ViTeNA: An SDN-Based Virtual Network Embedding Algorithm for Multi-Tenant Data Centers

Daniel Caixinha
INESC-ID Lisboa
Instituto Superior Técnico
Universidade de Lisboa, Portugal
daniel.caixinha@tecnico.ulisboa.pt

Pradeeban Kathiravelu
INESC-ID Lisboa
Instituto Superior Técnico
Universidade de Lisboa, Portugal
pradeeban.kathiravelu@tecnico.ulisboa.pt

Luís Veiga
INESC-ID Lisboa
Instituto Superior Técnico
Universidade de Lisboa, Portugal
luis.veiga@inesc-id.pt

***Abstract*—Data centers offer computational resources with various levels of guaranteed performance to the tenants, through differentiated Service Level Agreements (SLA). Typically, data center and cloud providers do not extend these guarantees to the networking layer. Since communication is carried over a network shared by all the tenants, the performance that a tenant application can achieve is unpredictable and depends on factors often beyond the tenant's control.**

We propose *ViTeNA*, a Software-Defined Networking-based virtual network embedding algorithm and approach that aims to solve these problems by using the abstraction of virtual networks. Virtual Tenant Networks (VTN) are isolated from each other, offering virtual networks to each of the tenants, with bandwidth guarantees. Deployed along with a scalable OpenFlow controller, *ViTeNA* allocates virtual tenant networks in a work-conservative system. Preliminary evaluations on data centers with tree and fat-tree topologies indicate that *ViTeNA* achieves both high consolidation on the allocation of virtual networks and high data center resource utilization.

I. Introduction

The creation of data centers allowed global access to huge computational resources, previously only available to large companies or governments. By renting the desired computational power, small companies (or even an individual) avoid large capital expenses. In current data center environments, a client can ask for a computational instance of various sizes, and the service provider assures levels of guaranteed performance (through an SLA) for that computational instance. This guarantee is possible due to the huge evolution of virtualization technologies. Nowadays, a hypervisor can control the behavior of the virtual machines (VMs) it hosts, ensuring that a VM cannot use more CPU than what it was requested (except when the hypervisor allows). In this way, tenants are not harmed by the misbehavior of the other tenants.

This simplicity of computational resources on demand has generated a lot of interest around the world. However, there are still a lot to improve in this area - considerably, the lack of network accounting in the renting of resources. Cloud providers do not offer network performance guarantees to their tenants. In fact, a tenant's compute instances or VMs communicate over the network shared by all tenants. Thus, the network performance that a certain VM can get depends on several factors including those outside the tenant's control, such as the network load on a given moment or the placement of that VM in the network. This is further aggravated by the oversubscribed nature of a data center network.

Lack of guarantees in a shared communication medium leads to unpredictable application performance often at tenants' cost. Machine virtualization has a considerable impact on network performance, where virtualized machines often present abnormally large packet delay variations, up to hundred times larger than the propagation delay between the considered two hosts. Moreover, TCP and UDP throughput can fluctuate rapidly (in the order of tens of milliseconds) between 1 Gb/s and zero, which shows that applications will have a very unpredictable performance [1]. Tenant applications in the cloud and data centers are often data intensive, such as video processing, scientific computing, or distributed data analysis. Hence a fluctuation in tenant virtual bandwidth allocation may severely degrade the performance achieved by an application. With intermittent network performance, MapReduce[2] applications will experience harsh issues when the data to be shuffled amongst mappers and reducers is quite large.

Software-Defined Networking (SDN) [3] is an abstraction that decouples the control plane from the data plane consisting of forwarding hardware such as switches and routers. Hence, the control mechanism can be extracted from the network elements and logically centralized in the SDN controller. The controller creates an abstraction of the underlying network, and thereby provides an interface to the higher-layer applications. SDN controllers can be leveraged to create a Virtual Tenant Network (VTN) on top of the data plane. Each of the tenants is given isolation guarantees at network level, with an illusion of a dedicated virtual network. VTN can be leveraged in data center and cloud networks to ensure that SLAs are met with Quality of Service (QoS) guarantees from the bottom-most level.

Virtual network embedding [4] aims to completely vir-

978-1-5090-3216-7/16/$31.00 ©2016 IEEE

tualize the network, providing performance isolation among tenants at the network level. So, virtual network embedding consists in the mapping of virtual networks (consisting of virtual nodes and links) onto the substrate network (consisting of physical nodes and links). Virtual network embedding is considered the main challenge in the implementation of network virtualization [5]. In the data center context, network virtualization is advantageous as it allows clients to define the desired network topology, which will be allocated within the infrastructure exactly as defined by the client. This enables seamless migrations of current or legacy networks to a data center environment, since the client defines the topology and the required guarantees (such as CPU or bandwidth between machines), which will be enforced by the embedding algorithm by placing the virtual network only where the resources are sufficient.

In this paper, we describe the design, implementation, and evaluation of *ViTeNA* virtual network embedding approach. *ViTeNA* leverages the global view of the network offered by SDN controllers for virtual network allocation for tenants in a data center network. This allows tenants to express their requirements in terms of bandwidth, which is then enforced through virtual networks. *ViTeNA* is designed as a scalable solution for data center environments. It further achieves high consolidation within the placement of virtual networks, and high utilization of the data center's physical resources - servers and network.

II. Background and Related Work

OpenFlow [3] is a southbound protocol and core enabler of SDN. Many SDN controllers such as OpenDaylight [6], ONOS [7], and Floodlight [8] have developed OpenFlow implementation in high-level languages. Supported by the Linux Foundation and many big players in the networking industry, OpenDaylight and ONOS have grown to be large and complex SDN projects, with various sub-projects and use cases. Floodlight remains fairly simple as a compact Java-based open source SDN controller.

OpenDaylight VTN offers virtual network provisioning, flow and QoS control over virtual network, and virtual network monitoring. However, OpenDaylight VTN focuses on network virtualization function; not on virtual network allocation guarantees. $MicroTE$ [9] uses OpenFlow as a framework to have centralized control over the data center and make routing decisions based on predictions of the traffic matrix. Current traffic engineering techniques do not apply well to a data center, as they are too slow to react to micro-congestions, and the reaction time must be under 2 seconds to be effective [9]. Hence, $MicroTE$ creates a hierarchical structure to make traffic measurements scalable with the data center size, having a centralized controller.

Unpredictability of network performance in data center environments is damaging to both tenants and service providers, since the tenant applications suffer from the unpredictability and the service provider can incur in avoidable revenue losses [10]. Oktopus [10] facilitates predictable networks, offering network guarantees to the tenants. SecondNet [11] utilizes a central unit that receives virtual network requests, and runs the embedding algorithm to process and allocate the virtual tenant network requests. But, unlike Oktopus, the routes calculated by the central node do not translate into routing rules to the switches, because in SecondNet the physical machines keep information about the routes of each VM it owns. Silo enables co-existence of tenants in a competitive environment for resources, though with a trade-off of reduced network utilization [12]. EyeQ offers a distributed transport layer for network performance isolation in multi-tenant environments [13].

Seawall [14] tackles the problem of fair bandwidth sharing and network performance isolation in data centers. It overcomes the lack of performance isolation at the network level by assigning weights to each entity (such as a VM, or a process inside a VM). Thus, Seawall devices a solution where the share of bandwidth obtained by the entity in each network link is proportional to its weight. Gatekeeper [15] shares some design ideas and goals with Seawall. However, it provides minimum bandwidth guarantees, by using Open vSwitch [16] in each server to control all the VMs within a server. Each VM has a virtual network interface card (vNIC), that connects to the Open vSwitch. A minimum receive bandwidth guarantee as well as a minimum send bandwidth guarantee is assigned to each VM. Minimum bandwidth guarantees are achieved using an admission control mechanism that limits the sum of guarantees to the available physical link bandwidth.

Heuristics based virtual network embedding algorithms have various specific objectives including low execution time. Survivable networks [17] aim to make the virtual network embedding with fast failure recovery times. Despite the differences in goals, all these algorithms focus on delivering bandwidth guarantees in a data center network. Server locality of the same virtual network embedding request is exploited and leveraged by these algorithms, to reduce the search space for a solution, thus reducing the algorithm execution time.

As virtual network embedding is an NP-hard problem, node mapping and link mapping phases are separated in typical heuristic-based algorithms. If the request is smaller than the VM capacity of the largest available server, upon a virtual network request, the system first attempts to do the node mapping by trying to accommodate everything inside a single server. If not, they try the servers on the same rack, followed by the adjacent racks, and so on, to minimize the distance. The choice of the first server rack to analyze varies from work to work, but it is either random or in a round-robin fashion. The link mapping phase only starts if the virtual network request completes the node mapping phase. If it does not, it can be put into a queue to be processed later or the request discarded to alert the client there are no resources available for their request.

This is a $greedy$ approach, since it picks the locally optimal choice at each branch in the road (i.e. it chooses the best solution that complies with the virtual network constraints). With this type of algorithms, the virtual networks also benefit from the reduced number of hops in the substrate network, which will yield low latency (as low as possible, but with no guarantees) between the VMs communicating in the virtual network. Some works [11], [18], [10], [19], [20] follow this approach. Path splitting (i.e. bifurcated traffic) and migration of VMs can also be considered for small data centers [19], as this does not significantly reduce the search space in a reasonable time for data centers beyond a few hundreds of nodes. Chowdhury et al [4] formulate the virtual network embedding as a mixed integer problem and solve it as a linear program by relaxing the integer constraints. Extending the later advances in networking and SDN, *ViTeNA* aims to improve virtual network embedding, offering a virtual tenant network for multi-tenant data centers.

III. *ViTeNA* Approach for Network Allocation

We will describe *ViTeNA* virtual tenant network allocation in this section. Although *ViTeNA* can be deployed on any network research topologies, we will limit our focus on the traditional tree-like topologies (tree and fat-tree) since they are still the most used data center topologies [21].

Figure 1 depicts a sample deployment of data center network in *ViTeNA* approach. The OpenFlow controller is the core of *ViTeNA*, as it is responsible for running the virtual network embedding algorithm to map the requests on the substrate network, and programs the switches to deploy the requested virtual networks. Each switch acts merely as a packet forwarder, per the rules dictated by the controller. The controller has a connection to every switch in the network, represented by the thinner dotted lines that are from the control plane in Figure 1. Though the data flows and control flows are differentiated using different lines in the diagram, the control plane does not necessarily need dedicated connections; it can use the same physical links of the data plane.

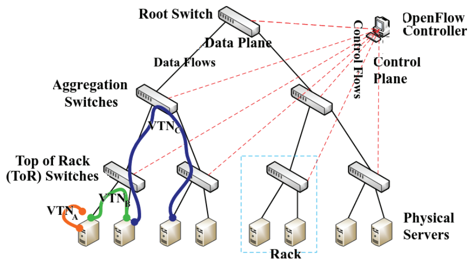

Fig. 1. Deployment landscape of *ViTeNA* with a tree topology.

ViTeNA network embedding algorithm tries to allocate the requests on the smallest available subset of the substrate network, like the other existing virtual network embedding approaches. It thus aims to maximize the proximity of VMs belonging to the same tenant, which results in minimizing the number of hops between those VMs. This is advantageous for two reasons: i) with less hops, the delay is generally reduced; and ii) keeping the VMs close (e.g. in the same rack) relieves the bandwidth usage in the upper links of the tree, where the bandwidth is scarcer in a data center [21]. With this approach, we will be able to accept more virtual network requests, since the core links will not be so likely to become the bottleneck of the data center.

Figure 1 shows the placement of three virtual networks in tree topology with depth equal to 3 and fanout equal to 2. The virtual network of tenant A (VTN_A) represents the best possible case where all the VMs of the virtual network can be mapped on the same physical server. In this case, there is no usage of the network (which saves bandwidth for future requests), and the bottleneck of the virtual network is only the speed within the server. In the virtual network of tenant B (VTN_B) the request could not be mapped to a single server, and hence it uses another server belonging to the same rack to accommodate the entire request. The virtual network of tenant C (VTN_C) shows a case where the virtual network could not be mapped in the same rack, and must use a server on the adjacent rack. As we can see, the bandwidth of the links on the top of the tree is only used in the worst cases (i.e. when the request is large or the data center is operating near saturation).

Besides the network embedding algorithm that ensures that the network can provide the bandwidth guarantees requested by each tenant, *ViTeNA* exploits the centralized information in the controller to provide fair bandwidth sharing (i.e. work-conservation) among tenants (non-existent in network embedding systems) and incremental consolidation of virtual network requests. Fair bandwidth sharing is achieved by instructing every switch used by a virtual network (which is determined by the embedding algorithm) to create a new queue for that virtual network. The queues in OpenFlow are used to provide QoS guarantees (in this case, bandwidth). Incremental consolidation of virtual network requests is enforced in the network embedding algorithm itself, choosing the location of a virtual network according to a best-fit heuristic on the VM placement. To do this, the algorithm leverages the current state of the network and the physical servers available to the OpenFlow controller.

A. *Software Architecture*

Figure 2 depicts each component of *ViTeNA* as well as the most important interactions between them. Mininet [22] open source network emulator has been used to emulate the data center networks, as it considered the de facto standard of OpenFlow emulators [23]. *ViTeNA* could be ported to a physical data center with a few or no changes in code, from the current Mininet-based emulations. The only exception is in the Linux Process, which in a real scenario would be running a hypervisor to manage the VMs inside that host. In this paper, this is simplified to an operating system managing processes, where each process will simulate a VM. We assume all the physical servers have equal CPU, so that in a virtual network request a tenant asks for a percentage of a CPU (instead of a CPU with a certain frequency).

Information Flow: The tenant expresses its demands in a virtual network request in an XML configurations file. This includes defining the number of VMs required (and the percentage of CPU of each one), as well as the bandwidth required between the VMs that will be connected (expressed in

MBit/s). This request is fed into the virtual network embedding algorithm that is running in the OpenFlow controller. Upon receiving the request, the embedding algorithm contacts the network information manager to get the current state of the network. Based on this state, the algorithm determines (if the request is accepted) where this virtual network will be allocated. As all the requests are processed by the controller, this updated view of the network involves zero control messages over the network (both to switches and hosts), since the controller just should update this information when it processes a new virtual network request.

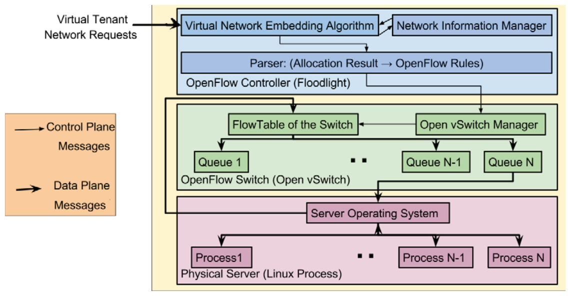
Virtual Tenant Network Requests
Virtual Network Embedding Algorithm
Network Information Manager
Parser: (Allocation Result → OpenFlow Rules)
OpenFlow Controller (Floodlight)
Control Plane Messages
Data Plane Messages
FlowTable of the Switch
Open vSwitch Manager
Queue 1
Queue N-1
Queue N
OpenFlow Switch (Open vSwitch)
Server Operating System
Process1
Process N-1
Process N
Physical Server (Linux Process)

Fig. 2. Software architecture of *ViTeNA*.

The controller then translates the output of the algorithm (i.e. the affected switches and hosts) to OpenFlow rule(s) to reprogram the switch(es). Upon receiving this message, the Open vSwitch manager creates a new queue for this virtual network, with the assured bandwidth present in the received message. It further installs a new rule in the switch's flow table to forward packets from a certain VM to the newly created queue. Hence, there will be a queue for each pair of linked VMs in a virtual network. Thus, a VM can use more bandwidth than its minimum when the link is not throttled. Moreover, the sharing of bandwidth between queues is made fairly according to the minimum bandwidth a queue has (a queue with a higher minimum bandwidth will use more spare bandwidth). Thus, the resource usage is maximized, since the tenants share unused bandwidth fairly, but at the same time get their minimum bandwidth guarantee when the network is saturated.

The VMs are represented by and implemented as Linux processes. To simulate tenant workloads, each process runs a traffic generator. Upon the necessary configurations, each process (i.e. VM) can communicate with other processes on the same virtual network, using either the operating system (in case the processes are on the same server), or contacting its adjacent switch, which will use the flow table to check to which queue it should forward this solicitation (in case the processes are on different servers). In this paper, we will only focus on communication inter-server as the intra-server communication is a responsibility of the hypervisor.

B. Virtual Network Allocation

As the core procedure of *ViTeNA*, algorithm 1 aims to guarantee that every VM pair in the virtual network request gets at least the requested bandwidth; and, to consolidate the virtual networks on the least amount of physical resources possible. Instead of making a consolidation algorithm that runs periodically, *ViTeNA* performs the consolidation incrementally at each request, merging the network embedding and the consolidation algorithms. Hence it avoids heavy migrations of VMs between servers, which would stop temporarily the work of those VMs and possibly generate a lot of control and data traffic. Periodic execution of the algorithm would diminish the controller performance. *ViTeNA* thus avoids performance degradation while processing the virtual network requests.

Algorithm 1 *ViTeNA* Virtual Network Embedding

```
 1: procedure isNetworkRequestAccepted($VNR$)
 2:     totalVMLoad ← getVMLoadFromVNR(VNR)
 3:     highestCPUAvailable ← getMostFreeCPU()
 4:     if (totalVMLoad < highestCPUAvailable) then
 5:         appropriateList ← findAppropriateList(totalVMLoad)
 6:         for all (server in appropriateList) do
 7:             serverCPUAvailable ← getCPUAvailable(server)
 8:             if (totalVMLoad < serverCPUAvailable) then
 9:                 allocVirtualNetwork(VNR, server)
10:                 Return True                              ▷ Request is accepted
11:     else
12:         server ← getMostFreeServer()
13:         firstServer ← server
14:         sortedVNR ← sortVNRByBWDemands(VNR)
15:         [preAllocatedVMs, remainingVMs] ← splitRequest(sortedVNR, highestCPUAvailable)
16:         VMsAlreadyAllocated ← (preAllocatedVMs, server)
17:         preAlloc(preAllocatedVMs, server)
18:         while (True) do
19:             server ← getNextServer(server)
20:             if server.$isFirstServer()$ then
21:                 cancelAllPreAllocs()
22:                 Return False                             ▷ Request is not accepted
23:             CPUAvailable ← getCPUAvailable(server)
24:             [preAllocatedVMs, remainingVMs] ← splitRequest(sortedVNR, CPUAvailable)
25:             BWDemands ← calcSumOfDemands(sortedVNR, VMsAlreadyAllocated, preAllocatedVMs, server)
26:             linksResidualBW ← calcResidualBW(VMsAlreadyAllocated, server)
27:             if (BWDemands > linksResidualBW) then
28:                 clearLastSplitRequest()
29:                 continue
30:             else
31:                 preAlloc(preAllocatedVMs, server, BWDemands)
32:                 VMsAlreadyAllocated ← VMsAlreadyAllocated + (preAllocatedVMs, server)
33:                 if (remainingVMs.$isEmpty()$) then
34:                     allocAllPreAllocs()
35:                     Return True                          ▷ Request is accepted
```

The algorithm is divided into two base cases depending on the virtual network request: i) when it fits in one physical server, and ii) when it needs to be spread across multiple servers. This is the first check made, comparing the total CPU load requested with the highest CPU available at the moment (line 4). If the request fits in one server, we want to find the server with the least free CPU that fits the request (i.e. best-fit). Once the server is found, we allocate this request on it (which includes updating the network state with this new request). To find the best-fit server, we first find a sub set as the search space (line 5). We will maintain 10 sets: the first keeps the servers with 0 to 10 % of CPU free, the second the servers with 10 to 20 % of CPU free, and so on. This reduces the search space for the appropriate physical server, which in a large data center environment can reduce the run time of the algorithm considerably.

If the request does not fit in one server, we sort the request by decreasing bandwidth (line 14), and allocate as much VMs as possible in the freest server on the entire data center. In this way, the most consuming demands are on the same physical server, which significantly relieves the load on the network (and thus we can accept more virtual network requests). After pre-allocating (since the request can be rejected) what is possible on the freest server, we try to allocate the rest on adjacent servers. In line 19, the getNextServer(*server*) function returns the servers on the same rack of *server*, then the servers on an adjacent rack, and so on. Next we check if we already tried on every server of the data center (line 20), and reject the request if so, cancelling all the pre-allocations.

In the next lines (23-26), we see if the server we are checking has connection(s) with sufficient bandwidth (as defined in the request) to the other server(s) already pre-allocated in previous iteration(s). If it does not have enough bandwidth, we try on the next server (lines 27-29). If it does, we pre-allocate the VMs mapped onto this server. Finally, we check if the set remaining VMs to be allocated is empty (lines 33-35): if it is, we allocate all the pre-allocations (i.e. commit) and return *True*; if it is not, we move on to the next server to allocate the remaining ones.

We expect to have a high consolidation (and consequently low fragmentation) of VMs within the servers, since the algorithm either allocates a whole server (if the request does not fit one server), or finds the best-fit server (if the request fits in one server).

IV. Implementation

Floodlight 1.1 has been leveraged as the core SDN platform for *ViTeNA* implementation. Mininet 2.2.1 and Open vSwitch 2.3.1 were used in emulating data centers with various topologies. The Floodlight controller is based on an event driven architecture. Hence, for a module to receive an OpenFlow PacketIn message, the module must subscribe for this type of messages. When the controller receives an OpenFlow message from a switch it will dispatch that message to all modules that have subscribed for that specific message type.

If the controller receives multiple messages from one or more switches, these messages are enqueued for dispatching because the controller only supports dispatching one message at a time. This means that the performance of the controller is dependent on the time it takes to process a message in each individual module, as the processing time of a single message is equal to the sum of all the processing time done in each individual module.

To add a new module to the Floodlight controller, a new Java package must be created directly in the code base of the controller. *Floodlight default properties file* defines which modules are launched when Floodlight is started. For a custom-made module to start up, its path should be added to the properties file. When the controller starts, it will start the modules, and set references between them per the inter-module dependencies. These dependencies are defined by implementing the appropriate methods (according to which dependencies one wants to set) of the `IFloodlightProviderService` interface. We modified the *link discovery* module of Floodlight. *ViTeNA* also includes 3 additional modules into Floodlight - i) Multipath Routing, ii) Queue Pusher, and iii) Virtual Network Allocator modules. We will now dive into the details of each of these four modules of *ViTeNA*.

A. Link Discovery Module

The controller needs to place the VMs of a request in physical servers that have links with enough bandwidth to accommodate what is requested in the XML file. Thus, each link should know how much free bandwidth it possesses. However, in Floodlight's original implementation, the Link object does not have such information. Hence, it was necessary to extend the link discovery module, so that each link would be aware of its spare bandwidth. This consisted in adding a new field to the Link class, and defining the appropriate *getters* and *setters* to configure this field.

The link's bandwidth isn't set automatically as one would expect (i.e. through the LLDP packets this Module sends). Mininet emulates all the virtual links with a bandwidth of 10 Gb/s, even if we configure it to have a lower bandwidth. We had to work around this, and we did it by initializing all link bandwidth when the controller starts, according to what is defined in a start-up XML file. To ease the process of running our controller, this XML file is automatically generated when the Mininet topology is created, according to the parameters in the Mininet script.

B. Multipath Routing Module

Floodlight's original routing module provides a Java API to use. Its $getRoute()$ method calculates the route between two endpoints by applying the Dijkstra's algorithm [24] to the graph that contains the network topology. This means that it always gives the shortest path between two nodes in the graph. Since we want to maximize the number of allocated virtual networks by the controller, we want to test every possible route between two endpoints. The shortest path between two endpoints may not have enough bandwidth to accommodate a certain request, and a longer path may have that required bandwidth. By choosing to use more than just the shortest path can turn many otherwise rejected requests into accepted ones.

As of our implementation, the multipath routing module registers itself as a receiver for events of the type `topologyChanged`. By receiving these events, the module builds a graph, adding and removing links or hosts as the events dictate. This graph represents the network topology. Having this graph, one must only apply a search algorithm on top of it to find the paths between two nodes. We use a Depth-First Search algorithm [25] to compute all possible paths between a pair of nodes.

C. Queue Pusher Module

ViTeNA queue pusher module is responsible for providing an API to create queues in Open vSwitches. Queues in Open vSwitch are created using the OVSDB protocol [26]. The queue pusher module creates queues, with only the Assured Rate configured (called `min-rate` in the OVSDB command) and no Ceil Rate (or `max-rate`) configured.

The queue pusher module uses the `ovs-vsctl` utility that comes pre-installed with the Open vSwitch, to create a new QoS entry and a new Queue below that QoS entry for each Queue the controller wants to create. When creating Queues, each one gets assigned an identifier, which should be unique per switch. The traffic will be directed to the Queue by matching this identifier, since the *enqueue* action receives it as argument.

D. Virtual Network Allocator Module

The allocation algorithm runs in the virtual network allocator module. When the request fits in one server, the sum of CPUs requested can be accommodated in the same physical server. In this case, this module should merely update the *global* data structures, with the information from the *local* data structures, that was gathered from the XML file.

When the request does not fit in one server, the request should be divided among two or more physical servers, and this module will start by sorting the links of the virtual network request in descending order (since the tenant can provide the XML file in any order). Then, it will allocate as much VMs as possible in the server that has the most CPU available. After that, it will try to allocate the remaining VMs in the neighbors of this server. It will start from the server adjacent to this one, and it will continue this logic until there are no remaining VMs.

In each iteration of going to the neighbor of the server with the freest CPU, this module uses the multipath routing module and the link discovery module. Every time it advances to an adjacent server, it uses the multipath routing module to get all paths between this server and the ones that already have allocated VMs. Upon getting these paths, the virtual network allocator module will use the link discovery module to check each link of each path, to make sure that those links have enough bandwidth to provide the guarantees required by this tenant's request.

Once all the VMs have a physical server assigned (assuming a request where this happens), the module knows this request is going to be accepted. So, it is necessary to translate this request's results into real network rules. This module uses the queue pusher to create the required queues on each OpenFlow switch and the necessary OpenFlow rules, returning `True` following that.

V. Evaluation

ViTeNA was evaluated on a computer with Intel® Quad-Core i7 870 @ 2.93 GHz processor, 12 GB DDR3 @ 1333 MHz RAM, and 450 GB Serial ATA @ 7200 rpm hard disk, on Ubuntu 14.04.3 LTS (Linux Kernel 3.13.0). The controller processes virtual network requests. We stop an experiment when the controller returns `False` to an allocation, as that means it cannot allocate any more virtual networks. Each experiment is run a thousand times to get the mean and variation of the results. To generate our dataset, we produced virtual network requests (XML files) where: a VM asks for a CPU that is generated randomly (using a uniform distribution) between 0.1 and 5%; the connections between VMs are also randomly generated (with a uniform distribution as well) between 0 and 10 Mbit/s. For each size of the virtual network requests (i.e. number of VMs in it), which we defined as going from 2 to 40, we generated 10000 virtual network requests.

A tree topology (depth = 3; fanout = 5) with 125 servers, which entails 31 switches and 155 links, and a fat-tree topology (factor k = 32, i.e. switches consist of 32 ports) with 128 servers, which entails 160 switches and 384 links, were emulated with Mininet for the evaluations.

A. Scalability to Data Center Environments

First, we evaluated the scalability of *ViTeNA* in data center scale. To this end, we measured the time it takes to process each virtual network request using the method `currentTimeMillis` from the controller Java API. Figure 3 depicts the results obtained with tree topology.

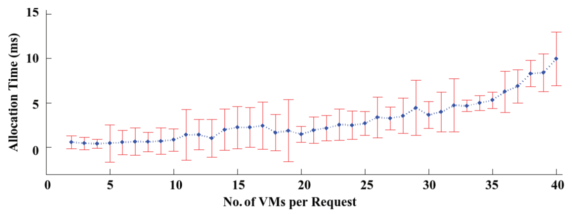

Fig. 3. Allocation for a virtual network request in tree topology.

Processing time less than 5 ms was observed for up to about 25 VMs in a request. SecondNet [11] achieves 10 ms in requests with 10 VMs, which is twice the processing time in requests with less than half of the VMs. *ViTeNA* consumes about 10 ms to process requests with 40 VMs. It should be noted that a request with 40 VMs is almost one third of the number of physical servers in the network. Even in these conditions, processing time did not grow abruptly, indicating the high scalability of *ViTeNA*.

Figure 4 depicts the allocation time for fat-tree. It can be noticed that the processing time using fat-tree topology is higher than the one observed for tree topology. Fat-tree peaks at around 15 ms, which is 5 ms more than that is observed with tree topology.

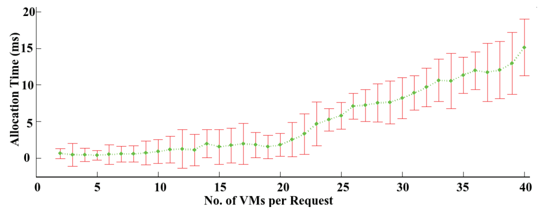

Fig. 4. Allocation for a virtual network request in fat-tree topology.

As fat-tree has a lot more links and switches than the tree topology, there are more paths between any two endpoints. Thus, the higher processing time can be explained by the extra work controller had to perform by checking more routes. Hence the extra processing time was not wasted, as with the fat-tree we received a total of 15688 accepted requests, *versus* 15055 with the tree topology.

B. High Consolidation

As *ViTeNA* takes server locality into account, allocating the VMs of a virtual network as close as possible, next we measured how consolidated the virtual networks are. A low number of hops leads to a low latency in the communication between the VMs of a virtual network. This metric is calculated by counting the numbers of physical servers in a virtual network allocation. The results of using a tree topology are depicted in Figure 5.

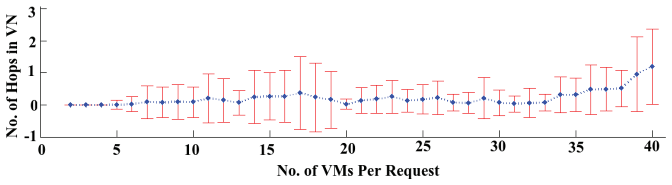

Fig. 5. Avg. number of hops in a virtual network in tree topology.

Figure 5 indicates that *ViTeNA* offers high consolidation of the virtual networks, since the average is almost always near zero. It starts out equal to zero up to 5 VMs per request, then the average is almost the same but the variance increases, meaning most of the virtual networks do not have any hop, but some do. It keeps this behavior along the line, with the average increasing less than linearly. On 40 VMs per request we get an average number of hops close to 1. 40 VMs in a request is significant, because the evaluated data center network has 125 servers, and still the virtual networks only needed one hop on average.

The average number of hops using a fat-tree topology is shown in Figure 6. Fat-tree exhibits a similar pattern to the one observed in tree topology case, and high consolidation is achieved in fat-tree topology as well. As we process more requests with this topology, we notice a higher average number of hops as well as the standard deviation. The extra requests we get with fat-tree topology are processed when the data center is near saturation (since we are stopping on the first rejected request in the tree topology, and with fat-tree topology we go further). Since the data center is near saturation, each request will more likely need a high number of hops, which explains the increase compared to the tree topology.

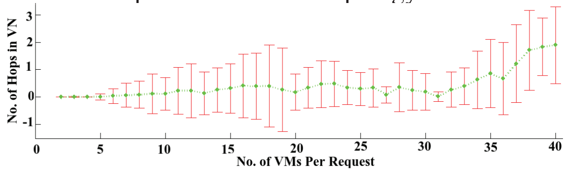

Fig. 6. Avg. number of hops in a virtual network in fat-tree topology.

C. High Resource Utilization

Resource utilization within the data center with *ViTeNA* was measured, by calculating the server and link utilization. Resource utilization is computed by calculating the utilization of the resources when the experiment stops, and dividing it by the full capacity. The resource utilization results using a tree topology are portrayed in Figure 7. As *ViTeNA* does a best-fit placement of the VMs within the servers, it was observed that most of the time *ViTeNA* achieves high server utilization. As *ViTeNA* already does an incremental consolidation, it does not need or have a consolidation algorithm running periodically in the controller.

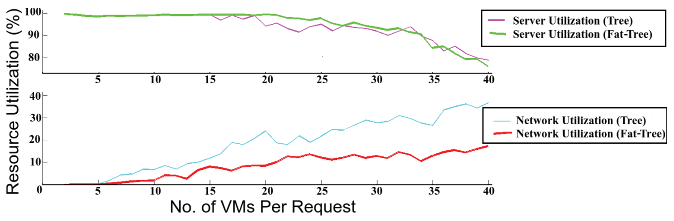

Fig. 7. Resource utilization in tree and fat-tree topologies.

The server utilization starts to drop when the number of VMs in a virtual network is around 20. This happens because with a request of this size (and larger), some of the VMs must be placed on different servers, which causes fragmentation of the CPU utilization by a server. This results in a lower server utilization. Obviously, the network utilization starts to grow when this happens, since we have more and more utilized links across the network. Moreover, the low network utilization is a result of getting all the servers full before we get some virtual networks that require link usage, as this is just a matter of which resource is exhausted first.

The results appear similar for both tree and fat-tree topologies. However, fat-tree allows the server utilization to remain high for longer (up to 25 VMs per request), where as in the tree topology it starts to drop at 15 VMs per request. This can be explained by higher number of requests served by fat-tree topology, since more requests allow to decrease the fragmentation in CPU usage across servers, which in turn causes the server utilization to increase. Nevertheless, fat-tree network utilization is significantly lower compared to the tree topology. This is due to the much higher number of links that fat-tree topology has (more than double of that of tree topology), which all add up to the denominator of this metric and causes it to decrease significantly.

D. Bandwidth Guarantees in a Work-conservative System

To evaluate the bandwidth guarantees in a work-conservative system, we created a topology with 8 hosts, where 4 hosts generate traffic towards the other 4. Each host generates 50 Mbit/s, and we simulate a 100 Mbit/s link using a queue with the `max-rate` parameter set to this value. We used the `iperf` tool to generate traffic with a constant bit-rate, each one generating traffic with a rate of 50 Mbit/s. We used a constant bit-rate to make sure that changes we see in the rate on the receiver side are due to the network changes and not from changes in the sending side. Figure 8 shows the graph generated accordingly.

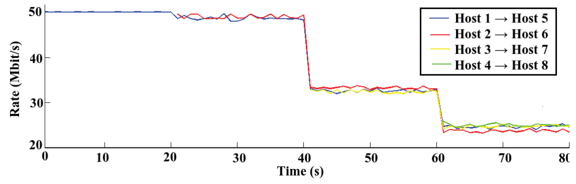

Fig. 8. Throughput achieved by multiple hosts sharing a single link.

In the beginning ($t = 0s$), only *Host 1* is generating traffic with the bit-rate stated above, and with *Host 5* as destination. As *Host 1* is the only one doing so, it gets the full bandwidth that it requested - 100 Mbit/s. This goes on until $t = 20s$, when *Host 2* starts to generate traffic towards *Host 6*, and there are two hosts generating traffic at 50 Mbit/s, which is the *link capacity* (i.e. the `max-rate` allowed by *Switch 2*). Each host on the right gets practically the bandwidth that its correspondent is generating; but we can already see the action of *Switch 2*, portrayed by the irregularities in both lines.

When we reach $t = 40s$, *Host 3* starts to generate traffic to *Host 7*. Now, the sum of the traffic generated by the hosts exceeds the *link capacity*', which will cause dropped packets. However, each host gets more than its assured bandwidth (25 Mbit/s), as the three of them divide the link, each one getting about 33 Mbit/s. Finally, at $t = 60s$, *Host 4* begins to generate packets towards *Host 8*. Now, the 100 Mbit/s link is divided by the four hosts, and each one gets about its assured bandwidth. Thus *ViTeNA* offers bandwidth guarantees to the tenants, while they utilize more resources when the resources are abundant.

VI. Conclusions and Future Work

Current data centers lack network performance guarantees, since all tenants interchangeably share the network. This makes the performance of tenant applications unpredictable, since it depends on factors outside of its control. This unpredictability severely prevents a wider cloud adoption, as many cloud use cases require network performance and isolation guarantees. These problems are solved using the abstraction of virtual networks. Virtual Tenant Networks (VTN) are isolated from each other, providing performance guarantees. Virtual network embedding algorithms attempt to solve this NP-hard problem of an efficient tenant network resource allocation.

ViTeNA is a virtual network embedding approach that extends an OpenFlow SDN controller to allocate virtual networks with bandwidth guarantees in a work-conservative system, providing a QoS-aware multi-tenanted data center. Evaluation on tree and fat tree topologies confirm that *ViTeNA* offers, 1) low execution time, 2) high consolidation on the allocation of virtual networks, and 3) high resource utilization of the data center resources. As a future work, *ViTeNA* should be extended for reliability and isolation guarantees, in addition to efficient network allocation.

Acknowledgements: This work was supported by national funds through Fundação para a Ciência e a Tecnologia with reference UID/CEC/50021/2013 and a PhD grant offered by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC).

References

[1] G. Wang and T. E. Ng, "The impact of virtualization on network performance of amazon ec2 data center," in *INFOCOM, 2010 Proceedings IEEE*. IEEE, 2010, pp. 1–9.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[4] N. M. K. Chowdhury, M. R. Rahman, and R. Boutaba, "Virtual network embedding with coordinated node and link mapping," in *INFOCOM 2009, IEEE*. IEEE, 2009, pp. 783–791.

[5] A. Fischer, J. F. Botero, M. Till Beck, H. De Meer, and X. Hesselbach, "Virtual network embedding: A survey," *Communications Surveys & Tutorials, IEEE*, vol. 15, no. 4, pp. 1888–1906, 2013.

[6] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture," in *2014 IEEE 15th International Symposium on*. IEEE, 2014, pp. 1–6.

[7] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "Onos: towards an open, distributed sdn os," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 1–6.

[8] R. Wallner and R. Cannistra, "An sdn approach: quality of service using big switch's floodlight open-source controller," *Proceedings of the Asia-Pacific Advanced Network*, vol. 35, pp. 14–19, 2013.

[9] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: fine grained traffic engineering for data centers," in *Proceedings of the Seventh Conference on emerging Networking EXperiments and Technologies*. ACM, 2011, p. 8.

[10] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 242–253.

[11] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "Secondnet: a data center network virtualization architecture with bandwidth guarantees," in *Proceedings of the 6th international conference on Emerging networking experiments and technologies*. ACM, 2010, p. 15.

[12] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: predictable message latency in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 435–448, 2015.

[13] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim, "Eyeq: practical network performance isolation at the edge," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 297–311.

[14] A. Shieh, S. Kandula, A. Greenberg, and C. Kim, "Seawall: performance isolation for cloud datacenter networks," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. USENIX Association, 2010, pp. 1–1.

[15] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes, "Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks." in *WIOV*, 2011.

[16] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending networking into the virtualization layer." in *Hotnets*, 2009.

[17] M. R. Rahman, I. Aib, and R. Boutaba, "Survivable virtual network embedding," in *NETWORKING 2010*. Springer, 2010, pp. 40–52.

[18] T. Benson, A. Akella, A. Shaikh, and S. Sahu, "Cloudnaas: a cloud networking platform for enterprise applications," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 8.

[19] M. Yu, Y. Yi, J. Rexford, and M. Chiang, "Rethinking virtual network embedding: substrate support for path splitting and migration," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 17–29, 2008.

[20] M. F. Zhani, Q. Zhang, G. Simona, and R. Boutaba, "Vdc planner: Dynamic migration-aware virtual data center embedding for clouds," in *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*. IEEE, 2013, pp. 18–25.

[21] K. Bilal, S. U. Khan, J. Kolodziej, L. Zhang, K. Hayat, S. A. Madani, N. Min-Allah, L. Wang, and D. Chen, "A comparative study of data center network architectures." in *ECMS*, 2012, pp. 526–532.

[22] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.

[23] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 493–512, 2014.

[24] S. Skiena, "Dijkstra's algorithm," *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, pp. 225–227, 1990.

[25] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[26] D. Palma, J. Goncalves, B. Sousa, L. Cordeiro, P. Simoes, S. Sharma, and D. Staessens, "The queuepusher: Enabling queue management in openflow," in *Software Defined Networks (EWSDN), 2014 Third European Workshop on*. IEEE, 2014, pp. 125–126.