Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing applications

José Santos*, Tim Wauters*, Bruno Volckaert* and Filip De Turck*
* Ghent University - imec, IDLab, Department of Information Technology
Technologiepark-Zwijnaarde 126, 9052 Gent, Belgium
Email: josepedro.pereiradossantos@UGent.be

***Abstract*— Nowadays, the Internet of Things (IoT) continues to expand at enormous rates. Smart Cities powered by connected sensors promise to transform public services from transportation to environmental monitoring and healthcare to improve citizen welfare. Furthermore, over the last few years, Fog Computing has been introduced to provide an answer to the massive growth of heterogeneous devices connected to the network. Nevertheless, providing a proper resource scheduling for delay-sensitive and data-intensive services in Fog Computing environments is still a key research domain. Therefore, in this paper, a network-aware scheduling approach for container-based applications in Smart City deployments is proposed. Our proposal has been validated on the Kubernetes platform, an open source orchestrator for the automatic management and deployment of micro-services. Our approach has been implemented as an extension to the default scheduling mechanism available in Kubernetes, enabling Kubernetes to make resource provisioning decisions based on the current status of the network infrastructure. Evaluations based on Smart City container-based applications have been carried out to compare the performance of the proposed scheduling approach with the standard scheduling feature available in Kubernetes. Results show that the proposed approach achieves reductions of 80% in terms of network latency when compared to the default scheduling mechanism.**

***Index Terms*—Smart Cities, IoT, Fog Computing, Resource Provisioning, Kubernetes**

I. Introduction

Recently, the Internet of Things (IoT) has been reshaping our cities by transforming objects of everyday life into smart devices. These devices are revolutionizing different domains of urban life, such as public transportation, street lighting and environmental monitoring. According to [1], the number of Low-Power Wide-Area Network (LPWAN) devices is expected to grow from less than 1 percent in 2016 to 8.9 percent by 2021, from 58 million devices in 2016 to over 1 billion by 2021, which will inevitably lead to a need for change in our current cloud infrastructure. To meet these requirements, Fog Computing [2] is emerging as a distributed cloud solution to meet the future demand of IoT applications [3]. It provides nearby resources to end devices to perform real-time processing, data analytics operations and storage procedures overcoming the limitations of traditional centralized cloud infrastructures. Waste management platforms, real-time video streaming and smart transportation systems are already envisioned Smart City scenarios [4]. Nevertheless, Fog Computing is still in its early stages thus research challenges in terms of resource provisioning and service scheduling persist. In fact, setting up a proper infrastructure to serve billions of IoT devices and their applications, in real-time, without dismissing the importance of energy efficiency, bandwidth usage and geographic distribution is still a challenge to be addressed [5], [6].

Container-based micro-services are currently revolutionizing the way developers build their applications [7]. An application is decomposed in a set of lightweight containers deployed across a large number of servers instead of the traditional single monolithic application. Each micro-service is developed and deployed separately, without compromising the application life-cycle. Currently, containers are the *de facto* alternative to the traditional Virtual Machines (VMs), due to their high portability and low resource usage. Furthermore, with their wide adoption, multiple orchestration solutions are being developed by IT companies and open-source communities. Among them, the most widely used today is called Kubernetes [8]. Kubernetes is an open-source orchestration platform for automatic deployment, scaling, and management of containerized applications. Although containers already provide a high level of abstraction, they still need to be properly managed, specifically in terms of resource scheduling, load balancing and server distribution, and this is where integrated solutions like Kubernetes come into their own [9]. Kubernetes simplifies the deployment of reliable, scalable distributed systems by managing their complete life-cycle work-flow. Although Kubernetes already supports policy-rich and topology-aware features, the service scheduling merely takes into account the number of requested resources on each host, which is rather limited when dealing with IoT-based services. Therefore, in this paper, a network-aware approach is presented to provide up-to-date information about the current status of the network infrastructure to enable Kubernetes to make more informed resource allocation decisions. Our approach has been designed and implemented as an extension to the default scheduling mechanism available in Kubernetes. Finally, evaluations have been performed to validate our proposal, specifically for container-based Smart City applications. The performance of our approach has been compared with the standard scheduling feature present in Kubernetes.

The remainder of the paper is organized as follows. In the next Section, related work is discussed. Section III introduces the Kubernetes architecture and its scheduling functionality. In Section IV, the proposed network-aware scheduling is

presented. Then, in Section V, the evaluation approach is described which is followed by the evaluation results in Section VI. Finally, conclusions are presented in Section VII.

II. Related Work

A handful of research efforts has been already performed in the context of resource provisioning for cloud environments. In [10], the problem of scheduling container-based micro-services over multiple VMs has been addressed. Their approach focused on reducing the overall turnaround time and the total traffic generated for the complete end-to-end service. In [11], a rank scheduling approach has been presented. It focuses on optimizing the resource usage by sorting each task based on its computational need. Furthermore, in [12], an adaptive approach for dynamic resource provisioning in the cloud based on continuous reinforcement learning has been discussed. Their approach considered the uncertainty present in the cloud market, where each interaction between different cloud providers can have an impact on attracting and maintaining customers.

In recent years, resource provisioning research specifically tailored to IoT and Smart Cities has been carried out. In [13], an approach for the prediction of future resource demands in terms of service usage and Quality of Service (QoS) in the context of multimedia IoT devices has been presented. Moreover, in [14], a programming infrastructure for the deployment and management of IoT services has been discussed. The proposal includes migration mechanisms for the reallocation of services between multiple Fog nodes. In [15], several container orchestration technologies have been evaluated to measure the performance impact of running container-based micro-services in Fog nodes. In [16], a model for the deployment of IoT applications in Fog Computing scenarios has been presented and implemented as a prototype called FogTorch. The proposal focused on fulfilling not only hardware and software demands but also QoS requirements, such as network latency and bandwidth.

Although the existing and ongoing research cited address resource provisioning issues present in Cloud and Fog Computing environments, they have not yet delivered an integrated solution since most research has been focused on theoretical modeling and simulation studies. The proposed network-aware approach in this paper builds further on [17], where a resource provisioning Integer Linear Programming (ILP) model for the IoT service placement problem in Smart Cities has been presented. The ILP formulation takes into account not only cloud requirements but also wireless constraints while optimizing IoT services for network latency, energy efficiency and bandwidth usage. Solutions implemented in our earlier work have been implemented as an extension to the scheduler feature already present in the Kubernetes platform. Our approach allows the Kubernetes scheduler to make more informed resource allocation decisions based on up-to-date information about the current network status. To the best of our knowledge, our approach goes beyond the current state-of-the-art by extending the Kubernetes platform with network-aware scheduling mechanisms for the proper provisioning of IoT-based services. Performance evaluations based on container-based Smart City services have been carried out to compare our approach with the standard scheduling feature present in Kubernetes.

III. Kubernetes: Empowering Orchestration of Smart City Applications

This section introduces the Kubernetes orchestration platform. First, the Kubernetes architecture and main concepts are presented. Then, the scheduling functionality of Kubernetes is discussed.

A. Kubernetes Architecture

The Kubernetes architecture is shown in Fig. 1. The architecture follows the master slave model, where at least one master node manages Docker [18] containers across multiple worker nodes (slaves). These worker nodes can be local physical servers and VMs or even public and private clouds. The Master is responsible for exposing the Application Program Interface (API) server, scheduling the service deployments and managing the overall cluster. The *API server* is implemented through a RESTFul interface, which provides an entry point to control the entire Kubernetes cluster. Users can send commands to the API server through the built-in Kubernetes Command Line Interface (CLI), known as *kubectl*. *Etcd* is a key-value pair distributed data storage used for service discovery, coordination of resources and sharing cluster configurations. Etcd allows the other components to synchronize themselves based on the desired state of the cluster. Scheduled jobs, created and deployed micro-services, namespaces and replication information are examples of data stored in Etcd. Furthermore, the Kubernetes node agent known as *Kubelet* is responsible for recognizing discrepancies between the desired state and the actual state of the cluster. When this happens, Kubelet launches or terminates the necessary containers to reach the desired state described by the API server. Kubelet runs on each node of the cluster and is also responsible for reporting events, resource usage, among others. Then, the *Controller Manager* is responsible for monitoring Etcd and the overall state of the cluster. If the state of the system changes, the Controller Manager communicates the desired state of the system through the API server.

Micro-services in Kubernetes are often tightly coupled forming a group of containers. This group is the smallest working unit in Kubernetes, which is called a *pod*. A pod represents the collection of containers and volumes (storage) running in the same execution environment [9]. The containers inside a pod share the same IP Address and port space (namespace). Containers in different pods are isolated from one another since they own different IP addresses, different hostnames, etc. The main limitation is that two services listening on the same port cannot be deployed inside the same pod. Based on the service requirements and on the available resources, the master schedules the pod on a specific node. Then, the assigned node pulls the container images from the image registry if needed

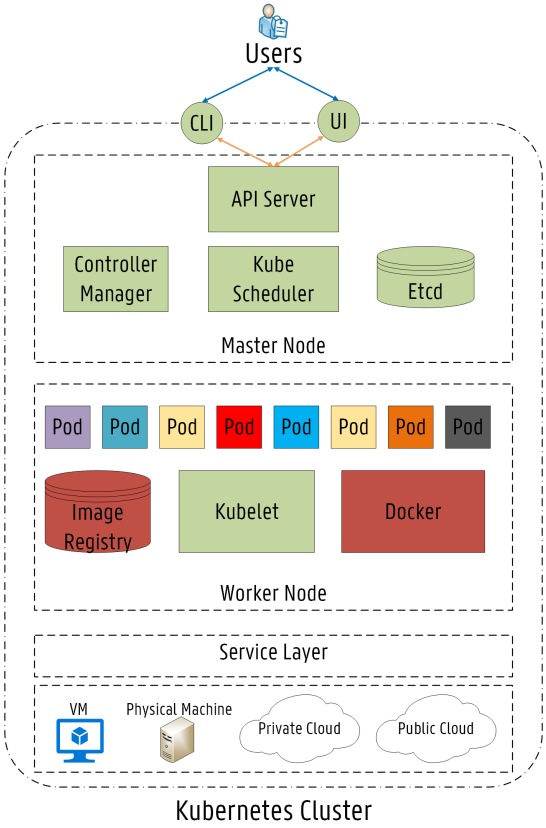

Fig. 1: High-level view of the Kubernetes Architecture.

and coordinates the necessary operations to launch the pod. The scheduling operation of assigning a node of the cluster to each pod is performed by the *Kube-scheduler (KS)*. The KS is the default scheduling component in the Kubernetes platform, which is responsible for deciding where a specific pod should be deployed. The KS operations are further detailed in the next section.

B. Kube-Scheduler (KS) - Scheduling features on Kubernetes

The KS decision making process is shown in Fig. 2. Every pod needing allocation is added to a waiting queue, which is continuously monitored by the KS. If a pod is added to the waiting queue, the KS searches for a suitable node for the deployment based on a two step process. The first step is called *node filtering*, where the KS verifies which nodes are capable of running the pod by applying a set of filters, also known as *predicates*. The purpose of filtering is to solely consider nodes meeting all specific requirements of the pod further in the scheduling process. The second operation is named *node priority calculation*, where the KS ranks each remaining node to find the best fit for the pod provisioning based on one or more scheduling algorithms, also called *priorities*. The KS supports the following predicates [8]:

1) **PodFitsResources:** If the free amount of resources (CPU and memory) on a given node is smaller than the one required by the pod, the node must not be further considered in the scheduling process. Therefore, the node is disqualified.
2) **NoDiskConflict:** This predicate evaluates if a pod can fit due to the volumes it requests, and those that are already mounted.
3) **NoVolumeZoneConflict:** This predicate checks if the volumes a pod requests are available through a given node due to possible zone restrictions.
4) **PodFitsHostPorts:** For instance, if the pod requires to bind to the host port 80, but another pod is already using that port on the node, this node will not be a possible candidate to run the pod and, therefore, it will be disqualified.
5) **CheckNodeMemoryPressure:** This predicate checks if a pod can be allocated on a node reporting memory pressure condition. Currently, Best Effort pods should not be placed on nodes under memory pressure, since they are automatically deassigned from the node.
6) **CheckNodeDiskPressure:** This predicate evaluates if a pod can be scheduled on a node reporting disk pressure condition. Pods can currently not be deployed on nodes under disk pressure since they are automatically deassigned.
7) **MatchNodeSelector (Affinity/Anti-Affinity):** By using node selectors (labels), it is possible to define that a given pod can only run on a particular set of nodes with an exact label value (node-affinity), or even that a pod should avoid being allocated on a node that has already certain pods deployed (pod-anti-affinity). These rules can be created by declaring *Tolerations* in the pod configuration files to match specific node *Taints*. Essentially, affinity rules are properties of pods that attract them to a set of nodes or pods while taints allow nodes to repel a

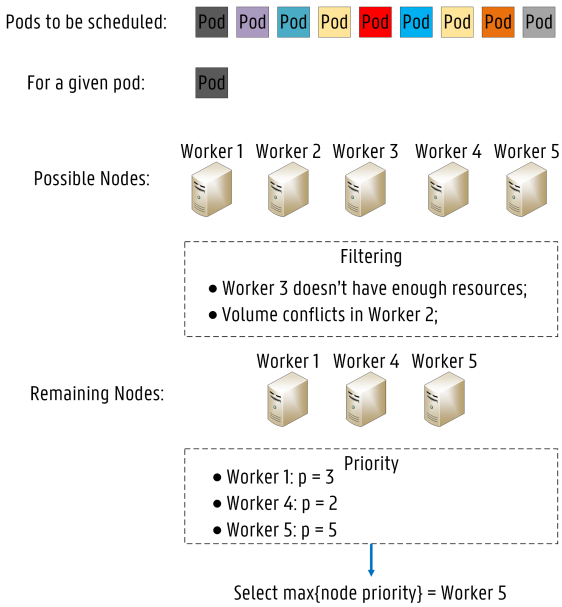

Fig. 2: The Scheduling procedure of the Kube-Scheduler.

given set of pods. Taints and tolerations ensure that pods are not deployed into inappropriate nodes. Both are important mechanisms to fine-tune the scheduling behavior of Kubernetes. Node selectors provide a flexible set of rules, on which the KS bases its scheduling decision by filtering specific nodes (node affinity/anti-affinity), by preferring to deploy certain pods close or even far away from other pods (pod affinity/anti-affinity), or just on node labels favored by the pod (taints and tolerations).

By using these predicates, the KS knows in advance which nodes are not suitable for the pod deployment, thus it will remove those from the list of possible candidates. On one hand, after completion of the filtering process, finding no capable nodes for the pod deployment is always a possibility. In that case, the pod remains unscheduled and the KS triggers an event stating the reason for the failed deployment. On the other hand, if several candidates are retrieved after completion of the filtering operation, the KS triggers the node priority calculation. The node priority calculation is based on a set of priorities, where each remaining node is given a score between 0 and 10, 10 representing "perfect fit" and 0 meaning "worst fit". Then, each priority is weighted by a positive number depending on the importance of each algorithm and the final score of each node is calculated by adding up all the weighted scores [8]. The highest scoring node is chosen to run the pod. If more than one node is classified as the highest scoring node, then, one of them is randomly selected. The KS supports the following priorities:

1) **LeastRequestedPriority:** The node is scored according to the fraction of CPU and memory (free/allocated). The node with the highest free fraction is the most preferred for the deployment. This priority function spreads the pods across the cluster based on resource consumption.
2) **MostRequestedPriority:** This priority algorithm is the opposite of the one above. The node with the highest allocated fraction of CPU and memory is the most preferred for the deployment.
3) **BalancedResourceAllocation:** This priority function ranks nodes based on the cluster CPU and Memory usage rate. The purpose is to balance the resource allocation after the pod provisioning.
4) **SelectorSpreadPriority:** This priority algorithm tries to minimize the number of deployed pods belonging to the same service on the same node or on the same zone/rack.
5) **CalculateAntiAffinityPriority:** This priority function scores nodes based on anti-affinity rules. For instance, spreading pods in the cluster by reducing the same number of pods belonging to the same service on nodes with a particular label.
6) **NodeAffinityPriority:** In this case, nodes are ranked according to node-affinity rules. For instance, nodes with a certain label are scored higher than others.
7) **InterPodAffinityPriority:** This priority algorithm scores nodes based on pod affinity rules. For example, nodes with certain pods already allocated are scored higher since it is preferred to deploy the given pod close to these pods.
8) **ImageLocalityPriority:** Remaining nodes are ranked according to the location of the requested pod container images. Nodes already having the requested containers installed are scored higher.
9) **TaintTolerationPriority:** This priority function scores nodes based on their taints and the correspondent tolerations declared in the pod configuration files. Remaining nodes are preferred according to the fewer number of intolerable taints on them for the given pod.

Predicates are evaluated to dismiss nodes that are incapable of running the given pod while priorities are designed to rank all the remaining nodes that can deploy the pod. For example, given a pod which requires half a core (0.5) CPU, the *PodFitsResources* predicate returns *False* for a node which only has 400 millicpu free. Furthermore, for the same pod, the *LeastRequestedPriority* priority ranks a node which has 2.5 CPU cores free higher than one which has only 600 millicpu left, even though both nodes can accommodate the pod (assuming they have the same CPU capacity). Additionally, it should be noted that the KS searches for a node for each pod, one at a time. The KS does not take into account the remaining amount of pod requests still waiting for deployment. After the scheduling decision is made, the KS informs the API server indicating where the pod must be scheduled. This operation is called *Binding*.

Another important feature of Kubernetes is resource requests and limits. Developers should specify resource requests and limits on the pod configuration files. A resource request is the minimum amount of resources (CPU and/or memory) needed by all containers in the pod while a resource limit is the maximum amount of resources that can be allocated for the containers in a pod. Pods can be categorized in three QoS classes depending on resource requests and limits:

1) **Best Effort (lowest priority):** A Best Effort pod has neither resource requests or limits on its configuration files for each of its containers. These pods are the first ones to get killed in case the system runs out of memory.
2) **Burstable:** A Burstable pod has all containers with resource requests lower than their resource limits. If a container needs more resources than the ones requested, the container can use them as long as they are free.
3) **Guaranteed (highest priority):** A guaranteed pod has resource requests for all its containers equal to the maximum resource needs that the system will allow the container to use (resource limit).

When pods have specified resources requests, the KS can provide better decisions in terms of scheduling and when pods have described resource limits, resource contention can be handled properly [19]. When several containers are running on the same node, they compete for the available resources. Since container abstraction provides less isolation than VMs, sharing physical resources might lead to a performance degradation called resource contention. These rules enable Kubernetes to

properly manage cluster resources. However, developers still need to properly set up these resource requests and limitations, because containers often do not use the entire amount of resources requested which leads to wasted resources. For instance, 6 pods have been deployed and each one is requesting 1Gb of RAM in a node with 6GB RAM capacity, but each pod only uses 500 MB of RAM. The KS could allocate more pods into that node, but due to the incorrect resource requests, it will never schedule.

IV. Network-Aware Scheduling extension in Kubernetes

Although the KS provides flexible and powerful features, the metrics applied in the decision making process are rather limited. Only CPU and RAM usage rates are considered in the service scheduling while latency or bandwidth usage rates are not considered at all. When IoT or Smart City environments are considered, latency restrictions are highly challenging since delay-sensitive applications, involving connected vehicles and disaster monitoring should react to dynamic changes in the order of milliseconds. If the threshold is exceeded, the service can become unstable, action commands may arrive too late and control over the service is lost. Therefore, it is crucial to ensure that the service is deployed on a node capable of providing low response times in the communication.

Another constraint is bandwidth. Even when the computational resources can handle all user requests, bandwidth limitations may cause network congestion and even service disruptions, which could eventually lead to service interruptions. As not all QoS levels can be maintained during these events, proper resource scheduling is needed for Fog Computing environments. A suitable provisioning approach must consider multiple factors, such as the applications' specific requirements (CPU and memory), the state of the infrastructure (hardware and software), the network status (link bandwidth and latency), among others. Thus, this paper proposes a network-aware scheduling extension to Kubernetes, which provides up-to-date information about the current status of the network infrastructure. Kubernetes describes three ways of extending the KS:

1) Adding new predicates and/or priorities to the KS and recompiling it.
2) Implementing a specific scheduler process that can run instead of or alongside the KS.
3) Implementing a "scheduler extender" process that the default KS calls out as a final step when making scheduling decisions.

The third approach is particularly suitable for use cases where scheduling decisions need to be made on resources not directly managed by the standard KS. The proposed network-aware scheduler has been implemented based on this third approach, since information on the current status of the network infrastructure is not available throughout the scheduling process of the KS. Essentially, when the KS tries to schedule a pod, the extender call allows an external process to *filter* and/or *prioritize* nodes. Two separate calls are issued to the extender, one for filtering and one for prioritizing actions. The arguments passed on to the *FilterVerb* endpoint consist of the set of nodes filtered through the KS predicates and the given pod while the arguments passed to the *PrioritizeVerb* endpoint also include the priorities for each node. The filter step is used to further refine the list of possible nodes. The prioritize call is applied to perform a different kind of priority function, where the correspondent node scores are added to the default priority list and used for the final node selection. Nevertheless, it should be noted that the three approaches are not mutually exclusive.

TABLE I: The RTT labels of our Fog Computing infrastructure

Node	RTT-A	RTT-B	RTT-C	RTT-D	RTT-E
Master	32 ms	32 ms	32 ms	32 ms	4 ms
Worker 1	64 ms	64 ms	4 ms	14 ms	32 ms
Worker 2	64 ms	64 ms	4 ms	14 ms	32 ms
Worker 3	64 ms	64 ms	4 ms	14 ms	32 ms
Worker 4	4 ms	14 ms	64 ms	64 ms	32 ms
Worker 5	4 ms	14 ms	64 ms	64 ms	32 ms
Worker 6	4 ms	14 ms	64 ms	64 ms	32 ms
Worker 7	64 ms	64 ms	14 ms	4 ms	32 ms
Worker 8	64 ms	64 ms	14 ms	4 ms	32 ms
Worker 9	64 ms	64 ms	14 ms	4 ms	32 ms
Worker 10	14 ms	4 ms	64 ms	64 ms	32 ms
Worker 11	14 ms	4 ms	64 ms	64 ms	32 ms
Worker 12	14 ms	4 ms	64 ms	64 ms	32 ms
Worker 13	32 ms	32 ms	32 ms	32 ms	4 ms
Worker 14	32 ms	32 ms	32 ms	32 ms	4 ms

Based on Kubernetes affinity/anti-affinity rules and node labels, a complete labeling of a Fog Computing infrastructure has been conducted. The infrastructure is shown in Fig. 3. As presented, the infrastructure is composed of a Kubernetes cluster with 15 nodes (1 master node and 14 worker nodes). Nodes are classified with labels $\{$*Min, Medium, High*$\}$ for keywords $\{$*CPU, RAM*$\}$, depending on their resource capacity. Additionally, nodes are classified in terms of device type, by classifying them with taints $\{$*Cloud, Fog*$\}$ for the keyword $\{$*DeviceType*$\}$ and according to their geographic distribution. Location taints enable the placement of services in specific zones or certain nodes. All these rules are important to fine-tune the scheduling behavior of Kubernetes, in particular, to help the scheduler make more informed decisions at the filtering step by removing inappropriate nodes. For instance, for a delay-sensitive service, a nearby node is more suited than a remote one for this time-critical scenario. Thus, Round Trip Time (RTT) values are assigned to each node as a label so that delay constraints can be considered in the scheduling process. The RTT labels of each node are listed in Table I.

A. Random Scheduler (RS)

The Random Scheduler (RS) has been implemented by extending KS through the filter endpoint. The random selection of the node is, in fact, a random pick after the KS filters out the inappropriate nodes.

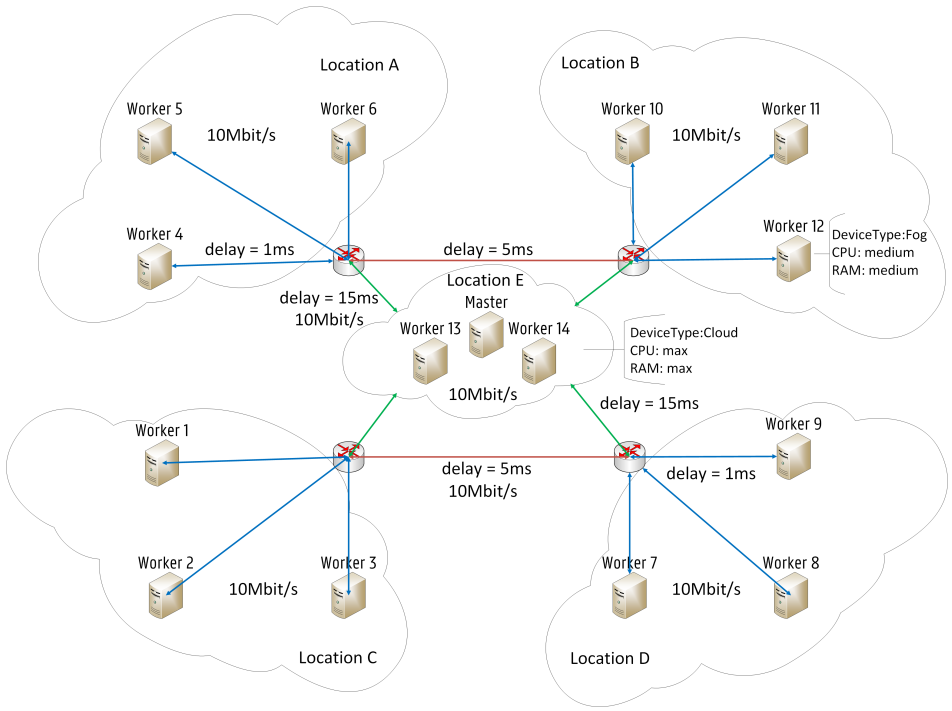

Fig. 3: A Fog Computing infrastructure based on the Kubernetes architecture

TABLE II: Software Versions of the Evaluation Setup.

Software	Version
Kubeadm	v1.13.0
Kubectl	v1.13.0
Go	go1.11.2
Docker	docker://17.3.2
Linux Kernel	4.4.0-34-generic
Operating System	Ubuntu 16.04.1 LTS

B. Network-Aware Scheduler (NAS)

The Network-Aware Scheduler (NAS) has been implemented by extending KS through the filter endpoint. The NAS Algorithm is shown in Alg. 1. The proposed NAS makes use of the strategically placed RTT labels to decide where it is suitable to deploy a specific service based on the target location specified on the pod configuration file. In fact, the node selection is based on the minimization of the RTT depending on the target location for the service after the completion of the filtering step. Additionally, in terms of bandwidth, NAS checks if the best candidate node has enough bandwidth to support the given service based on the bandwidth requirement label. If the bandwidth request is not specified in the pod configuration file, a default value of 250 Kbit/s is considered during the scheduling phase. After completion of the scheduling request, the available bandwidth is updated on the correspondent node label.

V. Evaluation Setup

The illustrated infrastructure has been implemented with Kubeadm [20]. The Kubernetes cluster has been set up on the imec Virtual Wall infrastructure [21] at IDLab, Belgium. The software versions used to implement the Kubernetes cluster are listed in Table II. All proposed schedulers have been implemented in Go and deployed in the Kubernetes cluster as a pod. In Fig. 4, the pod configuration file for the NAS is shown. As can be seen, the pod is composed by two containers: the extender and the NAS. The extender is responsible for performing the proposed scheduling operation while the NAS is in fact the actual KS. In Fig. 5, the scheduling policy configuration file for the NAS is shown.

The evaluation has been carried out on Smart City services performing unsupervised anomaly detection. This scenario has been previously presented in [22]. The purpose of this use case is to collect air quality data in the City of Antwerp to detect high amounts of organic compounds in the atmosphere based on outlier detection and clustering algorithms. The anomaly detection algorithms have been implemented as container APIs and then deployed as pods in the Kubernetes cluster. The deployment properties of each service are shown in Table III. In Fig. 6, the pod configuration file for the deployment of the birch-api service is shown. The desired location for the allocation of the service is expressed by the *targetLocation* label. The minimum needed bandwidth for the provisioning of the service is expressed by the *bandwidthReq* label. Moreover, the available bandwidth per node is 10 Mbit/s. Additionally, a pod anti-affinity rule has been added to each service so that pods belonging to the same service are not deployed together, meaning that a node can only allocate one instance of a certain pod. Additionally, the service chain of the Birch and the Kmeans service is composed by two pods, the API

Algorithm 1 NAS Algorithm

Input: Remaining Nodes after Filtering Process in
Output: Node for the service placement out

```
1: //Handle a provisioning request
2: handler(http.Request){
3:     receivedNodes = decode(http.Request);
4:     receivedPod = decodePod(http.Request);
5:     node = selectNode(receivedNodes, receivedPod);
6:     return node
7: }
8: //Return the best candidate Node (recursive)
9: selectNode(receivedNodes, receivedPod){
10:    targetLocation = getLocation(receivedPod);
11:    minBandwidth = getBandwidth(receivedPod);
12:    min = math.MaxFloat64;
13:    copy = receivedNodes;
14:    // find min RTT
15:    for node in range receivedNodes{
16:        rtt = getRTT(node, targetLocation);
17:        min = math.Min(min, rtt);
18:    }
19:    // find best Node based on RTT and minBandwidth
20:    for node in range receivedNodes{
21:        if min == getRTT(node, targetLocation){
22:            if minBandwidth ≤ getAvBandwidth(node){
23:                return node;
24:            }
25:            else
26:                copy = removeNode(copy, node);
27:        }
28:    }
29:    if copy == null
30:        return null, Error("No suitable nodes found!");
31:    else
32:        return selectNode(copy, receivedPod);
33: }
```

and the correspondent database. The deployment of these services has been performed to compare the performance of our implemented schedulers with the default KS.

VI. Evaluation Results

In Table IV, the execution time of the different schedulers is shown. As can be seen, the extender of both implemented schedulers decides on average between 4 and 6 ms. The default KS does not issue an extender call and, thus, the scheduling decision is made on average on 2.14 ms while the RS and the NAS require between 6 and 8 ms because of the extender procedure. Additionally, the binding operation execution time is similar among all the schedulers since the binding is actually performed by the KS on all the schedulers. The RS requires on average 3 seconds to allocate and initialize the required containers for the services while the KS and the NAS only need on average 2 seconds. In Fig. 7, the different allocation scheme for each of the schedulers is illustrated. As expected,

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
 labels:
  component: scheduler
  tier: control-plane
 name: network-aware-scheduler
 namespace: kube-system
spec:
 selector:
  matchLabels:
   component: scheduler
   tier: control-plane
 replicas: 1
 template:
  metadata:
   labels:
    component: scheduler
    tier: control-plane
    version: second
  spec:
   tolerations:
   - key: "function"
    operator: "Equal"
    value: "master"
    effect: "NoSchedule"
   serviceAccountName: network-aware-scheduler
   containers:
   - name: extender
    image: jpedro1992/network-aware-scheduler:0.0.3
    ports:
    - containerPort: 8100
   - name: network-aware-scheduler
    image: mirrorgooglecontainers/kube-scheduler:v1.12.3-beta.0
    command:
    - /usr/local/bin/kube-scheduler
    - --address=0.0.0.0
    - --leader-elect=false
    - --scheduler-name=network-aware-scheduler
    - --policy-configmap=network-aware-scheduler-config
    - --policy-configmap-namespace=kube-system
    livenessProbe:
     httpGet:
      path: /healthz
      port: 10251
     initialDelaySeconds: 15
    readinessProbe:
     httpGet:
      path: /healthz
      port: 10251
    resources:
     requests:
      cpu: '0.1'
    securityContext:
     privileged: false
    volumeMounts: []
   hostNetwork: false
   hostPID: false
```

Fig. 4: The pod configuration file for the NAS.

```
kind: ConfigMap
apiVersion: v1
metadata:
 name: network-aware-scheduler-config
 namespace: kube-system
data:
 policy.cfg: |
  {
    "kind" : "Policy",
    "apiVersion" : "v1",
    "metadata" : {
      "name": "network-aware-scheduler-config",
      "namespace": "kube-system"
    },
    "predicates" : [
      {"name" : "PodFitsResources"},
      {"name" : "PodFitsHostPorts"},
      {"name" : "NoDiskConflict"},
      {"name" : "NoVolumeZoneConflict"},
      {"name" : "PodToleratesNodeTaints"},
      {"name" : "MatchInterPodAffinity"}
    ],
    "extenders": [
      {
        "urlPrefix": "http://127.0.0.1:8100",
        "apiVersion": "v1",
        "filterVerb": "filter",
        "enableHttps": false
      }
    ]
  }
```

Fig. 5: The scheduling policy configuration file for the NAS.

the RS and the KS deployment schemes are not optimized for the desired location of the service since in their scheduling algorithm no considerations are made about network latency.

TABLE III: Deployment properties of each service.

Service Name	Pod Name	CPU Req/Lim (m)	RAM Req/Lim (Mi)	Min. Bandwidth (Mbit/s)	Replication Factor	Target Location	Dependencies
Birch	birch-api birch-cassandra	100/500 500/1000	128/256 1024/2048	2.5 5	4 3	A	birch-cassandra birch-api
Robust	robust-api	200/500	256/512	2	4	B	none
Kmeans	kmeans-api kmeans-cassandra	100/500 500/1000	128/256 1024/2048	2.5 5	5 3	C	kmeans-cassandra kmeans-api
Isolation	isolation-api	200/500	256/512	1	2	D	none

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: birch-api
spec:
  selector:
    matchLabels:
      app: birch-api
  replicas: 4
  template:
    metadata:
      labels:
        app: birch-api
        targetLocation: RTT-A
        minBandwidth: 2.5Mi
    spec:
      schedulerName: network-aware-scheduler
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - birch-api
            topologyKey: "kubernetes.io/hostname"
      containers:
      - image: jpedro1992/birch:2.0
        name: birch-api
        resources:
          requests:
            memory: "128Mi"
            cpu: '0.1'
          limits:
            memory: "256Mi"
            cpu: '0.5'
        ports:
        - containerPort: 5000
          name: http
          protocol: TCP
```

Fig. 6: The pod configuration file for the birch-api service.

TABLE IV: The execution time of the different schedulers.

Scheduler	Extender decision	Scheduling decision	Binding operation	Pod Startup Time
KS	-	2.14 ms	162.7ms	2.02 s
RS	5.32 ms	7.71 ms	178.2ms	3.04 s
NAS	4.82 ms	6.44 ms	173.1ms	2.10 s

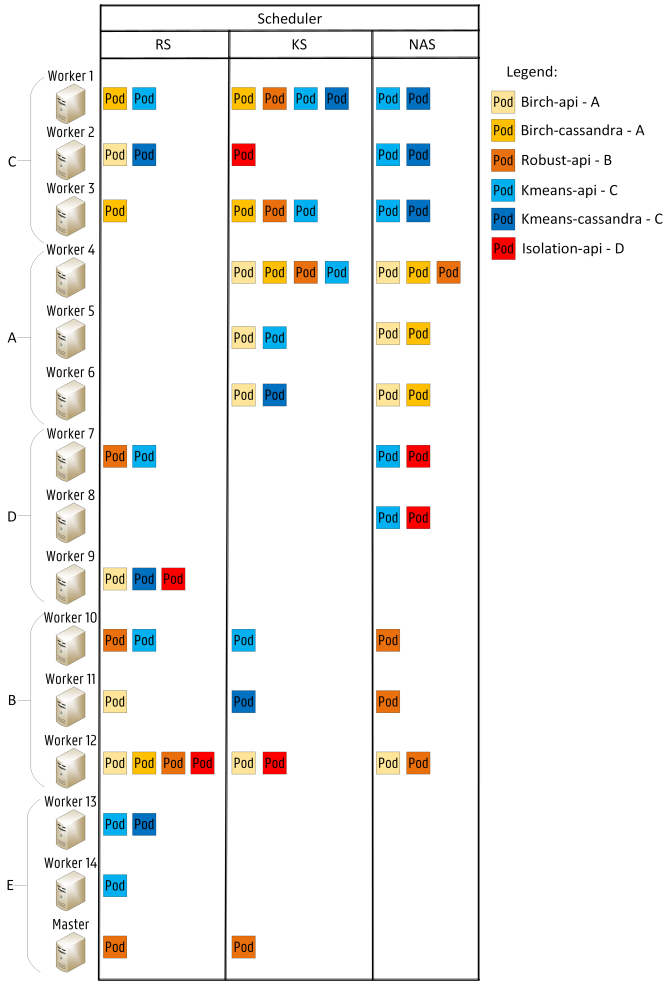

Fig. 7: The service provisioning schemes of the different schedulers.

For instance, the KS allocation scheme of the isolation-api service is fairly poor since both replicas are deployed in location B and C, respectively, while D is the desired location. The differences in the average RTT per scheduler are detailed in Fig. 8. As shown, the proposed NAS achieves significantly lower RTTs for each of the deployed services. For instance, the RTT is on average 4 ms when the NAS provisions the isolation-api service while for the RS and the KS, the RTT is on average 34 ms and 39 ms, respectively. By just increasing the KS execution time by 6 ms by issuing an extender call, the service provisioning in terms of network latency is highly improved. Furthermore, both KS and RS allocate pods on nodes already compromised in terms of network bandwidth since bandwidth requests are not considered in the scheduling process. KS overloads worker 1 and 4 by allocating on them 4 pods leading to service bandwidths of 14.5 Mbit/s and 12.5 Mbit/s for the worker 1 and 4, respectively, which surpasses the available bandwidth of 10 Mbit/s. This allocation scheme may lead to service disruptions due to bandwidth fluctuations. Results show that, for the overall service deployment, our NAS

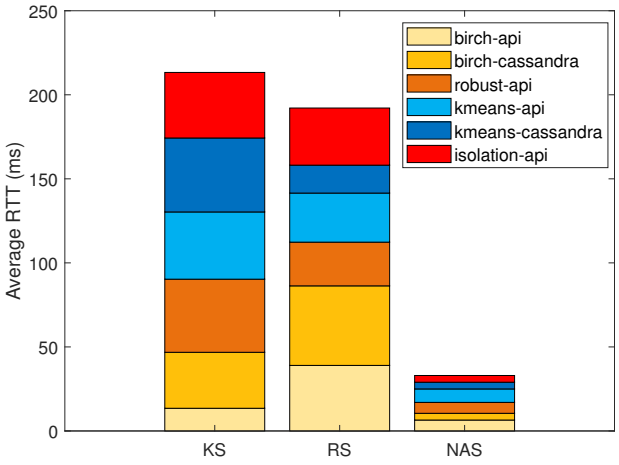

Fig. 8: Comparison of average RTT per scheduler for different pod-deployment scheduling strategies in a Smart City air quality monitoring scenario.

can improve the performance of the default KS by reducing the network latency by 80%. In summary, by combining the powerful Kubernetes scheduling features with Fog Computing concepts and their envisioned IoT demands, the proposed approach paves the way towards proper resource provisioning in Smart City ecosystems.

VII. Conclusions

In recent years, the combination of Smart Cities and Fog Computing has encouraged the development of effective orchestration mechanisms to guarantee the smooth performance of IoT services. In this paper, a network-aware scheduling approach for Smart City container-based applications is proposed, which provides informed resource allocation decisions based on the current status of the network infrastructure. Two popular open-source projects, Docker and Kubernetes, have been used to validate our approach. The modular and scalable design of the Kubernetes architecture provides a flexible abstraction between the micro-services and the underlying infrastructure. The proposed scheduling mechanism has been implemented as an extension to the default scheduling feature available in Kubernetes. Evaluations have been performed to validate the proposed scheduling algorithm. The performance of our approach has been compared with the standard scheduler in Kubernetes. Results show that the proposed NAS can significantly improve the service provisioning of the default KS by achieving a reduction of 80% in terms of network latency. As future work, additional scheduling strategies will be added to our scheduler to further refine the resource provisioning scheme in terms of mobility and resilience.

Acknowledgment

This research was performed partially within the projects "Service-oriented management of a virtualised future internet" and "Intelligent DEnse And Longe range IoT networks (IDEAL-IoT)" under Grant Agreement #S004017N, both from the fund for Scientific Research-Flanders (FWO-V).

References

[1] (2017) Cisco visual networking index: Global mobile data traffic forecast update, 2016–2021 white paper. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.pdf

[2] S. Sarkar, S. Chatterjee, and S. Misra, "Assessment of the suitability of fog computing in the context of internet of things," *IEEE Transactions on Cloud Computing*, vol. 6, no. 1, pp. 46–59, 2018.

[3] M. Chiang and T. Zhang, "Fog and iot: An overview of research opportunities," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, 2016.

[4] C. Perera, Y. Qin, J. C. Estrella, S. Reiff-Marganiec, and A. V. Vasilakos, "Fog computing for sustainable smart cities: A survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, p. 32, 2017.

[5] K. Velasquez, D. P. Abreu, D. Gonçalves, L. Bittencourt, M. Curado, E. Monteiro, and E. Madeira, "Service orchestration in fog environments," in *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE, 2017, pp. 329–336.

[6] S. Agarwal, S. Yadav, and A. K. Yadav, "An efficient architecture and algorithm for resource provisioning in fog computing," *International Journal of Information Engineering and Electronic Business*, vol. 8, no. 1, p. 48, 2016.

[7] N. Dmitry and S.-S. Manfred, "On micro-services architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, 2014.

[8] (2018) Kubernetes, automated container deployment, scaling, and management. [Online]. Available: https://kubernetes.io/

[9] K. Hightower, B. Burns, and J. Beda, *Kubernetes: Up and Running: Dive Into the Future of Infrastructure*. " O'Reilly Media, Inc.", 2017.

[10] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, "Multi-objective scheduling of micro-services for optimal service function chains," in *Communications (ICC), 2017 IEEE International Conference on*. IEEE, 2017, pp. 1–6.

[11] K. Ettikyala and Y. V. Latha, "Rank based efficient task scheduler for cloud computing," in *Data Mining and Advanced Computing (SAPIENCE), International Conference on*. IEEE, 2016, pp. 343–346.

[12] F. Bahrpeyma, H. Haghighi, and A. Zakerolhosseini, "An adaptive rl based approach for dynamic resource provisioning in cloud virtualized data centers," *Computing*, vol. 97, no. 12, pp. 1209–1234, 2015.

[13] M. Aazam, M. St-Hilaire, C.-H. Lung, and I. Lambadaris, "Mefore: Qoe based resource estimation at fog to enhance qos in iot," in *Telecommunications (ICT), 2016 23rd International Conference on*. IEEE, 2016, pp. 1–5.

[14] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwälder, "Incremental deployment and migration of geo-distributed situation awareness applications in the fog," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM, 2016, pp. 258–269.

[15] S. Hoque, M. S. de Brito, A. Willner, O. Keil, and T. Magedanz, "Towards container orchestration in fog computing infrastructures," in *Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual*, vol. 2. IEEE, 2017, pp. 294–299.

[16] A. Brogi and S. Forti, "Qos-aware deployment of iot applications through the fog," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1185–1192, 2017.

[17] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Resource provisioning for iot application services in smart cities," in *Network and Service Management (CNSM), 2017 13th International Conference on*. IEEE, 2017, pp. 1–9.

[18] (2018) Docker containerization unlocks the potential for dev and ops. [Online]. Available: https://www.docker.com/

[19] V. Medel, R. Tolosana-Calasanz, J. Á. Bañares, U. Arronategui, and O. F. Rana, "Characterising resource management performance in kubernetes," *Computers & Electrical Engineering*, vol. 68, pp. 286–297, 2018.

[20] (2018) Overview of kubeadm. [Online]. Available: https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm/

[21] (2018) The virtual wall emulation environment. [Online]. Available: https://doc.ilabt.imec.be/ilabt-documentation/index.html

[22] J. Santos, P. Leroux, T. Wauters, B. Volckaert, and F. De Turck, "Anomaly detection for smart city applications over 5g low power wide area networks," in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2018, pp. 1–9.