Boreas – A Service Scheduler for Optimal Kubernetes Deployment

Torgeir Lebesbye[1], Jacopo Mauro[2], Gianluca Turin[1(✉)], and Ingrid Chieh Yu[1]

[1] University of Oslo, Oslo, Norway
{torgeirl,gianlutu,ingridcy}@ifi.uio.no
[2] University of Southern Denmark, Odense, Denmark
mauro@imada.sdu.dk

Abstract. The advent of cloud computing radically changed the way organisations operate their applications and allows them to achieve high availability of services at affordable cost. Most cloud-computing platforms fostered Kubernetes for their container orchestration and service management. The scheduler is a key component of Kubernetes, as it is responsible for finding the placement of new service containers when they are deployed. The default scheduler is very fast, although often suboptimal. This can lead to inefficient placement of services, or more severely, inability to deploy.

We present a custom Kubernetes scheduler, dubbed *Boreas*, which is designed to evaluate bursts of deployment requests concurrently. Boreas finds the optimal placements for service containers with their deployment constraints by utilising a configuration optimiser. Results show that Boreas is able to find placements where the default Kubernetes scheduler fails, wasting less computing resources, or proving that no feasible deployment solution is possible.

Keywords: Services on the Cloud · Cloud service management · Kubernetes · Scheduling

1 Introduction

Kubernetes [5] has become the new standard for container orchestration and service management. Originally proposed by Google, Kubernetes is an open source project that provides a layer between the cluster operator and the applications running on the cluster. Its applications are implemented as collections of services, each developed, deployed and scaled individually.

The main components of a Kubernetes systems are the Pods, every one of them representing an instance of a scalable (micro)service. A pod generally hosts one or few containers which are the minimal units containing the service source code to execute with all the code dependencies. This division proved to be extremely useful to avoid software dependencies because when two services have conflicting modules they can be arranged in different containers within the same

© Springer Nature Switzerland AG 2021
H. Hacid et al. (Eds.): ICSOC 2021, LNCS 13121, pp. 221–237, 2021.
https://doi.org/10.1007/978-3-030-91431-8_14

pod. On the other hand, this flexibility is limited by the pod's need of being small so it can be quickly scaled to meet possibly increasing service demand.

Another central component in Kubernetes is the scheduler [14], i.e., the component responsible for finding the placement of new service pods when they need to be deployed. Kubernetes comes with a default scheduler that is very fast, scales to hundreds of nodes, but it is heuristic based. This means that dependency constraints (e.g., pod affinities) are not necessarily optimal, thus leading to possible waste of resources, and more severely, the scheduler may be incomplete, i.e., unable to deploy pods even when a possible schedule is available.

In general, the problem of finding the optimal pod deployment in Kubernetes is an extension of the bin-packing problem and therefore a NP-complete problem [17]. Kubernetes developers prioritized speed over scheduler completeness and optimality in the design of the default scheduler to allow Kubernetes to scale up to thousands of pods and nodes. However, Kubernetes is also used for systems that are not very dynamic and with a limited size. In such deployment scenarios, when speed is not the main priority, one would prefer a scheduler that can lead to more accurate and less resource consuming deployments.

In this paper, we introduce a custom Kubernetes scheduler, dubbed *Boreas*, that ensures optimal pod placements with deployment constraints. Boreas reduces the overall computing resource usage and increases the utilization of cloud computing infrastructure managed by Kubernetes at the cost of slower pod deployment. The core of Boreas is the optimization configuration tool Zephyrus2 [1] that relies on the Aeolus formal model [9] for provably optimal service deployment [6]. Boreas integrates Zephyrus with the architecture of Kubernetes through a proper adapter. When new pod deployment requests arrive, Boreas parses the deployment constraints of the new requests and, based on the available computational resources left, encodes the deployment problem for Zephyrus that is invoked to retrieve the optimal pods deployment solution, if any. In this paper, we describe the design principles and the architecture of Boreas. Moreover, we show empirically that in the presence of standard Kubernetes deployment constraints, Boreas is able to find a placement for the pods in cases where the default scheduler fails, demonstrating that Boreas can be a better alternative than the default scheduler for medium size cost-aware applications.

The rest of the paper is organized as follows. In Sect. 2 we give an introduction of Kubernetes, the pod deployment strategy and the optimization tool Zephyrus2. In Sect. 3 we introduce Boreas, its architecture, how it handles deployment constraints and its batch scheduling. In Sect. 4 we test Boreas and compare it with the default scheduler on some medium size deployment jobs. Section 5 gives related work and we conclude the paper in Sect. 6.

2 Preliminaries

In this section we briefly introduce the two main tools used in our approach: Kubernetes and Zephyrus.

Kubernetes. Kubernetes [5] is the most widely used container orchestration engine for the deployment and maintenance of container based applications.

Containers encapsulate the execution environments of a program, abstracting from the details of physical and virtual machines, and the deployment infrastructure. Compared to virtual machines, they provide the same advantages of virtualization but are more lightweight, offering a better scalability and maintainability. Containers are also *portable* across clouds [10], they require much less storage, and have faster booting time than virtual machines. For all these reasons, containers have recently been widely adopted giving rise to the need of platforms such as Kubernetes to orchestrate them. In the following, we restrict our attention to the main components of Kubernetes related to resource management, with a special focus on the scheduler.

Pods are the basic scheduling unit in Kubernetes. They are high-level abstractions for groups of containerized components which are usually run using a Docker engine [26]. A pod consists of one or more containers that are guaranteed to be co-located on the host machine and can share resources. A pod is deployed according to its resource requirements and has its own specified resource limits. For two or more pods to be deployed in the same node, the sum of the minimum amounts of resources required for the pods needs to be available in the node.

Services represent components that act as basic internal load balancers and ambassadors for pods. A service can be thought as a collection of pods that perform the same function and are viewed as a single entity. Kubernetes can deploy a service, keep track of pods of the service and route all needed communications to them. Services and pods in a Kubernetes cluster are organized within namespaces, allowing multiple applications to share the cluster resources.

Nodes are computing resource on which Kubernetes runs. One node functions as the master node,[1] and acts as a gateway and controller for the cluster by exposing an API for developers and external traffic. The master node carries out the scheduling and orchestrates the communication between other components. The other nodes, called workers, host pods. The worker nodes have explicit resource capabilities given as a set of labels that can specify its version, status, and particular features (e.g., presence of a GPU).

Autoscalers are responsible for ensuring that the number of pods deployed in the cluster matches the number of pods in its configuration. There is one autoscaler for each service, managing a group of identical, replicated pods which are created from pod templates and can be horizontally scaled by deploying or removing pods.

Scheduler is in charge of assigning pods to specific nodes in the cluster. The scheduler matches the operating requirements of a pod's workload to the resources that are available in the current infrastructure environment, and places pods on appropriate nodes. The scheduler is responsible for monitoring the available capacity on each node to make sure that workloads are not scheduled in excess of the available resources. The scheduler needs to know the total capacity of each node and the resources already allocated to the nodes.

[1] There can be more master nodes, but one will always be the main master node hosting the cloud controller.

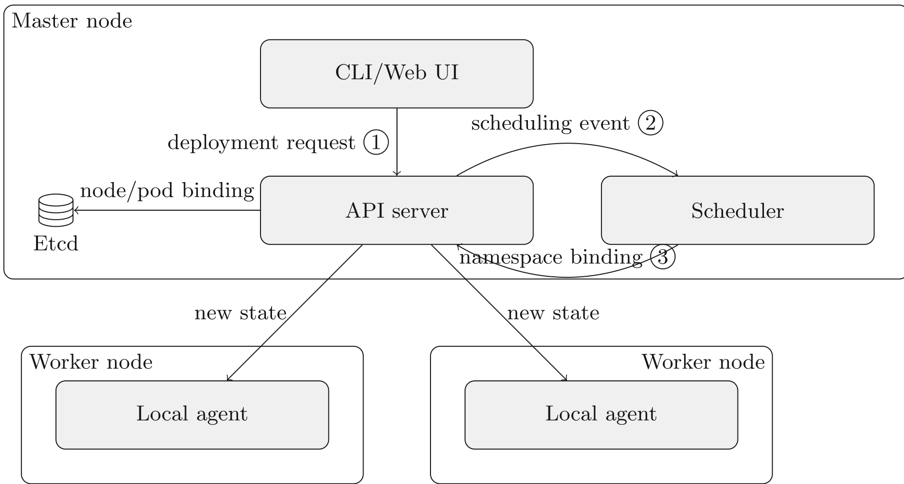

Fig. 1. Pod scheduling orchestration in Kubernetes.

While deploying a pod, it is possible to set deployment constraints that condition how it should be placed in the cluster. For example, it is possible to define pod *affinity* to place the affine pods on the same nodes. Similarly, by defining a pod *anti-affinity* it is possible to avoid deploying the pods on the same node. If the pods have an anti-affinity for themselves, every pod of a service will be deployed on a different node. Pods can also have affinities towards node types.

When a pod deployment is created, a chain of events is generated as illustrated in Fig. 1. When the deployment request is sent to the Kubernetes API server ①, the API server creates and exposes a scheduling event ②. Schedulers listen for such events and when an event targets them they process the request. The scheduler first identifies a node that is suitable for deploying the pod of the scheduling event and then sends a suggestion back to the API server in the form of a namespace binding between the pod and node ③. The API server, at this point, adds the binding to its own distributed storage (i.e., an Etcd server), allowing the local Kubernetes agent running on the selected node to instructs its container runtime to fetch and run the pod's container(s).

A pivotal point in this event chain is when scheduling events are processed by schedulers. The default Kubernetes scheduler iterates unassigned pods one at a time when assigning them to a node. It does so at an incredible speed (i.e., scheduling throughput of more than 50 pods/sec), but its implementation is heuristic-based, and it does not guarantee that pods are placed where they fit best if looking at all deployments as a whole.

The default scheduler identifies the most suitable node in the cluster in two steps [19]:

1. Filter: remove any node that lack any resources required by the pod, doesn't match explicit label or node name requirements, or that report memory or disk pressure.

2. Rank: the remaining nodes are then ranked using a set of priority functions. The ranking is calculated by weighting properties such as highest fraction of free resources (least requested), resource balance, service spread, pre-installed service requirements and affinity requirements.

One consequence of the default weighting of these priority functions is that the ranking will lean towards spread pods as much as possible.

Zephyrus2. Zephyrus2 [1] is a configuration optimizer originally designed to find the optimal placement of applications on virtual machines. Zephyrus2 requires a declarative description as input to specify the software components, the available virtual machines, and the deployment constraints.

The software components are specified in Aeolus [9], i.e., a component model for the definition and reasoning of cloud deployment plans. In Aeolus, software components are modeled as black-boxes that expose require- and provide-ports to capture required and provided functionalities respectively. Every software component consumes a given amount of resources. The virtual machines are modeled instead as locations. Each location has a name, a list of resources that it can provide, and an associated cost. The user can specify (deployment) constraints in an ad-hoc declarative language to define the desired final configuration. The constraints are powerful enough to express, e.g., the presence of a given number of components, their co-installation requirements, and their conflicts.

By exploiting modern SMT and CP technologies, Zephyrus2 finds a configuration distributing components on a set of locations such that: (i) the constraints reflecting the user requirements are satisfied, (ii) every functionality required by a deployed component is provided, (iii) in each location, the available resources are sufficient to cover the resource needs of all components deployed on it, and (iv) the values of some user-defined objective functions are minimized. The default objective-function is to obtain the final configuration with lower cost, choosing the one with the minimal amount of components in case of ties.

Zephyrus2 can be deployed as a Docker container, and it can be invoked by HTTP requests.

3 Boreas - An Optimal Kubernetes Scheduler

In this section we present the salient features of Boreas, how it can be deployed and how the deployment optimization problems are encoded and solved.

Boreas is a custom scheduler for Kubernetes that can replace the default scheduler or run alongside it. The modular system architecture of Kubernetes makes this framework highly configurable and extensible allowing to modify, extend or replace the default scheduler [20].

A graphical representation of the deployment of the Boreas scheduler in a Kubernetes master node is shown in Fig. 2. The Boreas scheduler and Zephyrus2 run in separate Docker containers and are arranged together in a service pod. They run on the master node alongside other Kubernetes system services. The

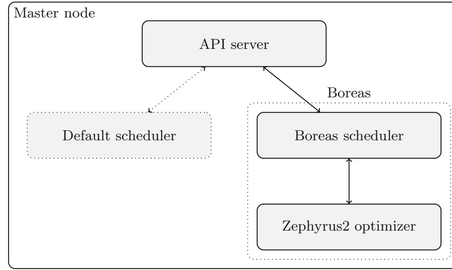

Fig. 2. Boreas from the master node perspective

Boreas scheduler communicates with the API server using the Kubernetes client API, and to Zephyrus2 using regular HTTP requests.

Boreas' workflow starts with collecting the pod deployment requests in batches. The batch size is limited to a maximum amount of events (99 by default). When this limit is reached or if a configurable number of seconds passes (e.g., 30 s by default), the accumulated requests are processed. The Boreas scheduler encodes the deployment of all the pod requests into an optimization problem for Zephyrus2 taking into account the request, the deployment constraints, and the current configuration of the cluster. Zephyrus2 is then invoked, and after processing the problem, it returns the optimal placement for each pod, if any. The Boreas scheduler parses the response and applies it by sending pod deployment instructions to the API server, like the default scheduler.

While Kubernetes is implemented in the Go programming language, any custom component can be implemented in another programming language due to its modular system architecture. Since we were interested in a proof-of-concept implementation, Boreas is implemented in Python. This choice does not call for efficiency, but since the heaviest task is the optimization of the configuration performed by Zephyrus2, the performance of the wrapping layer does not affect the overall scheduling performance. Boreas is constituted by about 400 lines of code and is freely available from the project's Github repository [3].

3.1 Deploying Boreas in Kubernetes

Boreas is deployed as a Kubernetes pod on the master node. The source code includes a deployment script that provides the configurations and privilege required to function as a custom scheduler. The script can be run by using `kubectl`, i.e., the command-line tool to control Kubernetes. Running the deployment script will download the containers and deploy the Boreas pod to the master node, as shown below.

```
$ kubectl create -f deployments/scheduler.yaml
serviceaccount/boreas-scheduler created
clusterrolebinding.rbac.authorization.k8s.io/boreas-
    ↪ scheduler-as-kube-scheduler created
deployment.apps/boreas-scheduler created
```

The Boreas pod will then be listed alongside the default services in the `kube-system` namespace, i.e., the default namespace used to run the pods implementing the core functionalities of Kubernetes.

```
$ kubectl get pods --namespace=kube-system
NAME                     READY   STATUS    RESTARTS   AGE
boreas-scheduler-<hash>  2/2     Running   0          60s
kube-scheduler-master    1/1     Running   0        1d12h
...
```

3.2 Integration with Zephyrus2

Since the Zephyrus2 container is running in the same pod as the Boreas scheduler, their containers can communicate using HTTP. The Boreas scheduler can therefore retrieve deployment configurations from Zephyrus2 simply by sending an HTTP post request to its container.

The Zephyrus2 tool was originally designed to minimize the cost of application deployment to virtual machines (VMs) [1]. While conceptually there is not a big difference between that problem and the placement of service pods on nodes in a Kubernetes cluster, in practice, extensive adjustments and conversions of the data and constraints had to be made before Zephyrus2 was able to process the placement of Kubernetes pods.

As a first operation, Boreas retrieves the status of every node in the cluster and encodes them into a location, as defined in the Aeolus model [9]. A node with its resources is simply seen as a location in which software components can be deployed. The CPU and the memory available on the node are seen as resources provided by the location. Since Zephyrus2 does not support fractional CPU specification while Kubernetes also allows millicores for pod consumption specification[2], the CPU values had to be rescaled of a factor of 1000. As an example, Listing 1.1 shows the JSON representation gathered for Zephyrus2 of a computation node. Lines 3–6 specifies that there is currently a node (`"num": 1`) that has a spare capacity of 3972 MB of RAM and 900 millicores.

[2] E.g. One CPU equals `1000m` where `m` stands for millicore and a Pod generally occupies few hundreds millicores.

Listing 1.1. Snippet of node encoding for Zephyrus2.

```
1 "locations": {
2   "k8s_worker_1": {
3     "num": 1,
4       "resources": {
5         "RAM": 3972,
6         "cpu": 900 }}}
```

In a Kubernetes cluster, services can be horizontally scaled by defining a number of running copies of pods within the so-called *replica set*. The API server creates individual scheduling events for each pod, including all the pods in a replica set. Passing every single request to Zephyrus2 as a separate request would greatly reduce its performance due to the increased number of components and constraints that would need to be considered. For this reason, Boreas compresses all requests for pods of a replica set into an equivalent unique request while processing events from the API server. The pod requests are then encoded in Zephyrus2 with the notion of a software component and a deployment constraint.

A pod is seen as a black box that requires a given amount of resources. As an example, Listing 1.2 shows the JSON representation of a `frontend` pod that requires 67 MB of RAM and 100 millicores.

Listing 1.2. Snippet of pod encoding for Zephyrus2.

```
12 "components": {
13   "frontend": {
14     "resources": {
15       "RAM": 67,
16       "cpu": 100 }}}
```

The deployment constraint is instead a conjunction of inequality that requires the installation of certain components in a given amount of resources and the metric to minimize. For example the deployment constraints requiring the installation of two frontend pods as encoded in Listing 1.2 is the following.

```
"specification": "frontend > 1; cost; (sum ?y in components: ?y)"
```

Here the first constraint `frontend > 1` imposes Zephyrus2 to search for configurations in which there are at least 2 `frontend` components. What follows after the semicolon is the definitions of the minimization metric used by Zephyrus2. In this case, Zephyrus2 proceeds to minimize the `cost` of the new deployment. Since no nodes have been defined by specifying a cost, by default, the nodes' costs are treated equally, and therefore this metric simply requires Zephyrus2 to minimize the number of nodes used for the deployment. The last part of the specification string `(sum ?y in components: ?y)` requires Zephyrus2 to break the possible ties between configurations using the same amount of nodes by further minimizing the total number of new pods deployed. In this specific case, since there are no

pod dependencies and the frontend was required in an amount strictly greater than 1, Zephyrus will produce a configuration using the least amount of nodes and deploying only two frontends.

The last ingredients taken into consideration by Boreas are affinities and anti-affinities constraints that allow to deploy pods on the same node or only in separate nodes. These constraints were a recent addition to the 1.6 version of Kubernetes [24], but are vital to guarantee the efficiency and reliability of the deployed application and thus frequently used in modern complex applications.

Listing 1.3. Example of affinity and anti-affinity relationships in Kubernetes.

```
1  affinity:
2    podAffinity:
3      requiredDuringSchedulingIgnoredDuringExecution:
4      - labelSelector:
5          matchExpressions:
6          - key: app
7            operator: In
8            values:
9            -  frontend
10       topologyKey: "kubernetes.io/hostname"
11   podAntiAffinity:
12     requiredDuringSchedulingIgnoredDuringExecution:
13     - labelSelector:
14         matchExpressions:
15         - key: app
16           operator: In
17           values:
18           -  backend
19       topologyKey: "kubernetes.io/hostname"
```

Kubernetes allows to define two types of intra pod affinities, a “hard” one that specifies rules that must be met for a pod to be scheduled and “soft” that specifies preferences that the scheduler will try to enforce but will not guarantee. Boreas, for the time being, considers the “hard” request since these are those that can not be violated and restrict the possible admissible configurations.

In Kubernetes, intra pod affinities and anti-affinities are expressed implicitly using labels assigned to pods, e.g., a pod can be affine to pods having a certain label. Labels allow a certain degree of flexibility but since they are not considered by the Aeolus formal model, Boreas has to compile all the affinity and anti-affinity relationships between pod labels into affinity and anti-affinity between pods. For this reason, Boreas gathers all the labels of batched pods, deployed, pods, and worker nodes and used them to create a reverse look-up function to represent a biunivocal relation between labels and components and nodes names, thus allowing to convert the constraints from labels to components and nodes. The affinity and anti-affinity constraints can thus be precisely defined in the declarative language supported by Zephyrus2.

As an example, Listing 1.3 presents a snippet for the definition of one affinity and one anti-affinity constraint for a `backend` pod deployable in Kubernetes. This snippet assumes that the `backend` has associated a label `app` with value `backend` while the `frontend` pod has associated the label `app` with value `frontend`. The `requiredDuringSchedulingIgnoredDuringExecution` in Lines 3 and 12 specify to Kubernetes that the two constraints are "hard" and must be satisfied during the scheduling. The pod affinity in Lines 4–9 states that the `backend` pod can be scheduled onto a node only if that node has at least one running pod with a label with the key `app` and value `frontend`. Similarly, the pod anti-affinity in Lines 13–18 state that a `backend` pod can not be installed on a node having a pod with a label with key `app` and value `backend`.[3] Finally, the `topologyKey` is used to define the domain of the application of the policy to a topology domain like node, rack, cloud provider zone, or cloud provider region. In this context we can abstract from these details, assuming that the policies apply to all the nodes.

As specified with constraint in Listing 1.3, Boreas detects that there is an affinity between the `backend` and the `frontend`, and an anti-affinity between two `backend` pods. The affinity is encoded as `(forall ?x in locations: (?x.backend ↪ > 0 impl ?x.frontend > 0))` that will require Zephyrus2 to consider configuration in which for all the possible locations `x` (i.e., for all the Kubernetes nodes), if the number of `backend` pods deployed on `x` (represented in Zephyrus2 as `?x.backend`) is greater than 0, then also on the same node the number of `frontend` must be greater than 0. This universal quantification of an implication thus excludes the possibility to have a node in which a `backend` is installed but no `frontend` is available. Similarly, Boreas will encode the self anti-affinity constraint as `(forall ?x in locations: (?x.backend <= 1))`. These constraints are added to the specification in conjunction with the constraints specifying the minimal amount of pods required (e.g., in conjunction with the constraint `frontend > 1`).

4 Evaluation

In this section we describe the experiments performed to compare Boreas w.r.t. the default scheduler proving that Boreas can deploy applications that the default scheduler can not.

Due to the lack of established benchmarks for deployment tasks, we set up two kinds of synthetic tests: i) a minimal test to prove that the heuristics of the default scheduler can prevent the full deployment of a simple application, and ii) a more elaborate affinity test using affinity and anti-affinity constraints in which the default scheduler behaves in a nondeterministic way, often preventing the deployment of the application.

[3] The Boreas scheduler supports hard pod affinities specified with the `In` operator. The full support of the other operators, e.g., `NotIn`, `Exists` and `DoesNotExist`) is trivial due to the fact that labels are finite at a given point in time and left as a future work.

The scheduling tests are regular Kubernetes deployment scripts, and were initiated with a single Kubernetes command line instruction (i.e., `kubectl create`). A test run is considered successful if the scheduler is able to find placements on worker nodes for all pods requests. Note that the schedulers may fail quite differently. The default scheduler processes as many service pods as possible, leaving some of them in a “Pending” state, meaning that it did not find space to deploy them. Boreas’s holistic approach leads instead to the deployment of all the pods or, if all the requests can not be satisfied, leave all pods in a pending state.

Due to the nondeterministic nature of the Kubernetes default scheduler, the evaluation tests were repeated 100 times and run on a small cluster of twelve Ubuntu 20.04 LTS servers running upstream Kubernetes 1.19. Each worker node contributed with 1 CPU and 4 GB of computing resources to the cluster and was built automatically using the open-source *infrastructure as code software tool* Terraform [31]. The Kubernetes software and its dependencies were installed and configured automatically using Ansible playbooks [2]. For solving the optimization problem, Zephyrus2 was configured to rely to OR-Tools [28], i.e., a state-of-the-art constraint solver. To reproduce the deployment, the scripts are available in the project’s Github repository [3].

Minimal Test. To verify the difference between Boreas and the default scheduler, the two schedulers were tasked with the deployment of a new system requiring the deployment of two `backend` pods and three `frontend` pods on two empty nodes, as visually depicted in Fig. 3. This deployment was set up to require all available CPU resources on two worker nodes.

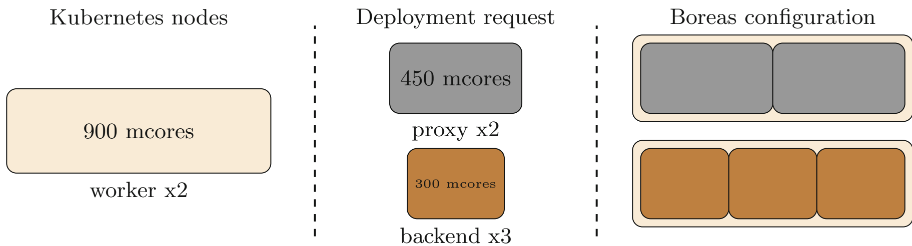

Fig. 3. Minimal test requirements and Boreas configuration

In the Boreas case, the scheduling requests are batched and, as expected, the optimal allocation was always found. On the other hand, the default scheduler had a hard time finding a placement for all five service pods. Its one-at-a-time approach forces it to select a placement for the first service pods without being able to plan for the resources needed for the other service pods. The default scheduler’s algorithm for ranking available worker nodes makes it prone to place the first service pods where they will block later service pods in a resource-scarce

scenario. Among all the 100 repetitions, none of the deployments were successful when using the default scheduler.

Affinity Test. In the second and more elaborate test we tested the deployment of an application constituted by a reverse proxy server such as Nginx [27] for incoming HTTP requests, frontend and backend components of a web application, and a message broker such as Redis [29] to queue long-running tasks from the backend to separate threads. The application is an instance of the typical web frontend with backend services and cache and is derived from the production ready Online Boutique[4] which can scale up to handle millions of users given the proper amount of resources.

With our deployment constraints, an optimal deployment for this system requires four worker nodes and, differently from the previous test, only 83% of the total amount of the CPU resources are needed. As illustrated in Fig. 4, for redundancy and load balancing purposes 3 backends, 3 frontends, and 3 message brokers are required. Moreover, the system also requires 2 proxy services for communication with the outside world. The backend and frontend have anti-affinity to themselves, thus requiring at most one copy of each in a node. Moreover, the frontend has an affinity to the backend requiring for performance reasons to be deployed in the same node.

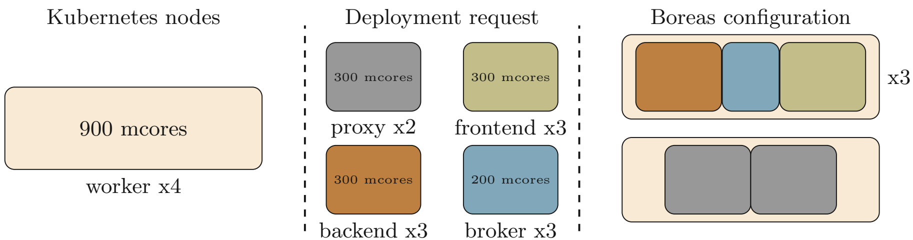

Fig. 4. Affinity test requirements and Boreas configuration

By repeating 100 times the deployment of this system starting from 4 empty nodes, we have noticed that the default scheduler has a success rate of 34%. As with the basic test, its failures result from the earlier placements of service pods blocking the placement of the ones that are scheduled later.[5] When the default scheduler fails, it will not be able to deploy one or two of the pods in the test, usually either a `proxy` or `backend` pod, due to the lack of a suitable node. Boreas, on the other hand, succeeded each test run, giving rise to the

[4] https://github.com/GoogleCloudPlatform/microservices-demo.

[5] Please note that even though the request are given at once with the command `kubectl create`, the default scheduler sequentializes the requests.

configuration depicted in Fig. 4.[6] The evaluation shows that there are resource-scarce scenarios or complex deployment constraints scenarios where the Boreas scheduler finds placement for services that the default scheduler is unable to find.

In the first set of minimal test, Boreas takes an average of 1.82 s to compute the deployment and 2.15 in the affinity test. Moreover, when the number of replicas in the affinity example is scaled to require a cluster of 8 or 12 nodes, (thus optimizing the placement of a total 22 or 33 pods respectively), Boreas takes 2.92 and 4.22 s in average to compute the optimal placement. For this reason, we conjecture that the majority of the time taken by Boreas for these simple optimization problems is spent on the exchange of messages and in the initialization of Zephyrus2. Trying to reduce the running time by integrating more tightly Zephyrus or an ad-hoc reasoner directly in the Boreas scheduler is beyond the scope of this work and left as future work.

We would like to note that the deployment optimization in Boreas, being an NP-hard problem, does not provide any time guarantee for the returning of the result. The resource consumption of the Boreas scheduler requires less than 50 MB and 400 millicores for scheduling up to 50 pods in 10 nodes. This amount of resources is negligible considering that the current recommended settings for a master node of Kubernetes with 11–100 nodes are 4 vCPUs and 15 GB of memory and slightly above the footprint of the default scheduler that consumes 27 MB and 5 millicore for handling a queue of 50 scheduling events. While the Boreas Scheduler has a low resource consumption, the NP-hardness of the optimization problem solved by Zephyrus2 can also have an impact on the footprint of the Zephyrus2 optimizer container that can vary depending on the nature of the optimization problem and the backend solver used to solve it. Based on Zephyrus2's benchmarks [1] we conjecture that Boreas can be used to deploy up to hundreds of pods in clusters with up to a dozen nodes in less than a minute.[7]

5 Related Work

Kubernetes is a complex ecosystem that rely of a set of plugins and extensions that improve and extend its functionalities. Aside for the scheduler, there are plenty of other approaches that substitute and complement the default implementation. For example, plugins like Istio [18] and Linkerd [22] complement the native handling service-to-service communication with a service mesh.

[6] Note that Boreas can compute configurations that are not robust like the one presented in Figure 4 that has two proxies deployed on the same node. It is the user responsibility to define all the constraints to make the final configuration robust stating, e.g., all the anti-affinity constraints.

[7] Additional example of bigger system requiring more computation time from Zephyrus2 (i.e., less than a minute for clusters up to 10 nodes) can be found in the project's Github repository [3].

If we restrict to consider Kubernetes schedulers, different scheduler have been designed to exploit deployment heuristics to try to optimize some resources. As an example, RLSK [15] is a deep Reinforcement Learning Scheduler for Kubernetes that uses reinforcement learning for the refinement of deployment heuristic. To improve resource distribution, Zhang [34] proposed to combine an ant colony and particle swarm optimization algorithms. Li et al. [25] introduced a dynamic Input/Output sensing scheduler for Kubernetes. The scheduler considers the disks pressure in the scheduling process and tries to balance the node disk I/O usage across the cluster dynamically. Similarly, Gaia [30] is a scheduler specifically designed to improve GPUs load distribution, treating GPU resources in the same way Kubernetes treats CPUs. Townend et al. [32] and Wang et al. [33] studied schedulers to reduce energy consumption and heat waste. Poseidon-Firmament [21] is instead a scheduler designed to be faster than the default Kubernetes scheduler on bigger clusters. Differently than Boreas, all these approaches, are neither complete nor optimal, polynomial in the size of the cluster and the number of pods to deploy and thus privileging speed over optimality.

Not focusing on Kubernetes, the closest works to ours is Aeolus Blender [6,7] that combines the first version of Zephyrus [8] with the Metis planner [23] and the Mandriva Armonic collection into a tool chain that automates ad-hoc deployment tasks. Differently that in our approach, the application domain of Aeolus blender was narrower and they did not combine the configuration optimizer with an established and constraint-rich orchestration tool. Similarly, the Jolie redeployment optimiser [11] used Zephyrus with a reconfiguration coordinator to redeploy micro-services when they are reconfigured. In this case, the service orchestration would be handled by the Jolie redeployment optimiser itself. SmartDepl [13] presents instead an extension to the Abstract Behavioural Specification language (ABS) [16] allowing users to specify costs and other deployment requirements using ABS classes and outputs a deployment configuration by using Zephyrus2. The final configuration can be simulated or formally checked by using the formal methods tools available for ABS.

Also relevant, is the work of Medea [12] that introduces a two-scheduler design for clusters where long-running service containers are deployed together with short-running batch containers. Medea, differently than us, was implemented as an extension to the Apache Hadoop cluster scheduler and finds placement for the long-running containers, leaving the short-running containers for the default scheduler in order to keep scheduling latency low.

6 Conclusion

In this work, we presented an alternative scheduler that optimizes the resource usage and costs of a Kubernetes cluster, i.e., the most used container orchestrator. The new scheduler, Boreas, relies on a configuration optimizer and on a formal model for the cloud deployment. We have shown that Boreas is able to deploy applications that the default scheduler failed to deploy.

Boreas can be used for Kubernetes clusters having only one scheduler per cluster or one scheduler per zone in the case the cluster is divided into zones. Our approach assumes that the components of the application have been profiled to establish their RAM and vCPU consumption, around 50 MB of RAM in the Kubernetes master node, and the selection of a suitable time window for the grouping of the deployment requests and their concurrent deployment. Solving a NP-hard problem, Boreas can not guarantee answers in a short amount of times and therefore, it is mainly targeting clusters encompassing a limited amount of computing nodes (e.g., dozen nodes) and applications that do not require high variability (i.e., less than a hundred new deployment requests per minute).

Boreas is only a proof-of-concept implementation that does not support all the deployment constraints recently introduced in Kubernetes. We plan to extend it further to capture all the possible varieties of deployment constraints (e.g., affinity constraints with different matching criteria for pod labels) and also improve the resolution of the optimization problem. Further evaluations and tests are required to study the impact of possible backup plans for situations in which the solver can not prove the optimality of the solution in time (e.g., use the best solution so far retrieved or the solution produced by the default scheduler). In particular, we are interested in leveraging Boreas to solve the local deployment problems created when topology spread constraints are used. These constraints, introduced in the 1.19 recent version of Kubernetes, are used to control how pods are spread across the cluster regions, zones, or nodes and can split the global problem of scheduling pods into smaller sub-problems that can be solved independently.

Aside from improving Boreas, we are also interested in providing comprehensible explanations for the DevOps operators managing a Kubernetes cluster when a system is not deployable. This can be achieved by exploiting the conflicting constraints found by Boreas when solving the deployment problem. Moreover, inspired by [4,13], we are also interested in introducing more complex deployment constraint directly in Kubernetes to describe dependencies between the pods that allow the cluster operators to avoid "domino" effects due to unstructured scaling actions that may cause cascading slowdowns or outages [35].

References

1. Ábrahám, E., Corzilius, F., Johnsen, E.B., Kremer, G., Mauro, J.: Zephyrus2: on the fly deployment optimization using SMT and CP technologies. In: Fränzle, M., Kapur, D., Zhan, N. (eds.) SETTA 2016. LNCS, vol. 9984, pp. 229–245. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47677-3_15
2. Ansible - simple IT automation. https://www.ansible.com
3. Boreas scheduler (source code). https://github.com/torgeirl/boreas-scheduler
4. Bravetti, M., Giallorenzo, S., Mauro, J., Talevi, I., Zavattaro, G.: Optimal and automated deployment for microservices. In: Hähnle, R., van der Aalst, W. (eds.) FASE 2019. LNCS, vol. 11424, pp. 351–368. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16722-6_21

5. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. Queue **14**, 70–93 (2016)
6. Catan, M., et al.: Aeolus: mastering the complexity of cloud application deployment. In: Lau, K.-K., Lamersdorf, W., Pimentel, E. (eds.) ESOCC 2013. LNCS, vol. 8135, pp. 1–3. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40651-5_1
7. Di Cosmo, R., Eiche, A., Mauro, J., Zacchiroli, S., Zavattaro, G., Zwolakowski, J.: Automatic deployment of services in the cloud with aeolus blender. In: Barros, A., Grigori, D., Narendra, N.C., Dam, H.K. (eds.) ICSOC 2015. LNCS, vol. 9435, pp. 397–411. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48616-0_28
8. Di Cosmo, R., et al.: Automated Synthesis and Deployment of Cloud Applications. In: ASE. ACM (2014)
9. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: a component model for the cloud. Inf. Comput. **239**, 100–141 (2014)
10. Fazio, M., Celesti, A., Ranjan, R., Liu, C., Chen, L., Villari, M.: Open issues in scheduling microservices in the cloud. IEEE Cloud Comput. **3**(5), 81–88 (2016)
11. Gabbrielli, M., Giallorenzo, S., Guidi, C., Mauro, J., Montesi, F.: Self-reconfiguring microservices. In: Theory and Practice of Formal Methods. Springer, Cham (2016)
12. Garefalakis, P., Karanasos, K., Pietzuch, P., Suresh, A., Rao, S.: Medea: scheduling of long running applications in shared production clusters. In: EuroSys. ACM (2018)
13. de Gouw, S., Mauro, J., Zavattaro, G.: On the modeling of optimal and automatized cloud application deployment. JLAMP **107**, 108–135 (2019)
14. Hightower, K., Burns, B., Beda, J.: Kubernetes: Up & Running: Dive into the Future of Infrastructure. O'Reilly Media, New York (2017)
15. Huang, J., Xiao, C., Wu, W.: RLSK: a job scheduler for federated kubernetes clusters based on reinforcement learning. In: IC2E. IEEE (2020)
16. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
17. Korte, B., Vygen, J.: Combinatorial Optimization: Theory and Algorithms, 5th edn. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-24488-9
18. Managing microservices with the istio service mesh (2017). https://kubernetes.io/blog/2017/05/managing-microservices-with-istio-service-mesh
19. Scheduler algorithm in kubernetes (kubernetes community docs). https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scheduling/scheduler_algorithm.md
20. Extending your Kubernetes Cluster (Kubernetes v1.19 Docs). https://v1-19.docs.kubernetes.io/docs/concepts/extend-kubernetes/extend-cluster
21. Poseidon-Firmament. https://v1-16.docs.kubernetes.io/docs/concepts/extend-kubernetes/poseidon-firmament-alternate-scheduler
22. Linkerd - A different kind of service mesh. https://linkerd.io/2/overview
23. Lascu, T.A., Mauro, J., Zavattaro, G.: Automatic deployment of component-based applications. Sci. Comput. Program. **113**, 261–284 (2015)
24. Lewis, I.: Advanced scheduling in kubernetes 2017. Kubernetes Blog online Posted 2017–03–31 (2017). https://kubernetes.io/blog/2017/03/advanced-scheduling-in-kubernetes

25. Li, D., Wei, Y., Zeng, B.: A dynamic I/O sensing scheduling scheme in Kubernetes. In: HPCC. ACM (2020)
26. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux J. **2014**(239), 2 (2014)
27. Nginx - a high-performance HTTP server and reverse proxy. https://nginx.com
28. Perron, L.: Operations research and constraint programming at Google. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, p. 2. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23786-7_2
29. Redis - an in-memory database. https://redis.io
30. Song, S., Deng, L., Gong, J., Luo, H.: Gaia scheduler: a kubernetes-based scheduler framework. In: ISPA/IUCC/BDCloud/SocialCom/SustainCom. IEEE (2018)
31. Terraform - an infrastructure as code software tool. https://www.terraform.io
32. Townend, P., et al.: Improving data center efficiency through holistic scheduling in kubernetes. In: SOSE. IEEE (2019)
33. Wang, S., Sheng, Q.Z., Li, X., Mahmood, A., Zhang, Y.: Energy minimization for cloud services with stochastic requests. In: Kafeza, E., Benatallah, B., Martinelli, F., Hacid, H., Bouguettaya, A., Motahari, H. (eds.) ICSOC 2020. LNCS, vol. 12571, pp. 133–148. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-65310-1_11
34. Wei-guo, Z., Xi-lin, M., Jin-zhong, Z.: Research on Kubernetes' resource scheduling scheme. In: ICCNS. ACM (2018)
35. Woods, D.: On infrastructure at scale: a cascading failure of distributed systems. https://medium.com/@daniel.p.woods/on-infrastructure-at-scale-a-cascading-failure-of-distributed-systems-7cff2a3cd2df