Abstract—Network function virtualization (NFV) allows service providers to implement network processing functionality in software using standard computing servers. As such, this approach precludes the need for deploying costly and proprietary hardware-based networking devices, i.e., black boxes, greatly reducing infrastructure and operational costs. Instead, operators can program and deploy desired network functions as software instances on commodity servers and further chain these functions to build end-user services, i.e., service function chaining (SFC). Although SFC mapping has received much focus in recent years, existing schemes assume that network functions are run in hypervisor-based virtual machines (VM). However VM-based strategies impose higher resource consumption at the server level and entail higher request processing/setup times. As a result, this paper analyzes container-based mapping approach which leverages VNF dependency relationships and incorporates resource constraints in the physical network. Performance results confirm that the proposed scheme gives notable gains versus VM hypervisor-based mapping, i.e., in terms of resource utilization, processing times and satisfied requests.

Keywords: Network function virtualization, service function chaining, hypervisor-based virtualization, virtual machines, container-based virtualization

I. INTRODUCTION

Network function virtualization (NFV) has emerged as a major solution technology in the ongoing push to softwarize data networks. In particular, NFV decouples network functions (NF) from their underlying hardware host platforms, allowing them to instead be implemented as software functions/instances on low-cost commodity servers [1]-[3]. One example here is splitting the baseband units (BBU) from the radio remote heads (RRH) in 5G cellular network [4]-[5]. In turn, these VNFs can be chained together to build different service function chains (SFCs) to meet a wide range of client service needs. Indeed, many NFV implementations today can readily achieve multi-gigabit packet processing speeds. As such, this paradigm is helping reduce operators’ needs to deploy and operate costly and proprietary hardware-based solutions, greatly improving provisioning flexibility, scalability, and reliability.

However, despite the benefits of NFV technology, a range of concerns still need to be resolved in order to ensure its wider deployment. Foremost, there is a need to develop proper virtual NF (VNF) placement and SFC mapping schemes to ensure client constraints (e.g., processor, memory, bandwidth, and delay) and also minimize operator resource consumption. Namely, each SFC request generally requires a given VNF sequence, where the interdependencies between atomic VNF are determined by client needs. These chains must be properly mapped onto underlying datacenter and networking and cloud substrate infrastructures. Clearly, inefficient SFC provisioning can result in increased resource consumption (lower revenues) and delays if requests have to traverse longer routes.

Now a wide range of solutions have been proposed to implement various SFC mapping objectives, a detailed review of which is also presented in some recent survey articles, see [6],[7]. For example, work in [8] formulates the SFC chaining as an optimization problem and proposes a mixed integer quadratic constrained program (MIQCP) solution for VNF mapping on to traditional VMs. Meanwhile, a backtracking scheme for VNF placement is also proposed in [9]. The authors in [10]-[12] also detail a dependent directional acyclic graph scheme that takes into account the priority dependence between VNF nodes and helps reduce bandwidth usage on substrate network links. Finally, the work in [13],[14] presents an integer linear program (ILP) scheme which maps SFC requests with multiple instances onto a single datacenter, i.e., in order to minimize network resource consumption. However, this approach may require traffic to traverse at longer paths to reach the node hosting the SFC, thereby yielding increased network bandwidth consumption (capacity exhaust).

However, for the most part, existing SFC mapping schemes generally assume that VNFs are mapped onto virtual machines (VM) running on physical servers, i.e., hypervisor-based mapping. However, hypervisor-based virtualization requires servers to install and run guest operating systems (OS) for each VM, leading to increased setup times and higher processor and memory utilization, i.e., both random access memory (RAM) and disk storage. In turn, these overheads will increase service latency and reduce the number of satisfied (carried) demands. It is here that the use of new container-based virtualization (containerization) solutions offers a new approach to streamline NFV provisioning. Specifically, containers have been proposed as an alternative to hypervisor-based VMs [15] and operate by creating abstractions for guest processes. Hence this approach can yield lower overheads versus VM-based strategies which require instantiating a new VM with a guest OS for each application (VNF instance).

In light of the above, this paper proposes the use of container-based SFC routing and mapping in NFV networks. Namely, a novel dependency-aware (DAS) scheme is proposed to improve resource utilization and lower cost. Note that dependency here implies an ordered list of required network services in incoming SFC requests. The proposed solution performs batch SFC mapping of such demands and also takes into account functional dependencies between VNFs and their interconnecting bandwidth requirements. Also, shortest path routing is used to reduce SFC delays. The framework assumes realistic operational settings with limited substrate resources.

Overall, this paper is organized as follows. First, Section II
overviews the advantages of container-based virtualization versus hypervisor-based VM technology. The new DAS mapping scheme is then detailed in Section III, followed by performance evaluation results in Section IV. Conclusions and future work directions are then presented in Section V.

II. CONTAINER-BASED VIRTUALIZATION

Hypervisor-based virtualization installs a guest operating systems (OS) for each type of VM requested by a client. Hence there is generally a high degree of duplication across similar VMs running on the same host, leading to reduced efficiency. For example, an application (VNF) with very low run-time requirements will still impose a large memory footprint. By contrast, container-based virtualization reduces such overheads by setting up isolated computing environments, i.e., termed as “containers”, which do not require dedicated resources (such as memory and disk space) prior to instantiation [16],[17]. Instead, containers use the same OS as the hosting node, i.e., no OS installation, and only consume resources when they actually run applications, e.g., such as VNFs. As expected, this approach gives much better memory resource utilization and also avoids duplication. These two approaches are summarized in Figure 1, where the container engine comprises of all applications running in the containers and their dependencies, e.g., libraries.

![Figure 1: Hypervisor versus container-based virtualization](image)

Now the open source Docker platform is a Linux-based containerization solution which provides standard runtime image formats [8]. This solution has seen wide deployment and usage and offers a promising framework for building more streamlined, lightweight NFV infrastructures. Specifically, Docker based containers can be used to customize and rapidly instantiate VNF services and dynamically re-size them based upon varying client demands and traffic flows. Furthermore, Docker container images can also be built and shared in a layered manner, i.e., once a new image is created on another node, this image can be further re-created using an existing local image. By contrast, each VM requires the whole image to be copied multiple times on every node—likely from an external node or location—giving much higher setup and instantiation times (versus containers). Accordingly, a novel container-based VNF orchestration and mapping scheme is developed and evaluated for realistic SFC demands, i.e., in terms of carried load (satisfied demands), resource utilization, and instantiation times.

III. CONTAINER-BASED MAPPING

The requisite notations for the network and client SFC demand model are presented first, followed by a detailed algorithmic description of the proposed DAS scheme, see also Figure 2 and Table I.

A. Substrate Network Topology

Generally, a physical substrate infrastructure is comprised of a set of datacenters interconnected by high-bandwidth links. This setup can be modeled as a graph, \( G=(D,E) \), where \( D \) is the set of datacenter nodes (on which VNFs are mapped) and \( E \) is the set of network links. Additionally, as per real-world operational settings, available substrate resources are generally limited/bounded. Namely, without loss of generality, the total memory at a datacenter is bounded by \( M_d \), the total processor capacity at a datacenter node is bounded by \( C_d \), and the available link bandwidth on a substrate link is bounded by \( B_d \).

B. Client Request (Demand) Model

Meanwhile the SFC demand model is used to represent all client requests. Foremost, the total (batch) set of demands is given by \( R \). Here, an individual request \( r \in R \) is further specified by the 4-tuple \( r=s, d, F_r, b_r \), where \( s \) and \( d \) are the source and destination nodes, respectively \((s,d) \in D)\), \( F_r \) is the set of desired VNF types, and \( b_r \) is the desired link bandwidth to interconnect the VNFs in the service chain. In particular, \( F_r=\{f_1, f_2, \ldots, f_t\} \), where \( t \in T \) is the set of ordered VNFs for request \( r \), and \( T \) is the set of all possible VNF types. Furthermore, each VNF datacenter needs specific resources, most notably, processor and memory, denoted by \( C_{f_t}, C_{f_t} \in \mathbb{Z}^+ \), and \( M_{f_t}, M_{f_t} \in \mathbb{Z}^+ \), respectively.

C. Dependency-Aware Scheme (DAS) Heuristic

The overall pseudocode for the proposed DAS algorithm is detailed in Figure 3. The scheme implements container-based dependency-aware SFC mapping. Consider the details.
The DAS algorithm assumes an input batch set of SFC requests, $R$, and proceeds to route them in an iterative manner based on first-in-first-out order, i.e., outer loop over index $i$, Figure 3. Akin to other SFC schemes [7],[12], the VNF mapping and connection routing phases here are done in a sequential manner. Namely, for each request $r$, the first VNF in the chain is processed and mapped to a specific datacenter, i.e., inner loop over index $j$, Figure 3. Here the algorithm determines the “best” datacenter to host the VNF by first identifying the source node, i.e., inner loop over index $k$, Figure 3. Specifically, this node must have sufficient resources in terms of memory ($M_d$) and processor ($C_d$), as well as egress link bandwidth on the edge $B_e$. Note that the datacenters hosting the mapped VNFs can be either be direct physical neighbors of the source node or non-physical neighbors, i.e., logical neighbors connected via VNF link connections over routers/switches (at intermediate datacenters). Furthermore, all VNF link connections are computed using the Dijkstra’s shortest path algorithm with link weights set to unity, i.e., minimum hop routing for resource minimization. The scheme also checks to make sure that all underlying physical links carrying the connection, $e \in E$, have sufficient available bandwidth, i.e., $B_e \geq B_r$ (Figure 3). Therefore, request $r$ is only successful if all its VNFs are successfully reserved along with their interconnecting bandwidth links.

Now once a SFC request, $r$, has been mapped, an operator has to reserve the required resources at its mapped datacenters and network links. Namely the available, i.e., free, resource levels at associated datacenters and links must be updated according to the SFC resource requirements. Hence in order to track these changes, the DAS algorithm use a temporary working copy of the infrastructure graph, $G'$. Namely, this graph is initialized to $G$ when starting to process each request, $r$, and the resources are iteratively deducted (reserved) from it for each successful VNF mapping and connection route setup. Hence if the request is successful, then the actual infrastructure topology graph, $G$, is updated (replaced) by the temporary working graph, $G'$. Otherwise, the request is failed and the temporary graph $G'$ is discarded and the algorithm proceeds to the next request.

Given input network substrate topology $G=(D,E)$ and set of incoming batch requests, $R$

/* Loop and process all requests */
for $i = 1$ to $|R|$

Make temporary graph copy $G'$, i.e., $G \rightarrow G'$

Remove $i$-th request, $r=<s_i, d_i, F_r, b_r>$

/* Loop and map all VNFs in request $r$ */
for $j = 1$ to $|F_r|$

/* Find proper node for function $f_i$ */
for $k=1$ to $|D|

Map $f_j$ to node $d_i$

if (($C_{f_i} < C_d$ & ($M_{f_i} < M_d$) & ($b_r < B_e$))

/* Reserve free resources */

$C_d = C_d - C_{f_i}$
$M_d = M_d - M_{f_i}$
$B_e = B_e - B_{f_i}$

Add node $d_i$ to graph $G'$

Add edge from $s_i$ to $d_i$ to graph $G'$

if (all VNFs in request $r$ mapped)

Request successful, copy $G' \rightarrow G$

else

Request $r$ failed
end if

end /*k loop*/

end /* j loop */

Clear $G'$

end /* i loop */

Figure 3: Pseudocode description for DAS algorithm

Some sample SFC mappings are shown in Figure 2 for two sample requests, $r_1$ and $r_2$. Namely, $r_1=<d_1,d_3,\{f_1, f_2, f_3\},50>$ and requests 3 SFC VNFs originating at datacenter $d_1$ and terminating at datacenter $d_3$. The respective VNFs for this request ($f_1$, $f_2$, $f_3$) must be mapped in the given (dependency) order as they may correspond to specific network applications, e.g., such as firewalls, intrusion detection/prevention engines, address translation devices, proxy servers, etc. Hence as per Figure 2, $f_1$ is mapped to the source datacenter, $f_2$ is mapped to a non-neighboring datacenter $d_3$, and $f_3$ is mapped to the destination datacenter. A similar mapping for the second request $r_2=<d_6,d_{10},\{f_1, f_2, f_3, f_4\},60>$ is also shown.

Overall, the DAS algorithm ensures that all node and link capacity constraints are met throughout the SFC provisioning process. Namely, the total sum of CPU and memory capacities required across all requests should not exceed the available physical resources at any node $d \in D$, i.e.,

$$\sum_{r \in R} \sum_{t \in T} n_{f_i}^{r_j} C_{f_i} \leq C_d, \forall d \in D,$$

(1)
Similarly, the sum of bandwidth required by all virtual path (connections) between the VNF should not exceed the available physical link bandwidth, i.e.,

$$\sum_{r \in R} m_r b_e \leq B_e, \forall e \in E$$

where $m_r$ denotes the number of virtual links in request $r$ that is mapped to a link $e$ (Table I).

$$\sum \sum n_{ij}^r M_{f_j} \leq M_d, \forall d \in D. \quad (2)$$

The hypervisor and container-based mappings are now tested for the DAS scheme with varying input batch request sizes and analyzed for a range of performance metrics, i.e., including resource utilization, processing times, number of satisfied requests, etc. The findings are now presented.

**IV. PERFORMANCE EVALUATION**

The DAS scheme is tested to compare the performance of VM hypervisor-and container-based VNF hosting mechanisms. The test infrastructure uses the NSF network topology shown in Figure 4, consisting of $|D|=16$ datacenters and $|E|=24$ links. Furthermore, each datacenter node has $C_r=50$ processors and $M_r=64$ gigabytes of storage capacity, and each link has $B_e=100$ units of bandwidth capacity. Meanwhile, each request $r \in R$ specifies 3 VNFs randomly from a predefined set, i.e., $|T|=5$. Here each VNF is essentially mapped to a datacenter (using either a VM or container) and has random uniform resource requirements, i.e., 1-5 processors ($C_f$), 1-5 gigabytes of memory ($M_f$), and 1-10 units of bandwidth ($B_f$).

Furthermore realistic empirical measurements are also used to gauge the relative overhead requirements of the hypervisor and container schemes. In particular, the *iperf* benchmark tool is used here by setting up an *iperf* server in the Amazon EC2 environment. Local *iperf* clients are then run in both VM and Docker container environments to note their associated overheads. In particular, the clients run the Ubuntu 12.04 LTS 4 operating system and/or the Docker version 0.7.2 on an Intel Core i7 processor with 47 gigabytes of memory. Resultant measurements indicate that an average Docker image requires approximately 10 megabytes of memory to run a single *iperf* function/instance versus 16 megabytes for a VM image. Since each request has 3 VNFs, 48 megabytes of memory are required if datacenters use VM hypervisors and 30 Mbytes if they use containers, i.e., 63% less memory usage.

**A. Number of Satisfied Functions and Requests**

The number of successful requests provides a measure of the load-carrying capacity of a particular scheme, i.e., a gauge of carrier revenue. Accordingly, Figures 5 and 6 plot the number of satisfied SFC demands as well as the number of satisfied VNFs, respectively, for varying input request batch sizes, $|R|$. As expected, both the VM and container-based mappings give full success for lower loads, i.e., under 20 SFC requests. However, the performance of VM-based mapping quickly levels off at medium-to-high loads and does not exceed 30 SFC requests (Figure 5). By contrast, the container-based approach gives full mapping success for almost 40 input SFC demands and then drops to sub-linear growth and levels off at about 56 requests, i.e., almost twice the carried load of the VM approach. Clearly, the increased memory usage of the VM approach is a key factor here, particularly when considering VNFs mapped across multiple (3) dispersed datacenter nodes (Figure 5). Similar relative performances are also observed in terms of the number of satisfied VNFs, where the container-based approach effectively gives less datacenter memory fragmentation, see Figure 6.

**B. Resource Consumption**

Next, the efficiency of the hypervisor and container-based SFC mapping strategies is gauged by measuring resource utilization. Within the context of NFV, this value is treated as a weighted sum of the total memory consumption and processor usage across all nodes (by the mapped VNFs), i.e., system resources, as well as the total link bandwidth usage of all inter-VNF connections. Hence the total resources consumption (usage) by a single SFC is given by the following:
where \( a_1, a_2 \) and \( a_3 \) are the respective weighting factors. Note that this study assumes equal weighting between all resource types, i.e., \( a_1=a_2=a_3=1 \).

Along these lines, Figure 8 shows the resultant aggregate processing times for the two strategies with varying numbers of satisfied SFC requests. These results show a clear advantage with the container-based solution, which gives roughly linear growth in processing time. By contrast, the VM-based method exhibits much higher super-linear processing times, i.e., as each request requires full OS setup/instantiation at each datacenter (and likely involves more complex host-level memory management at higher loads as well). For example, the VM approach takes 250 ms (1.6 seconds) to process 30 requests (90 requests) as compared to only 34 ms (116 ms) with the container-based approach. As per the above, dynamic services requiring rapid SFC setup (or even modification) will clearly benefit from the containerization approach.
particular, a novel dependency-aware service function chain mapping heuristic scheme is introduced to improve resource efficiency at the physical substrate level, i.e., processor, memory, and bandwidth consumption. The hypervisor and container-based strategies are then compared within this context using network simulation. Overall results show much better resource efficiency and carried network loads, as well as reduced request processing times with container-based function mapping. As such, service providers can achieve notable cost reduction by using this strategy, i.e., versus renting virtual machines from infrastructure providers. In addition, they can also reduce their response times, i.e., increase service velocity, and better respond to dynamic client demands. Along these lines, future studies can also look at the use of container-based methods for rapid failover switching to improve the reliability of service function chains.

REFERENCES


