Dynamic Resource Allocation and Placement of Cloud Native Network Services

Sagar Arora, Adlen Ksentini
Eurecom, Sophia Antipolis, France
firstname.lastname@eurecom.fr

***Abstract*—Cloud-native technologies have recently entered the telecommunication world. These technologies were specially designed for developing and orchestrating container-based applications. The new cloud-native network functions use container based virtualization instead of virtual machine-based virtualization. These network functions have low resource footprints and low deployment time, making them suitable for a distributed environment. To adopt these new network functions and cloud native approach the network function virtualization vision needs alterations. In this paper, we use cloud-native approach to provide resilience to cloud-native network services. We proposed dynamic resource allocation and placement algorithm for modeling and placing a simple cloud-native network service. The algorithm aims to minimize infrastructural resource utilization under the constraint of abiding service availability mentioned in the service level agreement.**

***Index Terms*—NFV, VNF, cloud native, placement**

I. Introduction

The initial release of NFV specification [1] was predominantly dependent on hypervisor-based Virtual Machines (VM) for virtualization. A recent improvement in container based virtualization has introduced cloud-native technologies, which are driving the virtualization in the cloud. The ETSI NFV group has recently published a report "Enhancements of the NFV architecture towards cloud-native and PaaS" [2], which introduces container-based Cloud-native Network Functions (CNF).

Container-based applications do not require full operating system like virtual machines. Making them lightweight and reducing their deployment time. They can be assigned vCPU at a granularity of 1milli cpu, where 1000m CPU is equivalent to 1vCPU [3]. Whereas in VMs, the minimum vCPU that can be assigned is 1vCPU. CNFs can use this finer vCPU assignment to have an improved infrastructural resource utilization in comparison to traditional VM based VNFs.

Network functions can be used alone or along with other network functions to provide network services. Latter are offered by service providers who either owe the infrastructure or lease it from infrastructure providers. While providing a service, they have to abide by service availability which is an important attribute of the Service Level Agreement (SLA). The ETSI NFV group has published specifications and guidelines related to the resilience and availability of network functions and network services [4]. They acknowledge higher availability is subjected to higher deployment and management cost. Making it important to find cost-availability trade-off. This trade-off has always been a challenge for service providers, in maximizing their profits.

Cloud-native network services have to reach the telco grade 99.999% availability. To achieve this with one instance of each CNF required to provide the service can be challenging. The cloud-native way to achieve this availability is to have multiple replicas of the CNFs composing a network service. This might lead to over provisioning of infrastructural resources, which increases the deployment and management cost. Hence, a decision problem arise; How many replicas of each CNF a cloud-native network service needs? Without over-provisioning computational resources to avoid the high cost and provide service availability as promised in the SLA.

In this paper, we propose a solution for this decision problem from the perspective of service providers. Existing related work in the field of cloud-native network functions does not address the above problem or consider cost and availability together as an attribute while placing them on cloud infrastructure. To fill this gap, we provide an algorithm to model a simple cloud-native network service.

A simple cloud native network service requires a single CNF to provide the network service functionality. We make two assumptions: first, service provider knows in advance the computational resources, importantly vCPU required by the service to serve the user demand; second, the nodes on which CNF replicas will be placed have enough storage and memory resource. We make the following contribution in this paper,

1) Cost and availability model for simple cloud-native network service
2) Dynamic Resource Allocation and Placement (DRAP) algorithm for design and placement of a simple cloud-native network service. DRAP provides a dictionary that contains the number of CNF replicas, placement on infrastructure node, and vCPU allocation for each replica. It abides by service availability as a constraint.

The rest of the paper is arranged in chronological order, initial background of cloud-native network functions, related work, our contributions, results and conclusion.

II. Background: Cloud Native Telco

Cloud-native [5] is a result of the growing demand for virtualization in IT industry. Container Orchestration Engine (COE), like Kubernetes is responsible for its integration within telco and especially in the NFV vision. In this paper we refer the terminologies from the ETSI cloud native report [2]. In

this section we provide a brief background of cloud-native network functions and their orchestration.

A. Cloud native Network Functions

Cloud-native applications are designed on microservices-based architecture with statelessness in consideration. In contrast, the traditional VM-based network functions are monolithic and stateful. Containerizing these monolithic applications will not receive the complete benefits of an agile cloud-native application. Re-engineering these traditional VM based VNFs to adapt cloud-native principles is a possible solution. Fig. 1 depicts a Kubernetes based cloud-native VNF.

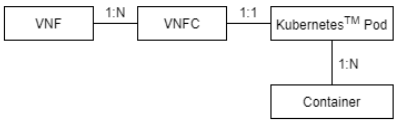


Fig. 1: Cloud native Network Function (CNF) [2] sec.6.2.4

Every VNF can have multiple Virtualized Network Function Components (VNFC) and each component is mapped to a Kubernetes pod. These Kubernetes pods can have multiple containers depending on the design of VNFC. A VNF is a logical entity whereas VNFCs are the functional blocks, which should follow cloud-native design principles. In a traditional VM-based VNF, the VNFC is mapped to a VM. It should be noted that different design possibilities for a CNF depending on the COE exist, which are not discussed in this paper.

B. NFV Orchestrator NFVO

A NFV orchestrator is responsible for resource and service orchestration. In a cloud-native environment, it will perform the same functions, but with new functional blocks relevant for cloud-native network functions. The ETSI cloud native report [2] proposes new functional blocks and their possible placement in traditional NFV-MANO architecture.

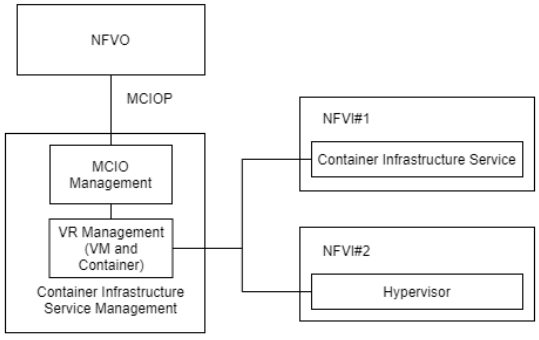


Fig. 2: Position of CISM and CIS in ETSI MANO [2], sec. 7.2.4.5

The cloud-native equivalent of hypervisor is Container Infrastructure Service (CIS), which provides all the runtime infrastructural dependencies for one or more container virtualization technologies. Container Infrastructure Service Management (CISM) is a cloud-native equivalent of Virtualized Infrastructure Manager (VIM). The functionality of VNFM can be integrated with CISM, but it has some pros and cons already discussed in the report. Managed Container Infrastructure Object Package (MCIOP) contains the placement, resource allocation and configuration related information for a container or VM-based network function. Fig. 2 depicts one of the six proposed architectures to place CISM and CIS. In this architecture, NFVO is capable of orchestrating container and VM-based network functions. However, it should be noted that NFV Infrastructure (NFVI) is outside the framework of NFV-MANO.

III. Related Work

Most of the existing related work focuses on the resource allocation and placement of traditional VM-based virtual network functions. Authors of [6] have proposed a model for joint vCPU to VM allocation and VM placement considering a simple CDN network service. They have used this model to highlight and address the cost and availability trade-off. The service availability model presented in our paper is inspired from this work. Authors of [7] provided a queuing theory-based approach to solve traditional VM-based VNF placement and resource assignment problem for a 5G network service. Their presented model considers a service function chain that can be useful for simple as well as complex network services.

IV. Simple Cloud Native Network Service

NFVO manages the life-cycle of a network service using a Network Service Descriptor (NSD), which is received from northbound entities such as, OSS/BSS or a slice orchestrator in case of Network slicing [8]. The NSD contains details about virtualized and physical network functions, virtual and physical links between them, etc. The NSD is used for modeling, placing, and scaling the instances of VNFs. Based on this, we consider along with NSD, NFVO receives the maximum vCPU required by the service and the required service availability. The calculation for the maximum vCPU required by a service is out of the scope.

A. Modeling a Simple Cloud Native Network Service

In a traditional VM-based network service, the network functions are connected via the Service Function Chain (SFC) concept [9]. In the previous sections we highlighted that a network service can have multiple different types of VNFs and PNFs. These VNFs may have replica instances depending on the design of the network service. The ETSI specification for cloud-native VNF implementation [10] acknowledge redundancy as a possible solution to provide resiliency.

Based on this, we propose a model for a cloud-native network service, which has multiple replicas of the same CNF, and the maximum vCPU that the service requires is divided among these replicas. The maximum vCPU consumed by the replicas will not exceed the maximum vCPU required by the service. The service load is distributed between replicas, which provide resilience to the service without over-provisioning the computational resources. This will be discussed in detail in in the availability section.

We named this service as *Simple Cloud Native Network Service* because there are multiple replicas of the "same kind of CNF". This CNF has one VNFC in a Kubernetes pod. This pod can have single or multiple containers.

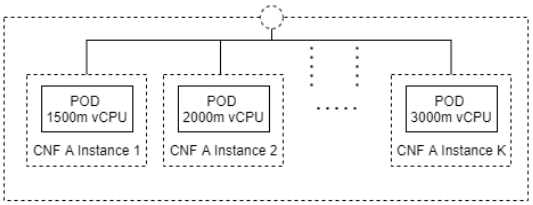


Fig. 3: Simple cloud-native network service S with K replicas of CNF A

B. Deploying a Simple Cloud Native Network Service

In Fig. 4 the NFVO receives a deployment request for a simple cloud native network service. The request contains NSD, maximum vCPU required by the service (R_{vCPU}), and required service availability (RA). NFVO starts with onboarding CNF images and then Dynamic Resource Allocation and Placement (DRAP) algorithm will generate Placement And vCPU Allocation (PACA) dictionary. It sends the dictionary and RA to CISM.

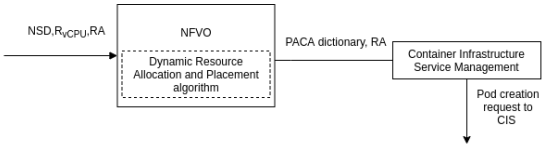


Fig. 4: Cloud native network service deployment flow diagram

CISM place CNF pod(s) on NFVI and allocate the vCPU as mentioned in the dictionary. Pod placement on NFVI is based on the dictionary. It should be noted that generally COE performs initial placement of pods, here, NFVO is performing initial placement.

V. Dynamic Resource Allocation and Placement Model

A. Preliminary

Now onward we use the notations highlighted in TABLE I. The notations are specific for a simple cloud native network service S. A pod refers to CNF instance as depicted in Fig. 3. Besides from the table, $X = (x_{ij})$ is the pod placement matrix, x_{ij} is 1 if pod i is placed on node j otherwise 0. $W = (w_{ij})$ is the vCPU allocation matrix and w_{ij} denotes the vCPU allocated to pod i placed on node j. All $i\epsilon[1, K]$ and $j\epsilon[1, N]$.

B. Cost Model

Consider an array $Y = (y_j)$ and $j\epsilon[1, N]$, where y_j denotes the status of a NFVI node j on which pods can be placed. y_j is 1 when node j is hosting at least one pod. Otherwise, node is not hosting any pod then 0.

NFVI is a cluster of physical machines providing infrastructural resources (computational, storage, and networking resources). The container infrastructure service instance can be present on VMs or bare-metal. We consider in this work bare-metal deployment of CIS in Fig. 2. We consider that each node of the cluster has a fixed cost L when it is hosting pod(s), otherwise, the cost is zero. In a Kubernetes node, this fixed cost can be calculated by computing the computational resources consumed by operating system processes, container runtime engine, and Kubernetes fix components. We formulate the cost model as,

$$D = L * \sum_{j=1}^{N} y_j + R_{vCPU} \tag{1}$$

D denotes the deployment cost for a simple cloud native network service. Generally, in Kubernetes, the CPU is always requested as an absolute quantity [11]. The unit of D is in milli vCPU.

TABLE I: Summary of Notations

N	Number of nodes available in NFVI (CIS) for pod placement
R_{vCPU}	Maximum vCPU in milli units required by the service
P_{max}, P_{min}	Maximum and minimum number of pods which can host the service
K	Actual number of pods hosting the service S
min_{vCPU}	Minimum vCPU in milli units which can be assigned to a pod
max_{vCPU}	Maximum vCPU in milli units which can be assigned to a pod
$C(j)$	vCPU capacity of node j in milli units, $j\epsilon[1, N]$
RA	Service availability required by the service
h_j	Failure probability of node j, $j\epsilon[1, N]$
g_i	Failure probability of pod i, $i\epsilon[1, K]$

C. Availability Model

We consider two types of service availability models depending on the Quality of Service (QoS) perceived by each user of the service. First, *relax availability* or minimal service model i.e., at any time at least one pod is accessible. In this model, the QoS perceived by each user can be degraded and might not be the same as requested in the SLA. This might happen due to the unavailability of some instances and their load being shared among available instances.

Second, *strict availability model*, which maintains the QoS perceived by each user as requested in the SLA. To achieve this, all the pod(s) and the node(s) hosting the respective pod(s) should be accessible. We made the following assumptions for both models,

- A pod i can fail with probability g_i, independent of the other pod(s) and node(s), irrespective of the load imposed on the pod, and resources allocated to the pod.
- A node j can fail with probability h_j, independent of the other node(s) and pod(s) running on it.

The above probabilities are already known to the service provider as a result of measurement studies or prior experience. A pod may be inaccessible due to its failure or node

failure which is hosting the pod. Pod failures can be correlated due to their dependence on the underlying node(s). Based on this, we define a correlated group of pods as the pods instantiated on the same node. The availability of a correlated group is subjected to the below models,

1) Relaxed availability model: Availability of a correlated group is subjected to the availability of,

- The node on which the group is hosted,
- At least one pod of the correlated group should be available

Probability that a correlated group j hosted on node j will be available is,

$$a_j = (1 - h_j) * (1 - \prod_{i \epsilon [1,K] | x_{ij}=1} g_i) \tag{2}$$

For the service to be available, at least one correlated group should be available. Considering that correlated groups fail independently, the service availability is defined as,

$$\begin{aligned} A(X) &= 1 - \Pr\{\text{All correlated group fail}\} \\ &= (1 - \prod_{j \epsilon [1,N] | \sum_{i=1}^{K} x_{ij}=1} (1 - a_j)) \end{aligned} \tag{3}$$

2) Strict availability model: Availability of a correlated group is subjected to the availability of,

- The node on which the group is hosted,
- All the pods of the correlated group should be available

The probability that a correlated group j hosted on node j will be available is,

$$a_j = (1 - h_j) * [\prod_{i \epsilon [1,K] | x_{ij}=1} (1 - g_i)] \tag{4}$$

To deliver per user perceived QoS as mentioned in the SLA, all the correlated groups should be available. Considering that correlated groups fail independently the service availability is defined as,

$$\begin{aligned} A(X) &= \Pr\{\text{All correlated group are available}\} \\ &= \prod_{j \epsilon [1,N] | \sum_{i=1}^{K} x_{ij}=1} a_j \end{aligned} \tag{5}$$

D. Constraints

Resource allocation and placement are subjected to constraints. We categories these constraints as infrastructural level or service level.

1) Infrastructural level constraints: Eq. 6 defines the capacity constraint, where the pod(s) hosted on node j can not exceed the available vCPU resources,

$$\sum_{i=1}^{K} w_{ij} \leqslant C(j) * y_j \tag{6}$$

Eq. 7 defines the pod hosting constraint, where a pod i can only be hosted on one node,

$$\sum_{j=1}^{N} x_{ij} = 1 \tag{7}$$

Eq. 8 defines vCPU limiting constraints,

$$\begin{aligned} & w_{ij} \leqslant max_{vCPU} * x_{ij} \\ & w_{ij} \geqslant min_{vCPU} * x_{ij} \\ & \forall i \epsilon [1, K], j \epsilon [1, N] \end{aligned} \tag{8}$$

2) Service level constraints: Eq. 9 defines the provisioning constraint to avoid over-provisioning. We are not considering pod overheads [12].

$$\sum_{j=1}^{N} \sum_{i=1}^{K} w_{ij} = R_{vCPU} \tag{9}$$

Eq. 10 defines the service availability constraint, where the service availability achieved by the placement algorithm should be equal to or higher than the required availability.

$$A(X) \geqslant RA \tag{10}$$

E. Problem Formulation

Service providers aims to minimize the service deployment cost as defined in eq. 1 and maintain service availability as per SLA. The variable component of that cost is the number of nodes hosting the service pods. We propose an Integer Linear Programming (ILP) formulation with an objective to minimize the number of nodes hosting the service pods while considering service availability constraint eq. 10.

$$Min \sum_{j=1}^{N} Y_j \tag{11}$$

The objective function also considers other constrains mentioned in eq. 6, 7, 8, 9. The value of minimum vCPU that can be allocated to a pod is fixed due to design reasons of the VNFC application running inside a pod. Maximum vCPU is fixed to avoid a pod from consuming vCPU resources allocated to other pods. By fixing these values, the number of maximum and minimum number of pods hosting the service is fixed.

$$\begin{aligned} & P_{max} = R_{vCPU} / min_{vCPU} \\ & P_{min} = R_{vCPU} / max_{vCPU} \\ & P_{min} \leqslant K \leqslant P_{max} \end{aligned} \tag{12}$$

VI. Heuristic Approach

Our problem formulation is similar to the well known bin packing problem. In a classical bin packing problem [13], the aim is to minimize the number of bins used to fit items of variable volume. These bins have a fixed volume. Whereas in our problem nodes with variable vCPU resemble to bins, and pods with variable size resemble to items. The classical bin packing problem is NP-hard [13] and there are heuristic algorithms to solve it. In our problem bin capacity is variable, the number of items is variable with variable capacity and there are additional constraints like service availability. Which results in a NP-hard problem as well. Considering this, we propose a heuristic algorithm, namely Dynamic Resource

Allocation and Placement (DRAP) to solve the problem in polynomial time.

The DRAP algorithm aims to minimize the number of nodes required to place the pods by adjusting the vCPU allocated to each pod. Adjustable vCPU allocation allows the algorithm to increase or decrease the number of pods based on the required service availability.

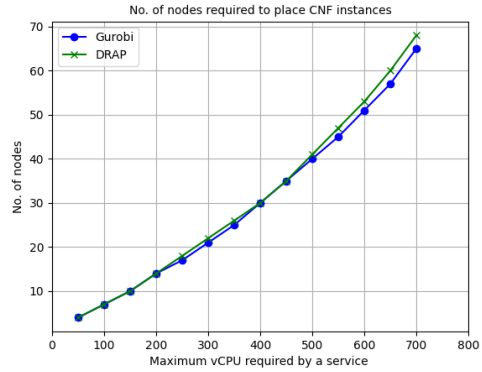
Input: $r > 0, C(j), N, R_{vCPU}, max_{vCPU}, min_{vCPU}$
Output: K, W, X

$i = 1$
Sort C(j) in decreasing order
while $t \geq min_{vCPU}/max_{vCPU}$ **do**
  for j in [1,N] **do**
    while $C(j) \geq 0$ **do**
      $w_{ij} = min(C(j), t * max_{vCPU})$
      if $w_{ij} \leq min_{vCPU}$ **then**
        $break$
      end if
      $C(j) = C(j) - w_{ij}$
      $x_{ij} = 1$
      if $sum(W) \geq R_{vCPU}$ **then**
        $break$
      end if
      if $i \leqslant P_{max}$ **then**
        $i = i + 1$
      else
        $break$
      end if
    end while
    if $A(X) \geq RA$ **then**
      $K = i$
      return W, X, K
    else
      $break$
    end if
  end for
  $t = t - r$
  reinitialize,K, W, X
end while

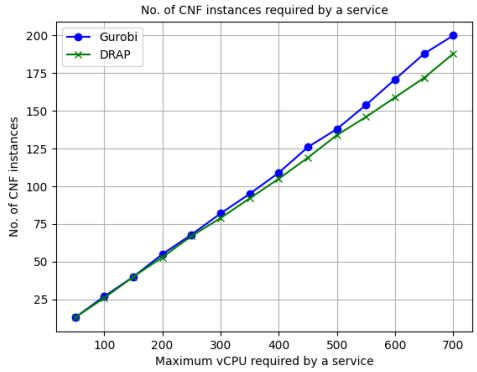
DRAP starts with sorting the nodes in decreasing order of available vCPU capacity. The tuning factor t is used to adjust the size of the pods. PACA dictionary W stores i pod number also pod name, vCPU allocation w_{ij} and node number j. The tuning rate r iterates by reducing the vCPU allocation from maximum to minimum possible. Algorithm iterates over several possible combinations of K, W and X until it finds the pod placement matrix X that satisfy the availability constraint, or it assigns the minimum possible vCPU min_{vCPU} to the maximum number of pods P_{max} that a service can have. If there is not enough capacity available in the cluster, algorithm will return no solution.

VII. Results

All the simulations were performed on Intel Core i5-9400F with 6 CPU@2.90GHz and 32GiB of RAM. Academic license of Gurobi optimizer was used to resolve our ILP. We fixed node and pod failure probabilities for all the nodes and pods constant, where $h_j = 0.001$, $g_i = 0.001$ for all $i \epsilon [1, K], j \epsilon [1, N]$. The minimum and maximum vCPU which can be allocated to pods are, 2000m and 4000m vCPU unit respectively.



(a) Number of nodes required



(b) Number of pods required

Fig. 5: Number of nodes and pods required by a cloud native network service

Fig. 5 depicts the number of nodes and pods required to host CNF instances of different cloud native network services. Each service requires different vCPUs, 50, 100 up to 700vCPUs respectively and 99.999% service availability. The services were placed on a cluster of 100 nodes, and each node has a capacity between 2 and 16vCPUs selected uniformly at random. The placement of each service was independent of the other services. The cluster nodes had the same vCPU distribution for all the services at the time of placement. The performance of DRAP is close to Gurobi in terms of reducing the number of nodes. Both of them provide similar availability for each service.

In Fig. 6 we considered a network service requiring 600vCPUs and 99.999% availability. The minimum and maximum vCPU allocation for pods fixed to 1000m and 3000m vCPU unit respectively. We placed the service on clusters having 200, 400 upto 2000 nodes. Every node of a cluster has 8vCPUs. The sharp increase in gurobi's execution time justifies that the problem is NP-hard. The heuristic algorithm DRAP provides

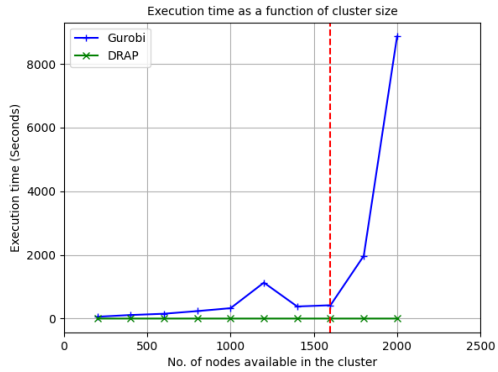
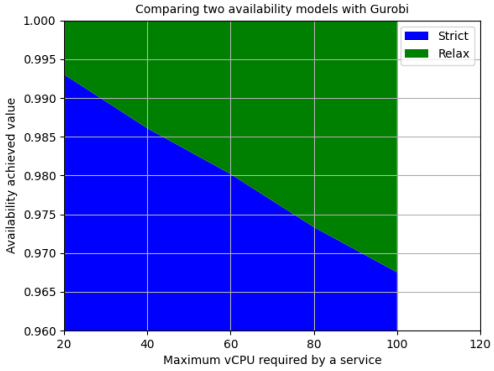
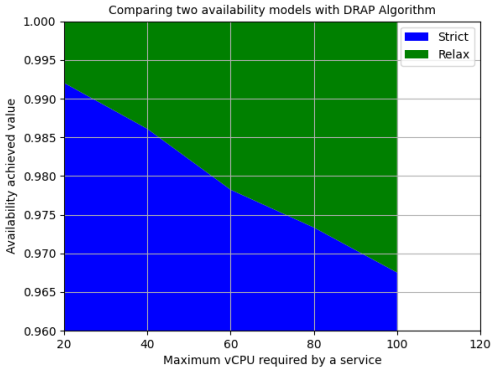


Fig. 6: Execution time as a function of cluster size

the solution in a very less amount of time. The time taken by DRAP to provide PACA dictionary varies between 13ms and 108ms.



(a) Gurobi



(b) DRAP

Fig. 7: Comparing two availability models

Fig. 7 compares the strict and relax availability models defined previously. We placed five services on a cluster of 50 nodes where each service requires 20, 40 up to 100vCPUs. Each cluster node has a capacity between 2 and 16vCPUs selected uniformly at random. The aim is to achieve maximum service availability, which can be promised by both models. For example, a service requesting 60vCPU with DRAP 97.823% of time there will no QoS degrade for any user and 99.999% time service will be available but some users might be affected. Whereas for Gurobi these values are 98.019% under strict and 99.999% under relax model. Gurobi and DRAP have nearly similar performance.

VIII. Conclusion

Cloud-native has just started to enter in the telecom world. Hence, there is a need to consider re-engineering the traditional VM-based network functions to benefit from the agility of containers. We used the cloud-native approach of having multiple replicas without over-provisioning resources to provide resilience. Our proposed algorithm benefits from allocating vCPU dynamically and at a finer granularity. It can be considered that by allocating vCPU at a finer granularity nodes capacity can be efficiently utilized. If a node has a low computational capacity, then the pod requirements can be adjusted. Our approach can be beneficial for service providers in reducing their infrastructural cost. The service we considered was simple because it has replicas of the same type of CNF. If there is a service which has different type of CNFs and each CNF has replicas, then there can be complications related to affinity, which are not considered in our algorithm. In our future work, we will work with a complex network service which considers a SFC.

References

[1] Network Functions Virtualisation (NFV); Management and Orchestration, ETSI GS NFV-MAN 001 V1.1.1, Dec. 2014
[2] Network Functions Virtualisation (NFV) Release 3; Architecture; “Report on the Enhancements of the NFV architecture towards Cloud-native and PaaS”, ETSI GR NFV-IFA 029 V3.3.1, Nov. 2019.
[3] Kubernetes.[Online].Available:https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
[4] Network Functions Virtualisation (NFV); Resiliency Requirements, ETSI GS NFV-REL 001, Jan. 2015.
[5] Cloudnative.[Online].Available:https://github.com/cncf/toc/blob/master/DEFINITION.md
[6] L. Yala, P. A. Frangoudis, G. Lucarelli and A. Ksentini, “Cost and Availability Aware Resource Allocation and Virtual Function Placement for CDNaaS Provision,” in IEEE Tran. on Network and Service Management, vol. 15, no. 4, pp. 1334-1348, Dec. 2018.
[7] S. Agarwal, F. Malandrino, C. F. Chiasserini and S. De, ”VNF Placement and Resource Allocation for the Support of Vertical Services in 5G Networks,” in IEEE/ACM Transactions on Networking, vol. 27, no. 1, pp. 433-446, Feb. 2019
[8] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini and H. Flinck, “Network Slicing and Softwarization: A Survey on Principles, Enabling Technologies, and Solutions,” in IEEE Communications Surveys & Tutorials, vol. 20, no. 3, pp. 2429-2453, thirdquarter 2018.
[9] A. M. Medhat, T. Taleb, A. Elmangoush, G. A. Carella, S. Covaci and T. Magedanz, ”Service Function Chaining in Next Generation Networks: State of the Art and Research Challenges,” in IEEE Communications Magazine, vol. 55, no. 2, pp. 216-223, Feb 2017.
[10] Network Functions Virtualisation (NFV) Release 3; Virtualised Network Function; “Specification of the Classification of Cloud Native VNF implementations”, ETSI GS NFV-EVE 011 V3.1.1, Oct 2018
[11] KubernetesCPU.[Online].Available:https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu
[12] Podoverhead.[Online].Available:https://kubernetes.io/docs/concepts/configuration/pod-overhead/
[13] B. H. Korte and J. Vygen, “Bin Packing,” in Combinatorial optimization: theory and algorithms, Berlin: Springer, 2006.