On Monolithic and Microservice deployment of Network Functions

Sachin Sharma[1], Navdeep Uniyal[2], Besmir Tola[3] and Yuming Jiang[3]
National College of Ireland[1], University of Bristol, U.K.[2] and NTNU, Norway[3]
Email: sachin.sharma@ncirl.ie[1], navdeep.uniyal@bristol.ac.uk[2], {besmir.tola, yuming.jiang}@ntnu.no[3]

***Abstract*—Network Function Virtualization (NFV) has recently attracted telecom operators to migrate network functionalities from expensive bespoke hardware systems to virtualized IT infrastructures where they are deployed as software components. Scalability, up-gradation, fault tolerance and simplified testing are important challenges in the field of NFV. In order to overcome these challenges, there is significant interest from research communities to scale or decompose network functions using the monolithic and microservice approach. In this paper, we compare the performance of both approaches using an analytic model and implementing test-bed experiments. In addition, we calculate the number of instances of monoliths or microservices in which a network function could be scaled or decomposed in order to get the maximum or required performance. Single and multiple CPU core scenarios are considered. Experimentation is performed by using an open source network function, SNORT and running monoliths and microservices of SNORT as Docker containers on bare metal machines. The experimental results compare the performance of monolith and microservice approaches and are used to estimate the validity of the analytic model. The results also show the effectiveness of our approach in finding the number of instances (monoliths or microservices) required to maximize performance.**

***Index Terms*—Network Functions; Performance; VNF Monoliths; VNF Microservices; VNF decomposition**

I. Introduction

In recent years, monolithic and microservice approaches for deploying applications have been successfully implemented in the field of cloud computing [1]. In the case of the monolithic approach, an application is deployed as a single software package that offers tens or hundreds of services. However, in the microservice approach, an application is decomposed into a set of independently deployed smaller sub-components (i.e., microservices) in which each sub-component does a small and simple task, and communicates through well-defined, light weight mechanisms.

Many cloud computing companies, such as Netflix and Amazon, have been deploying their applications using the microservice approach. For these companies, the microservice approach offers following main advantages [2]:

1) Scalability: The various components of an application have diverse performance footprints. Thus, some components might be more resource intense than others. In comparison to the monolithic approach, the microservice approach provides an efficient way of scaling up only resource intense components.
2) Up-gradation: Since small applications can be upgraded independently in short time, upgrading microservices is a lot easier and less disruptive (e.g., in terms of down-time) than upgrading an entire monolithic application.
3) Simplified development: The microservice approach fosters software re-usability in building different applications. It is easier to re-use smaller code fragments (i.e., microservices) than a large code base (e.g., monoliths). They can even be written in different programming languages. Evidently this simplifies the software development process, especially for the big code-development teams.
4) Simplified testing: Since the feature set of a microservice is much less compared to a monolithic application, it is easier to test and debug service components. Testing the overall service, however, might be more complex though.
5) Fault tolerance: When a critical failure (e.g., a segmentation fault) happens in a monolithic application, the whole application goes down. In case of microservices, however, only a small subset may be affected if a failure happens to a single or a subset of microservices.

In addition to aforementioned advantages, a microservice approach might introduce the following overheads compared to the monolithic approach:

1) Decomposition overhead: The microservice approach requires an application to be decomposed. Today, this is usually considered more complex and introduces additional efforts as compared to the monolithic approach. In the monolithic approach, these additional overheads are not there, as an application is not required to be decomposed.
2) Orchestration overhead: Orchestrating (i.e., coordinating, deploying, configuring, and managing) a set of microservices that constitute the overall service, may result in a larger overhead compared to orchestrating a single monolithic application as the number of managed entities is increased.
3) Communication overhead: Since microservices are loosely coupled and possibly distributed over a large set of physical or virtual hosts, they need to communicate over well defined APIs and protocols. Evidently, this

This work was done when authors Sachin Sharma and Navdeep Uniyal were working with NEC Laboratories Europe, Heidelberg.

increases communication costs.

4) Integration testing overhead: Since the communication patterns and protocols are more complex (e.g. due to asynchronous communication), integration tests might become cumbersome. This introduces a significant testing overhead and calls for a strict automated testing.

Today, microservices are mostly considered in the context of traditional software systems, such as web and cloud applications. Recently, however, this approach has also raised significant attention in the field of Network Function Virtualization (NFV) [3]. In NFV, network functions, such as load balancer, intrusion detection, and network address translation, which usually run on specialized, proprietary hardware, are implemented as software components on top of custom-off-the-shelf cloud infrastructure [4].

In this paper, we define microservice as a decomposed component of a network function and define monothith as a virtual network function that is run as a single instance. We deploy network functions using the monolithic and microservice approach and compare the performance of both monoliths and microservice version of network functions using a simple analytic model and experimental results.

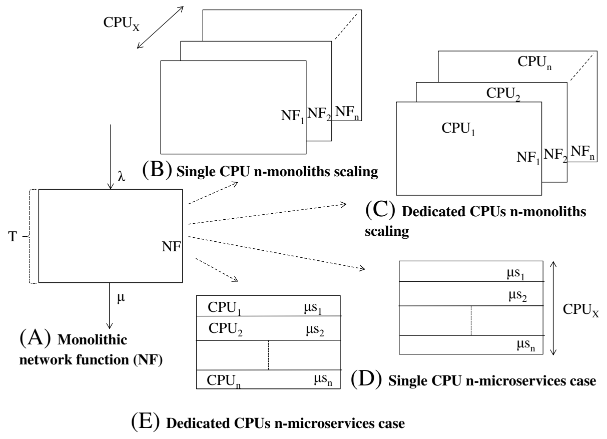
An M/M/1 queuing model is considered in order to estimate the performance using an analytic model. Without loss of generality while maintaining low complexity, we make use of an M/M/1 model and verify the suitability of this model through experimental results. We first provide the mathematical formulations for the performance (in form of average delay) of the monolithic and microservice version of network functions. We then calculate the number of instances of monoliths and microservices required to get maximum performance. We also experimentally calculate the performance using an open source network function, SNORT [5] and running monoliths and microservices as Docker containers on bare metal machines. The experimental results compare the performance of monolith and microservice approaches and verify the mathematical formulations. In addition, the experimental results verify the value of the number of instances for scaling or decomposition, given by the analytic model, with respect to experimental results.

Section II presents the mathematical formulations for the performance of network functions. Section III describes the experimental setup and Section IV provides results. The related work is provided in Section V and finally, Section VI concludes the paper by summarizing the main results.

II. Performance of Network Functions

In this section, we first derive the performance of the network function (NF) considered by using the analytic model and then calculate the performance when the NF is deployed using the monolithic and microservice approach implemented in an experimental testbed. Four different cases to deploy the NF are considered. These cases are: (1) Single CPU n-monoliths scaling, (2) Dedicated CPU n-monoliths scaling, (3) Single CPU n-microservices case and (4) Dedicated CPU n-microservices case. The former two cases are related to the monolithic approach and the later two are related to the microservice approach. We use queuing theory to model a network service, where an NF is modeled as a queuing system and its serving capacity is represented by the computational resources, i.e., CPU, used by the NF.

We use an M/M/1 queuing system to model an NF, as we assume that the NF is a single threaded function and run on one server. Our aim is to use a rather simple model but capable of providing us useful performance evaluation so that we find good parameters to take into account when scaling or decomposing a network function.



CPU$_X$
NF$_1$ NF$_2$ NF$_n$
(B) Single CPU n-monoliths scaling
CPU$_n$
CPU$_2$
CPU$_1$
NF$_1$ NF$_2$ NF$_n$
(C) Dedicated CPUs n-monoliths scaling
λ
T
NF
μ
(A) Monolithic network function (NF)
CPU$_1$ μs$_1$
CPU$_2$ μs$_2$
CPU$_n$ μs$_n$
(E) Dedicated CPUs n-microservices case
μs$_1$
μs$_2$
μs$_n$
CPU$_X$
(D) Single CPU n-microservices case

Fig. 1. Network Function (NF) Deployment

A. Considered Network function

Fig. 1A depicts the considered NF which has an incoming traffic rate λ and service rate μ. We assume that traffic arrives according to the Poisson process and serviced according to an exponential distribution. Let T be the average service time per packet. Then, μ is given by $\frac{1}{T}$. Here, T also includes the average idle time (I) per packet, which may be due to operations such as input-output operations and state reading time. Hence, $T = t + I$, where t is the service time if there is no idle time. Note that if idle time is greater than 0, the CPU allocated to the NF will not be utilized fully. Hence, CPU utilization will always be less than 100%.

When considering an M/M/1 queuing model [6], in order to calculate the performance of the NF, the average delay per packet, denoted as D, is given by:

$$D = \frac{1}{(\mu - \lambda)} = \frac{1}{\frac{1}{T} - \lambda} \quad (1)$$

B. Network Functions Deployment

In the following subsection, we describe the four cases in which an NF can be deployed.

1) Single CPU n-monoliths scaling: Fig. 1B describes the single CPU n-monoliths case in which an NF is scaled into n separate NF instances (NF_i, where $i = 1, 2, ..n$) and a single CPU (i.e., CPU_X) is allocated to process all the instances.

We assume that the incoming traffic is equally distributed among separate NF instances hence, the average traffic arrival rate per NF instance is be given by $\frac{\lambda}{n}$. Let the average service rate in this case be μ_{sin}^{Mono} and if we again assume an M/M/1 queuing model, the average delay per packet is given by:

$$D_1 = \frac{1}{(\mu_{sin}^{Mono} - \frac{\lambda}{n})} \tag{2}$$

For the case in which there is no idle time and also no overhead due to this scaled version of monoliths, the average delay of a packet will be equal to the delay D given in subsection II-A. Hence, in this case, μ_{sin}^{Mono} is derived as:

$$\mu_{sin}^{Mono} = \mu - \frac{(n-1)\lambda}{n} \tag{3}$$

For the case in which there is idle time or also overheads (e.g., due to traffic distribution among NF instances and context switching), the average delay per packet will be different than the delay given in subsection II-A. For simplicity, we assume that each NF_i has an equal average idle time of I per packet. Let O_{sin}^{Mon} denote the average time spent due to traffic distribution among different NF instances and context switching. Hence, when an NF is idle, another NF can utilize the CPU. Therefore, the average service time in this case is given by:

$$T_{sin}^{Mono} = \frac{1}{\mu - \frac{(n-1)\lambda}{n}} - \frac{1}{n}(\sum_{i=0}^{i=n-1} iI) + O_{sin}^{Mon} \tag{4}$$

and thereby, the average service rate (μ_{sin}^{Mon}) is given by the inverse of the average service time $1/T_{sin}^{Mono}$. We can now calculate the average delay per packet as:

$$D_1 = \frac{1}{(\mu_{sin}^{Mono} - \frac{\lambda}{n})} = \frac{1}{(\frac{1}{\frac{nT}{n-(n-1)\lambda T} - \frac{(n-1)I}{2} + O_{sin}^{Mon})} - \frac{\lambda}{n})} \tag{5}$$

2) Dedicated CPUs n-monoliths scaling: Fig. 1C describes the dedicated CPU n-monoliths case in which an NF is scaled into n separate NF instances and each NF gets a dedicated CPU ($CPU_1 .... \ or \ CPU_n$).

For calculation, we assume that traffic is equally distributed among different NFs and the traffic arrival rate per NF can be given by $\frac{\lambda}{n}$. Let the time spent in distributing traffic among different NFs is given by overhead O_{Ded}^{Mono}. The average service time per packet would be $T + O_{Ded}^{Mono}$ and thereby, the average service rate per NF (μ_{Ded}^{Mono}) is given by $\frac{1}{T+O_{Ded}^{Mono}}$. Note that T includes the idle time (I) in above calculations.

Again, if we consider the M/M/1 queuing model, the average delay per packet is given by:

$$D_2 = \frac{1}{(\mu_{Ded}^{Mono} - \frac{\lambda}{n})} = \frac{1}{(\frac{1}{T+O_{Ded}^{Mono}} - \frac{\lambda}{n})} \tag{6}$$

3) Single CPU n-microservices case: Fig. 1D describes the single CPU n-microservices case in which an NF is decomposed into n microservices ($\mu s_1 ... \mu s_n$) and all n microservices are processed by a single CPU (i.e., CPU_X).

Let the average service rate and time of a microservice be μ_{sing}^{micro} and T_{sing}^{micro} respectively. For a tandem queuing network [6] of microservices, the average delay of a packet is given by:

$$D_3 = \frac{n}{(\mu_{sin}^{micro} - \lambda)} = \frac{n}{(\frac{1}{T_{sin}^{micro}} - \lambda)} \tag{7}$$

For the case in which there is no idle time per micro-service and also no overhead due to the microservices or context switching, the average delay of a packet will be equal to the average delay (D) given in subsection II-A. Thereby, the value of μ_{sing}^{micro} in this case is given by $n\mu - (n-1)\lambda$.

For the case in which there is idle time per microservice or also overheads (e.g., due to context switching and microservices overheads), the average delay can be assumed to be different than the delay given in section II-A. For simplicity, we assume that each microservice waits for an equal amount of idle time and is given by $\frac{I}{n}$. Let O_{sin}^{micro} be the time spent in additional overheads (i.e., due to microservices and context switching). The service time is therefore given by:

$$T_{sin}^{micro} = \frac{1}{n\mu - (n-1)\lambda} - \frac{n-1}{n}\frac{I}{2} + O_{sin}^{micro} \tag{8}$$

By putting this value in Eq. 7, the average delay of a packet in the single CPU n-microservice case is given by:

$$\begin{aligned} D_3 &= \frac{n}{\frac{1}{\frac{1}{n\mu-(n-1)\lambda} - \frac{n-1}{n}\frac{I}{2} + O_{sin}^{micro}} - \lambda} \\ &= \frac{1}{\frac{1}{\frac{nT}{n-(n-1)\lambda T} - (n-1)\frac{I}{2} + nO_{sin}^{micro}} - \frac{\lambda}{n}} \end{aligned} \tag{9}$$

4) Dedicated CPUs n-microservices case: Fig. 1E describes the dedicated CPU n-microservices case where an NF is decomposed into n microservices ($\mu s_1 ... \mu s_n$) and a dedicated CPU (i.e., CPU_i) is allocated to each microservice.

As n CPUs are provided for n microservices, the average service time (T_{Ded}^{micro}) per microservice in this case should be reduced by n (i.e., it should be equal to $\frac{T}{n}$). However, there are some additional overheads (O_{Ded}^{micro}) due to microservices (e.g., header extraction in each microservice, bandwidth overheads etc.). Therefore, the average service time of a microservice (T_{Ded}^{micro}) will be given by $\frac{T}{n} + O_{Ded}^{micro}$.

For a tandem queuing network of microservices, the average delay will be given by:

$$D_4 = \frac{n}{\frac{n}{T+nO_{Ded}^{micro}} - \lambda} = \frac{1}{\frac{1}{T+nO_{Ded}^{micro}} - \frac{\lambda}{n}} \tag{10}$$

C. *Number of instances (n) of monoliths or microservices*

In this subsection, we calculate the value of n for all the above four cases so that we can achieve maximum or required performance (in form of delay). For the single CPU cases, we can achieve performance gain without increasing the number of resources (i.e., due to idle time we discussed above). In order to achieve the maximum performance in these cases, the average delay given by Eq. 5 and Eq. 9 should be minimum.

However, for the dedicated CPU cases, we need to dedicate additional resources in order to gain more performance. Hence, in these cases, the minimum number of resources could be selected, so that we can achieve the required performance (i.e., there is no need to achieve minimum performance in these cases).

In this paper, we use a graph based method to find the minimum or required value of the average delay. In this method, if all the values except n and D, D_1, D_2, D_3, D_4 in Eq. 5, Eq. 6, Eq. 9 and Eq. 10 are known, the graph is plotted where y axis is represented by the average delay and x-axis is represented by the number of instances of monoliths or microservices (n). The value of x is chosen from the graph that represents the minimum or required value of the average delay. This value gives the number of instances in which a network function should be scaled or decomposed in order to achieve minimum or required performance.

III. Experiment setup

In this section we will discuss the experiment setup used to find the performance of the monolithic and microservice approaches described in this paper.

In an NFV architecture, when deploying VNFs through a microservice approach, the VNFs can typically be function based or rule based. It would be easier to decompose a large and complex (in terms of functionality) VNFs (e.g., Virtual EPC and IMS) based on functions. However, if a VNF (e.g., forwarder) is small in terms of functionality, it would much easier to decompose such VNFs into rule based. There are several VNFs which are rule based such as forwarder, routers or IDS, and the rule based creation of microservices is a perfect example of simplicity and easy way to decompose network functions using a black box approach. In this paper, we perform the experimentation based on rule based decomposition. We used the Intrusion Detection System - SNORT - as a network function [5] and deployed it in four ways discussed in section II.

SNORT contains the number of rules combining signature, protocol and anomaly inspection methods to detect a malicious activity. We have added 200 such rules in a single SNORT NF, which is deployed on a single Docker container. Then, packets are forwarded according to the Poisson distribution. Each packet has 1000 bytes size and is matched against each rule in the SNORT and if a malicious activity is present, it is written in an alert file. Therefore, if malicious activities are reported by many rules, SNORT writes many alerts in the file for a single packet, taking long time in input/output operations. Hence, it increases the idle time of the SNORT NF. We randomly generated traffic producing different malicious activities and calculated the idle time by having 200 such rules in SNORT. We ran 50 different experiments (with different traffic characteristics) and calculated the average idle time. The average idle time (I) in our experiments was 0.248 ms.

In case of the single and dedicated CPU n-monoliths scaling case, n instances of the SNORT NF are deployed on a machine and traffic is equally divided among all the instances using a load balancer. Here, each instance and load balancer are run on a separate Docker container. In the single CPU n monolith case, all the n instances and load balancer are dedicated on a single CPU. However, in the dedicated CPU n-monoliths scaling case, each instance is given a dedicated CPU core.

In case of the single and dedicated CPU n-microservices case, the SNORT NF is decomposed into n sub-components, where each sub-component (i.e., microservice) contains $\frac{200}{n}$ rules and connected with another sub-component (i.e., microservice) in a tandem network. Here, each sub-component is run on a separate Docker container. In case of the single CPU n-microservice case, a single CPU core is allocated to forward traffic through all the sub-components (microservices). However, in case of the dedicated CPU n-microservice case, each microservice has given a dedicated CPU core.

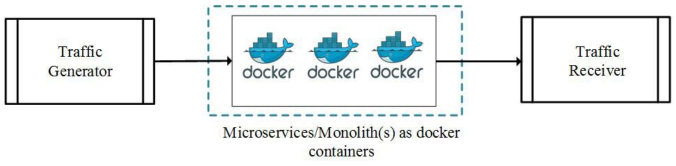


Fig. 2. Experiment setup

In our experiments, we varied n from 1 to 16 and followed the M/M/1 average model, where the incoming traffic rate is always kept less than the service rate. Our bare machines setup uses three machines and is shown in Fig. 2. We used a UDP traffic generator running on one machine directly connected to the other machine running SNORT, which inspect all the traffic going through it to the traffic receiver running on the third machine. We deployed our n-monolith and n-microservice cases in the second machine.

Each of the machine runs Ubuntu 16.04 on Intel Xeon Processor E5-2640 with 16 CPU Cores, 2.4GHz processor base frequency and a RAM of 125Gb. As shown in Fig. 2, traffic Generator is a simple UDP packet generator which generates packets following a Poisson distribution. For our experiments we have pinned the Docker containers to single or multiple CPU cores as per the case scenarios discussed in Section II. Each machine in the setup has a direct connection to each other using 10Gbps link. The traffic rate λ for the experiment is 0.97 per ms, considering no packet loss due to network link congestion. This choice of parameters is according to the M/M/1 queuing model with a stable queuing system where the incoming traffic rate (λ) should be less than service rate. The choice of the arrival rate is to exploit the dedicated processing capacity, yet having a stable queue by not exceeding the allocated service time, given by the processing capacity.

IV. Results

In this section, we first present the results obtained through the mathematical calculations obtained in Section II and then give the results obtained through experimentation performed on a SNORT NF running on Docker containers (as described in the previous section). We then compare experimental and mathematical results.

A. Analytical results

In order to calculate the average delay for n-monolithic and microservice case (where $n > 1$), we need to know the value of T, λ, I and overheads. We already know the value of λ and I through our emulation scenario (described in the previous section). The value of T is calculated from Eq. 1 by putting the value of λ and the average delay observed by running the experiment (mentioned in the previous section) on a single SNORT NF. So, we now calculate the value of overheads for different values of n. We then calculate the value of the average delay.

1) Overheads: Overheads in case of the single CPU n-monolithic case are traffic distribution overheads and context switching overheads. However, overheads in case of the single CPU n-microservice case are context switching overheads and microservices overheads (such as header translation at each microservices, communication overheads etc.).

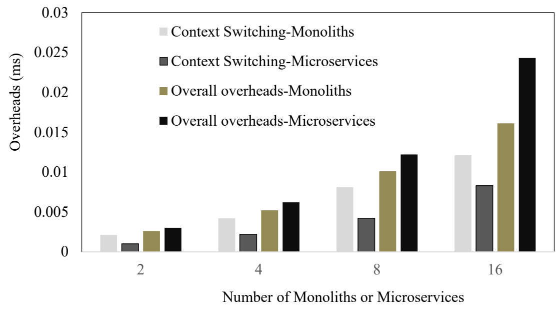


Fig. 3. Overheads for $n-$monoliths and $n-$microservices in case of a single CPU

The value of context switching overheads are taken by running the experiment of the single CPU n-monoliths and microservices cases (described in the previous section) on bare metal machines. We depict these overheads in Fig. 3. Fig. 3 shows that context switching overheads in case of the single CPU n-monolithic case are more than the overheads in case of single CPU n-microservices. These overheads are more in case the monolithic case because the monolithic function is a large application and may need to copy more number of state variables in context switching. On the other hand, as microservices are small functions, less number of state variables may be needed to be copied in context switching.

In our experiment results, we found that traffic distribution and microservice overheads are multiple of the number of instances of monoliths or microservices. This is because as the number of instances of monoliths or microservices increases, the traffic distribution and microservice overheads increase. In addition, the traffic distribution overhead is smaller than the microservice overhead. This is because in traffic distribution, the overheads are just due to redirecting traffic to a particular instances, however, in microservice overheads, the overheads are due to many factors such as header translation at each instance, bandwidth overheads etc. In our experiments, the traffic distribution overhead is about 0.0005 ms for two monoliths (i.e., for $n = 2$) and the microservice overhead is about 0.002 ms for two microservices (i.e., for $n = 2$). Hence, for a high value of n, these overheads are the overheads when n equal to 2 and multiplied with $n/2$. The overall overheads are then calculated by adding this value with the context switching overheads. The overall overheads are also shown in Fig. 3.

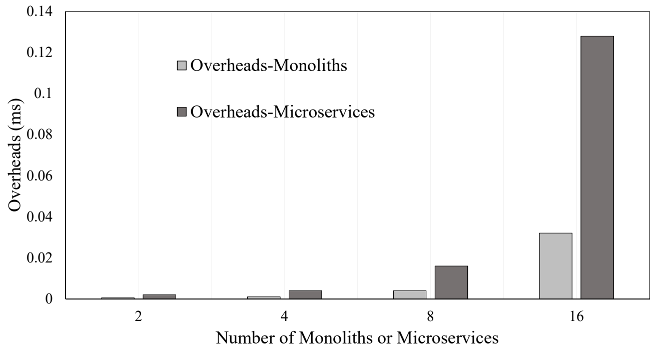


Fig. 4. Overheads for $n-$monoliths and $n-$microservices in case of the dedicated CPU cases

In case of the dedicated CPU $n-$ monolithic case, there are only traffic distribution overheads and in case of the dedicated CPU $n-$ microservice case, there are are only microservices overheads. These overheads are shown in Fig. 4.

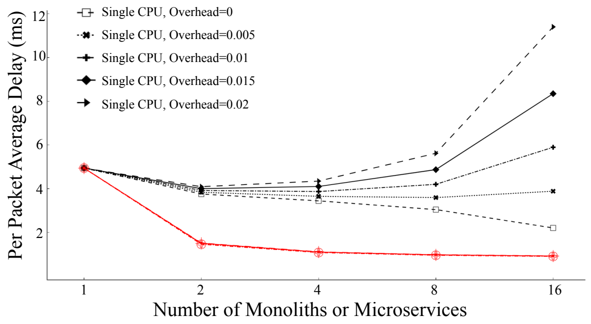


Fig. 5. Average delay mathematical results

2) Average Delay calculations: Fig. 5 shows the comparison of the average delay we calculated using the mathematical calculations presented in Section II. If overheads in case of the single CPU n-monoliths (O_{sin}^{Mono}) and microservice case

(nO_{sin}^{Micro}) are equal, the average delay given by Eq. 5 and Eq. 9 are equal. Therefore, we depict only a single line in Fig. 5 for both of these cases of the single CPU. In addition, if overheads in case of the dedicated CPU n-monoliths (O_{Ded}^{Mono}) and n-microservice cases (nO_{Ded}^{Micro}) are equal, the average delay given by Eq. 6 and Eq. 10 are equal. Hence, we again depict only a single line in Fig. 5 for these two cases of dedicated CPU.

In our mathematical results, we calculate the average delay for the overheads value 0, 0.005, 0.01, 0.015 and 0.02 ms using Eq. 5 and Eq. 10.

Fig. 5 shows that if the overhead is 0 ms, the average delay decreases in case of the single CPU when we increase n (i.e, the number of instances) from 1 to 16. This is because if the number of instances (monoliths or microservices) increases, the total idle time of the CPU (in case of a single CPU) will decrease. This is due to the fact that when one instance is idle, the other instance can take the CPU. For the case when the overhead value is 0.005 ms, the average delay decreases until we increase the number of instances to 8 and after that it starts increasing. In addition, if we keep the overhead as 0.01 ms, the average delay decreases until we increase the number of instances to 4. For the rest of the overhead values, the average delay decreases until we increase the number of instances to 2.

For the dedicated CPU case, as we have dedicated CPU per instance, we always get short average delay when we increase the number of instances from 1 to 16. However, this short delay is in the cost of dedicated CPUs.

Note that in real scenarios, the overhead value varies as we increase the number of instances (i.e., n). Therefore, in order to choose the best value of n we need to calculate the average delay at different overhead values (can be calculated using the description given in previous subsection) and have to choose the n which gives the shortest or acceptable average delay.

We calculated the average delay for the overhead values measured in the previous subsection for different values of n (i.e., from 1 to 16). We found that the single CPU n-monolithic and n-microservice case give the shortest average delay at $n = 4$. Therefore, in these cases, we should decompose or scale a network function into 4 instances to get the maximum performance. In case of the dedicated CPU n-monolithic and n-microservice case, we found the shortest average delay at $n = 16$. In fact, when $n = 4$, $n = 8$, and $n = 16$, the average delay is quiet close to each other.

In the next subsection, we will compare above best values of n with the best values obtained through experimental results.

B. Experimental Results

In this subsection, we report experimental results obtained through the emulations performed using the experiment scenario, described in section III. We emulated all the four cases we described before and depict the average delay and CPU utilization results.

The average delay is calculated as the mean value of the the time difference when a packet is received by the receiver and the packet is sent by the sender. The CPU utilization is calculated at the middle node where all the monoliths or microservices are deployed.

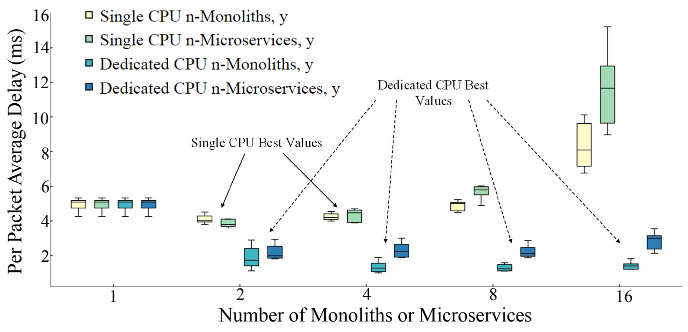


Fig. 6. Average delay experimental results

1) Average Delay Experimental Results: Fig. 6 shows the average delay in all the four cases. The results show that in the single CPU n-monolithic and n-microservice cases, there is low values of the average delay (shown by the single CPU best values in Fig. 6) when the SNORT NF is scaled or decomposed to 2 or 4 instances (i.e., at n=2 or 4). In addition, in the dedicated cases, there is low values of the average delay when the SNORT NF is scaled or decomposed to 2, 4, 8 and 16 instances (i.e., at n=2, 4, 8 and 16).

The above results are in line with mathematical results (see the previous subsection). The difference is only that the experimental results also show the best value of the average delay at n=2.

Note that in our experiments, the best value of n are same for the monolithic and microservice cases. However, as the overheads are different for monolithic and microservice cases for any n value, there may be cases in which the best value of n are different for monolithic and microservice cases.

The experimental results verify that performance of both scaled and microservice version of the SNORT NF in the dedicated CPU case is better than that of the single CPU case. This is obvious because additional CPUs are allocated in case of the dedicated CPU cases. We also see in Fig. 6 that the average delay in the dedicated CPU case for scaled monoliths is less than that of the delay observed in the decomposed microservices case for all values of n (i.e., from 2 to 16). This is because observed overheads (i.e., traffic distribution overheads) in case of scaled monoliths in the dedicated case are lesser than the microservice overheads in the dedicated case.

2) CPU Utilization: We argue that, for Virtualized Network Functions, the utilization of compute resources should be optimum and could be an important factor in decision making for decomposition and scaling. In this section we will compare the average CPU utilization in each case.

From Fig. 7, we can infer that the average CPU utilization in the single CPU cases increases on increasing the number

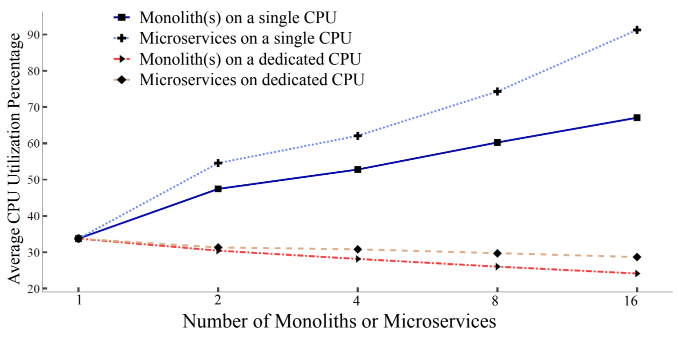


Fig. 7. Utilization per CPU

of instances. While scaling monoliths on a single CPU, the CPU utilization increases almost linearly from 34% to nearly 60% when the number of instances is increased from 1 to 16. On the other hand, the CPU utilization in the single CPU microservices case is higher than the single CPU monoliths case due to additional microservice overheads (discussed in Section 3). The CPU utilization increases up to 90% in the single CPU microservices case when the number of instances increases to 16.

In case a dedicated CPU is available for each instance, the average utilization per CPU tends to decrease from 34% to nearly 20% in case of the decomposed microservices as well as scaled monoliths cases. This is because the traffic load per CPU is decreased by dedicating a separate CPU to each instance.

C. Comparison between Mathematical and Experimental Results

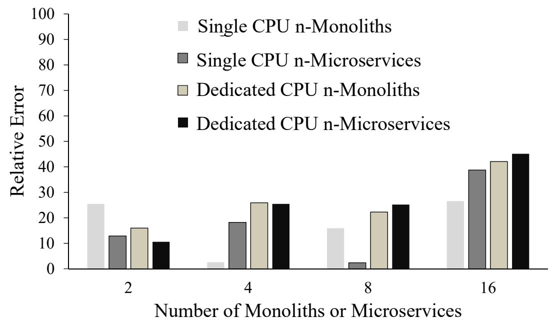


Fig. 8. Relative errors in experimental results. The mean value of the average delay from experimental results is taken to calculate the difference

Fig. 8 shows the relative error (denoted by $RError$) in the experimental results with respect to the mathematical results. The relative error is calculated as

$$RError = \frac{|Exp_{delay} - Math_{delay}|}{Exp_{delay}} \times 100 \qquad (11)$$

Here, Exp_{delay} is the mean value of the average delay from experimental results and $Math_{delay}$ is the average delay from the mathematical results.

Although the M/M/1 queuing model is rather simple and gives an approximation for the average delay, Fig. 8 shows that the relative error in experimental results is between 2%-48%. We notice that on average, the smaller the number of instances the smaller is the relative error with values confined within 25%. Note that finding the best analytic model is out of the scope of this work as we make use of a simple model just for gaining preliminary insights into the best configuration. Finding a more suitable model that generalizes a VNF and achieves even lower relative errors is left for future work where one possible way could be by using a queuing network instead of a single-queue single-server system. However, instead of analyzing the network function through a queuing network analyzer (QNA), as in [26], we could use more robust mathematical tools like the robust queuing network analyzer (RQNA) proposed in [28] which is shown to outperform the QNA in terms of relative errors for both heavy- and light-tail traffic distributions.

V. Related Work

There are many CAPex and OPex related advantages for operators to switch from Physical Network Functions to Virtual Network Functions. However, the major difference between the two approaches is based on the performance gains and loss. Most studies [4] [7] show that the choice should be subjective as in most cases physical devices perform better than the Virtual Network Functions. To enable the better migration, scaling and failure recovery of VNFs, there has been some tools and architectures proposed [8] which advocate decoupling of states from the network functions to achieve the aforementioned benefits.

Recently, there has been a surge in technologies like zero-copy [7], SR-IOV and DPDK [9] which allows VNFs to perform better on the commodity hardware with specialized NICs and dataplane technologies [10]. Also, there has been various tools and architectures proposed for better VNF placement, development and chaining to achieve the desired performance [12]. With these technologies, there is a high scope of improvement in performance of the VNFs running as VMs on the commodity hardware. Microservice architecture has gained popularity in recent time across all the software domains as a new and emerging software architecture. Microservice architecture is thought to be one of the major driver in enabling cloud-native devops [13], [14]. The existing software architectures like SOA (Service Oriented Architecture) is being compared with the microservices architecture to create a case for the large scale adoption [11]. Efforts are being made through studies to create an architecture which can be easily adopted by the industry [15] [11]. The introduction of containers creates an interesting case for experimenting the architecture in the networking domain. There has been platforms like openNetVM [24] developed, which advocates the use of containers for running VNFs combined with the technologies like DPDK. Recent study [25] shows that the performance of containers can be further improved, which provide a valid case to experiment with the microservice architecture for the VNF deployment.

Considering all the advancements and research done in the space of VNF performance, it would be interesting to weigh the benefits in terms of scaling, failure recovery and performance of monolith VNFs with that of micro-VNFs. In this regard, some research has been done to bring in the features of microservices like flexibility, scalability, failure recovery etc. in the scope of network functions [17]. The research in this direction is gaining momentum as it awaits major challenges in terms of management, decision making and architecture. On the other hand, with the development of 5G network and virtualization techniques, the microservice architecture can fit perfectly into the space and futher it also allows multiple vendors to come together with the expertise and work on the common goal of the virtualized network service development. Currently there are several Management and Orchestration systems for VNFs based on the standards defined by ETSI [23] which have been developed to manage and deploy the VNFs on the cloud infrastructure. With the inclusion of microservice architecture, there would arise a need to manage and orchestrate these micro network services which would be a cumbersome task to integrate such microservices and to define APIs between them [11].

The architecture proposed in [17] is targeted towards enabling microservices in the higher layer network services like virtual CDNs. Adopting microservices in such applications are proved to be useful not only in terms of better resource utilization but also shows improvement in the Quality of Service [17]. Packet Processing technologies like DPDK [9] and container technologies like docker [21] and lxc are a major technological force behind enabling microservice architectures and the work done in [17], [18], [19] and [20] are all based on these technologies. The authors in [26] have proposed a three-tier queuing network for modelling VNF chains. In [27], a similar model is used to represent a virtualized Evolved Packet Core control plane which the authors use to investigate the performance impact of scaling procedures and propose an automatic scaling algorithm. The proposed model is validated through simulation. However, in our case, we have targeted the single threaded VNF running on one server, hence using the M/M/1 queuing model, and the model is validated experimentally.

The Quality of Service improvements in Virtualized Network Microservices has been seen with the specific use cases like IP Multimedia Subsystem (IMS) [20], [17]. Most of the studies focus on the higher level network functions and the architecture of the network microservices. There has been some work done on creating a framework for the Service Function Chains of the micro network functions [19]. However, we could not find a study focusing on the measurements of the Quality of Service improvements on decomposing a monolith VNF into microservices. There has been some recent work on enabling the parallelism in network services for better resource utilization as well as the QoS improvements [22] which proves that the microservice model of VNFs could be useful in meeting strict QoS requirements.

VI. Conclusions

In this paper we focus on deploying virtual network functions using the monolithic and microservice approaches. We compared the performance of both the approaches using the mathematical and experimental results. We also calculated the number of instances in which a network function could be scaled or decomposed in both the approaches. Our results showed that scaling using the monolithic approach can result into more performance compared to the decomposition using the microservice approach (due to low overheads in the monolithic approach). In addition, we showed that by dedicating a CPU for each instance in the monolithic and microservice approach, we can get more performance. However, this performance gain is at the cost of additional CPUs. Therefore, we showed that the monolithic and microservice approach can utilize resources (e.g., CPU) fully by just increasing the number of instances and thereby, performance can also increase without investing more on resources (e.g, CPU). The results also showed the effectiveness of our method to calculate the number of instances in order to achieve maximum or required performance. In this paper, microservices are connected through a tandem network. However, in future work microservices connected in a mesh network will be considered.

Acknowledgment

This research was funded by the joint EU FP7 Cleansky Project, Contract No. 607584.

References

[1] J. Thnes, “Microservices,” IEEE Software, vol. 32(1), pp. 116-116, 2015.
[2] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, “Microservices: yesterday, today, and tomorrow”, arXiv:1606.04036v2, 2017
[3] Steven Van Rossem et. al., “NFV Service Dynamicity with a DevOps approach: Insights from a Use-case Realization”, IFIP/IEEE International Symposium on Integrated Network Management (IM), pp. 674-679, 2017
[4] B. Han, V. Gopalakrishnan, L. Ji and S. Lee, “Network function virtualization: Challenges and opportunities for innovations”, IEEE Communications Magazine, Vol. 53(2), pp. 90-97, 2015
[5] SNORT software and documentation: https://www.snort.org/
[6] Kishor S. Trivedi, “Probability and Statistics with Reliability, Queuing, and Computer Science Applications”, 2nd Edition, Wiley, 2001
[7] A. Panda, et. al., “NetBricks: Taking the V out of NFV”, 12th USENIX Conference on Operating Systems Design and Implementation (OSDI), pp. 203-216, 2016
[8] M. Kablan et. al., “Stateless Network Functions,” ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization, pp. 49-54, 2015
[9] Intel DPDK https://dpdk.org
[10] J. Hwang, K. K. Ramakrishnan and T. Wood, ”NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms,” in IEEE Transactions on Network and Service Management, vol. 12, no. 1, pp. 34-47, March 2015.
[11] Cerny, Tomas and Donahoo, Michael J. and Trnka, Michal, ”‘Contextual Understanding of Microservice Architecture: Current and Future Directions”’, SIGAPP Appl. Comput. Rev., pp. 29–45, 2018
[12] A. Bremler-Barr, Y. Harchol, and D. Hay, “OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions”, ACM SIGCOMM, pp. 511-524, 2016

[13] A. Balalaie, A. Heydarnoori and P. Jamshidi, ”Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture,” IEEE Software, pp. 42–52, 2016

[14] J. Kim et al., “Service provider DevOps for large scale modern network services,” 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), pp. 1391-1397, 2015

[15] P. D. Francesco and I. Malavolta and P. Lago, ”Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption,” IEEE International Conference on Software Architecture (ICSA), 2017

[16] M. P. Martinez et. al., ”‘Multivendor Deployment Integration for Future Mobile Networks”’, SOFSEM 2018: Theory and Practice of Computer Science, pp. 351–364, 2018

[17] Narjes Tahghigh Jahromi, Roch H. Glitho, Adel Larabi and Richard Brunner, ”‘An NFV and Microservice Based Architecture for On-the-fly Component Provisioning in Content Delivery Networks”’, 15th IEEE Annual Consumer Communications & Networking Conference (CCNC), pp. 1–7, 2018

[18] D. H. Luong, H. T. Thieu, A. Outtagarts and B. Mongazon-Cazavet, ”Telecom microservices orchestration,” 2017 IEEE Conference on Network Softwarization (NetSoft), Bologna, 2017, pp. 1-2.

[19] R. Kawashima and H. Matsuo, ”vNFChain: A VM-Dedicated Fast Service Chaining Framework for Micro-VNFs,” 2016 Fifth European Workshop on Software-Defined Networks (EWSDN), The Hague, 2016, pp. 13-18.

[20] A. Boubendir, E. Bertin and N. Simoni, ”A VNF-as-a-service design through micro-services disassembling the IMS,” 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), Paris, 2017, pp. 203-210.

[21] Dirk Merkel, ”‘Docker: lightweight Linux containers for consistent development and deployment,”’ Linux Journal 2014, 239, pages.

[22] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling Network Function Parallelism in NFV. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM ’17). ACM, pp. 43-56.

[23] ETSI NFV ISG. GS NFV-MAN 001 V1.1.1 network function virtualisation (NFV); management and orchestration, Dec. 2014.

[24] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K.K. Ramakrishnan, and T. Wood. 2016. OpenNetVM: A Platform for High Performance Network Service Chains. In Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization (HotMIddlebox ’16). ACM, New York, NY, USA, 26-31.

[25] Y. Zhao et. al., ” Performance of Container Networking Technologies”. In Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems (HotConNet ’17). ACM, New York, NY, USA, 1-6.

[26] J. Prados-Garzon et. al., ” Analytical modeling for Virtualized Network Functions” 2017 IEEE International Conference on Communications Workshops (ICC Workshops) 2017

[27] J. Prados-Garzon and A. Laghrissi and M. Bagaa and T. Taleb. ” A Queuing based dynamic Auto Scaling Algorithm for the LTE EPC Control Plane” 2018 IEEE Global Communications Conference (GLOBECOM 2018), Abu Dhabi, UAE. 2018.

[28] C. Bandi, D. Bertsimas and N. Youssef, (2015). ”Robust queueing theory”. Operations Research, 63(3), 676-700, INFORMS.