POLITECNICO DI TORINO

Master's Degree in Computer engineering

Master's Degree Thesis

Scheduling Jobs on Federation of Kubernetes Clusters

Supervisors

Prof. Fulvio RISSO

Dott. Alex PALESANDRO

Ing. Gabriele CASTELLANO

Candidate

Mattia LAVACCA

Academic year 2019-2020

Summary

In the last two decades the cloud has gained a lot of importance, indeed the current trend is to engineer the new web applications to be cloud native, thus to be split up in loosely-coupled micro-services, each one containerized and deployed as a part of a bigger application. The use of containers allows to cut oneself off the hosting physical hardware and operating system, letting to focus on the main purposes of a web application: to be widespread and high-available. The cloud allows to achieve this goal, by gathering the infrastructure control under the cloud provider tenants and implementing the IaaS (Infrastructure as a Service) and PaaS (Platform as a Service) paradigms: the computational, networking and storage resources are provided on demand to the cloud provider's customers as if they were services. A technology that broke through the cloud market is Kubernetes, a project kicked off by Google in 2014 that allows to automate deployment, scaling, and management of containerized applications. Beside the cloud, in recent years the edge computing has gained a lot of importance: it is a distributed computing paradigm that brings the computational and storage resources close to the final user, in order to improve the QoS standards in terms of latency and bandwidth.

The goal of the project in which this thesis is involved is to create a federation of Kubernetes clusters that cooperate at the network edge: many different tenants are connected together to cooperate in creating a federation of clusters with computational, storage and networking resources shared between them. In this scenario every tenant can make its own resource cluster available to the federation by sharing or leasing them out in a federated environment.

This model needs a job scheduler able to take into account both computational and inter-connectivity requirements for performing a scheduling that satisfies QoS policies set by the application. This work proposes a novel scheduling algorithm using an heuristic to pair every job to be deployed to a cluster of the federation: a deployment composed by a set of jobs (i.e., micro-services), each one with different constraints in terms of computation, memory, storage and inter-connection is represented by a deployment graph that is scheduled on the graph of underlying federated clusters. The function to be optimized by the algorithm is the cost for the leasing of the set of jobs.

Table of Contents

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 [2].

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 [3]) 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 [4]).

This work, carried out by the Computer Networks Group at Politecnico di Torino proposes an alternative to the Kubernetes scheduler, able to schedule in a gang-fashion sets of inter-connected and inter-dependent jobs (from now addressed as deployments) in a federation of clusters (possibly under different tenant's domains), while complying with an arbitrary number of policies, both on the clusters features and the inter-connections parameters between them. We address gang scheduling as the scheduling of a set of jobs with all their dependencies at once. The final goal of the algorithm is to minimize the monetary charge claimed by federated cluster's owners.

This thesis is structured as follows:

- **Chapter 2** provides an extensive presentation of Kubernetes, its architecture and concepts.
- **Chapter 3** describes the state of the art of scheduling algorithms and cluster management systems;
- **Chapter 4** formalizes the global design of the algorithm and its logical parts;
- **Chapter 5** provides an implementation of the algorithm and the software layers that allow its working;
- **Chapter 6** analyzes the goodness of the solution found and evaluates the performance of the algorithm;
- **Chapter 7** is the thesis conclusion and enunciation of future work and research directions.

Chapter 2

Kubernetes

In this chapter we analyse Kubernetes architecture, showing also its history and evolution through 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, hence we only provide here a description of its main concepts and components. Further details can be found in the official documentation [5].

The chapter continues with an introduction to other technologies and tools used to develop the solution, in particular **Virtual-Kubelet** [6], a project which allows to create virtual nodes with a particular behaviour, and **Kubebuilder** [7], a tool to build custom resources.

2.1 Kubernetes: a bit of history

Around 2004, Google created the **Borg** [8] 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" [8].

In 2013 Google announced **Omega** [9], 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".

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) [10]. Since then, Kubernetes has significantly 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 [11, 12].

2.2 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 figure 2.1.

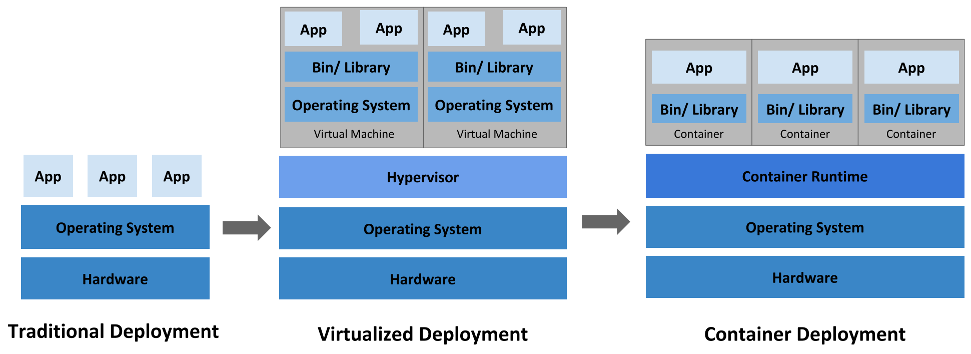


Figure 2.1: Evolution in applications deployment.

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 it is possible to 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 been proposed recently is **containerization**. Containers are similar to VMs, but they share the operating system with the host machine, 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 easily moved across different machines. Moreover, being "lightweight", containers are much faster than virtual machines: they can be booted, started, run and stopped with little effort and in a short time.

2.3 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. As depicted in figure 2.2, Kubernetes is by far the most used container orchestrator. We provide a description of such system in the following.

Kubernetes provides many services, including:

- **Service discovery and load balancing** A container can be exposed using the DNS name or using its own IP address. If traffic to a container is high, a load balancer able to distribute the network traffic is provided.
- **Storage orchestration** A storage system can be automatically mounted, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** The desired state for the deployed containers can be described, and the actual state can be changed to the desired state at a controlled rate. For example, it is possible to automate the

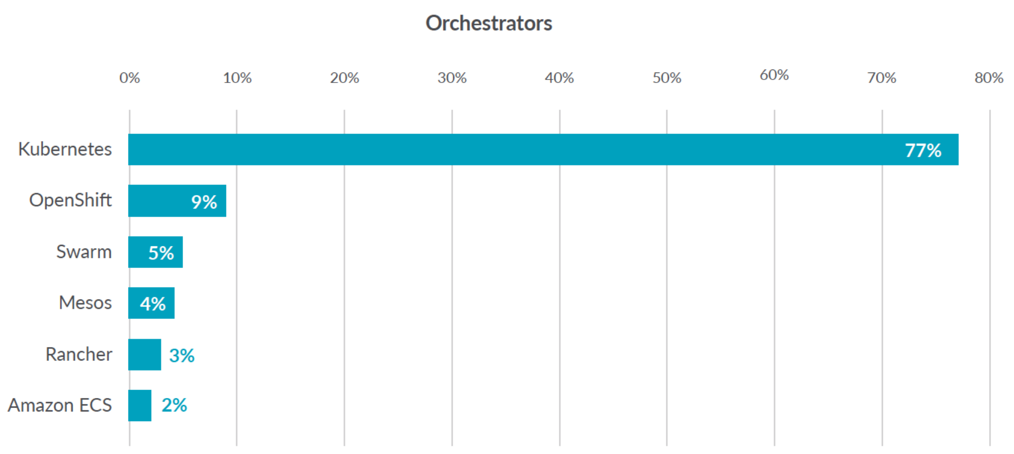


Figure 2.2: Container orchestrators use [13].

creation of new containers of a deployment, remove existing containers and adopt all their resources to the new container.

- **Automatic bin packing** Kubernetes is provided with a cluster of nodes that can be used to run containerized tasks. It is possible to set how much CPU and memory (RAM) each container needs, and automatically the containers are sized to fit in the nodes to make the best use of the resources.
- **Secret and configuration management** It is possible to store and manage sensitive information in Kubernetes, such as passwords, OAuth tokens, and SSH keys. It is possible to deploy and update secrets and application configuration without rebuilding the container images, and without exposing secrets in the stack configuration.

2.4 Kubernetes architecture

When Kubernetes is deployed, a cluster is created. 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**. Its role is to manage 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 on multiple nodes, providing fault-tolerance and high availability.

Figure 2.3 shows the diagram of a Kubernetes cluster with all the components linked together.

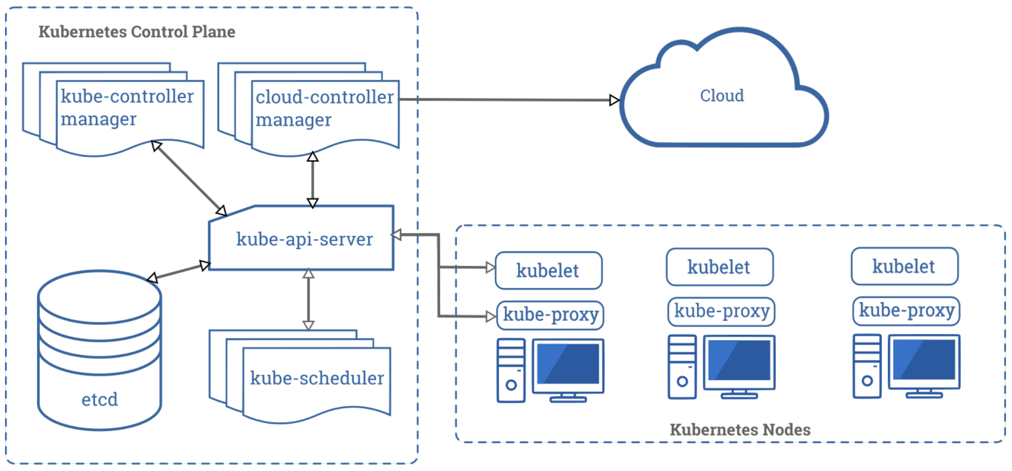


Figure 2.3: Kubernetes architecture

2.4.1 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). Although they can be run on any machine in the cluster, for simplicity, they are typically executed all together on the same machine, which does not run user containers.

API server

The API server is the component of the Kubernetes control plane that exposes the Kubernetes REST API, and constitites 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 [14], 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

The scheduler is the control plane component responsible of assigning the pods to the nodes. 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.

2.4.2 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.

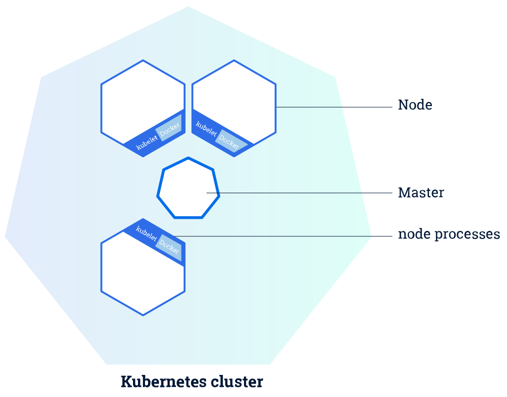


Figure 2.4: Kubernetes master and worker nodes [5].

2.5 Kubernetes objects

Kubernetes defines several types of objects, which constitutes its building blocks. Usually, a K8s resource object contains the following fields [15]:

- `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.;
- `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; once a resource is created, the system applies the desired state.
- **Read**: comes with 3 variants
  - **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 illustrate the main objects needed in the next chapters.

2.5.1 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.

2.5.2 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.

2.5.3 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.

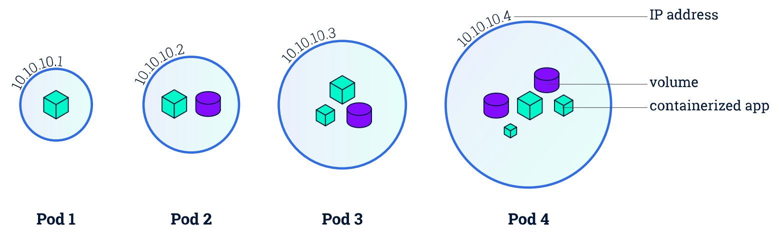


Figure 2.5: Kubernetes pods [5].

2.5.4 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**.

2.5.5 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. The listing 2.1 is an example of deployment.

Listing 2.1: Basic example of Kubernetes Deployment [5].

```
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
```

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.

2.5.6 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 contacting `<NodeIP>:<NodePort>`;
- **LoadBalancer**: exposes the Service externally using a cloud provider's load balancer;
- **ExternalName**: maps the Service to an external one so that local apps can access it.

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.

Listing 2.2: Basic example of Kubernetes Service [5].

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

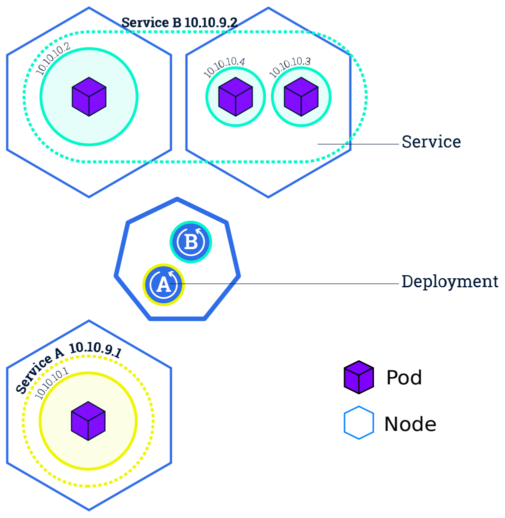


Figure 2.6: Kubernetes Services [5].

2.6 Virtual-Kubelet

Two Kubernetes-based tools which have been used during the development of this project 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 [6]. Virtual Kubelet is a Cloud Native Computing Foundation sandbox project.

The project offers a provider interface that developers need to implement in order to use it. The official documentation [6] 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”.

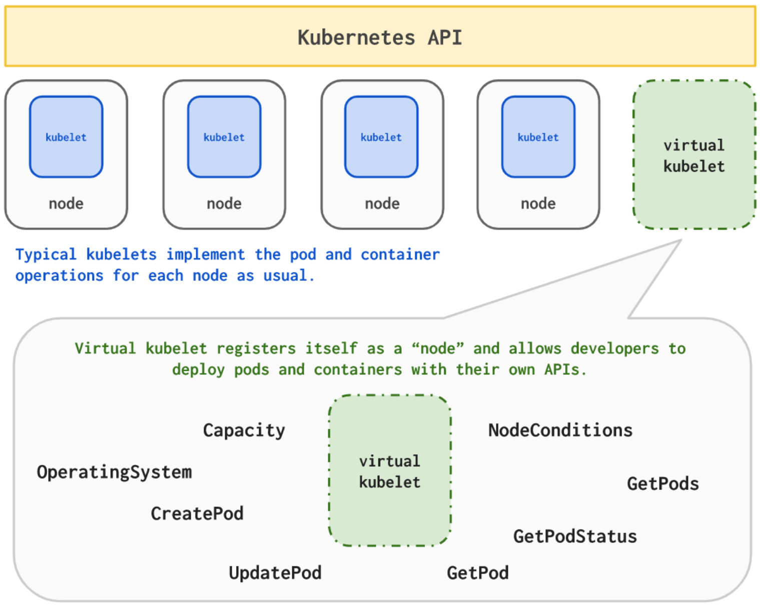

Kubernetes API
kubelet
node
kubelet
node
kubelet
node
kubelet
node
virtual kubelet
Typical kubelets implement the pod and container operations for each node as usual.
Virtual kubelet registers itself as a "node" and allows developers to deploy pods and containers with their own APIs.
Capacity
OperatingSystem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

Figure 2.7: Virtual-Kubelet concept [6].

2.7 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 a new CustomResourceDefinition is created, 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 [5]. 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 [16].

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 [7]:

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.

Chapter 3

Scheduling in clustered cloud: state of the art

Cloud computing manages a variety of virtualized resources, which makes scheduling a critical component. In the cloud, a client may utilize several thousand virtualized assets for every task. Consequently, manual scheduling is not a feasible solution. The basic idea behind task scheduling is to slate tasks to minimize time loss and maximize performance [17].

If a complex software is developed to be cloud-native (i.e. split up in multiple micro-services), it can be handled as a DAG. Independent tasks in a DAG can be executed by multiple VMs simultaneously, while related tasks will need to be executed in the correct sequential orders. The scheduling of multiple tasks to obtain the minimum makespan is a NP-complete problem, and so one can only hope for near-optimal task scheduling without performing exhaustive search.

3.1 Scheduling algorithms

There are two different types of cloud scheduling:

- static scheduling: in static scheduling all informations are known to scheduler about tasks and resources before execution. It has less runtime overhead;

- Dynamic scheduling: in dynamic scheduling information about task components is not known before execution. Task execution time may not be known. It has more runtime overhead.

Cost-Based scheduling

Cost based scheduling algorithms [18] are dedicated to finding efficient task-scheduling mechanisms for producing near-minimum makespan, while taking into account the requested monetary charges. Activity-based costing is a way of measuring both the cost of the resources and computation performance. In cloud computing, each application runs on a virtual system, where the resources are distributed virtually. In order to measure direct costs of applications, every individual use of resources (like CPU cost, memory cost, Input/output cost etc.) must be measured.

Earliest Deadline First scheduling

Earliest deadline first (EDF) [19] [20] is a dynamic priority scheduling algorithm. EDF selects tasks by giving priority to those with earlier deadlines. It means that the priority of a task is inversely proportional to its absolute deadline. A task which has a higher priority due to earliest deadline at a particular instant may have low priority at subsequent time instants due to the early deadline of another task. EDF typically executes in preemptive mode, i.e., currently executing task is preempted whenever another task with an earlier deadline becomes ready. EDF is an optimal algorithm which means if a task set is feasible then it is surely scheduled by EDF. Additonally, EDF does not specifically takes any assumption on periodicity of tasks.

Queue Based Job Scheduling

In this algorithm [21] the whole system is modeled with a queuing network. In the first stage, the Queue based Decision maker (QDM) unit calculates the probability of assigning each task to a specific cloudlet or public cloud, with the aim to reduce the average response time (the first objective). All the users' requests and initial queue and capacity of cloudlets/clouds are used as context information for this unit. The functionality of this unit is based on a model driven from queue theory. Solving the queue model by different number of tasks and extracting the best possible task assignment probabilities for each cloudlet/cloud has a large search space. The output of the QDM unit and communication duration between each user and cloudlets/clouds are inputs of the Ant Colony Optimization Decision Maker (ACODM) unit. The computed assignment probabilities from the QDM unit only minimizes the response time of each assignment, hence, the ACODM unit, uses the previously found assignment probabilities as heuristic information for the Ant Colony Optimization (ACO) algorithm. The ACODM unit finds better solution by considering communication duration in a way that the overall offload service duration of the MCC system gets minimized (second objective).

Priority Based scheduling

In these algorithms [22], each job requests a resource with determined priority. So comparison matrices for every jobs, according to resources accessibilities are computed, as well as comparison matrix of resources. For each of the comparison matrices, priority vectors (vector of weights) are computed and finally a normal matrix of all jobs named as Δ is created. The normal matrix γ of all resources is also computed. The next step of the algorithm is to compute Priority Vector of S (PVS), where S is the set of jobs. PVS is calculated by multiplying matrix Δ with matrix γ. The final step of the algorithm is to choose the job with maximum calculated priority, so a suitable resource is allocated to that job.

Greedy scheduling

These algorithms [23] formulate the assignments of tasks to servers as an integer-programming problem with the objective of minimizing the energy consumed by the servers of the data center. The researchers propose the most-efficient-server-first (MESF) task-scheduling scheme to minimize the energy consumption while keeping the response within a constrained time. Here, a task is a request for a job belonging to an application that may require a defined amount of resources and the creation of a VM to support the application. The job may be data transmission (uploading and downloading), data processing, software access and execution, or storage functions. Each task, as the corresponding VM, is then assigned to one of the available servers.

PSO-MA scheduling

The PSO-MAs (Particle Swarm Optimization-Memetic Algorithms) [24] are used for the permutation flow shop scheduling problem (PFSSP) with the objective to minimize the maximum completion time, which is a typical non-deterministic polynomial-time (NP) hard combinatorial optimization problem. In these algorithms, both PSO-based searching operators and some special local searching operators are designed to balance the exploration and exploitation abilities. In particular, the PSOMA applies the evolutionary searching mechanism of PSO, which is characterized by individual improvement, population cooperation, and competition to effectively perform exploration. On the other hand, the PSOMA utilizes several adaptive local searches to perform exploitation.

Min-min and max-min Scheduling

These algorithms [25] are batch-mode heuristics, where tasks are collected into sets called meta-tasks (MT). These sets are mapped at pre-scheduled times called mapping events. Min-Min begins with the set MT of all unassigned tasks. It has two phases. In the first phase, the set of minimum expected completion time (such that task has the earliest expected completion time on the corresponding machine) for each task in MT is found. In the second phase, the task with the overall minimum expected completion time from MT is chosen and assigned to the corresponding resource. Then this task is removed from MT and the process is repeated until all tasks in the MT are mapped. Max-Min is very similar to Min-Min, except for the second phase. Indeed, Max-Min assigns the task with maximum expected completion time to the corresponding resource..

3.2 Cluster scheduling

With the rise of elastic compute resources, cluster management has become an increasingly hot topic in system research and deveploment, and a number of competing cluster managers including Kubernetes, Mesos, and Docker are currently jockeying for the crown in this space [26]. Increasingly, many applications and websites rely on distributed back ends running in cloud data centers. In these data centers, clusters of hundreds or thousands of machines run workloads ranging from fault-tolerant, load-balanced web servers to batch data-processing pipelines and distributed storage stacks.

Current cluster schedulers rely on heuristics that prioritize generality, ease of understanding, and straightforward implementation over achieving the ideal performance on a specific workload. Cluster schedulers must meet a number of goals simultaneously: high resource utilization, user-supplied placement constraints, rapid decision making, and various degrees of “fairness” and business importance, all while being robust and always available.

3.2.1 Mesos

Mesos [27] was the first academic publication on modern cluster management and scheduling. The implementation is open source and appeared at a time when Borg was still unknown outside Google. The Mesos authors had the key insight, later reaffirmed by the Borg paper, that dinamically sharing the underlying cluster among many different workloads and frameworks (e.g., Hadoop, Spark, and TensorFlow) is crucial to achieving high resource utilization. Different frameworks often have different ideas of how independent work units (tasks) should be scheduled; indeed, the frameworks that Mesos targeted initially already had their own schedulers. To arbitrate resources among these frameworks without forcing them into the straitjacket of a single scheduling policy, Mesos neatly separates two concerns: a lower-level resource manager allocates resources to frameworks (e.g., subject to fairness constraints), and the higher-level framework schedulers choose where to run specific task (e.g., respecting data locality preferences). An important consequence of this design is that Mesos can support long-running service tasks just as well as short, high-throughput batch-processing ones (they are simply handled by different frameworks).

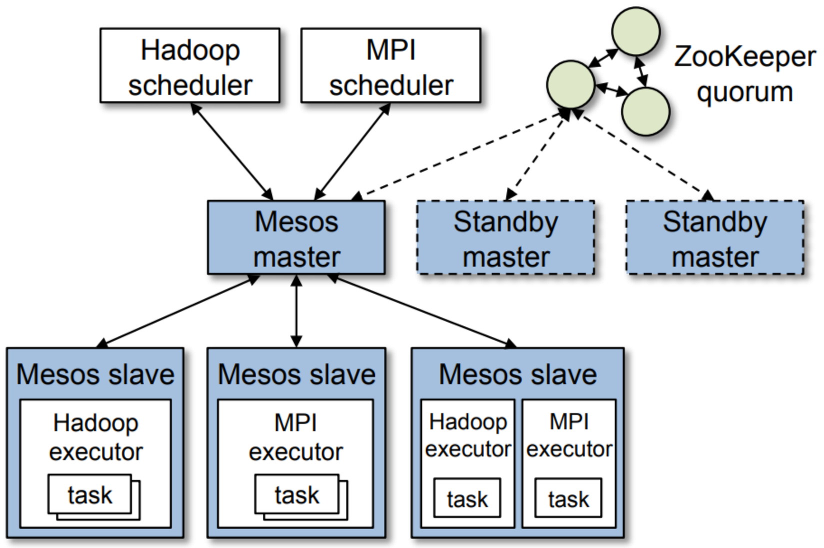


Figure 3.1: Mesos architecture

Mesos avoids having a complex API for frameworks to specify their resource needs. It does this by inverting the interaction between frameworks and the resource manager: instead of frameworks requesting resources, the Mesos resource manager offers resources to frameworks in turn. Each framework takes its pick, and the remaining resources are subsequently offered to the next one. Using appropriately sized offers, Mesos can also enforce fairness policies across frameworks, although this aspect has in practice turned out to be less important than perhaps originally anticipated. The Mesos offer mechanism is somewhat controversial: yet, Mesos's multischeduler design has been quite impactful: many other cluster managers have since adopted similar architectures, though they use either request-driven allocation (e.g., Hadoop YARN) or an Omega-style shared-state architecture (e.g., Microsoft's Apollo and HashiCorp's Nomad). Another deliberate consequence of the offer-driven design is that the Mesos resource manager is fairly simple, which improves its scalability.

3.2.2 Omega

Increasing scale and the need for rapid response to changing requirements are hard to meet with current monolithic cluster scheduler architectures. This restricts the rate at which new features can be deployed, decreases efficiency and utilization, and will eventually limit cluster growth. Omega [9] has been presented as a novel approach to address these needs using parallelism, shared state, and lock-free optimistic concurrency control. The Omega researchers identified the two prevalent scheduler architectures shown in Figure 3.2. Monolithic schedulers use a single, centralized scheduling algorithm for all jobs. Two-level schedulers have a single active resource manager that offers compute resources to multiple parallel, independent "scheduler frameworks", as in Mesos and Hadoop-on-Demand (YARN).

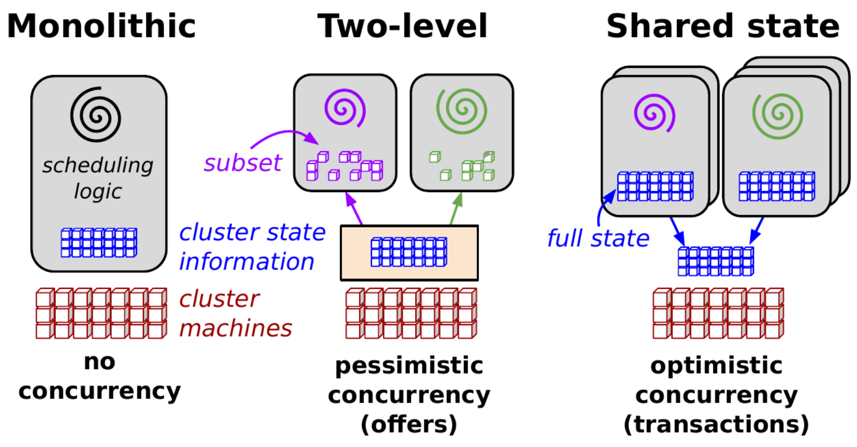

Monolithic
Two-level
Shared state
scheduling logic
subset
cluster state information
cluster machines
full state
no concurrency
pessimistic concurrency (offers)
optimistic concurrency (transactions)

Figure 3.2: scheduling architectures

Monolithic schedulers do not make it easy to add new policies and specialized implementations, and may not scale up to the cluster sizes the big companies were planning for. Two-level scheduling architectures appear to provide flexibility and parallelism, but in practice their conservative resource-visibility and locking algorithms limit both, and make it hard to place difficult-to-schedule "picky" jobs or to make decisions that require access to the state of the entire cluster. One of the proposed solutions is a new parallel scheduler architecture built around shared state, using lock-free optimistic concurrency control, to achieve both implementation extensibility and performance scalability.

The Omega paper observes that Mesos must offer all cluster resources to every framework to expose the full knowledge of the existing state (including, for example, preemptible tasks), but that a slow framework scheduler can adversely affect other frameworks and overall cluster utilization in the absence of optimistic parallel offers and a conflict-resolution mechanism. In response, the Mesos developers have devised concepts for such extensions to Mesos.

The Omega approach builds on many prior ideas. Scheduling using shared state is an example of optimistic concurrency control, which has been explored by the database community for a long time, and, more recently, considered for general memory access in the transactional memory community.

3.2.3 Sparrow

Even within what Mesos considers framework-level tasks, there may be another level of scheduling. Specifically, some data-analytics systems break their processing into many short work units. Spark, for example, often generates application-level "tasks" that run for only a few hundred milliseconds. Using such short tasks has many benefits: it implicitly balances load across the workers that process them; failures lose only a small amount of state; and straggler tasks that run much longer than others have smaller absolute impacts. Shorter tasks, however, impose a higher load on the scheduler that places them. In practice, this is usually a framework-level scheduler, or a scheduler within the job itself (as in standalone Spark). With tens of thousands of tasks, this scheduler might get overwhelmed: it might simply be unable to support the decision throughput required. Indeed, the research shows that the centralized application-level task scheduler within each Spark job scales to only about 1,500 tasks per second. Queueing tasks for assignment at a single scheduler hence increases their overall "makespan", as well as leaving resources idle while they await new tasks to be assigned by the overwhelmed scheduler. Sparrow [28] addresses this problem in a radical way: it builds task queues at each worker and breaks the scheduler into several distributed schedulers that populate these worker-side queues independently. Using the "power of two random choices" property, which says that (under certain assumptions) it suffices to poll two random queues to achieve a good load balance, Sparrow then randomly places tasks at workers. This requires neither state at the scheduler, nor communication between schedulers (and, hence, scales well by simply adding more schedulers). This research includes several important details that make the random-placement approach practical.

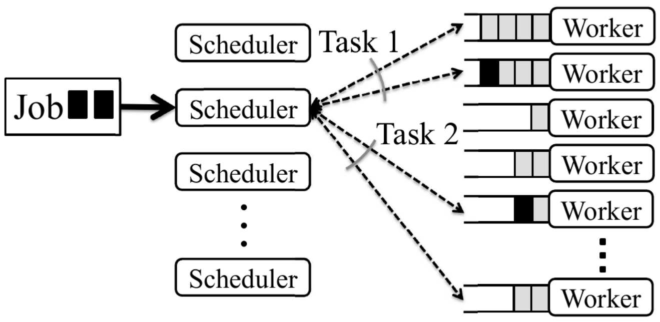


Figure 3.3: sparrow queues

Sparrow could in principle be used as a cluster-level scheduler, but in practice works best load-balancing application-level tasks over long-running workers of a single framework (which, for example, serves queries or runs analytics jobs). At the cluster-scheduler level, the task startup overhead normally makes tasks below tens of seconds in duration impractical because of package distribution, container launch, etc. Consequently, the open-source Sparrow implementation supplies a Spark application-level scheduler plug-in. Finally, while Sparrow's randomized, distributed decisions are scalable, they assume that tasks run within fixed-size resource slots and that queues of equal length amount to equally good choices. Several follow-up papers address some of these issues while maintaining the same distributed architecture for scalability and fault tolerance.

3.2.4 Borg

Borg [8] handles all aspects of cluster orchestration: it monitors the health of machines, restarts failed jobs, deploys binaries and secrets, and oversubscribes resources just enough to maintain key SLOs (Service Level Objectives), while also leaving few resources to sit idle. To achieve this, the Borg developers had to make many decisions: from choosing an isolation model (Google uses containers), to how packages are distributed to machines (via a torrent-like distribution tree), how tasks and jobs find each other (using a custom DNS-like naming system), why and how low-priority and high-priority workloads share the same underlying hardware (a combination of clever oversubscription, priority preemption, and a quota system), and even how to handle failures of the Borgmaster controller component (Paxos-based failover to a new leader replica). There is a treasure trove of neat tricks here (e.g., the automated estimation of a task's real resource needs in §5.5), as well as a ton of operational experience and sound distributed-system design. A related Queue article describes how Borg and Omega, also developed at Google, have impacted Kubernetes, developed with substantial inspiration from Borg.

3.2.5 Firmament

Distributed decisions brings benefits in scalability and fault tolerance, but should be taken in the presence of reduced (and only statistically sampled) information about cluster state. Centralized scheduler, which make all decisions in the same place, have the information to apply more sophisticated algorithms (for example, to avoid overloaded machines). Of course, this applies both at the cluster level and to application-level schedulers. The paper sets out to investigate whether fault-tolerance benefits not withstanding — distribution is indeed necessary for scalability. It notes that it is crucial to amortize the cost of decisions over many tasks, especially if the scheduler supports features that require reconsidering the existing placements, such as task preemption. Indeed, if the scheduler picked a

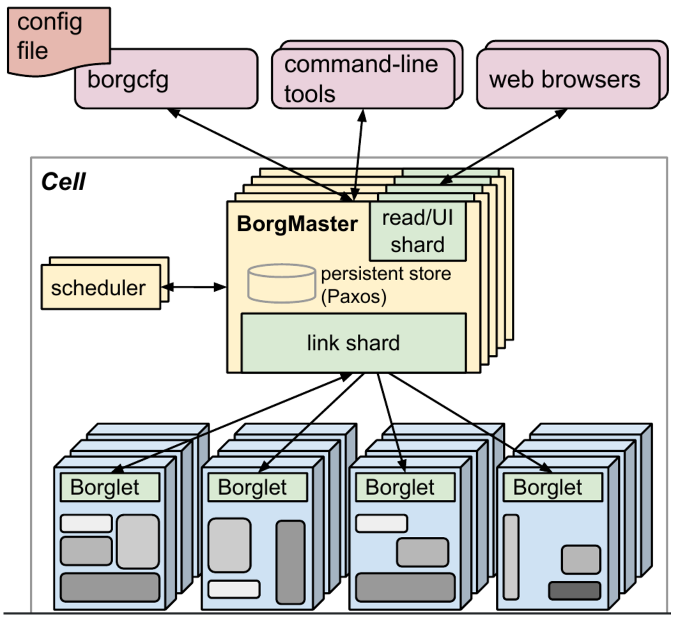


Figure 3.4: The high-level architecture of Borg

task off a queue, looked at a whole bunch of machines, and made some complex calculations just to decide where to put a single task, it probably would not scale well. Firmament [29] generalizes the Quincy scheduler, a system that uses a min-cost, max-flow constraint solver to schedule batch workloads. Because min-cost, max-flow solvers are highly optimized, their algorithms amortize the work well over many tasks. Applied naively, however, the Quincy approach cannot scale to large workloads over thousands of machines the constraint-solver runtime, which dominates scheduling latency, becomes unacceptably high.

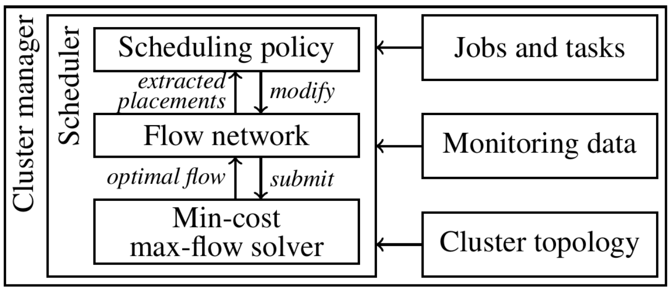


Figure 3.5: Firmament structure

To address this issue, Firmament concurrently runs several min-cost, max-flow algorithms with different properties and solves the optimization problem incrementally if possible, refining a previous solution rather than starting over.

With some additional optimizations, Firmament achieves subsecond decision times even when scheduling a Google-scale cluster running hundreds of thousands of tasks. This allows Sparrow-style application-level tasks to be placed within hundreds of milliseconds in a centralized way even on thousands of machines. The paper also shows that there is no scalability-driven need for distributed cluster-level schedulers, as Firmament runs a 250-times-accelerated Google cluster trace with median task runtimes of only four seconds, while still making subsecond decisions in the common case. The simulator and Firmament itself are open-source, and there is a plug-in that allows Kubernetes to use Firmament as a scheduler. The Firmament paper suggests that decision quality needs not to be compromised to solve a perceived scalability problem in cluster scheduling. Nevertheless, Sparrow- style distributed schedulers are useful in several use cases: for example, a fault-tolerant application-level load balancer that serves a fixed set of equally powerful workers might well use Sparrow's architecture.

3.3 Research to practice

Many companies already run their own installations of Mesos or Kubernetes on clusters of VMs provisioned on the cloud or on machines on their own premises. The problem of scaling cluster managers and their schedulers to very large clusters, however, is one that most readers will not have to face: only a few dozen companies run such large clusters, and buying resources on AWS (Amazon Web Services) or Google's or Microsoft's clouds is the easiest way to scale. Yet, scheduler scalability can also be an issue in smaller clusters: if you are running interactive analytics workloads with short tasks, a scalable scheduler may give you better resource utilization. Another important aspect is that cluster workloads are quite diverse, and scheduling policies in practice often require substantial hand-tuning using placement constraints. Indeed, this is what makes cluster scheduling different from the multiprocessor scheduling performed by the kernel in OSs: while most applications are fine with the general-purpose policies the kernel applies to assign processes to cores, current cluster-level placement policies do not work well for all workload mixes without manual operator help.

3.4 Machine learning contributions

Modern Machine Learning (ML) techniques help to automatically learn highly efficient, workload-specific scheduling policies. Many systems encode job stages and their dependencies as Directed Acyclic Graphs (DAGs). Efficiently scheduling DAGs leads to hard algorithmic problems whose optimal solutions are intractable. Given only a high-level goal (e.g., minimize average job completion time), ML can use existing monitoring information and past workload logs to automatically learn sophisticated scheduling policies. For example, instead of a rigid fair sharing policy, ML systems learn to give jobs different shares of resources to optimize overall performance, and learn job-specific parallelism levels that avoid wasting resources on diminishing returns for jobs with little inherent parallelism. The right algorithms and thresholds for these policies are workload-dependent, and achieving them today requires painstaking manual scheduler customization.

Chapter 4

Job Scheduling algorithm design

This thesis aims at creating a new scheduler algorithm for Kubernetes, in order to optimize the scheduling of multiple jobs across a set of federated clusters. Each Kubernetes cluster belonging to the federation will have its own scheduler, able to interpret advertisement messages coming from other federated clusters. Starting from the advertisement messages, the scheduler builds a representation of its own view of the federation. Whenever the cluster manager (or one of his customers) wants to deploy a set of dependent jobs in the federation, the scheduler takes the request and finds a good scheduling solution, by taking into account multiple scheduling policies, with the objective to perform a scheduling operation by ensuring a low monetary charge on the offloading entity.

4.1 The problem

The goal of the scheduler is to pair a set of jobs belonging to a deployment with a Kubernetes cluster where the job will run. The pairing between federation clusters and deployment jobs is affected by various policies, such as constraints in terms of computation and memory capabilities for nodes, and latency and bandwidth capabilities for edges. Hence, a node can be scheduled in a cluster only if the cluster satisfies the constraints of the job, and the edge between two clusters satisfies the inter-connectivity constraints posed by the dependency between the jobs scheduled on them. For the sake of simplicity and clarity, all the elements of the graphs (both nodes and edges) used in this section have only *one* label that represents a generic resource constraint (for the nodes) or an inter-connectivity constraint (for the edges). In chapter 5 we will describe an implementation of the proposed algorithm that also takes into account multi-dimensional label sets.

4.1.1 The deployment graph

The deployment of an application is represented by an undirected graph, a pair $D = (N_D, E_D)$, where N_D is a set of vertices representing jobs to be deployed, and E_D is a set of edges representing dependencies between them. A dependency between two jobs n_i, n_j represents the necessity for them to communicate to each other.

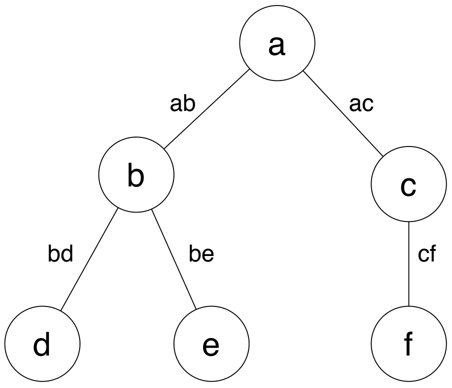


Figure 4.1: deployment graph

Each node $n \in N_D$ of the deployment graph is labelled by the label w_n (node weight) representing the amount of resources required by the job n to be properly executed.

Each edge $e \in E_D$ of the deployment graph is labelled by the label k_e (edge weight) representing the amount of inter-connectivity resources required by the job pair (i, j) joint together by e to comply with some QoS policies.

4.1.2 The federation graph

The federation infrastructure is represented by an undirected graph, a pair $F = (N_F, E_F)$, where N_F is a set of nodes representing Kubernetes clusters and E_F is a set of edges representing inter-connections between them.

Since the scheduling algorithm is applied in each cluster for scheduling its own applications, the topology of the graph is always *star-shaped*: each Kubernetes cluster receives the advertisement from each other cluster in the federation and from its point of view the resulting topology is a star with itself as the center. Even though from the architectural point of view the topology of the graph is star shaped, it may actually be *fully meshed*, since each cluster should be able to communicate *directly* to each other cluster in the federation (e.g., jobs b and d in fig. 4.1 must be able to communicate even if they are in different clusters). This goal is reached by considering that a virtual path between two edges e_i, e_j can be

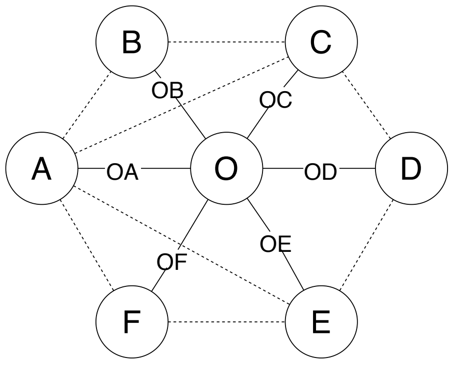


Figure 4.2: federation graph

established through the star center, to model the fact that a path between any Kubernetes clusters should exists and the above two edges can communicate to each other. By doing this assumption, the graph topology becomes virtually fully meshed, and the scheduler assumes that each Kubernetes cluster is connected to each other. As it is depicted in fig. 4.2, each actual connection is represented by a solid line, creating a star topology, whereas the routed connections are represented by a dotted line (the drawing is not completely connected for the sake of clarity and semplicity).

The scheduler assumes that each cluster can communicate with each other, and therefore it assumes a fully meshed topology.

Each node $n \in N_F$ of the deployment graph is labelled by two different labels:

- r_n (node resource) represents the quantity of resources make available by the cluster n to the federation;
- c_n (node cost) represents the price asked by the cluster n to host a job offloaded by another cluster of the federation. The cost is per resource unit.

Each edge $e \in E_D$ of the deployment graph is labelled by two different labels:

- l_e (link capability) representing the amount of inter-connectivity resources required by the job pair (i, j) joint together by e to comply with some QoS policies;
- c_e (link cost) represents the price asked by the edge e to to be traversed. It is a one-off price.

4.2 Data structures

In order to explain how the algorithm works, it is necessary to introduce three different data structures, the Sorted Jobs array (SJ), the Cost Matrix (CM), and the Node Affinity Matrix (NAM). All these matrices are related to a scheduling attempt, i.e., a tentative of pairing each deployment job with a federation Kubernetes cluster, hence they are instantiated and initialized with the proper values when a scheduling operation must be performed.

4.2.1 SJ: Sorted Jobs

The sorted jobs data structure is an array that contains all the jobs sorted according to their requested amount of resources summed to their neighborhood amount. The idea is to give higher scheduling priority to jobs that likely have the most impacting constraints, indeed, our heuristic attempts to assign the greatest amount of resources on the cheapest nodes. The SJ array is computed as in algorithm 1.

Algorithm 1 create SJ

```
1: function CREATESJ($N_D$)
2:     $SJ = map(node, float)$
3:     for each $i \in N_D$ do
4:         $y_i = w_i$
5:         for each $n \in$ Neighborhood($i$) do
6:             $y_i = y_i + w_n$
7:         end for
8:         $SJ[i] = y_i$
9:     end for
10:    sortDescending($SJ$).byValue()
11:    return $SJ$.values()
12: end function
```

4.2.2 CM: Cost Matrix

The Cost Matrix CM (depicted in figure 4.3) is a matrix n x N, where $n = |N_D|$ and $N = |N_F|$. The content of each cell $c_{ij} = CM[i][j], c_{ij} \in \Re^+$ represents the cost for the job $i \in N_D$ to be hosted by the cluster $j \in N_F$. The cost is computed by taking into account:

- the cost required by the node $j \in N_F$ to host the job $i \in N_D$ (considering that the cost requested by a cluster is given per unit);
- the cost necessary to use the links required by the inter-jobs communication, considering the jobs already scheduled in other Kubernetes clusters (e.g. in fig. 4.1 if the job a has already been paired with cluster $j \in N_F$, when the job

b has to be paired, it is necessary to take into account the cost of the edge connecting j and the candidate clusters).

The Cost Matrix is initialized with the price requested by the cluster $j \in N_F$ to host the job $i \in N_d$, by taking into account that the job i has an amount of requested resources and the cluster j sets a price per resource unit. At each iteration of the scheduling algorithm (i.e. whenever a pairing between a job and a cluster is performed) the matrix is updated according to algorithm 2.

Algorithm 2 CM update

```
1: function UPDATECM($n$, $SJ$, $m$, $N_F$)
2:     $K$ = neighborhood($SJ[n]$)
3:     for each $i \in K$ do
4:         for each $j \in N_F$ do
5:             CM[i][j] = CM[i][j] + $l(j, m)$
6:         end for
7:     end for
8: end function
```

	O	A	B	...	N
a	c_{aO}	c_{aA}	c_{aB}	...	c_{aN}
b	c_{bO}	c_{bA}	c_{bB}	...	c_{bN}
...	...	...	...	...	...
n	c_{nO}	c_{nA}	c_{nB}	...	c_{nN}

Figure 4.3: Cost Matrix (CM)

4.2.3 NAM: Node Affinity Matrix

Another data structure required by the algorithm is the Node Affinity Matrix (NAM). This matrix, depicted in fig. 4.4, is a matrix n x N, where $n = |N_D|$ and $N = |N_F|$. The content of each element $f_{ij} = FM[i][j], x \in]0, 1]$ represents the affinity that the job i has with the cluster j.

The values computation for those matrices is based on clusters resources: given clusters capabilities, we compute a value $x \in \Re$ (explained below) representing the affinity between a job and a cluster, then, the Φ function is used to remap it in the interval]0,1]; Φ is defined as follows:

$$\Phi(x) = \begin{cases} 1 & \text{if } x \leq 0 \\ e^{-ax} & \text{if } x > 0 \end{cases}$$

	O	A	B	...	N
a	f_{aO}	f_{aA}	f_{aB}	...	f_{aN}
b	f_{bO}	f_{bA}	f_{bB}	...	f_{bN}
...	...	...	...	...	...
n	f_{nO}	f_{nA}	f_{nB}	...	f_{nN}

Figure 4.4: Node Affinity Matrix (NAM)

The parameter a can be used to tune $\Phi(x)$ in order to have a variable slope. Fig. 4.5 shows how $\Phi(x)$ changes to varying of a.

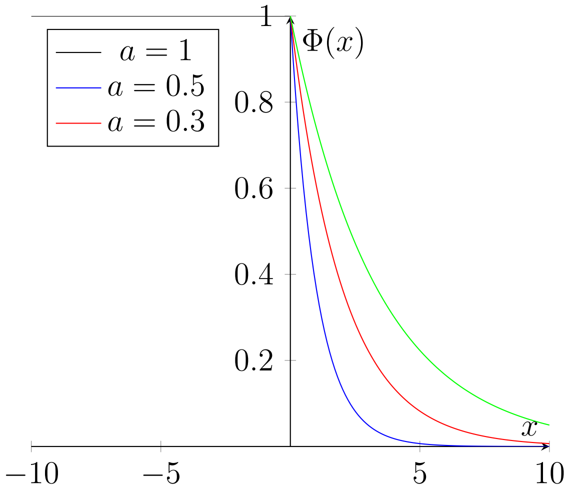


Figure 4.5: $\Phi(x)$ chart to vary of a

The affinity that each job $i \in N_D$ has with each cluster $j \in N_F$, is computed by taking into account the cluster size: for each job $i \in N_D$, given $N = neighborhood(i)$, the corresponding row values x_i are computed in relation to which percentage of jobs $n \in N$ the cluster $j \in N_F$ is able to host, considering their required resource. The higher the value of x_i, the higher the possibility that the cluster $j \in N_F$ is a good candidate to host the job $i \in N_D$. Once this value x_i has been computed, if $x_i \neq 1$, the affinity is mapped in the range $(0, 1]$ by means of $\phi(x)$, otherwise it is set to 1. In algorithm 3 the value x_i is computed as $y_i - r_j$, where:

- y_i is the total amount of resource required by job i and all its neighborhood;
- r_j is the quantity of resource owned by the cluster j.

Hence, the best possible affinity is 1 ($r_j > y_i$), and the worst one 0 (possible only if the cluster j has no resources for hosting job i).

At each stage of the recursive algorithm, the Node affinity matrix has to be re-computed with the new affinity values.

Algorithm 3 create NAM

```
1:  function createNAM($N_D$, $N_F$)
2:      $NAM = float[|N_D|][|N_F|]$
3:      for each $i \in N_D$ do
4:          $y_i = w_i$
5:          for each $n \in$ Neighborhood($i$) do
6:              $y_i = y_i + w_n$
7:          end for
8:          for each $j \in N_F$ do
9:              if $w_i > r_j$ then
10:                 $NAM[i][j] = 0$
11:             else
12:                 $NAM[i][j] = \Phi(y_i - r_j)$
13:             end if
14:         end for
15:     end for
16:     return $NAM$
17: end function
```

4.2.4 PA: Priority Array

The Priority Array (PA) is computed at each stage of the recursive function: its size is the same of N_F and is computed as follows: given the node affinity matrix and the cost matrix, each value PA_i is got by modifying the cost matrix values according to the affinity that each job $i \in N_D$ has with each cluster $j \in N_F$. The modification is applied by dividing each cost by the affinity values (a perfect affinity has a score of 1, therefore an affinity $a < 1$ increases the cost required to the node i to be hosted by the cluster j).

Every time a new element of the PA array is computed, it is necessary to verify whether the position of the job i on the cluster j does not break any constraint. This is verified by the function in algorithm 5. This function gets the current job and a candidate federation node, then iterates over all the solution nodes and checks whether there is an already scheduled job such that a connectivity constraint would be broken if the current job was placed on the candidate federation node.

Algorithm 4 create PA

```
function createPA($i$, $N_D$, $N_F$, $NAM$, $EAM$, $S$)
    $PA_i = N_F$
    for each $j \in N_F$ do
        if $NAM[i][j]$ != 0 then
            r = checkConnectivityConstraints($N_D[i]$, $N_F[j]$)
            if r == true then
                $D_i[j] = CM[i][j]/NAM[i][j]$
            end if
        end if
    end for
    return $PA_i$
end function
```

Algorithm 5 check connectivity constraints

```
function checkConnectivityConstraints($n$, $m$, $S$)
    for each $dn \in S$ do
        $ed$ = EdgeBetween($n$, $dn$)
        $ef$ = EdgeBetween($m$, $dn.associatedFedEdge$)
        if $e$ == null then
            return false
        end if
        if $ef$.canHost($ed$) == false then
            return false
        end if
    end for
    return true
end function
```

4.3 Algorithm

The brute-force algorithm that aims to find the best possible solution is based on a recursion that explores the whole space of the possibilities. At each state of the recursive process a new node $i \in N_D$ is evaluated in all the possible offloading positions $j \in N_F$, i.e. all possible matches between jobs and clusters are tried.

Given the complexity of the problem, such approach is nonviable. Pruning unfeasible branches in the solution space could help shortening the execution times. However, in order to obtain reasonable scheduling times, not all the feasible solutions could be explored. We propose an heuristic that optimizes such exploration so that the first feasible solution found is likely to be a good one. This also requires 'good' to be defined.

At each recursion state the match between the job i and the cluster j is performed by following the algorithm 7. A wrapper that initializes all the data structures and calls the recursive function is listed in algorithm 6.

4.3.1 Parameters

The wrapper function creates some parameters and pass them to the recursive function. They are listed below:

- S_J: sorted array of the jobs to be deployed
- N_F: array of the federation clusters;
- NAM: Node Array Matrix;
- CM: Cost Matrix;

The recursive function makes use of some new variables, in addition to the ones specified by the wrapper function:

- i: the current job to be deployed;
- S: current solution

Algorithm 6 Main algorithm

```
1: $SJ$ = CREATESJ($N_D$)
2: $NAM$ = CREATENAM($N_D$, $N_F$)
3: $CM$ = initCM($N_D$, $N_F$)
4: schedule(0, $SJ$, $N_F$, $CM$, $NAM$, [ ])
```

Algorithm 7 recursive function

```
 1: function SCHEDULE($i$, $SJ$, $N_F$, $CM$, $NAM$, $S$)
 2:     if $i == S_J$.length() then
 3:         output($S$)
 4:         $C_l = C_t$
 5:         return
 6:     end if
 7:     if $C_t > C_l$ then
 8:         return
 9:     end if
10:     $PA_i$ = CREATEPA($i$, $SJ$, $N_F$, $NAM$, $S$)
11:     sortAscending(PA)
12:     for each $f \in PA_i$ do
13:         $S = S + d$
14:         $CM_n$ = UPDATECM($i$, $S_J$, $f$, $N_F$)
15:         SCHEDULE( $i + 1$, $SJ$, $N_F$, $CM_n$, $NAM$, $S$)
16:         $S = S - d$
17:     end for
18: end function
```

4.4 Complexity evaluation

The scheduling problem is a well-known NP-complete problem. Since this algorithm stops when the first feasible solution is found, the worst case complexity is exponential as well, but the heuristic approach is structured in a way that, on the average case, a solution is found in polynomial time. This behavior is ensured by the fact that both deployment and federation nodes are sorted such that good placements are performed as first placements, leading to a feasible good solution without performing any backtracking.

Since some data structures used by the algorithm have to be created, it is also necessary to evaluate their complexity:

- SJ creation: $O(N_D)$;
- CM creation: $O(N_D * N_F)$;
- NAM creation: $O(N_D * N_F)$.

The resulting complexity of the preparatory phase is then $O(N_D * N_F)$.

In addition to the main complexity brought in by the recursive operation $O(N_D)$, the global complexity of the algorithm is also affected by the following parts:

- PA update and sorting: $O(N_F * N_F)$ (we consider insertion in sorted array);
- CM update: $O(N_F)$.

If we assume that the CM update can be packed in the PA creation operation, the resulting complexity, is then $O(N_D * N_F^2)$, that, summed to the creation operations has a global complexity of $O(N_D * N_F) + O(N_D * N_F^2)$. Hence, the global computational complexity of the algorithm, in case no backtracking occurs, is $O(N_D * N_F^2)$.

Chapter 5

Job Scheduling algorithm implementation

This scheduling algorithm is designed to work in Kubernetes for scheduling jobs across clusters belonging to the same federation. For this reason this algorithm is implemented in the Golang language: this will facilitate its deployment as Kubernetes scheduler module that will take the pods and will schedule them on the various clusters of the federation. In order to create an efficient version of the algorithm, a software layer able to provide certain functionalities has been developed. This software layer is composed by 2 main libraries:

- graph library;
- label library;

5.1 Graph library

The entities taken as input by the scheduler are the federation (a set of Kubernetes clusters) and the deployment (a set of interconnected services). Since both of them can be seen as a set of nodes interconnected by edges, they can be represented by a graph.

We wrote a library that allows to create, manage and perform a set of operation on graphs; this library defines a set of interfaces:

- Graph: the graph entity, declares all the methods for managing all the operation requested by the algorithm explained in chapter 4.
- Element: the single entity that can be added to a graph; it can be either a node or an edge and declares all the operation that are common to nodes and edges.

- Node: the node entity, declares all the operations that can be performed on a node.

- Edge: the edge entity, declares all the operations that can be performed on a edge.

5.1.1 Graph interface

Listing 5.1: graph interface

```
type Graph interface {
    NewNode(id string) Node
    AddNode(n Node)
    RemoveNode(id string)
    NewEdge(id, xid, yid string) Edge
    AddEdge(e Edge)
    RemoveEdge(id string)
    AddNodeLabel(id string, label labels.Label)
    AddEdgeLabel(id string, label labels.Label)
    Node(id string) Node
    Edge(id string) Edge
    Nodes() map[string]Node
    Edges() map[string]Edge
    EdgeBetween(xid, yid string) Edge
}
```

The Graph interface in listing 5.1 declares all the methods for adding and removing new elements, such as NewNode, that allows to create a new node belonging to the graph, AddNode, that adds a node to the graph, NewEdge, that creates a new Edge belonging to the graph, and AddEdge, for adding the Edge given as argument to the graph. Nodes and edges can also be removed by the methods RemoveNode and RemoveEdge. AddNodeLabel and AddEdgeLabel allows to attach new labels to nodes or edges. Nodes lists all the nodes of the graphs and Edges lists all the edges of the graph. Node and Edge are used to retrieve a node or an edge, given its unique string identifier as argument. Last, the EdgeBetween methods retrieves the edge interconnecting the two nodes represented by the unique identifiers given as arguments.

The actual data structures of the Graph interface have been implemented as follows.

- The adjacency matrix to get the edge interconnecting two nodes has been implemented by a map[string]map[string]Edge, that, given the two strings as the unique identifiers of the nodes, retrieves the edge interconnecting them. This implementation gives us random access to the matrix.

- The Nodes entity is represented by a map[string]Node, that gives random access to a Node, given its unique identifier.
- The Edges entity is represented by a map[string]Edge, that gives random access to an Edge, given its unique identifier.

5.1.2 Element interface

Listing 5.2: element interface

```
type Element interface {
    ID() string
    AddLabel(Labels.Label)
    RemoveLabel(Labels.Label)
    Labels() map[string]labels.Label
    CanHost(Element) (bool, error)
    GetWeight() map[string]float64
    GetResource()  map[string]float64
    GetCost() map[string]float64
}
```

The Element interface in listing 5.2 declares all the methods that are common to the Node and the edge, therefore must be implemented by both of them. ID allows to retrieve the unique identifier of the element, AddLabel and RemoveLabel can be used to add or remove a label to/from the element, and Labels retrieves the list of labels belonging to the element. CanHost tells whether the element given as input can be hosted by the current element. GetWeight, GetResource and GetCost are used to retrieve the total amount of resources owned by the element and the cost for that resources.

5.1.3 Node interface

Listing 5.3: node interface

```
// Node is a graph node
type Node interface {
    Element

    plug(e Edge, nn Node)
    unplug(e Edge)
    EdgeNeighbor(id string) Edge
    GetNeighbors() map[string]Node
    SetNeighborhoodWeight(map[string]float64)
    GetNeighborhoodWeight() map[string]float64
    ConsumeResources(map[string]float64)
}
```

The Node interface in listing 5.3 includes all the methods declared by Element in addition to a set of methods proper of the Node. plug and unplug, given an edge and a node, allows to plug or unplug a link between two nodes, EdgeNeighbor takes the unique identifier of a node neighbor as argument, and retrieves the edge interconnecting to it. GetNeighbors returns the list of neighbors of the node, while SetNeighborhoodWeight and GetNeighborhoodWeight allow to set and get the weight of the node's neighborhood. Last, ConsumeResources decreases the resources amount owned by the node.

The concrete struct that implements the Node interface makes use of the following data structures:

- map[string]Edge: edges connected to it.
- map[string]Node: node's neighbors.
- map[string] labels .Label: the labels associated to it.

5.1.4 Edge interface

Listing 5.4: edge interface

```
type Edge interface {
    Element

    From() Node
    To() Node
}
```

The Edge interface in listing 5.4 includes all the methods declared by Element in addition to a set of methods proper of the Edge. From and To return the nodes connected to the endpoints of the edge.

The actual struct that implements the Edge interface makes use of map[string] labels .Label to store the labels associated to it.

5.2 Label library

The Label library specifies a set of interfaces and structs that can be used to create labels of different types starting from string definitions. Then the labels can be attached to nodes or edges in order to specify element descriptions, scheduling behaviours or policies, etc. Labels can be very different between each other and it is possible to define new types of them only by adding new structs implementing the desired interfaces, without modifying the label creation or addition processes. In order to achieve this goal we made extensive use of the reflection library of Golang (section 5.2.2).

5.2.1 Labelling interfaces

In order to define a new label it is necessary to implement a certain number of interfaces (there can be many interfaces, but in this first version of the label library, there is a limited amount of them).

Label interface

Listing 5.5: label interface

```
type Label interface {
    ID() string
    Kind() string
    Type() string
    Description() string
    SetID(string)
    SetKind(string)
    SetType(string)
    SetDescription(string)
    Validity() bool
    OverallValidity() bool
}
```

The Label interface 5.5 is the main interface that has to be implemented to create a new label. Each label struct is uniquely identified by a pair of two different values:

- kind: the main category of the label, e.g. resource, domain, etc.
- type: the type of the label, given its kind, e.g. cpu, latency, preemption, etc.

There is a pair of methods required to check the validity of the label:

- Validity: it is called after each custom setting methods, as explained in section 5.2.5.
- OverallValidity: it is called once all the custom setting methods have been called, as explained in section 5.2.5.

Hosting and hosted resource label interfaces

Listing 5.6: hosted resource label interface

```
type HostedResourceLabel interface {
    CanBeHosted(Label) (bool, error)
}
```

Listing 5.7: hosting resource label interface

```
type HostingResourceLabel interface {
    CanHost(Label) (bool, error)
}
```

All the labels aiming to represent a scheduling policy must implement both the HostingResourceLabel (section 5.7) and the HostedResourceLabel (section 5.6), which declare two dual methods:

- CanHost(Label) (bool, error): returns a boolean value that tells whether the current label satisfies the policy described by the label given as argument;
- CanBeHosted(Label) (bool, error): returns a boolean value that tells whether the label given as argument satisfies the policy described by the current label.

Consumable resource label interface

Listing 5.8: Consumable Resource Label interface

```
type ConsumableResourceLabel interface {
    TotalAmount() interface{}
    Amount() interface{}
    SetAmount(interface{})
    Consume(interface{})
}
```

If a struct implements the ConsumableResourceLabel interface, then it must have kind="resource". This interface represents a type of physical resource that can be consumed, such as CPU, memory, etc. The methods to implement have the following meaning:

- TotalAmount returns the original amount of resources;
- Amount returns the amount of resource available;
- SetAmount allows to set the amount of original amount of resources;
- Consume allows to consume resources, i.e., the amount of available resources is decreases by the specified amount.

A central parameter that must be used by the structs that implement the ConsumableResourceLabel interface is validity, a boolean checked by the labeller entity (section 5.2.5) in order to ensure the correctly instantiation of the label. Since the setting methods of the ConsumableResourceLabel interface take a generic interface as parameter, the cast of the arguments to the correct type must be done inside the methods, by setting the validity attribute to true or false, depending on the casting outcome.

Purchasable resource label interface

Listing 5.9: Purchasable Resource Label interface

```
type PurchasableResourceLabel interface {
    Cost() float64
    SetCost(interface{})
}
```

The PurchasableResourceLabel interface (section 5.9) represents a type of resource that can be purchased, i.e., in order to use the element, there is a cost to cope with. The PurchasableResourceLabel and ConsumableResourceLabel can be used jointly for representing resources that are bought and have an amount that decreases when used (e.g., memory), or can be used separately (e.g., label for a link can be Purchasable but not consumable).

5.2.2 Golang reflection

We made extensive use of the reflection library in order to derive the actual type of a label, starting from its textual representation. For example, given a label described by a string according to a certain format, by using this approach we can fetch the type of that label such described and instantiate it; once the label has been instantiated, we call its setter methods in order to create the object representation of the received textual one.

Reflection in general

Reflection in computing is the ability of a program to examine its own structure, particularly through types; it is a form of metaprogramming. Go is statically typed. Every variable has a static type, that is, exactly one type known and fixed at compile time: int, *MyType, [ ]byte, and so on. One important category of type is interface types, which represent fixed sets of methods. An interface variable can store any actual (non-interface) value as long as that value implements the interface's methods. An extremely important example of an interface type is the empty interface interface{}, that represents the empty set of methods and is satisfied by any value at all, since any value has zero or more methods.

Reflection goes from interface value to reflection object

At the basic level, reflection is just a mechanism to examine the type and value pair stored inside an interface variable. There are two types we need to know about in package reflect: Type and Value. Those two types give access to the contents of an interface variable, and two simple functions, called reflect .TypeOf and reflect .ValueOf, retrieve reflect .Type and reflect .Value pieces out of an interface value (also, from the reflect.Value it's easy to get to the reflect.Type).

Reflection goes from reflection object to interface value

Like physical reflection, reflection in Go generates its own inverse. Given a reflect.Value it is possible to recover an interface value using the Interface method; in effect the method packs the type and value information back into an interface representation and returns the result.

To modify a reflection object, the value must be settable

Settability is a property of a reflection Value, and not all reflection Values have it. The CanSet method of Value reports the settability of a Value. Settability is a similar like addressability, but stricter. It is the property that a reflection object can modify the actual storage that was used to create the reflection object. Settability is determined by whether the reflection object holds the original item.

Listing 5.10: Golang reflection

```
var x int = 5

p := reflect.ValueOf(&x)
fmt.Println(p.CanSet())      // false

v := p.Elem()
fmt.Println(v.CanSet())      // true

v.SetInt(3)
fmt.Println(v.Interface())   // 3
fmt.Println(x)               // 3
```

In listing 5.10 if we pass a copy of x (instead of the pointer) to reflect .ValueOf, so the interface value created as the argument to reflect .ValueOf is a copy of x, not x itself. Thus, if the statement v.SetInt(3) were allowed to succeed (it actually causes panic), it would not update x, even though v looks like it was created from x. Instead, it would update the copy of x stored inside the reflection value and x itself would be unaffected. The reflection object p is not settable, but it is not p we want to set, it is (in effect) *p. To get to what p points to, we call the Elem method of Value, which indirects through the pointer, and save the result in a reflection Value called v. At this point, by calling v.SetInt(3), it is possible to set the x value to 3.

Listing 5.11: Golang reflection fields

```
type T struct {
    A int
    B string
}

t := T{23, "field"}
s := reflect.ValueOf(&t).Elem()
typeOfT := s.Type()

for i := 0; i < s.NumField(); i++ {
    f := s.Field(i)
    fmt.Printf("%d: %s %s = %v\n", i,
        typeOfT.Field(i).Name, f.Type(), f.Interface())
}
// Printed strings:
// 0: A int = 23
// 1: B string = field
```

Listing 5.11 shows an example that analyzes a struct value, t. typeOfT is set to its type and iterated over the fields using straightforward method calls. Note that the names of the fields are extracted from the struct type, but the fields themselves are regular reflect.Value objects.

5.2.3 Labels data structures

Label definition relies on the definition of two different data structures, TypeRegistry and TypeAssociations, as shown in listing 5.12.

Listing 5.12: label types

```
var TypeRegistry = map[string]map[string]reflect.Type{
    "resource": {
        "cpu": reflect.TypeOf(CpuLabel{}),
        "cpuOffloading": reflect.TypeOf(CpuOffloading{}),
        "latency":      reflect.TypeOf(LatencyLabel{}),
    },
    "policy": {
        "linkCost": reflect.TypeOf(LinkCost{}),
        "preemption": reflect.TypeOf(PreemptionLabel{}),
    },
}

var TypeAssociations = map[string]map[string]string{
    "resource": {
        "cpuOffloading": "cpu",
        "latency": "latency",
    },
}
```

Type Registry

Type registry structure is designed to provide double random access to the Reflect. Type of a label, given its kind and type, as described in section 5.2.1. Every time a new label has to be added to the system, it is necessary to create a new entry of the correct kind in this data structure.

Type associations

Type associations is a data structure similar to the type registry structure, that aims to specify the pairing between different types of labels (e.g., in order to verify whether a job node with a cpuOffloading label can be hosted by a federation node, it is necessary to compare the job's cpuOffloading label with the node's cpu label). Every time a new label has to be added to the system, it is necessary to explicitly provide the pairing between the appropriate labels.

5.2.4 Labels comparison

Listing 5.13: labels comparison

```go
func ElementCanHost(fe, de Element) (bool, error) {

    // iterate over all the labels of the deployment element
    for _, l1 := range de.Labels() {

        // if the label l1 implements the resourceLabel, canHost is called
        if _, ok := l1.(labels.HostedResourceLabel); ok == true {

            // if fe doesn't have a label of the same type required by de,
            // the fe element is not suitable for the matching with de
            relatedType, ok := labels.TypeAssociations[l1.Kind()][l1.Type()]
            if ok != true {
                return false, errors.New("labels association not found")
            }

            l2, ok := fe.Labels()[l1.Kind() + "-" + relatedType]
            if ok == false {
                return false, errors.New("incompatible elements")
            }

            // Check if fe is suitable for the matching with de
            ok, err := l2.(labels.HostingResourceLabel).CanHost(l1)
            if err != nil{
                return false, err
            }

            if !ok {
                return false, nil
            }
        }
    }

    return true, nil
}
```

Whenever two elements (node or edge) must be compared in order to verify whether a federation node can host a job, it is necessary to iterate over all the federation element labels and compare each of them against the paired label (specified in TypeAssociations) of the job element. As shown in listing 5.13, this section of algorithm is divided in 5 different main parts:

1. iterate over all the labels of the deployment element (row 4);
2. if the current label implement the HostedResourceLabel, then it must be checked against the HostingResourceLabel of the correspondent type of the federation element (row 7);
3. the associated Federation label type is fetched from the TypeAssociation data structure (row 11);
4. it is checked that the federation element owns the related label type (row 16);
5. the two labels are compared each other through the implemented CanHost method (row 21).

5.2.5 Labeling mechanism

Listing 5.14: newLabel function

```go
// NewLabel creates a new label of the given type,
// calls the setters and return a new instance of the requested label
func (l *graphLabeler) NewLabel(k, t, graphType string, args map[string]interface{}) (labels.Label, error) {

    // check if the kind exists; if not, exit
    var tr reflect.Type
    var ok bool
    if tr, ok = labels.TypeRegistry[k][t]; !ok {
        return nil, errors.New("invalid kind and type")
    }

    // Create a new instance of the k kind and t type
    nl := reflect.New(tr)

    // Label interface methods that must always be called
    var defaultMethods = map[string]interface{}{
        "SetID":    strings.Join([]string{k, "-", t}, ""),
        "SetType": t,
        "SetKind": k,
        "SetGraphType": graphType,
    }

    nlRet := nl.Interface().(labels.Label)

    // set the protected mandatory values
    for k, v := range defaultMethods {
        if err := l.configureLabel(nl, k, v); err != nil {
            return nil, err
        }
    }

    // Each provided method is called by passing the related arg
    for k, v := range args {

```

```
35         // force methodName to be camelcase
36         methodName := "Set" + strings.Title(k)
37 
38         // check that the SetKind and SetType methods aren't manually setting
39         if _, ok := defaultMethods[methodName]; ok {
40             return nil, errors.New("invalid parameter")
41         }
42 
43         if err := l.configureLabel(nl, methodName, v); err != nil {
44             return nil, err
45         }
46 
47         if nlRet.Validity() == false {
48             return nil, errors.New("invalid attribute value")
49         }
50     }
51 
52     if nlRet.OverallValidity() == false {
53         return nil, errors.New("invalid overall label configuration")
54     }
55 
56     return nlRet, nil
57 }
```

The NewLabel function gets as input a set of parameters and returns a new label of the required type having all the fields set. This methods is tailored for the Kubernetes environment: each time a new advertisement message is accepted by the cluster, a new resource described by a set of strings is created. The set of strings describing the advertised resource embeds the information about the labels of the resource. This methods allows to convert a set of labels described only by means of strings to the correspondent object.

The NewLabel methods fetches the reflection .Type of a label, given its kind and type, then it allocates a new label of the desired type and sets all the fields of the labels by means of the provided arguments. This method basically provides the functionality to create a new label starting from its formal description in string. When a new label must be created, the NewLabel function is called, and a list of arguments is passed:

- kind of the label;
- type of the label;
- graphType of the graph the label will belong to;
- list of the arguments to give to the setting functions of the label.

This function makes use of the configureLabel function (section 5.15), needed to set the fields of the label my means of the reflection, without knowing them a priori. As shown in listing 5.14, first, the type of the label to create is fetched by means of the TypeRegistry data structure (row 8), then it is instantied a new label of the fetched type (row 13); after that the mandatory methods declared by the Label interface are called (row 26) for setting the ID, kind, type and graphType fields of the label. Then the arguments given as parameter are iterated (row 33), the setting method names are derived from the parameter names (row 36), it is checked that the default methods are not set twice. Last the configureLabel method is called (row 43) and the label validity is checked. Once all the methods have been called, the overall validity of the label is checked by means of the OverallValidity method. The validity checkers are deepened in 5.2.6.

Listing 5.15: configureLabel function

```
1  // ConfigureLabel calls the setter method represented by the k string
2  // on the current struct passing the v arguments
3  func (l *graphLabeler) configureLabel(nl reflect.Value, k string, v interface{}) error {
4      // fetch the method by its name
5      m := nl.MethodByName(k)
6  
7      // If the method doesn't exist, continue
8      if m.Kind() == reflect.Invalid {
9          return errors.New("method " + k + " doesn't exist")
10     }
11 
12     // check if the method is valid
13     if m.Type().NumIn() != 1 ||
14         m.Type().IsVariadic() {
15 
16         return errors.New("method " + k + " is invalid")
17     }
18 
19     // configure the input value
20     in := []reflect.Value{reflect.ValueOf(v)}
21 
22     // call the fetched method
23     m.Call(in)
24 
25     return nil
26 }
```

The configureLabel method takes a reflect.Value (the label to configure), a string (the name of the setting function to call) and an interface (the value to pass to the setting function). First the method to set is fetched by means of the MethodByName function, that given as input the method name returns the method to call (row 5), then some checks on the type of the methods are executed (rows 8, 13). Last the argument to pass to the method is converted to a Reflect.Value object (row 20), and the method is called passing the argument.

It is worth to notice that this approach for deriving label objects from the string descriptions requires a strict formatting of the text to convert in labels:

- the setting methods of the labels must necessarily be composed by the string "Set" + fieldName;
- it is not possible to explicitly set default fields (id, kind, type, graphType);
- each field to configure in the label must have a setting method.
- the correctness of the value passed to the setting methods must be provided by the label implementation itself.

5.2.6 Label validity

The label validity mechanism is a technique devoted to provide the correctness check of each label. The correctness of the each must be ensured by the label itself, by setting the valid parameter after each setting method. The validity check is a two-step process:

- each argument passed to a setting method must be checked in type (through a cast operation);
- once the configuration process ends up, the overall validity of the label is checked with the OverallValidity method, in charge to check that all the required setters have been called, the attributes values belong to the correct domain (e.g., $field1 \in \Re^+$), and the inter-values constraints (e.g., $field1 > field2$) are respected.

Listing 5.16: CpuOffloading label

```
type CpuOffloading struct {
    ...
    request float64
    limit   float64
    ...
    valid bool
    ...
}

func (l *CpuOffloading) Validity() bool {
    return l.valid
}

func (l *CpuOffloading) OverallValidity() bool {
    if  l.request <= 0 || l.limit <= 0 || l.request > l.limit {
        l.valid = false
    }

    return l.valid
}

// SetRequest sets the request attribute of the label
func (l *CpuOffloading) SetRequest(request interface{}) {
    var ok bool

    if l.request, ok = request.(float64); ok {
        l.valid = true
    } else {
        l.valid = false
    }
}
```

```

// SetLimit sets the limit attribute of the label
func (l *CpuOffloading) SetLimit(limit interface{}) {
    var ok bool

    if l.limit, ok = limit.(float64); ok {
        l.valid = true
    } else {
        l.valid = false
    }
}
```

The listing 5.16 shows an example of label with the validity checkers correctly set. This label represents the request of a job to have a certain amount of resources from a federation cluster. According to the Kubernetes behavior, the amount of resource is represented by a pair of values: request and limit. The setting methods for both limit and request takes as input an `interface{}` parameter that must be cast to float; if the cast operation does not succeed, i.e., the parameter is not a float value, the valid parameter is set to false, therefore, the label is marked as invalid.

After both `SetLimit` and `SetRequest` have been called, it is necessary to call the `OverallValidity` method, which checks that both SetRequest and Setlimit have been called (the default value of a float is 0), that they does not have invalid values (≤ 0) and the request is lower or equal than the limit.

Chapter 6

Experimental evaluation

This scheduler algorithm will be containerized and deployed in Kubernetes to be used as the alternative to the default scheduler, when deploying applications to be spread across a federation. In the real environment the algorithm has been designed for, data inputs are built as follows:

- the federation graph is incrementally built by means of the advertisement messages received from the other clusters of the federation;
- the deployment graph is received by another module that, on behalf of the scheduler, translates the descriptive language in which the deployment is formalized to the graph format handled by the scheduler.

Since the algorithm needs to be tested and evaluated in its performance, we created an artificial environment that simulates several sets of clusters joint in a federation, and various sets of jobs interconnected to create different representations of deployments to be scheduled across the federation. The formalism in which the data are expressed is faced in sections 6.1 and 6.2.

6.1 Input data

The input data consists in two different graphs:

- federation graph;
- deployment graph.

This data are formalized by means of .dot [30], a descriptive language that makes use of some constructs to represent arbitrary graphs with both nodes and edges labeled. An example of dot language is depicted in listing 6.1; this descriptive syntax can express both functional and aesthetic features of the graph. By means of the rendering tool provided by Graphviz, starting from the aforementioned listing, it is possible to render the image of the graph in fig. 6.1.

Listing 6.1: dot notation

```
graph{
    rankdir="LR";
    node[style="filled", color="#FF7F50"]
    0[label="cpuOffloading - request:156.84 - limit:115.46", label="n0"];
    1[label="cpuOffloading - request:456.02 - limit:381.22", label="n1"];
    2[label="cpuOffloading - request:260.01 - limit:186.07", label="n2"];
    3[label="cpuOffloading - request:399.18 - limit:270.57", label="n3"];
    4[label="cpuOffloading - request:126.07 - limit:122.97", label="n4"];
    3 -- 4[label="latency - value:31.54", label="n3-n4"];
    4 -- 1[label="latency - value:47.62", label="n4-n1"];
    3 -- 2[label="latency - value:58.39", label="n3-n2"];
    2 -- 1[label="latency - value:79.27", label="n2-n1"];
    3 -- 1[label="latency - value:52.39", label="n3-n1"];
    1 -- 0[label="latency - value:33.99", label="n1-n0"];
    0 -- 4[label="latency - value:37.54", label="n0-n4"];
    2 -- 0[label="latency - value:25.04", label="n2-n0"];
}
```

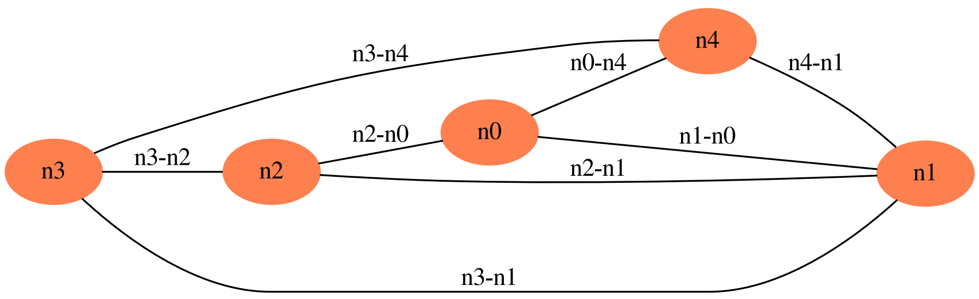


Figure 6.1: input dot representation

6.2 Output data

The algorithm takes as input the two graphs and computes a scheduling schema, by trying to find a match between each job of the deployment graph and a cluster of the federation graph. Every time a job is assigned to a cluster, the amount of resources available to that cluster is decreased by the amount required by the job, and whenever the job is withdrawn from a cluster (because it brought to an unfeasible solution), the amount of consumed resources is restored to the cluster.

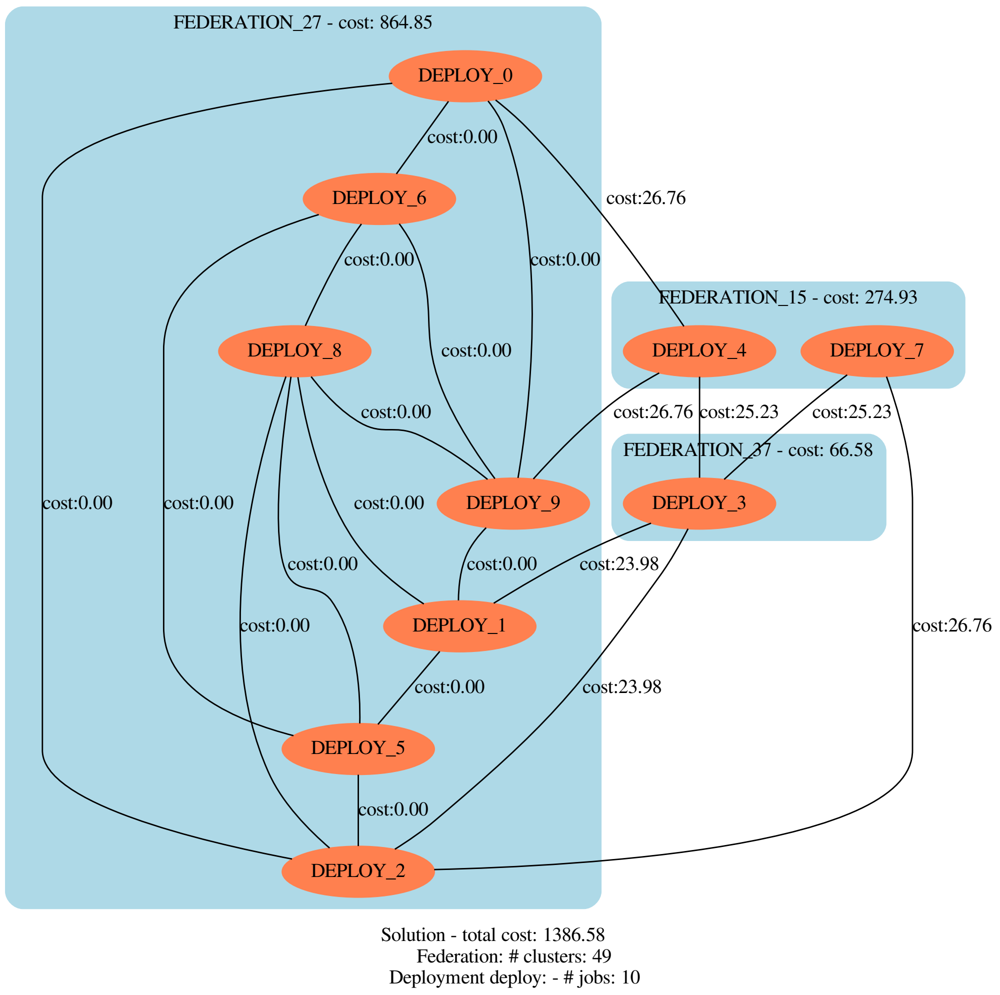

FEDERATION_27 - cost: 864.85
DEPLOY_0
DEPLOY_6
DEPLOY_8
DEPLOY_9
DEPLOY_1
DEPLOY_5
DEPLOY_2
FEDERATION_15 - cost: 274.93
DEPLOY_4
DEPLOY_7
FEDERATION_37 - cost: 66.58
DEPLOY_3
cost:0.00
cost:26.76
cost:25.23
cost:23.98
Solution - total cost: 1386.58
Federation: # clusters: 49
Deployment deploy: - # jobs: 10

Figure 6.2: output dot representation

A feasible scheduling schema is found when all the jobs are assigned to a cluster. In that moment the recursive algorithm ends up, the goodness of the solution is evaluated and a .dot file representing the solution is created. The output file can be treated by the Graphviz rendering software, which generates an image similar to figure 6.2.

6.3 Test-bed

6.3.1 Kubernetes resource quotas

Since this scheduling algorithm has been designed to be deployed in Kubernetes, we created a test-bed that aims to replicate as much as possible the Kubernetes environment and its rules; the main one is the resource quotas approach [31]. In a nutshell, when several users or teams share a cluster with a fixed number of nodes, there is a concern that one team could use more than its fair share of resources. A resource quota, defined by a ResourceQuota object, provides constraints that limit aggregate resource consumption per namespace. It can limit the quantity of objects that can be created in a namespace by type, as well as the total amount of compute resources that may be consumed by resources in that project.

Resource quotas work as follows. Different teams work in different namespaces. The administrator creates one ResourceQuota for each namespace. Users create resources (pods, services, etc.) in the namespace, and the quota system tracks usage to ensure it does not exceed hard resource limits defined in a ResourceQuota. If the creation or update of a resource violates a quota constraint, the request will fail with HTTP status code 403 FORBIDDEN with a message explaining the constraint that would have been violated. If quota is enabled in a namespace for compute resources like cpu and memory, users must specify requests or limits for those values; otherwise, the quota system may reject pod creation.

6.3.2 Policies

For the test-bed two policy labels have been used: cpu for nodes and latency for edges, that work as follows:

- cpu labels: assigned to nodes, they contain two fields that allow to specify request and limit of cpu required by the current job, as specified by the Kubernetes approach;
- latency labels: assigned to edges, they contain one field that specifies the maximum amount of latency bare by the labeled connection.

6.3.3 Generated graphs

The test-bed for the effectiveness and performance evaluations has been built by using a set of randomly created deployments. We used a *python* script to create a graph with random labels and random connections, starting from a given number of nodes and connections. The script creates both the federation and the deployment graphs, starting from two configuration files. The configuration files are read by the program, that then generates two different graphs with the following features:

- the deployment graph is an arbitrary connected graph, with the edges generated by randomly choosing the linked nodes from the randomly created node-set;
- the federation graph is a completely connected graph generated from a random graph with star topology, having the local node as the star-center. The connections between the external nodes are created by summing the edges needed to route from the first to the second one.

The deployment configuration

Listing 6.2: deployment configuration file

```
min_request: 100
max_request: 500

min_limit: 100
max_limit: 1000

min_latency: 10
max_latency: 80

n_nodes: 5
n_edges: 20
```

The *deployment-config* file (listing 6.2) specifies a deployment graph where the nodes represent the jobs and the edges represent the interconnections between jobs. The parameters are described as follows:

- min_request - max_request: the lower and upper bounds of the random generated request value for each node;
- min_limit - max_limit: the lower and upper bounds of the random generated limit value for each node;
- min_latency - max_latency: the lower and upper bounds of the random generated latency value for each edge;
- n_nodes: the amount of jobs for the generated deployment;
- n_edges: the amount of interconnections for the generated deployment.

The federation configuration

Listing 6.3: federation configuration file

```
max_amount: 3000
min_amount: 1000
```

```

min_request: 100
max_request: 1000

min_limit: 100
max_limit: 1500

min_cost: 0.4
max_cost: 1

min_latency: 4
max_latency: 15

min_latency_cost: 0.1
max_latency_cost: 0.2

n_nodes: 500
```

The *federation-config* (listing 6.3) specifies a federation graph where the nodes represent the Kubernetes clusters and the edges represent the links between clusters. The parameters are described as follows:

- min_amount - max_amount: the amount of resources owned by the cluster
- min_request - max_request: the lower and upper bounds of the random generated request value for each cluster;
- min_limit - max_limit: the lower and upper bounds of the random generated limit value for each cluster;
- min_cost - max_cost: the lower and upper bounds of the random generated cost value for each cluster;
- min_latency - max_latency: the lower and upper bounds of the random generated latency value for each link;
- min_latency_cost - max_latency_cost: the lower and upper bounds of the random generated cost value for each cluster;
- n_nodes: the amount of clusters in the generated federation.

6.3.4 Data sets

The tests have been performed as follows:

- 4 different federation graphs have been generated, each one with different parameters and a different amount of nodes [50, 100, 200, 500];

- 10 different deployments graph have been generated each one with different parameters and a variable amount of nodes [5, 10, 20, 40, 70, 100, 200, 50, 30, 10].

For each federation graph, all the deployment graphs have been scheduled in an incremental manner, by following the order of the above array.

6.4 Objective function evaluation

The main evaluation is based on the comparison between the scheduling algorithm that makes use of the heuristic described in chapter 4 (from now called *heuristic approach*) and a version of the same algorithm that does not make any use of sorting logic both in deployment and in federation graphs (from now called *random approach*). Both the algorithms have been tested in all the testing environments described above, and the results have been compared, in order to understand whether the heuristic approach leads to better results. We tried to schedule the whole set of deployments in all the federation graphs, and the first outcome is that not all the deployments have been scheduled in all the federation graphs. Indeed, only the federation with 200 and 500 nodes succeeded to host all the jobs, while the federation of 50 and 100 nodes hosted only a subset of them. This is due to the lack of enough resources in the federation: the smallest graphs did not have enough resources to host all the jobs.

6.4.1 Global evaluation

The first plot in figure 6.3 shows the comparison between the total cost of the deployments in the two test-cases: it is noticeable that the cost (as it is supposed to be) increases with the amount of scheduled jobs. The red line represents the total cost of the random approach, while the blue line refers to the heuristic one. The grey line is the delta between the two lines and is measured by the right-y axis. It is worth to notice that the difference between the two approaches increases on the increase of the federation size, reaching an amount of 70 when all the deployments have been scheduled (it means that the random approach has a cost that is approximately 30% higher than the heuristic one). Another meaningful aspect is that the scheduling schemas found in the federation of size 200 have a lower delta than the schemas computed for the federation of 500 nodes: it is explainable by the fact that when a federation has more nodes, likely there is an higher number of better choices than in federations with less nodes, therefore the algorithm has an easier way to find the best suitable nodes and the total cost decreases.

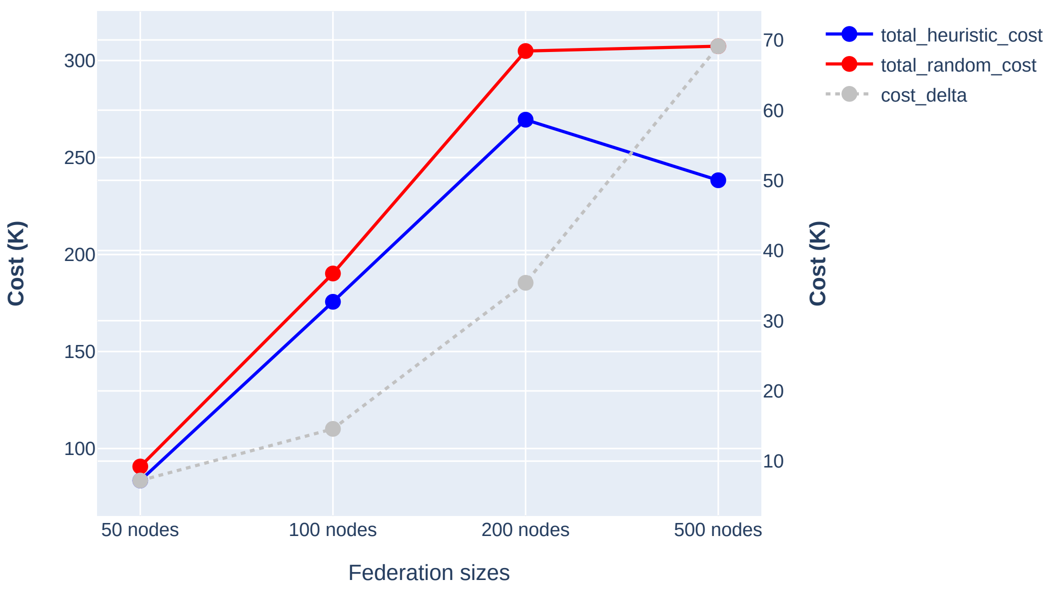


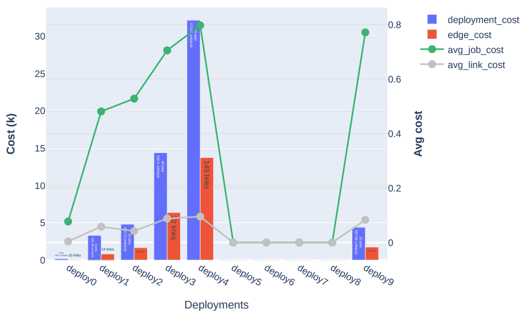
Figure 6.3: Objective function difference

6.4.2 Local evaluations

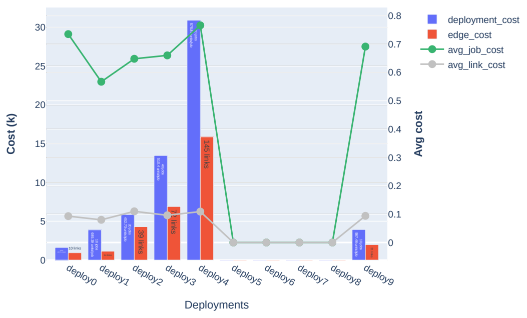
The plots in this section show how the objective function increases for both nodes and edges with the increase of the deployment sizes, comparing the result of the heuristic approach with the random approach. The plots are structured as follows:

- the left one is related to the heuristic approach, while the right one is related to the random approach;
- the x axis represents the deployments, in the order above described;
- the y axis represents the cost: thousands the left ones and units the right ones, for each plot;
- the y axis of the bars is the left one, while the axis of the scatter is the right one, for every plot;
- blue bars represent the total job cost for each deployment;
- red bars represent the total link cost for each deployment;

- the green scatters represent the cost per unit of computation for each deployment;
- the grey scatters represent the cost per link for each deployment.



(a) Heuristic



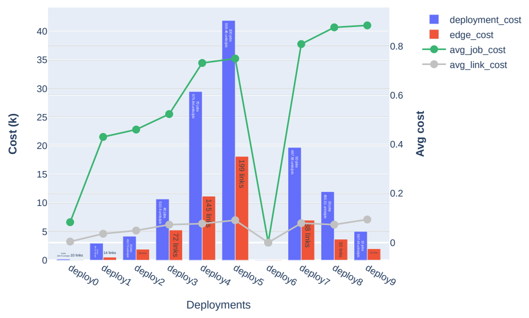
(b) Random

Figure 6.4: Deployment costs in a federation of 50 nodes

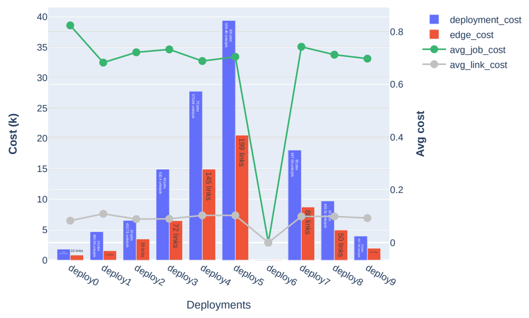
Plots related to the federation with 50 nodes (figure 6.4) clearly show which deployments have been scheduled; indeed, once the federation has been filled in with the first five deployments (5, 10, 20, 40, 70 jobs), there was not enough space to host more jobs, except for the last deployment, of size 5. It is suddenly clear how the cost per computation unit increases with the decrease of the available space for the heuristic approach, while in the random graph it does not follow any specific trend, but instead it is completely random. This behavior is explainable by the fact that the best scheduling options are chosen first by the heuristic approach. Then, once the cheaper nodes have been filled, the algorithm starts considering more expensive nodes, thus the cost per computation units increases over the x axis.

The scheduling operation in the federation of 100 nodes (figure 6.5) succeeds to scheduled all the jobs but one, the biggest (200 jobs). Even here the increase of the cost per computation unit is evident, along with the increase of cost per link, that follows the same trend (it increases over the x axis). The trends of the random approach are random as well.

The plot in figure 6.6 is the first one that succeeded to schedule all the deployments, and puts in evidence one interesting thing: the fact that the trend above highlighted has an outlier: the second deployment (10 nodes) has a cost per computation unit higher than the cost of the next deployment. This behavior is explainable by observing the average size of the jobs in that deployment: it is

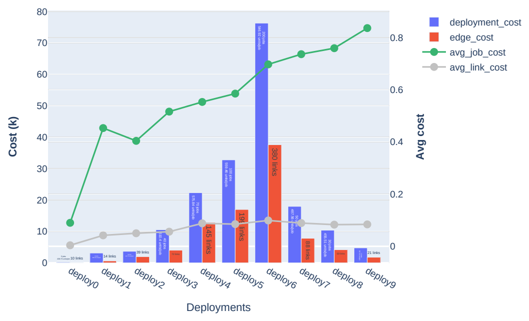


(a) Heuristic

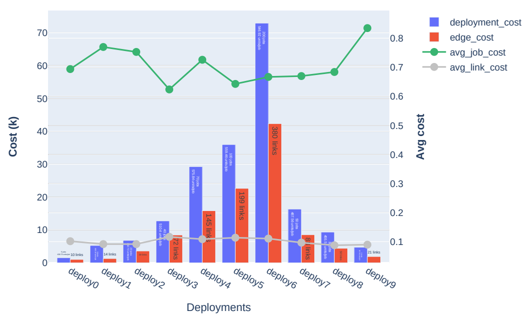


(b) Random

Figure 6.5: Deployment costs in a federation of 100 nodes



(a) Heuristic



(b) Random

Figure 6.6: Deployment costs in a federation of 200 nodes

685 units/job, that, compared to the average job size of the deployments (528 units/job) is 30% higher; this means that it is more difficult to schedule the jobs of that deployment, because of the lack of that amount of resources in the federation nodes, and then nodes with higher cost are chose for that deployment.

In the last plot 6.7, regarding the federation with 500 nodes, we can notice the consolidation of all the above described trends, and to put in evidence the overall behavior: in the heuristic approach the average cost per computation unit of the deployments does not even reach the average value kept by the random approach, then it is clear how the heuristic approach computes a solution in a smarter and better way, choosing first the cheaper and the bigger nodes. The size

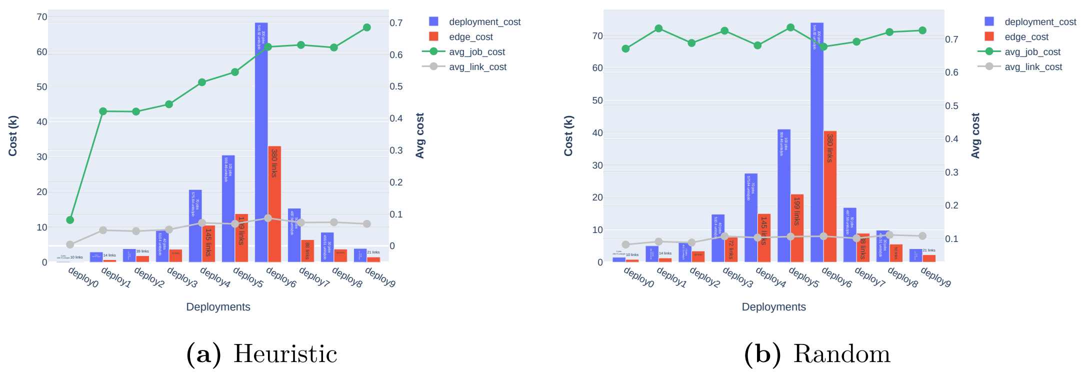


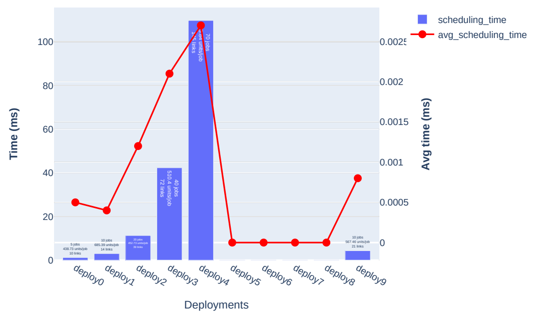
Figure 6.7: Deployment costs in a federation of 500 nodes

of the federation nodes is an important parameter, since the higher the size, the higher the number of jobs that can be scheduled in that node, leading to a decrease of the link cost, and to the possibility of getting stuck in an unfeasible recursion branch because of the tightening of the link constraints (to schedule jobs in nodes that does not have good links toward their neighbors).

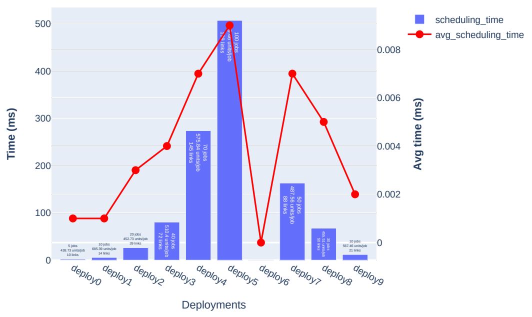
6.5 Performance evaluation

In this section we evaluated the performance of the scheduling algorithm under the same conditions above described. Since the performance of the random approach are comparable to the performance of the heuristic approach (the algorithm is the same, except for a sorting operation, as previously stated), only the performance of the heuristic approach has been plotted. The plots are structured as follows:

- the x axis represent the deployments, in the order above described;
- the y axis represent the cost for each plot;
- the y axis of the bars is the left one, while the axis of the scatters is the right one, for each plot;
- the blue bars represent the total scheduling time in milliseconds for each deployment;
- the red scatters represent the average time of deployment per computing resource unit for each deployment;



(a) 50 nodes

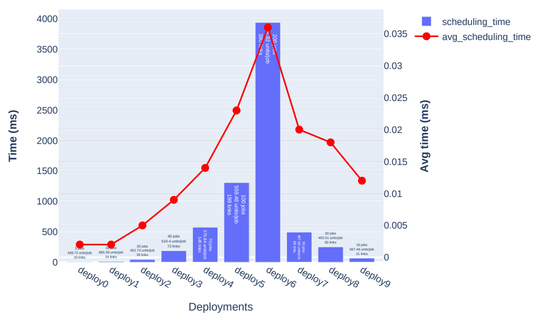


(b) 100 nodes

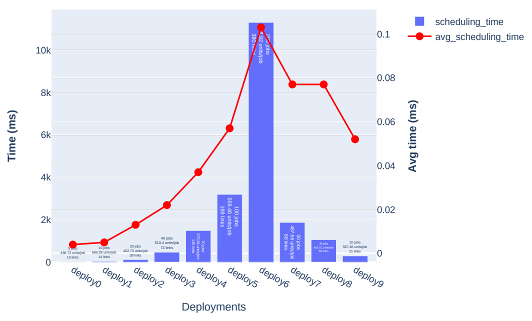
Figure 6.8: Performance evaluation in federations of 50 and 100 nodes

In the first two plots, representing the scheduling time in a federation of 50 and 100 nodes, it is possible to notice how the total scheduling time increases with the increase of the deployment size; there is a monotonic increase both in the total scheduling time and in average scheduling time. The *deploy1* outlier is due to the average size of the jobs composing the deployment (as already explained in the previously section): the higher the job sizes, the lower the scheduling time per computation unit, given a constant job scheduling time. One interesting fact is that the time does not increase linearly with the federation size; in fact in a federation with 50 nodes the total scheduling time is around 120 ms for the deployment with 70 jobs, while the schedule of the same deployment in the federation with 100 nodes took around 500 ms. Nevertheless the curve is not even exponential (as we would expect by the exponential nature of the problem), but is described by a polynomial function with order higher than linear (figure 4.4).

In the last two plots (figure 6.9) we depicted the scheduling performance in federations with 200 and 500 nodes. It is worth to notice the monotonic increase of the scheduling time with the increase of the deployment size (deploy0 - deploy6) and the subsequent monotonic decrease with the decrease of the deployment size (deploy6 - deploy9): the scheduling time of the deployment 9 (10 jobs) is much greater than the time employed for deployment 1 (same size). This happened because of resource saturation: in that case the branches of the recursion tree led to more unfeasible states respect to the deployment 1. Every time an unfeasible state is explored, a backtracking operation is required (with a consequent waste of time). This behavior exists only because we tried to schedule many deployments in the same federation at the same time. The normal operating condition is that whenever a job has been scheduled in a node, the node updates its advertisement



(a) 200 nodes



(b) 500 nodes

Figure 6.9: Performance evaluation in federations of 200 and 500 nodes

message, leading to a decrease of the federation size. This fact ensures that the scheduling times of deployments having the same size is comparable if performed with advertisement messages correctly updated.

Chapter 7

Conclusion and future work

This work aims at proposing a novel algorithm to perform scheduling operations in a gang-fashion, i.e., a job with all its dependencies at once, while taking into account several sets of policies. In this first implementation we succeeded to:

- create a software infrastructure that allows to define multiple policies
- perform gang scheduling while considering two different policies, one for nodes and one for edges;
- schedule jobs in a feasible (under exponential, $O(N_D * N_F^2)$) time.

In chapter 6 we evaluated the effectiveness and the performance of the algorithm, and obtained the following outcomes:

- the scheduling cost is much lower (the amount depends on the graphs values) than a random algorithm that does not make use of any heuristic approach;
- we scheduled deployments with size up to 100 jobs in federation with 100 nodes in a time $\leq 500ms$;
- we succeeded to schedule a deployment with 200 jobs in a federation with 500 nodes in a time $\sim 10s$;

In the future work on the algorithm we aim to:

- define a greater amount of labels (hence metrics);
- make the scheduler able to take into account more than one label both for nodes and edges;
- improve the scheduler heuristic by taking into account not only the affinity related to the clusters size but also the connectivity capabilities of the clusters;

- perform further analysis about the goodness of the solution found and the algorithm performance;

Last, we will create a Kubernetes scheduler based on this algorithm: the advertisement messages received from the other peers of the federation will allow us to create a virtual node (by means of the virtual kubelet), that represents a remote cluster giving us the possibility to offload jobs to it. The virtual node entity will be handled by a brand-new module that will create the logic representation of a new node in the federation graph. Then the scheduler will be able to schedule jobs across multiple clusters, such that we will verify its behavior in the environment it has been designed for, without building ad-hoc test-beds.

Bibliography

[1] *8 facts about real-world container use.* URL: `https://www.datadoghq.com/container-report/` (cit. on p. 1).

[2] 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/` (cit. on p. 1).

[3] *Minikube project git repository.* URL: `https://github.com/kubernetes/minikube` (cit. on p. 1).

[4] *Kubernetes Federation git repository.* URL: `https://github.com/kubernetes-sigs/kubefed` (cit. on p. 1).

[5] *Kubernetes official documentation.* URL: `https://kubernetes.io/docs/home/` (cit. on pp. 3, 10, 12–15).

[6] *Virtual-kubelet git repository.* URL: `https://github.com/virtual-kubelet/virtual-kubelet` (cit. on pp. 3, 14, 15).

[7] *Kubebuilder git repository.* URL: `https://github.com/kubernetes-sigs/kubebuilder` (cit. on pp. 3, 15).

[8] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. «Large-scale cluster management at Google with Borg». In: *Proceedings of the European Conference on Computer Systems (EuroSys).* Bordeaux, France, 2015 (cit. on pp. 3, 24).

[9] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. «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` (cit. on pp. 3, 22).

[10] Ferenc Hámori. *The History of Kubernetes on a Timeline.* June 2018. URL: `https://blog.risingstack.com/the-history-of-kubernetes/` (cit. on p. 4).

[11] 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/` (cit. on p. 4).

[12] Kalyan Ramanathan. *5 business reasons why every CIO should consider Kubernetes.* Oct. 2019. URL: `https://www.sumologic.com/blog/why-use-kubernetes/` (cit. on p. 4).

[13] Eric Carter. *Sysdig 2019 Container Usage Report: New Kubernetes and security insights.* Oct. 2019. URL: `https://sysdig.com/blog/sysdig-2019-container-usage-report/` (cit. on p. 6).

[14] Diego Ongaro and John Ousterhout. «In search of an understandable consensus algorithm». In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14).* 2014, pp. 305–319 (cit. on p. 7).

[15] *Kubernetes API official documentation.* URL: `https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/` (cit. on p. 10).

[16] *Kubernetes Operator pattern.* URL: `https://kubernetes.io/docs/concepts/extend-kubernetes/operator/` (cit. on p. 15).

[17] AR Arunarani, D Manjula, and Vijayan Sugumaran. «Task scheduling techniques in cloud computing: A literature survey». In: *Future Generation Computer Systems* 91 (2019), pp. 407–415 (cit. on p. 17).

[18] S Selvarani and G Sudha Sadhasivam. «Improved cost-based algorithm for task scheduling in cloud computing». In: *2010 IEEE International Conference on Computational Intelligence and Computing Research.* IEEE. 2010, pp. 1–5 (cit. on p. 18).

[19] J. H. Anderson, V. Bud, and U. C. Devi. «An EDF-based scheduling algorithm for multiprocessor soft real-time systems». In: *17th Euromicro Conference on Real-Time Systems (ECRTS'05).* July 2005, pp. 199–208. DOI: `10.1109/ECRTS.2005.6` (cit. on p. 18).

[20] *graphviz.* URL: `https://microcontrollerslab.com/earliest-deadline-first-scheduling/` (cit. on p. 18).

[21] Shima Rashidi and Saeed Sharifian. «A hybrid heuristic queue based algorithm for task assignment in mobile cloud». In: *Future Generation Computer Systems* 68 (2017), pp. 331–345 (cit. on p. 18).

[22] Shamsollah Ghanbari and Mohamed Othman. «A priority based job scheduling algorithm in cloud computing». In: *Procedia Engineering* 50.0 (2012), pp. 778–785 (cit. on p. 19).

[23] Ziqian Dong, Ning Liu, and Roberto Rojas-Cessa. «Greedy scheduling of tasks with time constraints for energy-efficient cloud-computing data centers». In: *Journal of Cloud Computing* 4.1 (2015), pp. 1–14 (cit. on p. 19).

[24] Bo Liu, Ling Wang, and Yi-Hui Jin. «An effective PSO-based memetic algorithm for flow shop scheduling». In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 37.1 (2007), pp. 18–27 (cit. on p. 19).

[25] Kobra Etminani and M Naghibzadeh. «A min-min max-min selective algorihtm for grid task scheduling». In: *2007 3rd IEEE/IFIP International Conference in Central Asia on Internet*. IEEE. 2007, pp. 1–7 (cit. on p. 19).

[26] Malte Schwarzkopf and Peter Bailis. «Research for Practice: Cluster Scheduling for Datacenters». In: *Commun. ACM* 61.5 (Apr. 2018), pp. 50–53. ISSN: 0001-0782. DOI: `10.1145/3154011`. URL: `https://doi.org/10.1145/3154011` (cit. on p. 20).

[27] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. «Mesos: A platform for fine-grained resource sharing in the data center.» In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22 (cit. on p. 20).

[28] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. «Sparrow: Distributed, low latency scheduling». In: Nov. 2013, pp. 69–84. DOI: `10.1145/2517349.2522716` (cit. on p. 23).

[29] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. «Firmament: Fast, centralized cluster scheduling at scale». In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 99–115 (cit. on p. 25).

[30] *graphviz*. URL: `https://www.graphviz.org/` (cit. on p. 54).

[31] *resource-quotas*. URL: `https://kubernetes.io/docs/concepts/policy/resource-quotas/` (cit. on p. 57).