Towards delay-aware container-based Service Function Chaining in Fog Computing

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*— Recently, the fifth-generation mobile network (5G) is getting significant attention. Empowered by Network Function Virtualization (NFV), 5G networks aim to support diverse services coming from different business verticals (e.g. Smart Cities, Automotive, etc). To fully leverage on NFV, services must be connected in a specific order forming a Service Function Chain (SFC). SFCs allow mobile operators to benefit from the high flexibility and low operational costs introduced by network softwarization. Additionally, Cloud computing is evolving towards a distributed paradigm called Fog Computing, which aims to provide a distributed cloud infrastructure by placing computational resources close to end-users. However, most SFC research only focuses on Multi-access Edge Computing (MEC) use cases where mobile operators aim to deploy services close to end-users. Bi-directional communication between Edges and Cloud are not considered in MEC, which in contrast is highly important in a Fog environment as in distributed anomaly detection services. Therefore, in this paper, we propose an SFC controller to optimize the placement of service chains in Fog environments, specifically tailored for Smart City use cases. Our approach has been validated on the Kubernetes platform, an open-source orchestrator for the automatic deployment of micro-services. Our SFC controller has been implemented as an extension to the scheduling features available in Kubernetes, enabling the efficient provisioning of container-based SFCs while optimizing resource allocation and reducing the end-to-end (E2E) latency. Results show that the proposed approach can lower the network latency up to 18% for the studied use case while conserving bandwidth when compared to the default scheduling mechanism.**

***Index Terms*—Resource Provisioning, Service Function Chain, Fog Computing, IoT, Kubernetes**

I. Introduction

In recent years, the fifth-generation mobile network (5G) rapidly started gaining popularity due to the wide adoption of network virtualization and Cloud technologies. Augmented reality, tactile Internet, autonomous vehicles are among the envisioned 5G use cases. These services will have stringent quality of service (QoS) requirements in terms of network bandwidth, mobility coverage, end-to-end (E2E) latency, among others [1]. To fully leverage from this kind of services, researchers have introduced Network Function Virtualization (NFV) [2], [3] since traditional hardware is unable to meet the high demanding requirements introduced by these use cases. NFV decouples network functions from the physical devices on which they run to be executed by software running on Virtual Machines (VMs), thus achieving the purpose of reducing Operational Expenditures (OPEX) and Capital Expenditures (CAPEX) while easing the deployment of new services [4]. Nevertheless, several challenges still remain to fully benefit from NFV. One important challenge is called Service Function Chain (SFC) [5], [6]. SFC encompasses an emerging set of technologies aiming to enable mobile operators and cloud providers to dynamically reconfigure softwarized Network Services (NSs) without having to implement changes at the hardware level. Thus, providing a flexible and cost-effective alternative to today's static network environment. Services must be connected in a specific order forming an SFC that each user has to traverse to achieve a particular NS as shown in Fig. 1. All circles represent different service functions while the arrows show how traffic is steered in the network. Users are then routed through the SFC according to the service graph, which results in optimized resource provisioning and reduced operational costs.

Recently, Cloud computing is also evolving towards a distributed paradigm called Fog Computing [7] due to the massive impact of the Internet of Things (IoT). Smart Cities [8] powered by IoT are transforming different domains of urban life, such as public transportation, environmental monitoring, and health-care to improve citizen welfare. Fog Computing aims to provide computing resources at the edges of the network, thus helping to meet the demanding constraints introduced by IoT (e.g. low latency, high mobility). Service providers may benefit from Fog Computing by deploying their applications across geographically distributed clouds, so that real-time processing, storage procedures and data analytics can be brought closer to end users, overcoming the limitations of traditional centralized cloud infrastructures [9]. Autonomous vehicles and environmental monitoring are among the envisioned Fog use cases benefiting from these architectures. Furthermore, container-based micro-services are currently revolutionizing the way developers build their software applications [10]. An application is decomposed in a set of small, self-contained containers deployed across a large number of servers instead of the traditional code-heavy monolithic application. Nowadays, containers are the *de facto* alternative to the conventional VMs, due to their low overhead and high portability.

In this paper, an SFC controller is presented to optimize the placement of container-based service chains in Fog Computing environments, since most SFC studies only focus on Multi-access Edge Computing (MEC) use cases. The main differ-

978-1-7281-4973-8/20/$31.00 © 2020 IEEE

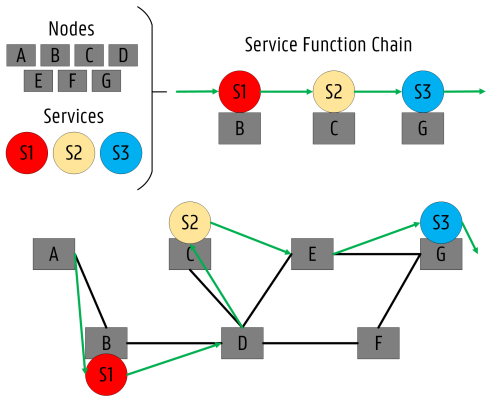

Fig. 1: An example of a Service Function Chain deployment.

ence between MEC and Fog Computing is in the considered interactions (i.e. between Edges and Cloud). MEC focuses on deploying services close to end-users to reduce latency and avoid congestion in the network core. In Fog Computing, bi-directional communications between Edges and Cloud are crucial due to the hierarchical architecture. For instance, a service is allocated in the cloud due to high computational requirements but needs to interact with another service, which may be located in the Fog. These interactions (e.g. necessary bandwidth) must be guaranteed. MEC is currently not taking these bi-directional communications into account. The proposed SFC controller has been implemented as an extension to the default scheduling feature available in Kubernetes [11], an open-source container management platform originally developed by Google, which simplifies the deployment of scalable distributed systems by managing the complete orchestration life-cycle of containerized applications. Finally, evaluations have been performed to validate our approach, specifically for container-based Smart City use cases. The proposed SFC controller enables Kubernetes to efficiently allocate container-based SFCs while maintaining bandwidth conservation and reducing the E2E latency.

The remainder of the paper is organized as follows. In the next Section, related work is discussed. Section III highlights the importance of SFC in Fog-Cloud environments and introduces the proposed SFC controller and its provisioning algorithm. In Section IV, the evaluation setup is described which is followed by the evaluation results in Section V. Finally, conclusions are presented in Section VI.

II. Related Work

In recent years, SFC allocation and resource provisioning issues gained significant attention in the fields of NFV, MEC and Cloud computing. In [12], the challenge of allocating Virtual Network Functions (VNFs) has been addressed. Their approach focused on finding the required number of VNF instances and their optimal placement while minimizing operational costs and maximizing network utilization, without violating service level agreements. The authors presented an Integer Linear Programming (ILP) formulation for finding the optimal solutions for small scale networks, while a heuristic-based algorithm has been proposed for larger-scale networks instead. In [13], the SFC allocation problem has been studied. The authors focused on determining the optimal VNF placement while minimizing the E2E latency and maximizing resource efficiency. In [14], the SFC orchestration problem in 5G virtualized infrastructures has been addressed. The authors modeled the problem as a robust binary optimization. Furthermore, in [15], a mixed ILP model has been presented as a solution to ease the SFC orchestration in a Cloud environment. The model finds the optimal VNF placement while deciding whether to re-instantiate or migrate the VNFs and minimizing the SFC delays. In [16], an ILP formulation has been presented to solve the SFC allocation problem, while considering E2E latency and data rate requirements. The authors also proposed a heuristic algorithm to address the scalability issue of the ILP-based solution. Our work goes beyond the state-of-the-art since the SFC orchestration problem has not been addressed for Fog Computing environments, where considering the interactions between Fog and Cloud is highly important for a proper SFC allocation.

Recently, a handful of research efforts has been performed in the context of resource provisioning in Fog Computing environments that combine aspects coming from Cloud computing, network virtualization and sensor networks. In [17], a resource management approach based on demand predictions has been presented. Their work focused on allocating resources based on users' demand fluctuations by using cost functions, different types of services and pricing models for new and existing customers. Simulation results showed that the suggested model achieves a fair performance by preallocating resources based on user behavior and future usage predictions. In [18], the IoT resource provisioning issue has been modeled as an ILP formulation. Application QoS metrics and deadlines for the provisioning of each type of application have been considered in their approach. Additionally, in [19], the Fog resource provisioning problem has been addressed. The authors studied the trade-off between maximizing the reliability and minimizing the overall system cost. A highly computationally complex ILP model has been presented. Then, the authors presented simulation results coming from a heuristic-based algorithm able to find suboptimal solutions, albeit achieving better time efficiency. Nevertheless, none of the aforementioned studies considered realistic latency-sensitive services with actual E2E latency demands envisioned to be supported by future 5G networks or considered the strict requirements coming from SFC or container-based applications. Furthermore, most research has only been focused on theoretical modeling and simulation studies, which limit their applicability to real deployments.

Previously, in [20], we have tackled the problem of resource provisioning in Fog Computing. The present work builds further on our previous one since the SFC placement issue has now been addressed. SFC capabilities for Fog-Cloud environments are still quite unexplored. To the best of our knowledge, our approach goes beyond the current state-of-the-art by extending a well-known platform called Kubernetes with

SFC controlling mechanisms enabling the efficient allocation of container-based SFCs, specifically tailored for Smart City use cases. Furthermore, a practical implementation of the proposed SFC controller has been evaluated to show the full applicability of our approach. By combining Fog Computing alongside SFC concepts, our work paves the way towards an efficient resource provisioning of SFCs in softwarized Fog Computing infrastructures.

III. Towards SFC in Fog Computing

This section introduces the proposed SFC controller mechanism followed by the discussion of its provisioning algorithm.

A. The SFC Controller

The SFC controller has been implemented as an extension to the Kubernetes platform, based on previous work presented in [21]. Although Kubernetes makes use of containers as the underlying mechanism to deploy micro-services, additional layers of abstraction exist over the container runtime environment to provide scalable life-cycle orchestration features. In Kubernetes, micro-services are often tightly coupled together forming a group of containers. This is the smallest working unit in Kubernetes, which is called a pod [22]. A pod represents the collection of containers and storage (volumes) running in the same execution environment. Additionally, Kubernetes provides a feature called *Service*, which is an abstract way to define a logical set of Pods and expose applications running on them as an NS [23]. By using this abstraction, there is no need to use a service discovery mechanism since pods have their own IP address and a single Domain Name System (DNS) name is assigned to a set of pods, which makes load-balancing a straightforward process across them. The rationale behind this abstraction process comes from the pods' volatility as they may be terminated, meaning that pods running at a certain moment may be different than the ones which are providing the service a few days later. This could lead to service disruptions. For instance, imagine two services, a frontend and a backend service. If pods are constantly being terminated and rescheduled, how could the frontend service keep track of which IP address it needs to connect to the backend service? Thus, the actual pods that compose the backend service may change, but users should not need to be aware of that, nor should they need to keep track of them. The SFC controller logic solves the issue of routing between different services in the SFC. An example of a container-based SFC in Kubernetes is shown in Fig. 2. Nevertheless, Kubernetes does not provide scheduling features to properly allocate SFCs. Kubernetes allocates pods based only on available resources (e.g. CPU and RAM usage rates), without making any consideration about the complete E2E service or even any concern about latency or bandwidth limitations. Furthermore, Kubernetes provisions pods, one by one, without taking into account previous pod allocations. In fact, the component that assigns pods to specific nodes in Kubernetes is called Kube–Scheduler (KS). The KS is the default scheduling feature in the Kubernetes platform, which is responsible for deciding on which adequate nodes

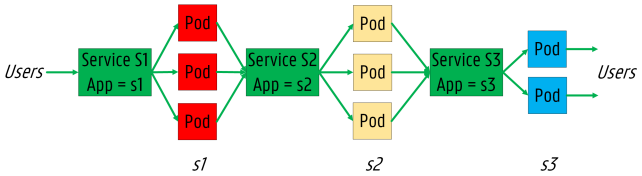

Fig. 2: An example of a container-based Service Function Chain deployment in Kubernetes.

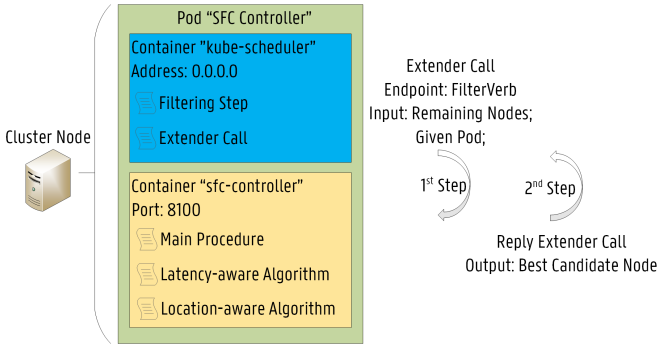

Fig. 3: The detailed Pod architecture of the SFC controller.

pods should be allocated. The SFC controller logic has been implemented as a "scheduler extender" process that the KS calls out as a final step when a scheduling decision is needed, which makes use of previous pod provisioning information to optimize the SFC allocation. The presented approach has been implemented in Go and deployed in the Kubernetes cluster as a pod. The pod architecture of the SFC controller is illustrated in Fig. 3. Essentially, every pod requiring 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 an adequate node for the placement. Firstly, KS executes the node filtering operation, where KS verifies which nodes are capable of running the pod by applying a set of filters. The purpose of filtering is to solely consider nodes meeting all specific pod requirements further in the scheduling process. Thus, inadequate nodes are already removed from the list of possible candidates by applying these filters. Then, KS calls out the SFC controller to make the final decision on which cluster node the service must be provisioned based on the remaining set of nodes. Each scheduling request is handled by the SFC controller, where a suitable node is selected based on two provisioning strategies: *Latency-aware* and *Location-aware*. Both algorithms are detailed next.

B. Provisioning Algorithm of the SFC controller

The main procedure of the SFC controller is stated in Alg. 1. First, the SFC controller gathers allocation information through pod labels defined on the pod configuration file. These pod labels are listed in Table I. Second, the provisioning algorithm is selected based on the *Policy* label. Two policies are currently supported: *Latency-aware* and *Location-aware*. Both algorithms are shown in Alg. 2 and Alg. 3, respectively.

TABLE I: Extending pod labels with SFC information.

Label	Description
Network Service Header	The specific SFC identifier (String).
Chain Position	The position of the given pod in the SFC.
Total Services	The total number of services in the SFC.
Target Location	The preferred location for the deployment.
Policy	The preferred allocation policy.
Min Bandwidth	The minimum expected bandwidth.
Prev Service	The previous service in the SFC.
Next Service	The next service in the SFC.

Algorithm 1 Main procedure of the SFC controller

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

```
// Return the best candidate Node
selectNode(nodes, pod):
  policy = getPolicy(pod)
  minB = getBandwidth(pod)
  nsh = getServideHeader(pod)
  if policy == Latency
    node = getLatencyNode(nodes, pod, minB, nsh)
    if node ≠ null then
      // Store pod info in Hash Table
      addPod(getKey(pod), node)
      // update available Link bandwidth
      updateB(getNodeB(node) − minB, node)
      return node
  else if policy == Location then
    node = getLocationNode(nodes, pod)
    if node ≠ null then
      addPod(getKey(pod), node)
      updateB(getNodeB(node) − minB, node)
      return node
  // Otherwise → max residual bandwidth Link
  node = getLinkNode(nodes);
  if node ≠ null then
    addPod(getKey(pod), node)
    updateB(getNodeB(node) − minB, node)
    return nodeLink
  else
    return null, Error("No suitable node!!")
```

On one hand, if *Latency-aware* is preferred, the SFC controller selects the best candidate node based on the calculation of Dijkstra's shortest path algorithm [24]. Provisioning records are kept of the previously allocated pods based on the *Network Service Header* label. If any of those corresponds to the same NS, the shortest paths will be calculated for each of the possible nodes. Otherwise, if not a single pod has been already allocated in the network corresponding to the same NS, the node selection is made as if the Location-aware policy was selected. Regarding bandwidth, each candidate node is checked to confirm that it has enough bandwidth to support the given pod based on the *Min Bandwidth* label. Thus, the node with the lowest combined shortest paths and enough bandwidth

Algorithm 2 Latency-aware algorithm of the SFC controller

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

```
// Return the best candidate Node based on Latency
getLatencyNode(nodes, pod, minB, nsh):
  // Find pods belonging to this nsh
  podList = getPodList(nsh, pod)
  if podList ≠ null then // Calculate Shortest Paths
    node = getNodeDelaySP(nodes, podList)
  else // Select Node as if Location-aware was selected
    node = getLocationNode(nodes, pod)
  return node
```

Algorithm 3 Location-aware algorithm of the SFC controller

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

```
// Return the best candidate Node based on Location
getLocationNode(nodes, pod):
  copy = nodes;
  loc = getLocation(pod)
  minD = getMinDelay(nodes, loc)
  // Node selected based on min delay & Bandwidth
  for node in range nodes do
    if minD == getDelay(node, loc) then
      if minB ≤ getNodeB(node)
        return node
      else
        copy = removeNode(copy, node)
  if copy == null then
    return null
  else // Repeat the Process (Recursive)
    return getLocationNode(copy, pod)
```

will be selected for pod deployment. On the other hand, if the *Location-aware* policy is chosen, the node selection is based on minimizing latency depending on the *Target Location* label, since certain pods may be preferred to be deployed on a certain Fog location or even in the Cloud, as they require a high amount of resources. Additionally, it is verified whether each candidate node has enough bandwidth to support the given service. After completion of each scheduling request, pod allocation information is stored as a provisioning record to be consulted in further scheduling requests and the node's available bandwidth is updated. Thus, the SFC controller knows exactly the available bandwidth between scheduling requests, which allows it to make informed decisions based on latency and bandwidth information. Finally, if no suitable node is found after policy execution, link costs are calculated for each possible node. The node with the maximum residual bandwidth link adequate to support the expected minimum bandwidth is selected to allocate the pod. Otherwise, it is not possible to allocate the service without compromising bandwidth. Thus, similar to the KS, an event is triggered due to the failed pod deployment (i.e. pod eviction).

TABLE II: Software Versions of the Evaluation Setup.

Software	Version
Kubeadm	v1.13.4
Kubectl	v1.13.4
Go	go1.11.5
Docker	docker://18.09.2
Linux Kernel	4.4.0-34-generic
Operating System	Ubuntu 16.04.1 LTS

In summary, the proposed SFC controller filters inappropriate nodes based on the KS filtering step and then makes use of the implemented pod labels to choose the best candidate node from the filtered ones to the desired scheduling policy. The SFC controller supports two policies, latency-aware and location-aware, upon which it can select nodes based on minimizing latency established by the calculation of the shortest paths or based on the target location for the pod deployment, respectively. Similar to the KS, the SFC controller optimizes the allocation of each pod, one by one. Thus, our implementation can find a sub-optimal solution when compared with ILP-based solutions, however, in smaller execution time. It should be noted that a dynamic SFC controller suitable for dealing with bandwidth fluctuations and delay changes is required. This, however, is out of the scope of this paper.

IV. Evaluation Setup

In this section, the testbed infrastructure used for the Kubernetes setup is described. Then, the two use cases for the evaluation are introduced. First, the Waste Management scenario is presented, which is followed by the Surveillance Camera use case.

A. Testbed Infrastructure

The Kubernetes cluster has been set up on the imec Virtual Wall infrastructure [25] at IDLab, Belgium. The Fog-Cloud infrastructure illustrated in Fig. 4 has been implemented with Kubeadm [26]. The software versions of all the components used to set up the Kubernetes cluster are listed in Table II.

B. Waste Management Use Case

Waste Management is viewed as one of the key services enabled by IoT technology in future Smart Cities [27]. Waste bins are located everywhere (e.g. restaurants, office buildings, retail stores), but picking up garbage has been traditionally an inefficient service for years. Garbage trucks follow a given route without knowing if bins are empty or full. Another issue is that waste bins may get overloaded before the planned cleaning. This results in high maintenance and fuel costs. IoT can tackle this issue by collecting waste bin data. For instance, sensors can be installed into waste bins to tell which bins are full. Furthermore, by sending the collected data to a fog-Cloud infrastructure, route planning services can be executed to find the optimal route for each truck based on bin fill levels. Thus, drivers do not waste time driving to empty bins and broken bins may be repaired quickly. Trucks and drivers can access this service through a dashboard available as a mobile application, enabling them to improve their customer service. Therefore, an IoT-based waste management service provides a more efficient waste collection through route optimization and higher driver productivity. The objective of this use case is to enable the real-time access to waste bin information. In Fig. 5, the container-based SFC for the waste management use case is illustrated and the correspondent deployment requirements are shown in Table III.

C. Surveillance Camera Use Case

Over the last few years, crowd surveillance has become increasingly important due to the possibility of identifying individuals or even objects in highly crowded areas. Nevertheless, several issues still need to be addressed, including data transfer over limited bandwidth and high latency in sensor-Cloud communication. For instance, imagine a surveillance camera requiring a continuous streaming bandwidth of 15 Mb/s. Sending the entire data from the video camera to the Cloud translates into approximately 4.86 TB/monthly for a single camera. Therefore, it is essential to adopt Fog infrastructures to perform data analysis operations locally, thus reducing the amount of data transferred to the Cloud. Surveillance cameras placed on particular streets or crowded areas send continuous video streams to a Fog-Cloud infrastructure where face recognition algorithms are performed in a distributed manner. Fog nodes located close to the surveillance cameras receive their video streams and perform a first-level analysis, such as face detection and feature extraction tasks. Then, Fog nodes send the results to the Cloud for global analysis operations, such as face matching and recognition operations. Afterwards, global outcomes can be presented in a central dashboard in a control room. Additionally, police officers may access the detection results through a mobile application. This distributed approach has been previously presented in [28], as a proper manner to enable anomaly detection in Fog Computing architectures for delay-sensitive IoT services. An IoT-based surveillance camera service provides a more efficient way of recognizing individuals in crowded areas by distributing tasks between Fog and Cloud. The objective of this use case is to provide a near real-time face detection system. In Fig. 6, the container-based SFC for the surveillance camera use case is illustrated and the correspondent deployment requirements are shown in Table IV.

D. Use Case Deployment in Kubernetes

The deployment of both use cases has been performed to compare the performance of our SFC controller with the default KS. All services have been deployed based on a pod configuration file. For example, the pod configuration file for the api service is shown in Fig. 7. It should be noted that a pod anti-affinity rule has been added to each service so that pods belonging to the same service cannot be deployed together, meaning that a node can only allocate one instance of a particular pod for a certain service.

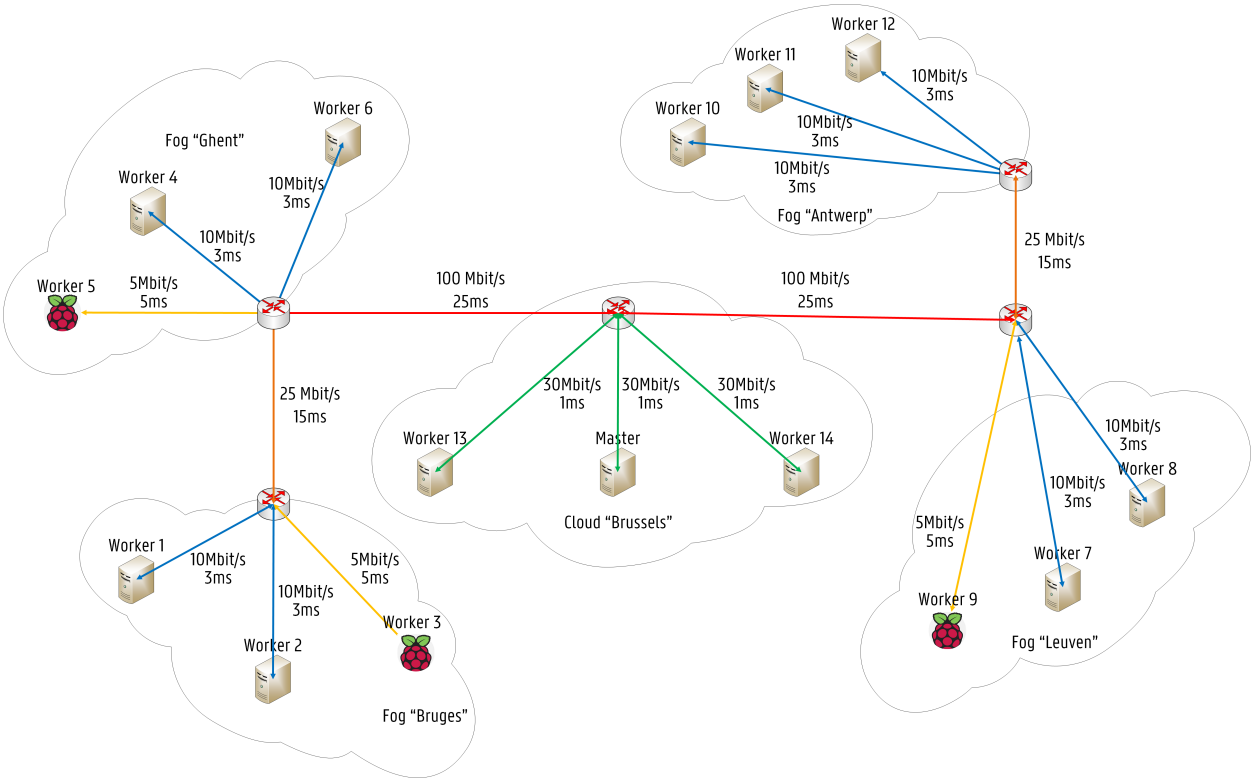

Fig. 4: A Fog-Cloud Infrastructure based on the Kubernetes architecture.

TABLE III: Deployment properties of the Waste Management Use Case.

Network Service Header	Pod Name	Chain Position	Total Services	Policy	Target Location	Min. Bandwidth (Mbit/s)	CPU Req/Lim (m)	RAM Req/Lim (Mi)	Replication Factor
Waste	api	1	4	Latency-aware	Any	4.0	250/500	256/512	3
	waste-db	2		Latency-aware	Any	5.0	500/1000	1024/2048	4
	route-planner	3		Location-aware	Brussels	8.0	500/1000	1024/2048	4
	server	4		Latency-aware	Any	4.0	250/500	256/512	3

TABLE IV: Deployment properties of the Surveillance Camera Use Case.

Network Service Header	Pod Name	Chain Position	Total Services	Policy	Target Location	Min. Bandwidth (Mbit/s)	CPU Req/Lim (m)	RAM Req/Lim (Mi)	Replication Factor
Camera	fd-ext	1	4	Latency-aware	Any	8.0	500/1000	512/1024	4
	fm-recog	2		Location-aware	Brussels	8.0	1000/2000	2048/4096	2
	cam-db	3		Latency-aware	Any	2.5	500/1000	1024/2048	2
	dashboard	4		Latency-aware	Any	5.0	250/500	256/512	4

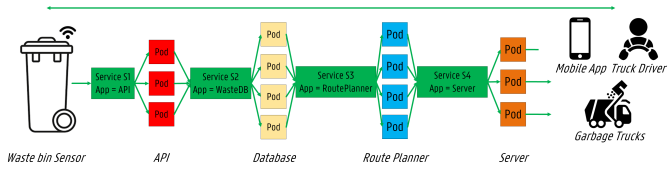

Fig. 5: The container-based Service Function chain envisioned for the Waste Management Use Case.

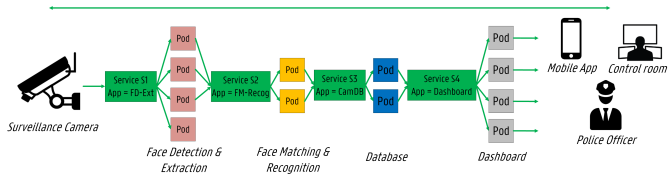

Fig. 6: The container-based Service Function chain envisioned for the Surveillance Camera Use Case.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
spec:
  selector:
    matchLabels:
      app: api
  replicas: 3
  template:
    metadata:
      labels:
        networkServiceHeader: waste
        chainPosition: 1pos
        totalChainServ: 4serv
        app: api
        targetLocation: Any
        minBandwidth: 4.0Mi
        policy: Latency
        nextService: waste-db
    spec:
      schedulerName: sfc-controller
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - api
            topologyKey: "kubernetes.io/hostname"
      containers:
      - image: jpedro1992/api:1.0
        name: api
        resources:
          requests:
            memory: "256Mi"
            cpu: '0.25'
          limits:
            memory: "512Mi"
            cpu: '0.5'
        ports:
        - containerPort: 5000
          name: http
          protocol: TCP
```

Fig. 7: The pod configuration file for the api Service.

TABLE V: The execution time of the different approaches.

Scheduler	Extender decision	Scheduling decision	Pod Startup Time
KS	-	3.21 ms	2.83 s
SFC controller	6.08 ms	8.38 ms	2.96 s

V. Evaluation Results

In this section, the evaluation results are detailed. First, the execution time of the different approaches is presented, followed by the average latency expected for each use case. Finally, the service bandwidth per node for the different scheduling approaches is shown.

A. Execution Time

In Table V, the execution time of the different schedulers is shown. The execution time has been averaged over 15 consecutive runs. The KS does not issue an extender call and, thus, the scheduling decision for each pod deployment is made on average on 3.2 ms, while the SFC controller requires on average 6.08 ms because of the extender procedure. The pod startup time corresponds to the duration between the deployment command until the moment it takes to allocate and instantiate the given containers in the cluster. Both KS and the SFC controller require on average between 2 and 3 seconds to allocate the required containers, since the main difference in execution time is only determined by the extender procedure and how the final node is chosen.

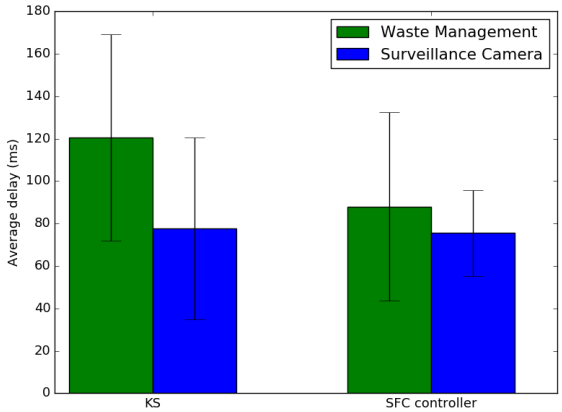

Fig. 8: The expected service latency for each of the schedulers.

B. Pod Allocation Scheme

In Table VI, the different allocation scheme for each of the schedulers is shown. As expected, the KS deployment scheme is not optimized for the service's desired location or service latency, since no considerations are made about location or latency in its scheduling algorithm. Thus, KS allocates multiple pods on a single node since it tries to balance the load in the cluster according to CPU and RAM usage rates. For instance, the KS allocation scheme for the route planner or the fm-recog service is fairly poor since no pods are deployed in the preferred location (Brussels).

C. Network Bandwidth

In Table VII, the expected service bandwidth per node for the different scheduling approaches is presented. It should be noted that bandwidth values in bold mean that the cluster node is overloaded based on the available bandwidth previously shown in Fig. 4. As shown, KS allocates pods on nodes already compromised in terms of network bandwidth. For instance, KS overloads worker 1 and 4 by allocating to them at least 4 pods leading to service bandwidths of 26.0 Mbit/s and 36.5 Mbit/s for the workers 1 and 4, respectively, which surpasses the available bandwidth of 10.0 Mbit/s. This provisioning scheme may lead to service disruptions due to bandwidth fluctuations. In contrast, the proposed SFC controller takes into account the available bandwidth while making scheduling decisions, which leads to informed decisions not only in terms of latency but also in terms of bandwidth.

D. Network latency

In Fig. 8, the expected service latency for each of the schedulers is detailed. As shown, the proposed SFC-controller achieves lower delays for each of the deployed services when compared with the default KS. In spite of overloading several nodes, KS is not able to find optimal paths for the SFCs. The proposed SFC controller can optimize the SFC latency while conserving network bandwidth. In this particular allocation scheme, the SFC-controller improves the performance of the default KS by reducing the network latency by 18%.

TABLE VI: The pod allocation scheme of the different schedulers.

		Scheduler	
Use Case	Service	KS	SFC controller
Waste Management	api	[Worker 5, Worker 11, Worker 12]	[Worker 3, Worker 5, Worker 7]
	waste-db	[Worker 1, Worker 4, Worker 10, Master]	[Worker 4, Worker 10, Worker 11, Worker 14]
	route-planner	[Worker 1, Worker 4, Worker 11, Master]	[Worker 6, Worker 13, Worker 14, Master]
	server	[Worker 6, Worker 10, Master]	[Worker 13, Worker 14, Master]
Surveillance Camera	fd-ext	[Worker 1, Worker 4, Worker 5, Worker 12]	[Worker 1, Worker 2, Worker 12, Master]
	fm-recog	[Worker 3, Worker 4]	[Worker 13, Master]
	cam-db	[Worker 3, Worker 4]	[Worker 7, Worker 8]
	dashboard	[Worker 1, Worker 4, Worker 10, Master]	[Worker 8, Worker 9, Worker 10, Worker 11]

TABLE VII: The expected service bandwidth per node for the different scheduling strategies.

Node	KS	SFC-controller
Worker 1	**26.0 Mbit/s**	8.0 Mbit/s
Worker 2	-	8.0 Mbit/s
Worker 3	**10.5 Mbit/s**	4.0 Mbit/s
Worker 4	**36.5 Mbit/s**	5.0 Mbit/s
Worker 5	**12.0 Mbit/s**	4.0 Mbit/s
Worker 6	4.0 Mbit/s	8.0 Mbit/s
Worker 7	-	6.5 Mbit/s
Worker 8	-	7.5 Mbit/s
Worker 9	-	5.0 Mbit/s
Worker 10	**14.0 Mbit/s**	10.0 Mbit/s
Worker 11	**12.0 Mbit/s**	10.0 Mbit/s
Worker 12	**12.0 Mbit/s**	8.0 Mbit/s
Worker 13	-	20.0 Mbit/s
Worker 14	-	17.0 Mbit/s
Master	22.0 Mbit/s	28.0 Mbit/s

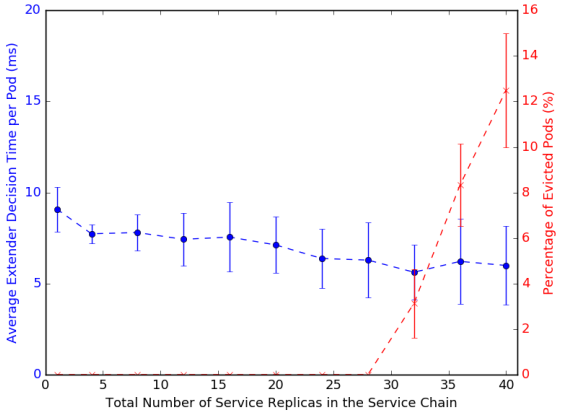

Fig. 9: The average execution time per Pod of the SFC controller extender call.

E. Scalability

In Fig. 9, the execution time per Pod of the SFC controller extender call is shown. The number of service replicas is increased to evaluate how the SFC controller handles the allocation of a high number of service instances in the Service Chain. As shown, the extender decision time decreases while the number of service replicas increases, since only a small amount of nodes will still be free in terms of resources, specifically in terms of bandwidth based on the previously presented infrastructure. Increasing the number of replicas will also lead to pod evictions since nodes will already be exhausted and no node will be available for the service allocation. Nevertheless, the SFC controller can cope with a high number of replicas without compromising the decision time as long as resources are available.

In summary, the proposed SFC-controller optimizes the resource provisioning in Kubernetes according to network latency and bandwidth, which is currently not supported by the default KS.

VI. Conclusions

In recent years, Cloud computing is evolving towards a distributed paradigm called Fog Computing, which aims to provide a distributed cloud infrastructure by placing computational resources close to end-users. Additionally, mobile operators are researching efficient ways of connecting different services in a specific order forming an SFC to fully benefit from network virtualization. The deployment of SFCs will allow mobile operators to profit from the high flexibility introduced by network softwarization. Nevertheless, SFC research only focuses on MEC use cases and only a few studies consider Fog-Cloud environments. Therefore, in this paper, an SFC controller has been presented as a scheduling approach to efficiently place container-based service chains in Fog-Cloud environments, specifically tailored for Smart City use cases. The popular open-source project Kubernetes has been used to validate the proposed approach. The SFC controller has been implemented as an extension to the scheduling features available in Kubernetes, enabling the allocation of container-based SFCs while optimizing resource provisioning and reducing the E2E latency. Evaluations have been performed to compare the proposed solution with the default scheduling feature available in Kubernetes. Results show that the proposed approach can significantly reduce the network latency while conserving bandwidth just by increasing the scheduling decision time by only 6ms per pod. As future work, dynamic strategies will be added to our SFC controller to further refine the allocation scheme in terms of bandwidth fluctuations and delay changes.

Acknowledgment

This research was performed within the project "Intelligent DEnse And Longe range IoT networks (IDEAL-IoT)" under Grant Agreement #S004017N, from the fund for Scientific Research-Flanders (FWO-V).

REFERENCES

[1] M. Agiwal, A. Roy, and N. Saxena, "Next generation 5g wireless networks: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 1617–1655, 2016.

[2] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2015.

[3] S. Abdelwahab, B. Hamdaoui, M. Guizani, and T. Znati, "Network function virtualization in 5g," *IEEE Communications Magazine*, vol. 54, no. 4, pp. 84–91, 2016.

[4] B. Yi, X. Wang, K. Li, M. Huang *et al.*, "A comprehensive survey of network function virtualization," *Computer Networks*, vol. 133, pp. 212–262, 2018.

[5] D. Bhamare, R. Jain, M. Samaka, and A. Erbad, "A survey on service function chaining," *Journal of Network and Computer Applications*, vol. 75, pp. 138–155, 2016.

[6] Y. Xie, Z. Liu, S. Wang, and Y. Wang, "Service function chaining resource allocation: A survey," *arXiv preprint arXiv:1608.00095*, 2016.

[7] R. Mahmud, R. Kotagiri, and R. Buyya, "Fog computing: A taxonomy, survey and future directions," in *Internet of everything*. Springer, 2018, pp. 103–130.

[8] H. Arasteh, V. Hosseinnezhad, V. Loia, A. Tommasetti, O. Troisi, M. Shafie-Khah, and P. Siano, "Iot-based smart cities: a survey," in *2016 IEEE 16th International Conference on Environment and Electrical Engineering (EEEIC)*. IEEE, 2016, pp. 1–6.

[9] 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.

[10] S. Newman, *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[11] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," 2016.

[12] F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and O. C. M. B. Duarte, "Orchestrating virtualized network functions," *IEEE Transactions on Network and Service Management*, vol. 13, no. 4, pp. 725–739, 2016.

[13] A. Alleg, T. Ahmed, M. Mosbah, R. Riggio, and R. Boutaba, "Delay-aware vnf placement and chaining based on a flexible resource allocation approach," in *2017 13th International Conference on Network and Service Management (CNSM)*. IEEE, 2017, pp. 1–7.

[14] A. Marotta, F. D'Andreagiovanni, A. Kassler, and E. Zola, "On the energy cost of robustness for green virtual network function placement in 5g virtualized infrastructures," *Computer Networks*, vol. 125, pp. 64–75, 2017.

[15] H. Hawilo, M. Jammal, and A. Shami, "Orchestrating network function virtualization platform: Migration or re-instantiation?" in *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*. IEEE, 2017, pp. 1–6.

[16] D. Harutyunyan, S. Nashid, B. Raouf, and R. Riggio, "Latency–aware service function chain placement in 5g mobile networks," in *IEEE Conference on Network Softwarization (NetSoft 2019)*, 2019.

[17] M. Aazam and E.-N. Huh, "Dynamic resource provisioning through fog micro datacenter," in *2015 IEEE international conference on pervasive computing and communication workshops (PerCom workshops)*. IEEE, 2015, pp. 105–110.

[18] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar, "Towards qos-aware fog service placement," in *2017 IEEE 1st international conference on Fog and Edge Computing (ICFEC)*. IEEE, 2017, pp. 89–96.

[19] J. Yao and N. Ansari, "Fog resource provisioning in reliability-aware iot networks," *IEEE Internet of Things Journal*, 2019.

[20] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Resource provisioning in fog computing: From theory to practice," *Sensors*, vol. 19, no. 10, p. 2238, 2019.

[21] ——, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *Proceedings of the IEEE Conference on Network Softwarization (NETSOFT), Paris, France*, 2019, pp. 24–28.

[22] K. Hightower, B. Burns, and J. Beda, *Kubernetes: up and running: dive into the future of infrastructure*. " O'Reilly Media, Inc.", 2017.

[23] (2019) Kubernetes, automated container deployment, scaling, and management. [Online]. Available: https://kubernetes.io/

[24] Y. Y. G. Jianya, "An efficient implementation of shortest path algorithm based on dijkstra algorithm [j]," *Journal of Wuhan Technical University of Surveying and Mapping (Wtusm)*, vol. 3, no. 004, 1999.

[25] (2019) The virtual wall emulation environment. [Online]. Available: https://doc.ilabt.imec.be/ilabt-documentation/index.html

[26] (2019) Overview of kubeadm. [Online]. Available: https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm/

[27] A. Medvedev, P. Fedchenkov, A. Zaslavsky, T. Anagnostopoulos, and S. Khoruzhnikov, "Waste management as an iot-enabled service in smart cities," in *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*. Springer, 2015, pp. 104–115.

[28] 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.