a "parallel scheduler architecture built around shared state, using lock-free optimistic concurrency control, in order to achieve both implementation extensibility and performance scalability" [16].

In the middle of 2014, Google presented **Kubernetes** as on open-source version of Borg. Kubernetes was created by Joe Beda, Brendan Burns, and Craig McLuckie, and other engineers at Google. Its development and design were heavily influenced by Borg and many of its initial contributors previously used to work on it. The original Borg project was written in C++, whereas for Kubernetes the Go language was chosen.

In 2015 Kubernetes v1.0 was released. Along with the release, Google set up a partnership with the Linux Foundation to form the **Cloud Native Computing Foundation** (CNCF) [5]. Since then Kubernetes has incredibly grown, achieving the CNCF graduated status and being adopted by nearly every big company; nowadays it has become the de-facto standard for container orchestration [20, 15].

3.2 Applications deployment evolution

Kubernetes is a portable, extensible, open-source platform for running and coordinating containerized applications across a cluster of machines. It is designed to completely manage the life cycle of applications and services using methods that provide consistency, scalability, and high availability.

What does "containerized applications" means? In the last decades, the deployment of applications has seen significant changes, which are illustrated in the picture below.

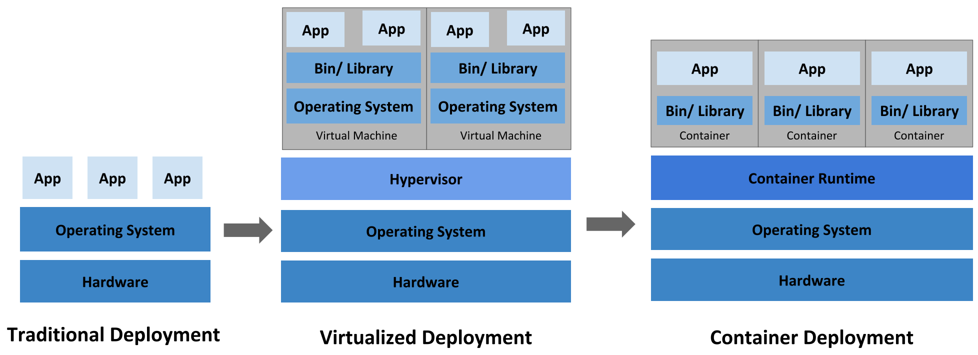

Figure 3.1. Evolution in applications deployment [10].

Traditionally organizations used to run their applications on physical servers. One of the problems of this approach was that resource boundaries between applications could not be applied in a physical server, leading to resource allocation issues. For example, if multiple applications run on a physical server, one of them could take up most of the resources, and as a result, the other applications would starve.

A possibility to solve this problem would be to run each application on a different physical server, but clearly it is not feasible: the solution could not scale, would lead to resources under-utilization and would be very expensive for organizations to maintain many physical servers.

The first real solution has been **virtualization**. Virtualization allows to run multiple Virtual Machines on a single physical server. It grants isolation of the applications between VMs providing a high level of security, as the information of one application cannot be freely accessed by another application. Virtualization enables better utilization of resources in a physical server, improves scalability, because an application can be added or updated very easily, reduces hardware costs, and much more. With virtualization you can group together a set of physical resources and expose it as a cluster of disposable virtual machines. Isolation certainly brings many advantages, but it requires a quite 'heavy' overhead: each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware.

A second solution which has arrived recently is **containerization**. Containers are similar to VMs, but they share the operating system among the applications, relaxing isolation properties. Therefore, containers are considered a lightweight form of virtualization. Similarly to a VM, a container has its own filesystem, CPU, memory, process space etc... One of the key features of containers is that they are portable: as they are decoupled from the underlying infrastructure, they are totally portable across clouds and OS distributions. This property is particularly relevant nowadays, with cloud computing: a container can be moved without any problem across different machines. Moreover, being "lightweight", containers are much faster than virtual machines: they can be booted, started, run and stopped with very a little effort and in a short time.

3.3 Container orchestrators

When hundreds or thousands of containers are created, the need of a way to manage them becomes essential; container orchestrators serve this purpose. A container orchestrator is a system designed to easily manage complex containerization deployments across multiple machines from one central location. This management includes the containers themselves, the hosts, networking, storage etc... The most famous orchestrators are Kubernetes, Docker Swarm and Apache Mesos; the one which we will focus on and describe in the following is Kubernetes.

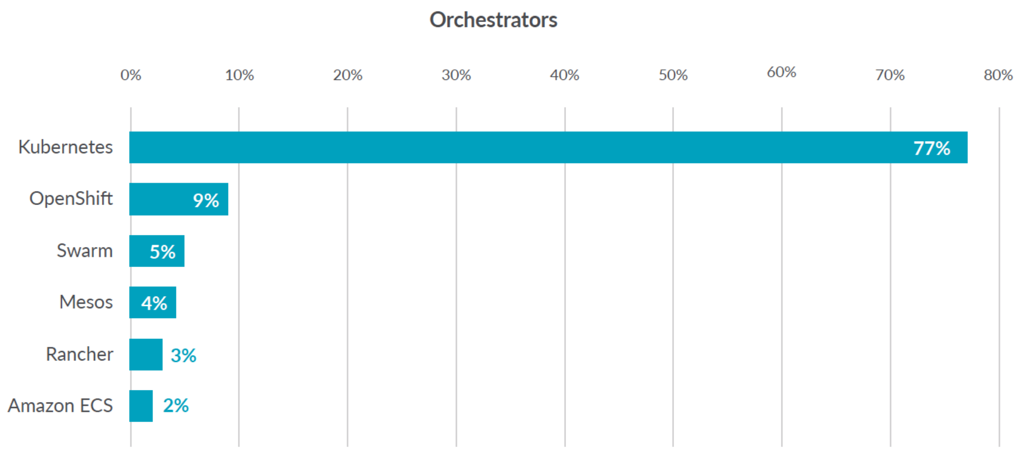

Figure 3.2. Container orchestrators use [3].

Kubernetes provides you with [10]:

- **Service discovery and load balancing** Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration** Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
- **Automatic bin packing** You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- **Self-healing** Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- **Secret and configuration management** Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH

keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration

After having illustrated the main high-level features of Kubernetes, we can finally analyse its components and architecture.

3.4 Kubernetes architecture

When you deploy Kubernetes, you have a cluster. A Kubernetes cluster consists of a set of machines, called **nodes**, that run containerized applications. At least one of the nodes hosts the control plane and is called **master**: it manages the cluster and expose an interface to the user. The **worker** node(s) host the **pods** that are the components of the application. The master manages the worker nodes and the pods in the cluster. In production environments, the control plane usually runs across multiple machines and a cluster runs multiple nodes, providing fault-tolerance and high availability.

Here's the diagram of a Kubernetes cluster with all the components linked together.

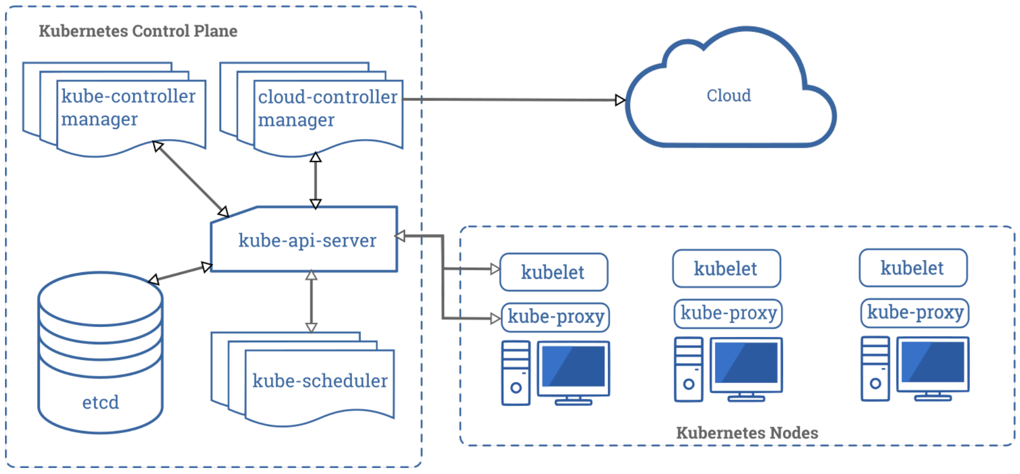

Figure 3.3. Kubernetes architecture [10].

3.4.1 Control plane components

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod). They can be run on any machine in the cluster. However, for simplicity, they are typically executed all together on the same machine, which do not run user containers.

API server

The API server is the component of the Kubernetes control plane that exposes the Kubernetes REST API: it is the front end for the Kubernetes control plane. Its function is to intercept REST request, validate and process them. The main implementation of a Kubernetes API server is `kube-apiserver`. It is designed to scale horizontally, which means it scales by deploying more instances. Moreover, it can be easily redounded to run several instances of it and balance traffic among them.

etcd

`etcd` is a distributed, consistent and highly-available key value store used as Kubernetes' backing store for all cluster data. It is based on the Raft consensus algorithm [17], which allows different machines to work as a coherent group and survive to the breakdown of one of its members. `etcd` can be stacked in the master node or external, installed on dedicated host. Only the API server can communicate with it.

Scheduler

Control plane component that is responsible for instantiating the pods. The one provided by Kubernetes is called `kube-scheduler`, but it can be customized by adding new schedulers and indicating in the pods to use them. `kube-scheduler` watches for newly created pods not assigned to a node yet, and selects one for them to run on. To make its decisions, it considers singular and collective resource requirements, hardware / software / policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.

kube-controller-manager

Component that runs controller processes. It continuously compares the desired state of the cluster (given by the objects specifications) with the current one (read from `etcd`). Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process. These controllers include:

- Node Controller: responsible for noticing and reacting when nodes go down.
- Replication Controller: in charge of maintaining the correct number of pods for every replica object in the system.
- Endpoints Controller: populates the Endpoint objects (which links Services and Pods).
- Service Account & Token Controllers: create default accounts and API access tokens for new namespaces.

cloud-controller-manager

This component runs controllers that interact with the underlying cloud providers. The `cloud-controller-manager` binary is a beta feature introduced in Kubernetes 1.6. It only runs cloud-provider-specific controller loops. You can disable these controller loops in the `kube-controller-manager`.

`cloud-controller-manager` allows the cloud vendor's code and the Kubernetes code to evolve independently of each other. In prior releases, the core Kubernetes code was dependent upon cloud-provider-specific code for functionality. In future releases, code specific to cloud vendors should be maintained by the cloud vendor themselves, and linked to `cloud-controller-manager` while running Kubernetes. Some examples of controllers with cloud provider dependencies are:

- Node Controller: checks the cloud provider to update or delete Kubernetes nodes using cloud APIs.
- Route Controller: responsible for setting up network routes in the cloud infrastructure.
- Service Controller: for creating, updating and deleting cloud provider load balancers.
- Volume Controller: creates, attaches, and mounts volumes, interacting with the cloud provider to orchestrate them.

3.4.2 Node components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

Container Runtime

The `container runtime` is the software that is responsible for running containers. Kubernetes supports several container runtimes: Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface).

kubelet

An agent that runs on each node in the cluster, making sure that containers are running in a pod. The `kubelet` receives from the API server the specifications of the Pods and interacts with the `container runtime` to run them, monitoring their state and assuring that the containers are running and healthy. The connection with the `container runtime` is established through the Container Runtime Interface and is based on gRPC.

kube-proxy

`kube-proxy` is a network agent that runs on each node in your cluster, implementing part of the Kubernetes Service concept. It maintains network rules on nodes, which allow network communication to your Pods from inside or outside of the cluster. If the operating system is providing a packet filtering layer, `kube-proxy` uses it, otherwise it forwards the traffic itself.

Addons

Features and functionalities not yet available natively in Kubernetes but implemented by third parties pods. Some examples are DNS, dashboard (a web gui), monitoring and logging.

3.5 Kubernetes objects

Kubernetes defines several types of objects, which constitutes its building blocks. Usually, a K8s resource object contains the following fields [8]:

- `apiVersion`: the versioned schema of this representation of the object;
- `kind`: a string value representing the REST resource this object represents;
- `ObjectMeta`: metadata about the object, such as its name, annotations, labels etc...

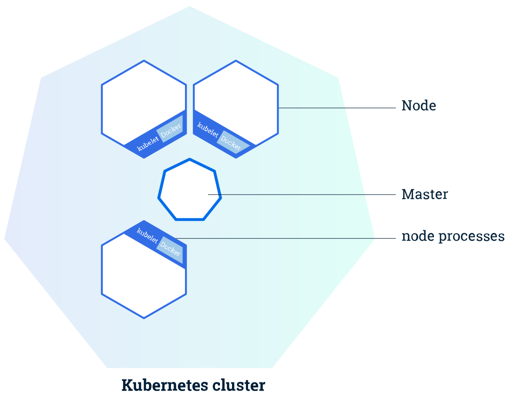

Figure 3.4. Kubernetes master and worker nodes [10].

- `ResourceSpec`: defined by the user, it describes the desired state of the object;
- `ResourceStatus`: filled in by the server, it reports the current state of the resource.

The allowed operations on these resources are the typical CRUD actions:

- **Create**: create the resource in the storage backend; after a resource is create the system will apply the desired state.
- **Read**: comes with 3 forms
  - **Get**: retrieve a specific resource object by name;
  - **List**: retrieve all resource objects of a specific type within a namespace, and the results can be restricted to resources matching a selector query;
  - **Watch**: stream results for an object(s) as it is updated.
- **Update**: comes with 2 forms
  - **Replace**: replace the existing spec with the provided one;
  - **Patch**: apply a change to a specific field.
- **Delete**: delete a resource; depending on the specific resource, child objects may or may not be garbage collected by the server.

In the following we will illustrate the main objects we will need in the next chapters.

3.5.1 Label & Selector

Labels are key-value pairs attached to a K8s object and used to organize and mark a subset of objects. Selectors are the grouping primitives which allow to select a set of objects with the same label.

3.5.2 Namespace

Namespaces are virtual partitions of the cluster. By default, Kubernetes creates 4 Namespaces:

- **kube-system**: it contains objects created by K8s system, mainly control-plane agents;
- **default**: it contains objects and resources created by users and it is the one used by default;
- **kube-public**: readable by everyone (even not authenticated users), it is used for special purposes like exposing cluster public information;
- **kube-node-lease**: it maintains objects for heartbeat data from nodes.

It is a good practice to split the cluster into many Namespaces in order to better virtualize the cluster.

3.5.3 Pod

Pods are the basic processing units in Kubernetes. A pod is a logic collection of one or more containers which share the same network and storage, and are scheduled together on the same pod. Pods are ephemeral and have no auto-repair capacities: for this reason they are usually managed by a controller which handles replication, fault-tolerance, self-healing etc...

3.5.4 ReplicaSet

ReplicaSets control a set of pods allowing to scale the number of pods currently in execution. If a pod in the set is deleted, the ReplicaSet notices that the current number of replicas (read from the `Status`) is different from the desired one (specified in the `Spec`) and creates a new pod. Usually ReplicaSets are not used directly: a higher-level concept is provided by Kubernetes, called **Deployment**.

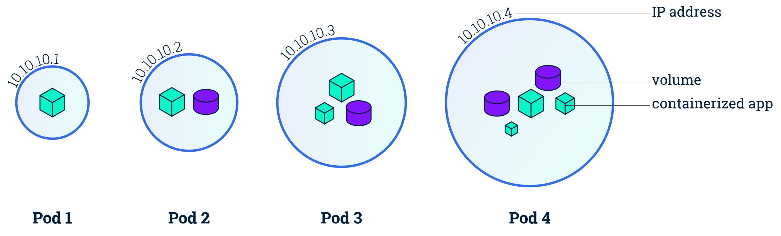

Figure 3.5. Kubernetes pods [10].

3.5.5 Deployment

Deployments manage the creation, update and deletion of pods. A Deployment automatically creates a ReplicaSet, which then creates the desired number of pods. For this reason an application is typically executed within a Deployment and not in a single pod.

This is an example of a Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Listing 3.1. Basic example of Kubernetes Deployment [10].

The code above allows to create a Deployment with name `nginx-deployment` and a label `app`, with value `nginx`. It creates three replicated pods and, as defined in the `selector` field, manages all the pods labelled as `app:nginx`. The template field shows the information of the created pods: they are labelled `app:nginx` and launch one container which runs the nginx DockerHub image at version 1.7.9 on port 80.

3.5.6 Service

A Service is an abstract way to expose an application running on a set of Pods as a network service. It can have different access scopes depending on its ServiceType:

- **ClusterIP**: Service accessible only from within the cluster, it is the default type;
- **NodePort**: exposes the Service on a static port of each Node's IP; the NodePort Service can be accessed, from outside the cluster, by requesting `<NodeIP>:<NodePort>`;
- **LoadBalancer**: exposes the Service externally using a cloud provider's load balancer;
- **ExternalName**: maps the Service to an external service so that internal apps can access it.

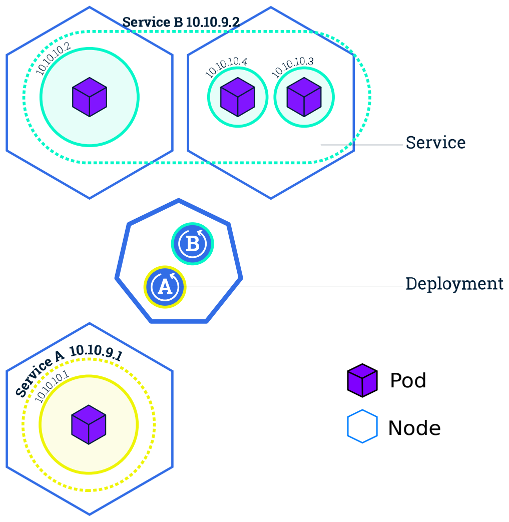

Figure 3.6. Kubernetes Services [10].

The following Service is named `my-service` and redirects requests coming from TCP port 80 to port 9376 of any Pod with the `app=MyApp` label.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: myApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Listing 3.2. Basic example of Kubernetes Service [10].

3.6 Virtual-Kubelet

Two Kubernetes-based tools which have been used during the developing work are Virtual-Kubelet and Kubebuilder. Virtual Kubelet is an open source Kubernetes kubelet implementation that masquerades a cluster as a kubelet for the purposes of connecting Kubernetes to other APIs [22]. Virtual Kubelet is a Cloud Native Computing Foundation sandbox project.

The project offers a provider interface developers need to implement in order to use it. The official documentation says that “providers must provide the following functionality to be considered a supported integration with Virtual Kubelet

1. Provides the back-end plumbing necessary to support the lifecycle management of pods, containers and supporting resources in the context of Kubernetes.
2. Conforms to the current API provided by Virtual Kubelet.
3. Does not have access to the Kubernetes API Server and has a well-defined callback mechanism for getting data like secrets or configmaps.” [22]

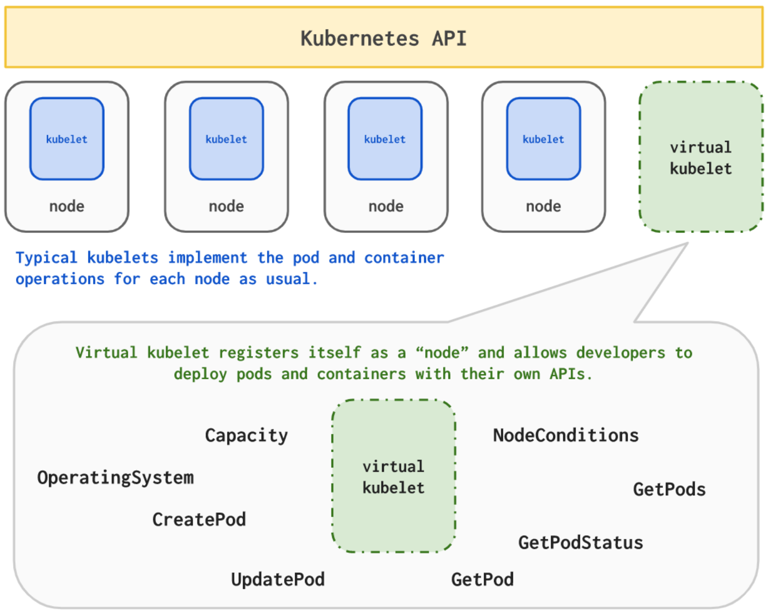
Kubernetes API
kubelet
node
kubelet
node
kubelet
node
kubelet
node
virtual kubelet
Typical kubelets implement the pod and container operations for each node as usual.
Virtual kubelet registers itself as a "node" and allows developers to deploy pods and containers with their own APIs.
Capacity
OperatingSystem
CreatePod
UpdatePod
virtual kubelet
NodeConditions
GetPods
GetPodStatus
GetPod

Figure 3.7. Virtual-Kubelet concept [22].

3.7 Kubebuilder

Kubebuilder is a framework for building Kubernetes APIs using Custom Resource Definitions (CRDs) [7].

CustomResourceDefinition is an API resource offered by Kubernetes which allows to define Custom Resources (CRs) with a name and schema specified by the user. When you create a new CustomResourceDefinition, the Kubernetes API server creates a new RESTful resource path; the CRD can be either namespaced or cluster-scoped. The name of a CRD object must be a valid DNS subdomain name.

A **Custom Resource** is an endpoint in the Kubernetes API that is not available in a default Kubernetes installation and which frees users from writing their own API server to handle them [10]. On their own, custom resources simply let you store and retrieve structured data. In order to have a more powerful management, you also need to provide a custom controller which executes a control loop over the custom resource it watches: this behaviour is called Operator pattern [11].

Kubebuilder helps a developer in defining his Custom Resource, taking automatically basic decisions and writing a lot of boilerplate code. These are the main actions operated by Kubebuilder:

1. Create a new project directory.

2. Create one or more resource APIs as CRDs and then add fields to the resources.
3. Implement reconcile loops in controllers and watch additional resources.
4. Test by running against a cluster (self-installs CRDs and starts controllers automatically).
5. Update bootstrapped integration tests to test new fields and business logic.
6. Build and publish a container from the provided Dockerfile. [7]

Chapter 4

Protocol

The goal of the protocol is to allow communication between different *Kubernetes* clusters so that applications, or in general *jobs*, of each cluster can be sent and executed on other domains. This allows to set up a ‘pool of clusters’ in which free resources of each member are made available and shared with others.

The general idea behind this protocol is to exploit periodic *advertisement* messages in which a cluster exposes its capabilities. These messages are then used to build a local table that will be analysed to decide where to send the jobs. If the lookup fails for any reason, the protocol provides the possibility to launch an ad-hoc auction for that specific request.

4.1 Point to point connections

In this first scenario there are different K8s clusters, belonging to various entities (companies or users) that may have previously signed a sort of agreement to communicate between each other. Therefore, each member knows the others and has a direct connection with them.

Each cluster periodically sends its *advertisement* and receives the ones from others, maintaining an *advertisement table*.

Whenever a user submits a job to its cluster (a personal one or the one of his company), it decides whether to execute it locally or to delegate it to someone else. In the latter case, the cluster (that will be called *home cluster*) makes a lookup in its table: an algorithm determines the best set of clusters to host the job and, when found, *request* messages are sent to the chosen clusters; they will reply with an implicit *ack/nack* to confirm, or not, the instantiation of the requested job.

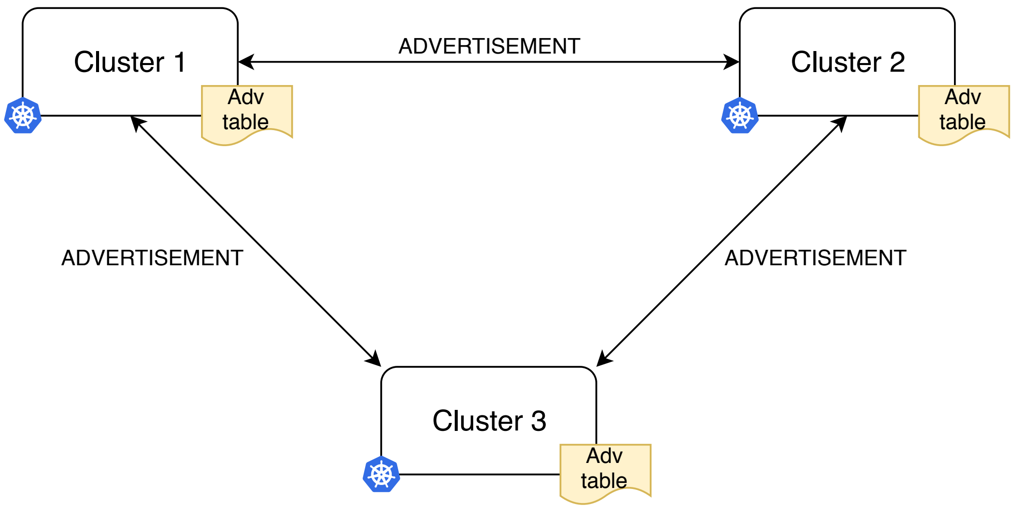

Figure 4.1. *Advertisement* messages exchange.

A job can be composed by several components, which can be scheduled on different clusters: the networking module will deal with the interconnection between all the cluster needed to execute the application.

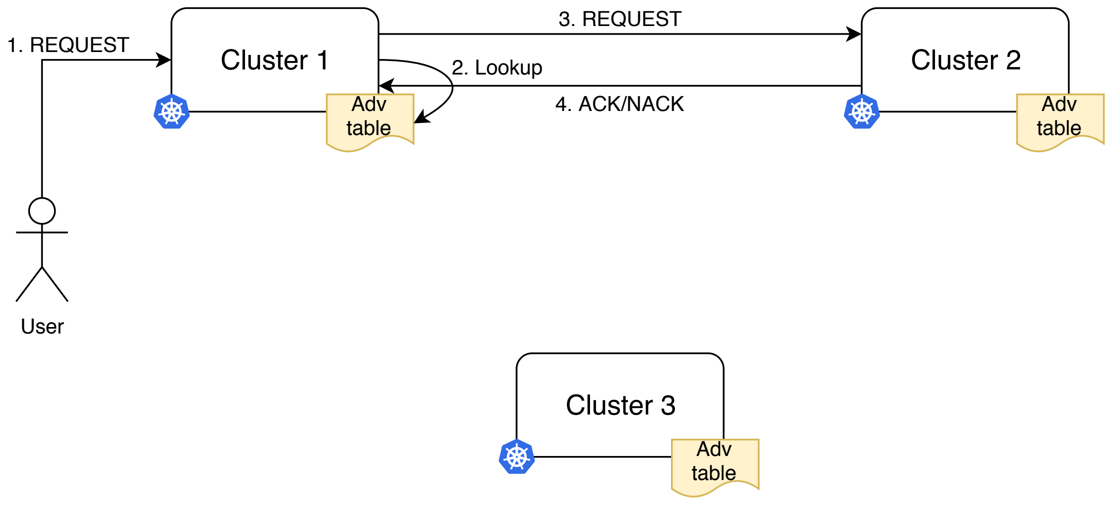

Figure 4.2. *Request* submitted by a user and sent to another cluster.

4.1.1 Ad-hoc request

The protocol supports also the possibility that the lookup in the *advertisement table* fails (e.g. not enough announced resources or request policies cannot be satisfied).

In this case, the *home cluster* can make a specific hosting request to other clusters, triggering an auction in real-time. Each cluster sends its *offer*; the *home cluster* collects all of them and decides where to send the single components or the whole application; the *requests* are followed, as in the previous scenario, by an *ack/nack*.

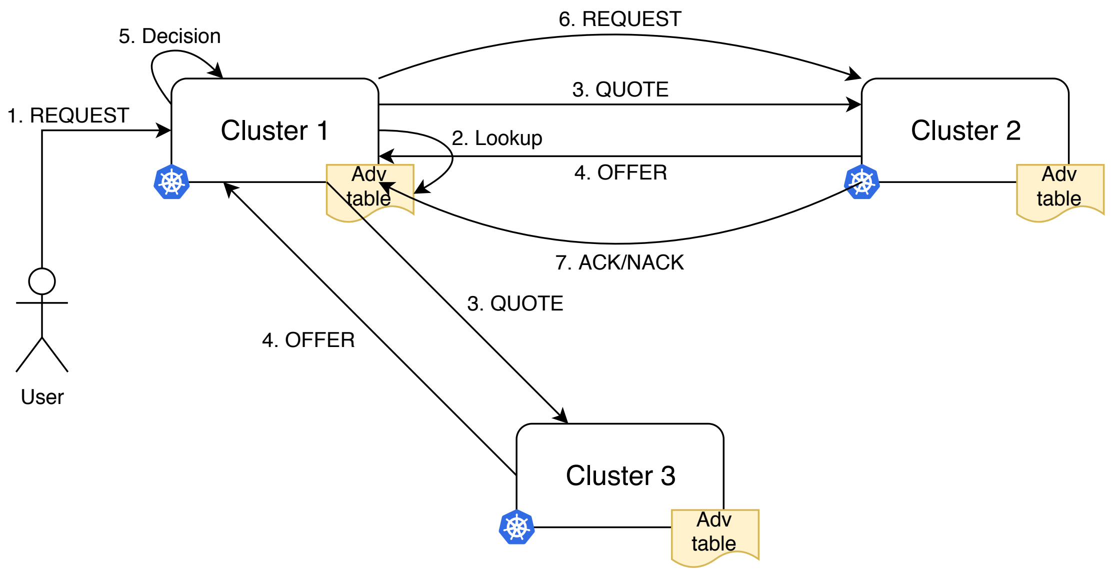

Figure 4.3. Request served with an auction.

Offer messages are different from *advertisements*: *offer* contains the prices for the deployments of the requested application; *advertisement* contains the prices for the cluster resources (cpu, ram, images...). Therefore, *offers* do **not** modify the *advertisement table*: they are specific messages related to the current auction and forgotten when the auction expires, as visible in the messages details in sections 4.2.1 and 4.2.6.

4.1.2 Operating and release phases

While the application is running on remote clusters, the *home cluster* checks periodically if its offloaded components are still alive and the clusters exchange traffic for the communication between the different components.

When the user does no longer need the application, he sends a *release* message to his *home* cluster, which is forwarded it to all the clusters that are currently hosting one or more components of the application itself.

4.2 Messages

4.2.1 Advertisement

Two types of messages are sent:

- full *advertisement*: it contains all the information about the cluster; it is sent every 30 minutes;
- partial *advertisement*: it contains only the free available resources (*availability*) and is also used as *hello message*; it is sent every 10 minutes.

```
{
    "cluster_id": string,
    "resources": [
        "image1": {...},
        ...
    ],
    "availability": [...],
    "prices": [...],
    "network": {...},
    "flags": [...],
    "timestamp": datetime,
    "timeToLive": datetime,
    "status": string
}
```

Listing 4.1. *Advertisement* message.

- `cluster_id`: unique id of the cluster;
- `resources`: images already mounted;
- `availability`: free resources (cpu, memory...);
- `prices`: prices of the previous resources (images and availability);
- `network`: network information about the cluster (e.g. internal CIDR, gateway IP...);
- `flags`: various flags for different purposes (e.g. aggregation information);
- `timestamp`: instant in which this message has been created;
- `timeToLive`: time of validity of this advertisement;

- `status`: status of the advertisement set by the receiving cluster (e.g. the advertisement has been accepted or not).

The above fields are mandatory for every Advertisement message, but new fields can be added in other implementations. The `flags` field serves exactly this purpose: a cluster can define its own flags and, if needed, the additional fields related. If a cluster does not understand a certain flag, it will simply ignore it and all the fields it does not know. For example, a cluster could add the flag `aggregated = true` indicating that the resources announced in its advertisements are the sum of its own resources and the ones of other clusters. When doing this, it can be necessary to add a field `advertisements` in which it inserts the advertisement messages it is aggregating. Another example could be the flag `aggregatable = false` which prevents other clusters from aggregating the advertisement.

The `status` fields allows the receiving cluster to insert its own information in the advertisement message of another, so that the sender can read them. For example it can specify if the advertisement has been accepted or not.

Advertisement table

The *advertisement table* maintains the information about all the clusters that have sent their *advertisement*. When a cluster builds its table, it can enrich the messages with additional data, for example network information (e.g. latency, bandwidth, etc.). If an *hello* message is not received from a cluster for more than 30 minutes (3 messages lost) its entry is deleted.

Cluster_id	*Images*	*Availability*	*Prices*	*Latency*	*Timestamp*	*TimeToLive*
cluster1_id	image1 image2	cpu memory	cpu memory image1 image2	L1	$T1_0$	$T1_1$
cluster2_id	image2 image3	cpu memory	cpu memory image2 image3	L2	$T2_0$	$T2_1$

Table 4.1. Advertisement table.

4.2.2 Request (user → cluster)

The *request* message contains all the data about the application the user wants to execute.

```
{
  "app_id": string,
  "app_info": {
     "app_components": [...],
     ...
  },
  "policies": [...],
  "timestamp": datetime,
  "timeToLive": datetime
}
```

Listing 4.2. *Request* message.

Its main fields are the following:

- `app_id`: id of the application;
- `app_info`: information about the application to be executed (e.g. components of the application);
- `policies`: rules the user sets for his application (e.g. blacklist of clusters, network requirements...).

4.2.3 Request (cluster $\rightarrow$ cluster)

Once the *home cluster* has received the *request* from the user, it decides, using its *advertisement table*, to which clusters it will assign the various application components or, potentially, the whole application; it keeps track of these assignments in the *application table*.

```
{
  "app_id": string,
  "app_components": [...]
}
```

Listing 4.3. *Request* message sent by *home cluster* to others.

Applications table

Each cluster maintains two tables: the first one listing its own applications hosted by other clusters, the second one with applications coming from other clusters, hosted locally. Since the two tables have the same structure, we show here only the application table (Table 4.2).

App_id	*Components*	*Assigned_to / Belongs_to*
app1_id	component1	cluster1_id
	component2	cluster2_id
app2_id	component2	cluster2_id
	component3	cluster3_id

Table 4.2. Applications table.

4.2.4 Acknowledge

When a cluster has been chosen by another and receives the *request* message, it it sends an instantiation confirm or, if it is no longer available, an explicit refusal.

```
{
  "app_id": string,
  "cluster_id": string,
  "confirm": boolean
}
```

Listing 4.4. *Acknowledge* message.

4.2.5 Quote

It contains the components of the application for the auction in progress.

Note: `cluster_id` is the id of the *home cluster*

```
{
  "cluster_id": string,
  "app_id": string,
  "app_components": [...],
  "timestamp": datetime,
  "timeToLive": datetime
}
```

Listing 4.5. *Quote* message.

4.2.6 Offer

It contains the offers of the remote cluster for the auction in progress.

Note: `cluster_id` is the id of the *remote cluster*

```
{
  "cluster_id": string,
  "app_id": string,
  "offers": [
    "app": {
       "interest": number,
       "price": number
    },
    "component1_id": {
       "interest": number,
       "price": number
    }
    ...
  ],
}
```

Listing 4.6. *Offer* message.

- `offers`: list of the offers to host the whole application (`app` field) or the single components; if `interest = 0` the item cannot be assigned to the cluster.

4.3 Broker

In this scenario each cluster sends its *advertisement* as before, but clusters communicate to a broker instead of a foreign domain. Therefore, the broker maintains the *advertisement table* with the information about all clusters connected to it. This way, the involved clusters do no longer need to directly know each other; furthermore, also the joining to the 'cluster pool' is simplified: it is only necessary to make a deal with the broker to get to know all the other members and to be announced to them.

The protocol acts exactly as in the point-to-point scenario; the only difference is that the *advertisement table* is kept and queried on the broker and not locally. Possibly, a cluster can periodically pull the table and keep a local copy to have faster lookups and to decrease the load on the broker. After the lookup in the table and the selection of the cluster set, the communication between the clusters can always pass through the broker or it can specify them the IP addresses to use to speak with each other directly.

The broker can also provide a *certified* service: it can for example suggest a given cluster because it has a very good network connection or because it has a high stability and low downtime. Therefore, the broker can introduce additional value in the system, helping cluster to select which one is the best *foreign cluster* for their services.

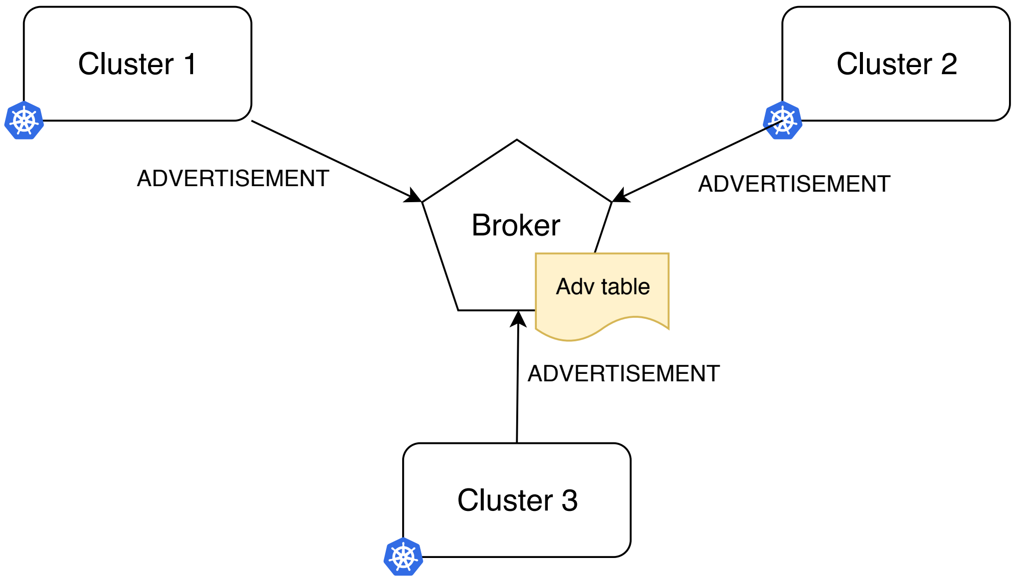

Figure 4.4. *Advertisement* messages exchange with broker.

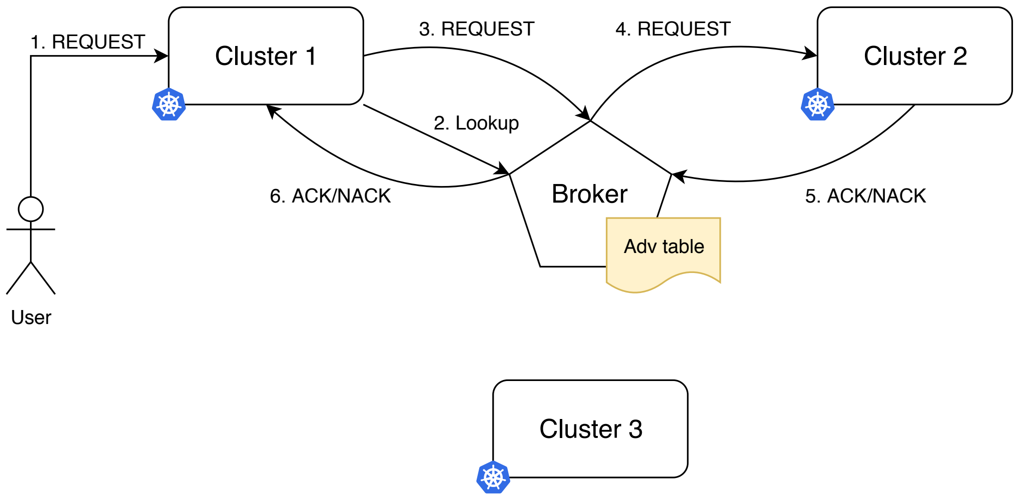

Figure 4.5. *Request* submitted by a user and sent to another cluster through the broker.

Chapter 5

Software architecture

This chapter presents an overview of the developed software architecture, showing the general structure of the solution and deepening the part related to the protocol, which is the one addressed by this thesis.

5.1 Global architecture

The developed solution is split into 3 main components:

- jobs scheduling;
- inter-cluster communication;
- networking.

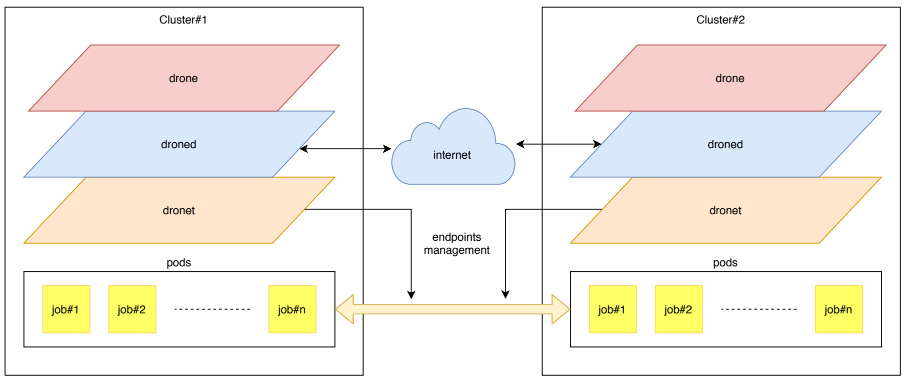

Figure 5.1. Software components.

We will describe the second component, which is in charge of realizing the communication between the clusters through the protocol described in chapter 4. The information provided by this component is used by the scheduling module to distribute jobs across the clusters and by the networking one to set up tunnels and endpoints which enable the network connection between the clusters.

The software architecture is illustrated in the picture below. For the sake of simplicity, we show only two clusters, but the communication may occur between an arbitrary number of them.

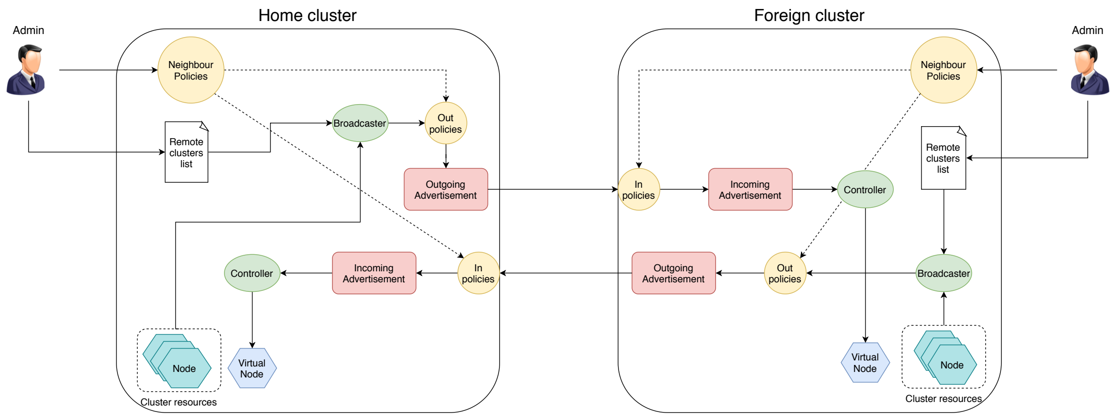

Figure 5.2. Architecture.

The administrator can define some neighbour policies which will be used to accept or modify the incoming Advertisement and will be applied to outgoing messages. These policies are injected in two modules and will be applied during execution. Two chains can be identified:

- **Outgoing chain**:
  1. A list of the remote clusters is provided (by the admin or through a discovery protocol).
  2. The `broadcaster` reads this information to create a client to the foreign clusters.
  3. The `broadcaster` reads the cluster resources and, after applying some policies, creates an `Advertisement`.
  4. The `Advertisement` is pushed to the foreign cluster.
- **Incoming chain**:
  1. An `Advertisement` is received from the foreign cluster.

2. The `Advertisement` is checked by a policy block.
3. If the `Advertisement` is accepted it is sent to the `controller`.
4. The `controller` creates a `virtual node` with the information taken by the `Advertisement`.

In the following a deeper description of the main modules is provided.

5.2 Broadcaster

The broadcaster is in charge of sending to other clusters the Advertisement message, containing the resources made available for sharing and their prices. Before exiting, the message can be modified by the outgoing policies: this allows for example to differentiate the previous information for every receiving cluster.

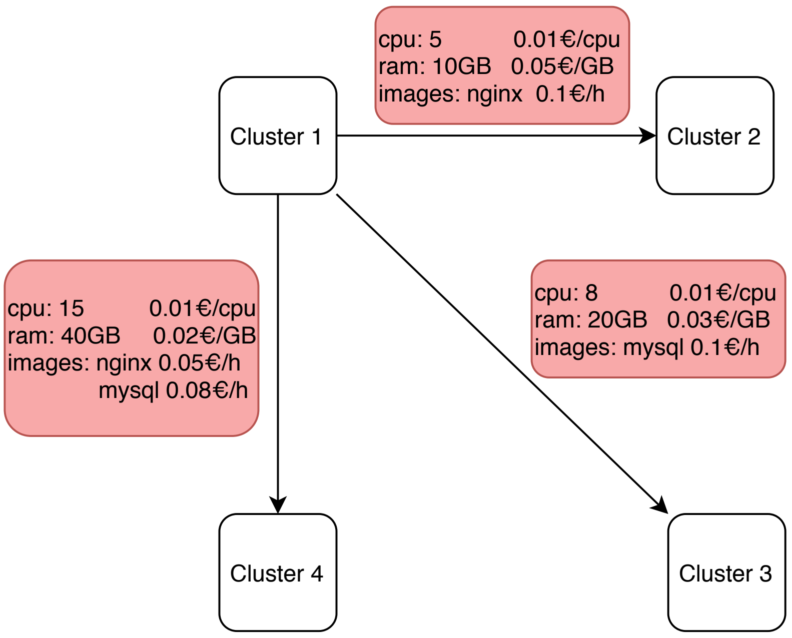

Figure 5.3. Broadcaster differentiation example.

There are two main use case scenarios we are interested in:

- Commercial agreement: clusters sell their resources to others and get paid accordingly.
- Sharing: clusters add their resources to a shared 'pool' and get from it when they need them, without a payment.

The second use case is particularly interesting: imagine when you are at home, there are 4 laptops but nobody is using them; their resources are completely available and unused. With sharing, you can offload part of your jobs to them, reducing the load on your machine and enabling the execution of more powerful applications which your computer alone could not bear. The same scenario can be applied in an office, among the clusters in the internal network.

5.3 Controller

The controller is the module that receives Advertisement messages and creates the virtual nodes with the announced resources. Doing so, the scheduler can read the virtual nodes and decide where to offload the jobs it receives, also considering the requested prices. Before being processed by the controller, messages are checked by the incoming policies block, which can accept or refuse an Advertisement.

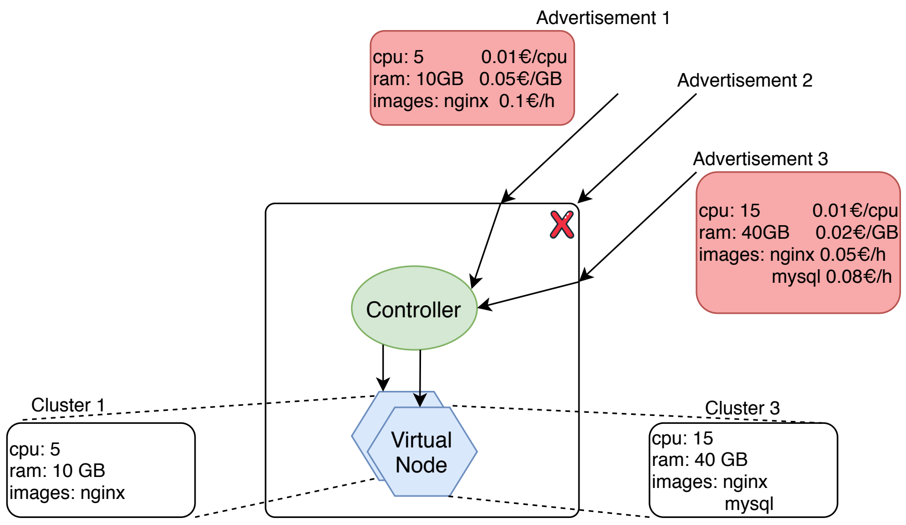

Figure 5.4. Controller reconciliation from Advertisement to Virtual Node.

In the example above, the Advertisement from cluster 2 is discarded, whereas the other messages received are accepted and translated in two virtual nodes. A possible question could be: “Why do I need to create a virtual node? I could simply watch the Advertisements and schedule over them”. This is true, but it would force to adopt a custom scheduler that observes Advertisement resources: instead, the native scheduler uses nodes to take its decisions. Therefore this solution works even with the standard Kubernetes scheduler, although a custom one can provide better results as it considers more rules and constraints.

5.4 Policies

As discussed before, the behaviour of the broadcaster and controller depends on the policies which have been set. A common way to set policies is to use the **Open Policy Agent** (OPA), an open source, general-purpose policy engine that enables unified, context-aware policy enforcement across the entire stack. OPA is a CNCF incubating-level project [14].

OPA decouples policy decision-making from policy enforcement. When your software needs to make policy decisions it queries OPA, providing structured data (e.g. JSON, YAML...) as input. The data that the service or its users publish can be inspected and transformed using OPA's native query language, Rego. **Rego** is a high-level declarative language, inspired by Datalog but which extends it to support structured document models such as JSON[1].

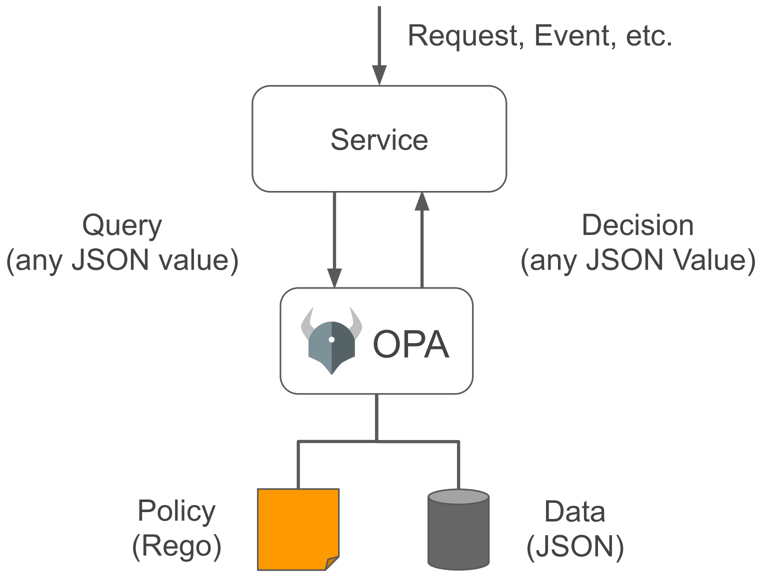

Figure 5.5. OPA policy decoupling [13].

OPA and Rego are domain-agnostic so you can describe almost any kind of objects in your policies, from which users are allowed to access which resources to which times of day the system is accessible. Moreover, policy decisions are not limited to simple yes/no or allow/deny answers: they can produce any structured data as output.

[1]Check the official documentation for further information about Rego language basics `https://www.openpolicyagent.org/docs/latest/policy-language/`

Generally speaking, a policy is a set of rules that controls the behaviour of a software service. It can describe rate-limits, names of trusted servers, the clusters an application should be deployed to, and more. Authorization is a special kind of policy that says which people (or machines) can perform which actions on which resources. Authorization is sometimes confused with Authentication (how people or machines prove their identity). Authorization often exploits the results of authentication (username, user attributes, groups...), but makes decisions based on far more information than just who the user is; some policy decisions have even nothing to do with users.

Today policy is often a hard-coded feature of the software service it administers. Open Policy Agent allows the decoupling of policy from that software service, so that the people responsible for policy can read, write, analyze, and in general manage policy separately from the service itself [13].

The proposed solution requires mainly two types of policies:

- admission/validation policies: they could be applied to incoming Advertisement to remove the ones we are not interested in or validate the data we are receiving;
- mutation policies: they could be used both for incoming Advertisement (e.g. some values are invalid so they are modified with legal ones) and outgoing Advertisement, for example if we want to differentiate the resources announced or their prices on the basis of the destination cluster.

As explained, OPA offers a convenient way to define these types of policies, but being very generic it can be also used for more complicated ones, like scheduling policies: for example, the user could influence the scheduling of the pods across the various clusters on the basis of some parameters he has chosen.

Chapter 6

Implementation

This chapter will present the implementation of the software solution discussed in the previous chapter. For the reasons detailed in section 3.3, Kubernetes has been chosen as orchestrator. The developed code has been written in Go, which is the language in which Kubernetes and all related projects are written.

The implementation is composed by two main modules:

- **Advertisement Operator**: an operator that broadcasts Advertisement messages and reconciles the ones received, creating virtual nodes. This component leverages the concept of Custom Resource Definition presented in section 3.7 and exploits Kubebuilder for code generation.
- **Kubernetes VK provider**: implementation of a provider for the Virtual-Kubelet project, previously illustrated in section 3.6.

6.1 Advertisement operator

The Advertisement operator is the core of the implementation developed during this work. It is composed by the logical blocks described in chapter 5 and detailed in the image below. In the following we will possibly refer to the cluster that **sends** the Advertisement as *home* cluster and the one that **receives** it as *foreign* cluster.

Before launching the operator two steps are required:

- install the Advertisement CRD on your cluster;
- install all the ConfigMaps containing the kubeconfig of every *foreign* cluster you will connect to.

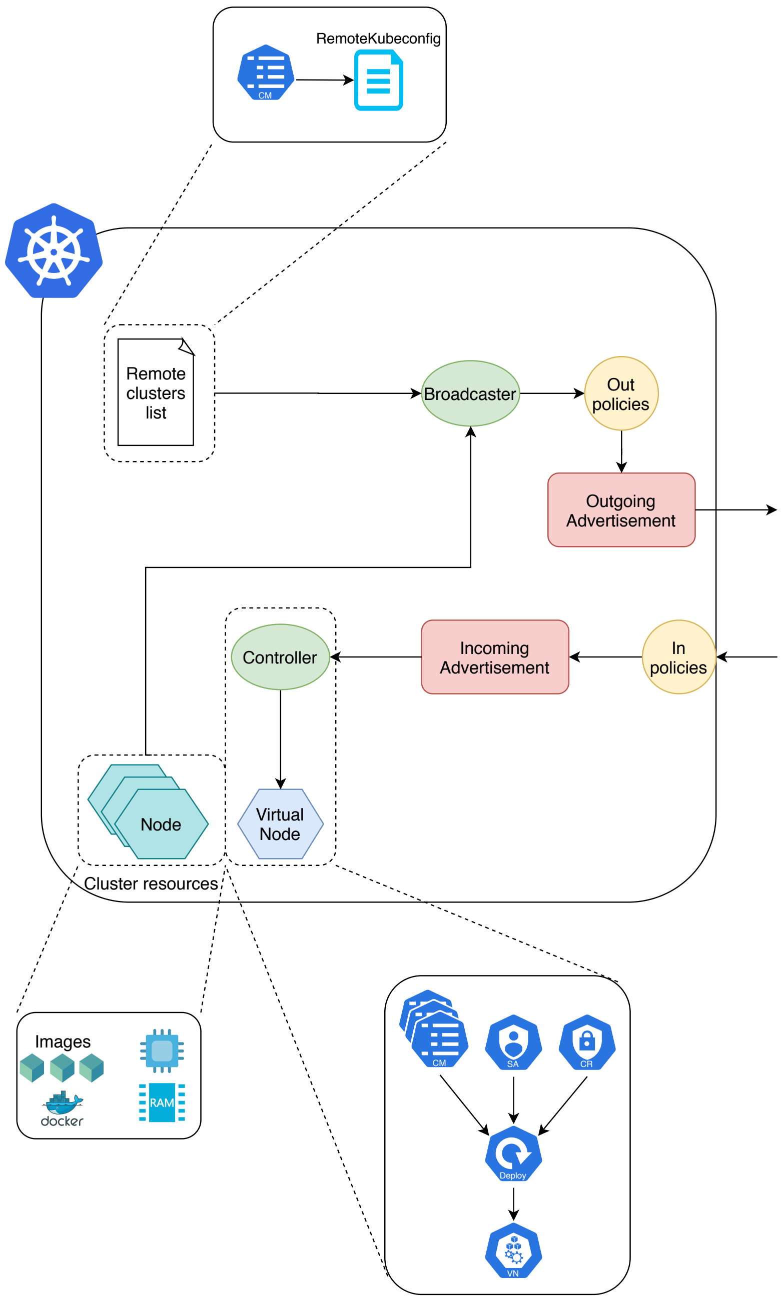

Figure 6.1. Implementation architecture.

6.1.1 Advertisement API definition

The Advertisement message has been modelled as a Custom Resource, which easily allows to extend Kubernetes API. Using Kubebuilder, a `struct` object with the requested fields has to be defined: the tool provides a `Makefile` which generates a CustomResourceDefinition from the designed `struct`.

As most of Kubernetes objects, an `Advertisement` contains the four fields:

- `metav1.TypeMeta` and `metav1.ObjectMeta`: the metadata of the object (e.g. name, namespace, kind, resourceVersion...);
- `Spec`: contains the desired state of the object;
- `Status`: defines the observed state of the object.

```go
// Advertisement is the Schema for the advertisements API
type Advertisement struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec AdvertisementSpec `json:"spec,omitempty"`
    Status AdvertisementStatus `json:"status,omitempty"`
}
```

Listing 6.1. Advertisement struct definition.

`AdvertisementSpec` holds the information the *home* cluster is announcing to others. Every cluster has a unique `ClusterId`, which is a simple string possibly derived by the DNS domain name of the cluster itself, and sends the `Images` already mounted on it and its `Availability` (in terms of cpu, memory etc...); `ContainerImage`[1] is a struct defined by Kubernetes API containing the names by which the image is known and its size in bytes, `ResourceList`[2] is a map where the key is the name of the resource and the value is its quantity.

Each one of the above resources have an associated price in the `Prices` field, with a content similar to the following:

```
prices["cpu"] = 2
prices["memory"] = 3
prices["nginx"] = 10
```

The `NetworkInfo` field contains network information about the cluster, for example the address space of its pods (`PodCIDR`). Finally, the struct contains a `Timestamp` and a `TimeToLive` fields.

[1]https://godoc.org/k8s.io/api/core/v1#ContainerImage

[2]https://godoc.org/k8s.io/api/core/v1#ResourceList

```
type NetworkInfo struct {
    PodCIDR string `json:"podCIDR"`
    GatewayIP string `json:"gatewayIP"`
    // +optional
    SupportedProtocols []string `json:"supportedProtocols,omitempty"`
}

// AdvertisementSpec defines the desired state of Advertisement
type AdvertisementSpec struct {
    ClusterId string json:"clusterId"
    // +optional
    Images []corev1.ContainerImage `json:"images,omitempty"`
    Availability corev1.ResourceList `json:"availability"`
    Prices corev1.ResourceList `json:"prices"`
    Network NetworkInfo `json:"network"`
    Timestamp metav1.Time `json:"timestamp"`
    TimeToLive metav1.Time `json:"timeToLive"`
}
```

Listing 6.2. AdvertisementSpec struct definition.

`AdvertisementStatus` is set by the receiving cluster and filled with information the sending cluster can read. `Acknowledgement` field holds the status of the Advertisement (Accepted, Pending, Refused); `ForeignNetwork` contains network information about cluster *foreign* which are needed by cluster *home* in order to set up network connection; `ObservedGeneration` is needed by the controller to distinguish different types of requests (we will explain this in section 6.1.3).

```
// AdvertisementStatus defines the observed state of Advertisement
type AdvertisementStatus struct {
    Acknowledgement string `json:"acknowledgement"`
    ForeignNetwork NetworkInfo `json:"foreignNetwork"`
    // +optional
    ObservedGeneration int64 `json:"observedGeneration,omitempty"`
}
```

Listing 6.3. AdvertisementStatus struct definition.

The installation of the CRD creates two API endpoints that can be contacted:

- `/apis/protocol.drone.com/v1/namespaces/default/advertisements`: the list of all Advertisements;
- `/apis/protocol.drone.com/v1/namespaces/default/advertisements/{id}`: the specific Advertisement with the given `id`.

6.1.2 Broadcaster

The broadcaster is in charge of sending to other clusters the Advertisement message. It has been implemented as a standalone component launched by the main process in another goroutine.

The `StartBroadcaster` function receives as parameters the ID of the *home* cluster (`clusterId`); `localKubeconfig` and `foreignKubeconfig` can be used for debugging purposes but are not needed when deploying in Kubernetes. The function reads the kubeconfigs of the *foreign* clusters from the ConfigMaps previously installed: with this information it can send its Advertisement to each of them.

```go
func StartBroadcaster(clusterId string, localKubeconfig string,
    foreignKubeconfig string) {

    // get a client to the local cluster
    localClient, err := pkg.NewK8sClient(localKubeconfig, nil)
    if err != nil {
        log.Error(err, "Unable to create client to local cluster")
        return
    }

    // get configMaps containing the kubeconfig of the foreign clusters
    configMaps, err :=
        localClient.CoreV1().ConfigMaps("default").List(metav1.ListOptions{})
    if err != nil {
        log.Error(err, "Unable to list configMaps")
        return
    }
    for _, cm := range configMaps.Items {
        if strings.HasPrefix(cm.Name, "foreign-kubeconfig") {
            go GenerateAdvertisement(localClient, foreignKubeconfig,
                cm.DeepCopy(), clusterId)
        }
    }
}
```

Listing 6.4. StartBroadcaster function.

The `GenerateAdvertisement` function generates an Advertisement CR every 10 minutes and post it to a *foreign* cluster. It receives a client to the local cluster and the ConfigMap containing the kubeconfig to the foreign cluster; `foreignKubeconfigPath` can be used for debugging purposes. First of all it tries to create a client to the *foreign* cluster: after 3 failed attempts the function returns with an error message. After that, an Advertisement object is forged (using the resources read from the Nodes of the *home* cluster) and sent to the *foreign* cluster, which will create (or update if it already exists) it.

```
func GenerateAdvertisement(localClient *kubernetes.Clientset,
    foreignKubeconfigPath string, cm *v1.ConfigMap, clusterId string) {

    var remoteClient client.Client
    var err error
    var retry int
    // extract the foreign cluster id from the configMap
    foreignClusterId := cm.Name[len("foreign-kubeconfig-"):]

    // create a CRDclient to the foreign cluster
    for retry = 0; retry < 3; retry++ {
        remoteClient, err = pkg.NewCRDClient(foreignKubeconfigPath, cm)
        if err != nil {
            log.Error(err, "Unable to create client to remote cluster
                "+foreignClusterId+". Retry in 1 minute")
            time.Sleep(1 * time.Minute)
        } else {
            break
        }
    }
    if retry == 3 {
        log.Error(err, "Failed to create client to remote cluster
            "+foreignClusterId)
        return
    } else {
        log.Info("created client to remote cluster " + foreignClusterId)
    }

    for {
        nodes, err :=
            localClient.CoreV1().Nodes().List(metav1.ListOptions{LabelSelector:
            "type != virtual-node"})
        if err != nil {
            log.Error(err, "Unable to list nodes")
            return
        }

        adv := CreateAdvertisement(nodes.Items, clusterId)
        err = pkg.CreateOrUpdate(remoteClient, context.Background(), log, adv)
        if err != nil {
            log.Error(err, "Unable to create advertisement on remote cluster
                "+foreignClusterId)
        } else {
            log.Info("correctly created advertisement on remote cluster " +
                foreignClusterId)
        }
        time.Sleep(10 * time.Minute)
    }
}
```

Listing 6.5. GenerateAdvertisement function.

6.1.3 Controller

The controller is the module that watches Advertisement resources. When an Advertisement is received, it is checked by the `checkAdvertisement` method that applies the policies defined by the administrator and update its status accordingly; this way the sending cluster can have an acknowledgement by reading the status. If the object is accepted, the Deployment containing the Virtual-Kubelet is created: this will spawn the virtual node with the resources announced.

The main logic is quite simple, but there is a problem: the `Reconcile` method offered by Kubebuilder is triggered for **every** modification of the watched Custom Resource. This can be cumbersome because the `Request` message received does not contain the type of the request (create/update/delete). Since the `Reconcile` method updates the Advertisement status, an infinite loop can be easily started. To deal with this problem the following solution has been adopted:

- `DELETE`: if the Advertisement has been deleted, the `Get` method will fail and the error will be ignored.
- `UPDATE`: the metadata of a K8s object contains a `Generation` field that is incremented for all changes, except for changes to metadata or status. This field is copied in the Advertisement status at creation and for every update of the spec: by comparing `Generation` field with the last `ObservedGeneration`, status updates can be ignored.

```go
func (r *AdvertisementReconciler) Reconcile(req ctrl.Request) (ctrl.Result,
    error) {

    // get advertisement
    var adv protocolv1.Advertisement
    if err := r.Get(ctx, req.NamespacedName, &adv); err != nil {
        // reconcile was triggered by a delete request
        log.Info("Advertisement " + req.Name + " deleted")
        return ctrl.Result{}, client.IgnoreNotFound(err)
    }

    // The metadata.generation value is incremented for all changes, except
        for changes to .metadata or .status
    if adv.Status.ObservedGeneration == adv.ObjectMeta.Generation {
        return ctrl.Result{}, client.IgnoreNotFound(nil)
    }

    // filter advertisements and create a virtual-kubelet only for the
        accepted ones
    adv = checkAdvertisement(adv)
    if err := r.Status().Update(ctx, &adv); err != nil {
        log.Error(err, "unable to update Advertisement status")
        return ctrl.Result{}, err
    }

    if adv.Status.Acknowledgement != "ACCEPTED" {
        return ctrl.Result{}, errors.NewBadRequest("advertisement ignored")
    }

    // create configuration for virtual-kubelet with data from adv
    vkConfig := createVkConfig(adv)
    err := pkg.CreateOrUpdate(r.Client, ctx, log, vkConfig)
    if err != nil {
        return ctrl.Result{}, err
    }

    // launch virtual-kubelet deployment
    deploy := pkg.CreateVkDeployment(adv)
    err = pkg.CreateOrUpdate(r.Client, ctx, log, deploy)
    if err != nil {
        return ctrl.Result{}, err
    }

    log.Info("launching virtual-kubelet for cluster " + adv.Spec.ClusterId)

    return ctrl.Result{}, nil
}
```

Listing 6.6. Reconcile method.

6.2 Kubernetes provider

The Kubernetes provider is the component that implements the interface exposed by Virtual-Kubelet. Many companies have implemented their own provider (Alibaba, Azure, HashiCorp...), which allows a Kubernetes cluster to see and work transparently with their cloud platforms. The provider that has been implemented does the same with a K8s cluster, ‘hiding’ the whole cluster behind a virtual Node, hence avoiding to expose precise information about the cluster itself to *foreign* parties.

```
type PodLifecycleHandler interface {
    // CreatePod takes a Kubernetes Pod and deploys it within the provider.
    CreatePod(ctx context.Context, pod *corev1.Pod) error

    // UpdatePod takes a Kubernetes Pod and updates it within the provider.
    UpdatePod(ctx context.Context, pod *corev1.Pod) error

    // DeletePod takes a Kubernetes Pod and deletes it from the provider.
    DeletePod(ctx context.Context, pod *corev1.Pod) error

    // GetPod retrieves a pod by name from the provider (can be cached).
    GetPod(ctx context.Context, namespace, name string) (*corev1.Pod, error)

    // GetPodStatus retrieves the status of a pod by name from the provider.
    GetPodStatus(ctx context.Context, namespace, name string)
        (*corev1.PodStatus, error)

    // GetPods retrieves a list of all pods running on the provider (can be
        cached).
    GetPods(context.Context) ([]*corev1.Pod, error)
}
```

Listing 6.7. Virtual-Kubelet provider interface [22].

All the methods above refer to a `Provider` object. The main information contained in the one implemented are the `client` to the *foreign* cluster, the `config` used to create the virtual node and the `notifier` function which allows the provider to notify the virtual-kubelet about pod status changes.

```
type KubernetesProvider struct {
    client *kubernetes.Clientset
    nodeName string
    operatingSystem string
    internalIP string
    daemonEndpointPort int32
    config KubernetesConfig
    startTime time.Time
    notifier func(*v1.Pod)
}
```

Listing 6.8. KubernetesProvider struct definition.

`KubernetesConfig` contains the configurable parameters for the virtual-kubelet: the path to the *foreign* Kubeconfig file, the resources to set in the virtual node (`CPU, Memory, Pods`) and the `Namespace` to map (i.e. only the pods belonging to this namespace will be sent to the *foreign* cluster).

```
type KubernetesConfig struct {
    RemoteKubeConfigPath string `json:"remoteKubeconfig,omitempty"`
    CPU string `json:"cpu,omitempty"`
    Memory string `json:"memory,omitempty"`
    Pods string `json:"pods,omitempty"`
    Namespace string `json:"namespace,omitempty"`
}
```

Listing 6.9. KubernetesConfig struct definition.

6.2.1 Pod lifecycle methods

The implementation of the interface methods shown in listing 6.7 is quite similar; the main difference is the action which is performed on the Pod (create/update/delete); therefore we will only describe the `CreatePod()` and `GetPod()`.

When a pod is submitted (e.g. `kubectl create -f pod.yaml`) on the *home* cluster, the `CreatePod()` method is triggered. First of all it checks if the pod is controlled by a DaemonSet; a DaemonSet is a controller which ensures that all Nodes run a copy of a Pod. These pods are typically related to internal network, storage, logging or monitoring, so they must not be sent to the *foreign* cluster. Second, the pod is transformed so that it can be correctly posted on *foreign* cluster and it is created. Finally the local pod is notified by the updates of its remote copy.

```go
func (p *KubernetesProvider) CreatePod(ctx context.Context, pod *v1.Pod)
    error {
    ctx, span := trace.StartSpan(ctx, "CreatePod")
    defer span.End()

    // Add the pod's coordinates to the current span.
    ctx = addAttributes(ctx, span, namespaceKey, pod.Namespace, nameKey,
        pod.Name)

    log.G(ctx).Infof("receive CreatePod %q", pod.Name)

    // ignore DaemonSet pods
    if pod != nil && pod.OwnerReferences != nil && len(pod.OwnerReferences)
        != 0 && pod.OwnerReferences[0].Kind == "DaemonSet" {
        msg := fmt.Sprintf("Skip to create DaemonSet pod %q", pod.Name)
        log.G(ctx).WithField("Method", "CreatePod").Info(msg)
        return nil
    }

    // prepare a pod to be pushed on foreign cluster and try to create it
    podTranslated := H2FTranslate(pod)

    podServer, err :=
        p.client.CoreV1().Pods(p.config.Namespace).Create(podTranslated)
    if err != nil {
        return errors.Wrap(err, "Unable to create pod")
    }
    log.G(ctx).Info("Pod", podServer.Name, "successfully created on remote
        cluster")

    p.notifier(pod)

    return nil
}
```

Listing 6.10. CreatePod method.

`GetPod()` returns a pod given its `name` and `namespace`. The pod received from the *foreign* cluster is transformed by a function that is the inverse of the one used to post the pod and returned by the method.

```go
func (p *KubernetesProvider) GetPod(ctx context.Context, namespace, name
    string) (pod *v1.Pod, err error) {
    ctx, span := trace.StartSpan(ctx, "GetPod")
    defer func() {
        span.SetStatus(err)
        span.End()
    }()

    // Add the pod's coordinates to the current span.
    ctx = addAttributes(ctx, span, namespaceKey, namespace, nameKey, name)

    log.G(ctx).Infof("receive GetPod %q", name)
    opts := metav1.GetOptions{}
    podServer, err :=
        p.client.CoreV1().Pods(p.config.Namespace).Get(name,opts)
    if err != nil {
        if kerror.IsNotFound(err) {
            return nil, errdefs.NotFoundf("pod \"%s/%s\" is not known to the
                provider", namespace, name)
        }
        return nil, errors.Wrap(err, "Unable to get pod")
    }
    podInverted := F2HTranslate(podServer, p.config.RemoteNewPodCidr)
    return podInverted, nil
}
```

Listing 6.11. GetPod method.

6.2.2 Pod translation methods

The `H2FTranslate()` (Home-to-Foreign) function receives a pod from *home* cluster and prepares a pod to be pushed on the *foreign* one, copying only the necessary fields (`Name, Namespace, Labels, Containers`). An `Affinity` field is added so that the pod on *foreign* cluster cannot be scheduled on a virtual-kubelet node. Finally, some original values are saved in `Annotations` in order to retrieve them in the inverse translation.

```go
func H2FTranslate(pod *v1.Pod) *v1.Pod {
    // create an empty ObjectMeta, copying only "Name" and "Namespace" fields
    objectMeta := metav1.ObjectMeta{
        Name: pod.ObjectMeta.Name,
        Namespace: pod.ObjectMeta.Namespace,
        Labels: pod.Labels,
    }
    // copy all containers from input pod
    containers := make([]v1.Container, len(pod.Spec.Containers))
    for i := 0; i < len(pod.Spec.Containers); i++ {
        containers[i].Name = pod.Spec.Containers[i].Name
        containers[i].Image = pod.Spec.Containers[i].Image
    }

    affinity := v1.Affinity{
        NodeAffinity: &v1.NodeAffinity{
            RequiredDuringSchedulingIgnoredDuringExecution: &v1.NodeSelector{
                NodeSelectorTerms: []v1.NodeSelectorTerm{
                    v1.NodeSelectorTerm{
                        MatchExpressions: []v1.NodeSelectorRequirement{
                            v1.NodeSelectorRequirement{
                                Key: "type",
                                Operator: v1.NodeSelectorOpNotIn,
                                Values: []string{"virtual-node"},
                            },
                        },
                    },
                },
            },
        },
    }
    // create an empty Spec, copying only Containers and Affinity fields
    podSpec := v1.PodSpec{
        Containers: containers,
        Affinity: affinity.DeepCopy(),
    }

    metav1.SetMetaDataAnnotation(&objectMeta, "home_nodename",
        pod.Spec.NodeName)
    metav1.SetMetaDataAnnotation(&objectMeta, "home_resourceVersion",
        pod.ResourceVersion)
    metav1.SetMetaDataAnnotation(&objectMeta, "home_uuid", string(pod.UID))
    metav1.SetMetaDataAnnotation(&objectMeta, "home_creationTimestamp",
        pod.CreationTimestamp.String())

    return &v1.Pod{
        TypeMeta: pod.TypeMeta,
        ObjectMeta: objectMeta,
        Spec: podSpec,
        Status: pod.Status,
    }
}
```

Listing 6.12. Home-to-Foreign function.

`F2HTranslate()` (Foreing-to-Home) function performs the inverse translation, taking the original value of the fields `UID`, `ResourceVersion`, `CreationTimestamp`, `NodeName` from the `Annotations`; possibly, the `PodIP` is changed too. Eventually, the values saved in `Annotations` are deleted.

```go
func F2HTranslate(podForeignIn *v1.Pod, newCidr string) (podHomeOut *v1.Pod) {
    podHomeOut = podForeignIn.DeepCopy()
    podHomeOut.SetUID(types.UID(podForeignIn.Annotations["home_uuid"]))
    podHomeOut.SetResourceVersion(podForeignIn.Annotations["home_resourceVersion"])
    t, err := time.Parse("2006-01-02 15:04:05 -0700 MST",
        podForeignIn.Annotations["home_creationTimestamp"])
    if err != nil {
        _ = fmt.Errorf("Unable to parse time")
    }
    podHomeOut.SetCreationTimestamp(metav1.NewTime(t))

    if podHomeOut.Status.PodIP != "" {
        newIp := changePodIp(newCidr, podHomeOut.Status.PodIP)
        podHomeOut.Status.PodIP = newIp
        podHomeOut.Status.PodIPs[0].IP = newIp
    }

    podHomeOut.Spec.NodeName = podForeignIn.Annotations["home_nodename"]
    delete(podHomeOut.Annotations, "home_creationTimestamp")
    delete(podHomeOut.Annotations, "home_resourceVersion")
    delete(podHomeOut.Annotations, "home_uuid")
    delete(podHomeOut.Annotations, "home_nodename")
    return podHomeOut
}
```

Listing 6.13. Foreign-to-Home function.

Chapter 7

Results

This chapter will show how the implementation described in the previous chapter has been tested to derive some qualitative results that characterize the current prototype.

7.1 Test environment

As seen in chapter 6, the implemented solution heavily relies upon Kubernetes API resources and libraries. For this reason we are more interested in the **end-to-end** testing rather than unit testing, because we assume that the objects offered by Kubernetes work properly.

In order to have a test environment that could be easily destroyed and recreated, the tests have been carried out using two simple types of cluster: **Minikube** [12], a local single-node K8s cluster with all Kubernetes features, and **KinD** [6], a tool for running local Kubernetes clusters within Docker containers. Both of them are not intended to be used in production, but it is not unusual for a single user to install them on his local machine (especially Minikube). Therefore they perfectly suit one of the use cases presented in the introduction in chapter 1, where different users with their small clusters want to share resources among them; for example this could be the case of a university lab or a company office.

All the tests have been run on my personal laptop using two or three virtual machines running Ubuntu 18.04 (kernel version 4.15.0-88) one with 2 cpu cores, 8 GB of ram and 100 GB of disk, the others with 2 cpu cores, 4 GB of ram and 20 GB of disk; Kubectl version was the latest (1.17.3), whereas Minikube hosted Kubernetes v1.16.2 on Docker 19.03.7.

7.2 Functional tests

Functional tests aim at verifying that the end-to-end chain operates as expected:

1. launch the Advertisement operator on both clusters;
2. check that *home-Advertisement* is created on *foreign* cluster and vice versa;
3. upon the receipt of the Advertisement a Virtual-Kubelet Node must be created;
4. when a pod is scheduled on the virtual node it must be launched on the other cluster.

The first component that has been tested is the Virtual-Kubelet Kubernetes provider (section 6.2). Initially it was executed as a process outside the cluster: it created a Node and sent all the pods scheduled on it (on *home* cluster) to the *foreign* one, returning the updates on its status (Pending, Running etc...). When everything was working, the Virtual-Kubelet was moved in a pod managed by a Deployment.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: vkubelet
  name: vkubelet
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      run: vkubelet
  template:
    metadata:
      labels:
        run: vkubelet
    spec:
      serviceAccountName: user1
      containers:
      - image: dronev2/virtual-kubelet
        name: vkubelet
        command: ["/usr/bin/virtual-kubelet"]
        args: ["--provider", "kubernetes",
               "--provider-config", "/app/config/vkubelet-cfg.json",
               "--disable-taint"]
        volumeMounts:
          - name: provider-config
            mountPath: /app/config/vkubelet-cfg.json
          subPath: vkubelet-cfg.json
        - name: remote-config
          mountPath: /app/kubeconfig/remote
          subPath: remote
      volumes:
        - name: provider-config
          configMap:
            name: vk-config
        - name: remote-config
          configMap:
            name: foreign-kubeconfig
```

Listing 7.1. Virtual-Kubelet Deployment.

As shown in the example above, the Virtual-Kubelet needs some arguments:

- `provider`: the name of the provider which must be executed (in this case `kubernetes`);
- `provider-config`: the configuration file with the information needed by the provider;
- `disable-taint`: a flag which allows to schedule pods on the virtual node created.

Moreover, the Deployment mounts two Volumes containing the configuration file for the provider and the kubeconfig of the *foreign* cluster.

To develop and test the Advertisement operator we followed a similar set of steps, starting with out-of-cluster execution and ending with the Deployment shown below. In this case we only need a parameter with the `cluster-id` of the cluster; the generation of the Virtual-Kubelet Deployment is made by the operator in the `Reconcile()` method.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: advertisement-operator
  name: advertisement-operator
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      run: advertisement-operator
  template:
    metadata:
```

```
    labels:
      run: advertisement-operator
  spec:
    serviceAccountName: user1
    containers:
    - image: dronev2/advertisement-operator
      name: advertisement-operator
      command: ["/usr/bin/advertisement-operator"]
      args: ["--cluster-id", "cluster1"]
```

Listing 7.2. Advertisement operator Deployment.

7.3 Performance and scalability tests

Tests presented in this section aim at a preliminary assessment of the performance and scalability of the solution. These tests were executed on an Ubuntu 18.04 virtual machine with 8 GB of RAM and 2 CPU cores. The cluster installed was Minikube [12].

We could not test with real cluster because of the difficulties in the setup of the required hardware, hence we emulated them by creating Advertisement CR manually: the Advertisement operator reacted creating a Virtual-Kubelet for every message. The script below sets up the cluster installing the Advertisement CRD and the necessary ConfigMaps; after that, it launches the Advertisement operator and creates the Advertisement CRs.

```
#!/bin/bash

clusterNum=100

# configure clusters installing the Advertisement CRD and creating the
    configMaps containing the kubeconfig
kubectl apply -f adv-crd.yaml
for ((i=1;i<=clusterNum;i++)); do
    id=cluster${i}
    kubectl create configmap foreign-kubeconfig-${id}
        --from-file=remote=kubeconfig
done

# create Advertisement operator deployment
kubectl apply -f adv-deploy.yaml

# create Advertisements
for ((i=1;i<=clusterNum;i++)); do
  sed -i -e "s/clusterX/cluster$i/g" adv.yaml
  kubectl apply -f adv.yaml
  sed -i -e "s/cluster$i/clusterX/g" adv.yaml
```

```
done

exit 0
```

Listing 7.3. Bash script to configure the cluster and create Advertisements.

This is the Advertisement used for this test.

```
apiVersion: protocol.drone.com/v1
kind: Advertisement
metadata:
  name: advertisement-clusterX
spec:
  clusterId: "clusterX"
  images:
    - names:
        - "nginx"
    - names:
        - "mongodb"
  availability:
    cpu: "3"
    memory: "2Gi"
    pods: "100"
  network:
    gatewayIP: 172.17.0.3
    podCIDR: 10.244.0.0/16
  prices:
    cpu: "1"
    memory: "2"
    nginx: "20"
    mongodb: "50"
  timestamp: 2020-03-13T13:30:59Z
  timeToLive: 2020-03-13T13:50:59Z
```

Listing 7.4. Sample Advertisement used in the tests.

7.3.1 Test results

In order to evaluate the performance of the solution, some time intervals have been analysed, in particular the ones between the following time instants:

- Advertisement `creationTimestamp`;
- Virtual-Kubelet Deployment `creationTimestamp`;
- Virtual-Kubelet Pod `startTime`;
- Virtual-Kubelet Node `creationTimestamp`.

Initially, the script has been launched with `clusterNum=10`, returning the results illustrated in the figure below.

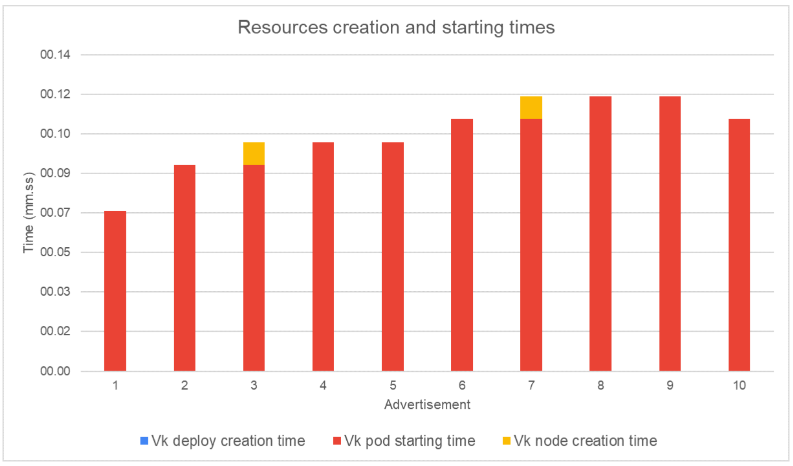

Figure 7.1. Observed times with 10 Advertisements.

We can see that the Virtual-Kubelet Deployment is created immediately when an Advertisement is received, whereas its Pod requires 9-10 seconds to start on average, and this value generally increases with the number of Advertisements. Virtual-Kubelet Node creation, after the Pod has started, is nearly immediate too.

After this first test, the script was executed again, but with `clusterNum=100`. The results, displayed in figure 7.2, follows the same trend observed with 10 Advertisement: Virtual-Kubelet Deployment and Node creation times are almost always null, while the Pod starting time grows, up to nearly 2 minutes.

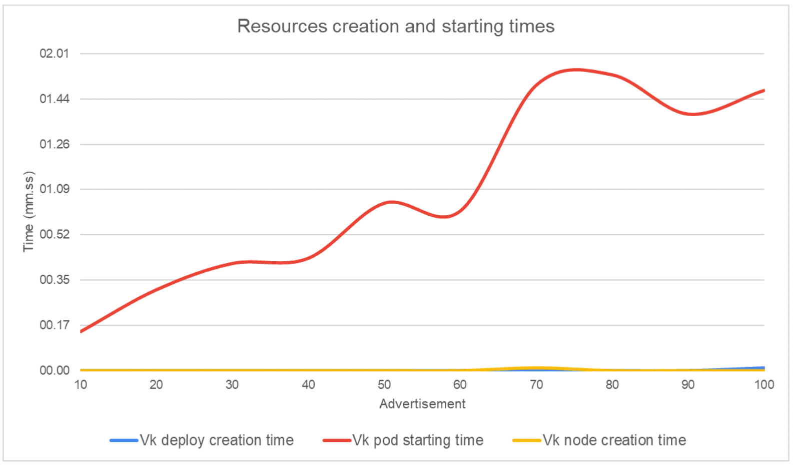

Figure 7.2. Observed time trends with 100 Advertisements.

Apparently the solution does not scale in the number of Advertisements. However, an additional test proves the opposite: if we create a new Advertisement **after some time**, when all the previous 100 have stabilized (i.e. the Virtual-Kubelet Node has been created and is in `Ready` status), the required times are the same of the first graph, 8 seconds from Advertisement CR creation to Virtual-Kubelet Node creation. This means that the problems occurs when we receive many Advertisements **in a short period of time**, because each of them triggers a chain of K8s objects creation, but there are no problems in storing many of them and managing multiple Virtual-Kubelet Nodes.

The last performance test was created to evaluate if the efficiency of the Kubernetes default scheduler would be affected by the presence of many Virtual-Kubelet Nodes. Therefore we ran a simple `nginx` Deployment and discovered it was not disturbed at all: each pod was created on a different node and reached `Running` status in 6 seconds on average, independently from the number of nodes (including virtual nodes) active in each cluster.

7.3.2 Test limitations

The tests described are obviously limited and cannot be considered as actual "scalability tests", but they are a first achievement. Moreover, consider they were launched on a Minikube cluster, therefore all the Virtual-Kubelet Pods had to run on only one real Node: there were 100 pods running **on the same Node**, and this is the limitation set by Kubernetes[1]. In fact, the creation of an additional Deployment was blocked by the system, but in spite of all that, the cluster continued to work properly.

[1]check the official documentation `https://kubernetes.io/docs/setup/best-practices/cluster-large/`

Chapter 8

Conclusions and future works

This thesis has presented some of the modules that compose the project we are developing at the Computer Networks Group at Politecnico di Torino. The goal of this project is to enable dynamic sharing of resources between different clusters, with a smart scheduler that distributes the components of each application and a networking module that allows the communication among them. The target is Kubernetes, but the protocol and the software architecture are generic and independent from the underlying technology.

The analysis of the related works shows that this issue has started to be addressed only in the last years and there is not a stable and widespread solution available yet; moreover, the official proposal from Kubernetes has some limitations that we want to overcome providing a **transparent** and **decentralized** solution. Therefore, the prototype that has been developed adopts the approach of extending the API exposed by Kubernetes using Custom Resource Definition, allowing a user to exploit the advantages of sharing without needing to modify his interaction with Kubernetes. The qualitative tests that have been carried out shows that this implementation does not introduce any noticeable overhead in the deployment of applications. The solution seems also to scale with an increasing number of clusters, as long as Advertisements are created with a certain delay between them, in order to allow the cluster to set up all the necessary resources; however, even in this situation the cluster should be ready in some minutes and this additional cost needs to be paid only when bootstrapping.

Starting from the work done, the first feature that should be implemented in the future works is the policy management; as explained in chapter 5, Open Policy Agent will be used for this purpose. Another issue that still needs to be formalized and addressed is about the setup of the foreign clusters each cluster can communicate with: at the moment this information is installed as ConfigMaps in the cluster before the operator is executed, so that it can read them, but a sort of 'discovery protocol' could be defined (or already existing protocols could be used, like UPnP[1]). A further extension that has only been designed but not yet realized is the concept of the broker illustrated in section 4.3, which could enable many additional features.

Finally, some aspects that have already been discussed and conceptualised but still need to be implemented are for example the completeness of the Virtual-Kubelet (kubelet logs, exec etc...), the addition of network information such as latency or bandwidth in the Advertisement, the resource locking on the *foreign cluster* upon the receipt of an Advertisement or the creation of an environment on the *foreign cluster* with all the resources (CRDs, Volumes, ConfigMaps etc...) the user has defined on his *home cluster*.

[1]`https://en.wikipedia.org/wiki/Universal_Plug_and_Play`

Bibliography

[1] *8 facts about real-world container use.* URL: https://www.datadoghq.com/container-report/.

[2] Uchechukwu Awada. «Hybrid Cloud Federation: A Case of Better Cloud Resource Efficiency». In: July 2018.

[3] Eric Carter. *Sysdig 2019 Container Usage Report: New Kubernetes and security insights.* Oct. 2019. URL: https://sysdig.com/blog/sysdig-2019-container-usage-report/.

[4] Joan Engebretson. *Will Kubernetes Be the Operating System for 5G? AT&T News Suggests Yes.* Feb. 2019. URL: https://www.telecompetitor.com/will-kubernetes-be-the-operating-system-for-5g-att-news-suggests-yes/.

[5] Ferenc Hámori. *The History of Kubernetes on a Timeline.* June 2018. URL: https://blog.risingstack.com/the-history-of-kubernetes/.

[6] *KinD git repository.* URL: https://github.com/kubernetes-sigs/kind.

[7] *Kubebuilder git repository.* URL: https://github.com/kubernetes-sigs/kubebuilder.

[8] *Kubernetes API official documentation.* URL: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/.

[9] *Kubernetes Federation git repository.* URL: https://github.com/kubernetes-sigs/kubefed.

[10] *Kubernetes official documentation.* URL: https://kubernetes.io/docs/home/.

[11] *Kubernetes Operator pattern.* URL: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

[12] *Minikube project git repository.* URL: https://github.com/kubernetes/minikube.

[13] *OPA documentation.* URL: https://www.openpolicyagent.org/docs/latest/.

[14] *OPA git repository.* URL: https://github.com/open-policy-agent/opa.

[15] Kalyan Ramanathan. *5 business reasons why every CIO should consider Kubernetes.* Oct. 2019. URL: https://www.sumologic.com/blog/why-use-kubernetes/.

[16] Malte Schwarzkopf et al. «Omega: flexible, scalable schedulers for large compute clusters». In: *SIGOPS European Conference on Computer Systems (EuroSys).* Prague, Czech Republic, 2013, pp. 351–364. URL: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf.

[17] *The Raft Consensus Algorithm.* URL: https://raft.github.io/.

[18] *TOP-IX website.* URL: https://www.top-ix.org/it/home/.

[19] *Understanding the cloud service broker model.* URL: https://searchnetworking.techtarget.com/definition/cloud-federation.

[20] Steven J. Vaughan-Nichols. *The five reasons Kubernetes won the container orchestration wars.* Jan. 2019. URL: https://blogs.dxc.technology/2019/01/28/the-five-reasons-kubernetes-won-the-container-orchestration-wars/.

[21] Abhishek Verma et al. «Large-scale cluster management at Google with Borg». In: *Proceedings of the European Conference on Computer Systems (EuroSys).* Bordeaux, France, 2015.

[22] *Virtual-kubelet git repository.* URL: https://github.com/virtual-kubelet/virtual-kubelet.

[23] Gianluca Zangara et al. «A Cloud Federation architecture Based on Open Technologies». In: Nov. 2015.