SDN-Based Framework for Infrastructure as a Service Clouds

Heli Amarasinghe, Ahmed Karmouch
School of Electrical Engineering and Computer Science, University of Ottawa, Canada
hamar040@uottawa.ca, akarmouc@uottawa.ca

Abstract— Infrastructure as a service (IaaS) has attracted significant attention from cloud research communities. While compute and storage resource management has been developed and studied to a greater extent, the use of network resource management in the context of IaaS is still in its early stages. There is a need for a comprehensive virtualization framework capable of providing users with low-level network and compute resource control while improving underlying resource utilization. We propose a Software-Defined Networking (SDN) IaaS framework that explicitly integrates network virtualization, including computing and storage, into a cloud platform. Our proposed framework abstracts data-center compute and network resources into a virtualized pool of resources, links them to logically compose virtual networks, and performs automated configurations to serve IaaS requests from users. We designed and constructed an SDN test-bed to verify the operation of the proposed framework. We successfully demonstrated our system's ability to meet our design objectives: (i) providing high degree of control over connectivity and Quality of Service (QoS), (ii) fully automated service delivery, (iii) fast and low overhead resource provisioning. We evaluated the feasibility and scalability of the proposed framework using statistics gathered from its deployment.

Index Terms — Cloud Computing, Composition, Infrastructure-as-a-Service, Network virtualization, Resource Management, Software-defined-networking

I. INTRODUCTION

Cloud computing has introduced a new era of utility computing by promising flexible resource provisioning, scalable deployment and reduced capital and operational expenditure. Cloud service model introduces three layers of services: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) [1]. Within these, IaaS has become popular due to recent advances in computing and networking infrastructure virtualization.

Virtualization allows infrastructure providers to partition as well as combine physical resources to construct logically isolated custom resource units. Such, custom resource allocation permitted by virtualization can potentially improve data-center resource utilization. Also, virtualization is generally used to conceal complexities of implementation in the underlying layers. Thus, in modern data-centers, hypervisor-based virtualization technologies are widely adopted to manage compute resources. In addition, the success of compute resource virtualization is due to the operational simplicity provided by well-defined abstraction mechanisms [2] [3]. But computer networks have evolved by introducing new protocols to deliver new service, thereby increasing design and operational complexity. A number of researchers have pointed out that lack of flexibility, dynamicity and controllability of interconnecting virtual networks impede the ability of business to rely on cloud services. Network virtualization therefore demands powerful abstraction capabilities to continue advancements while keeping the operational complexities hidden. Existing network virtualization technologies alone, such as virtual local area network (VLAN) and virtual private network (VPN), do not provide adequate solutions.

Software-Defined Networking (SDN), with its de facto standard application OpenFlow [4], is seen as a promising technology capable of addressing these needs. SDN serves to configure/route traffic flows dynamically and flexibly to satisfy user application requirements. OpenFlow's success is largely due to its wide adoption by the industry and network equipment manufactures. OpenFlow is capable of abstracting underlying hardware and allows fine-grained control over network traffic paths. Hence, particularly in an IaaS cloud environment, OpenFlow can deliver highly customizable virtualized network infrastructure. We envision that a comprehensive engineering solution that automates on-demand provisioning of virtual infrastructure with flow-level control over network traffic will attract more businesses towards cloud services. A framework with such characteristics is capable of not only enhancing user application performance but also improving resource utilization for IaaS providers. However, existing commercial and open-source IaaS platforms such as Amazone AWS [5], Openstack [6] and Cloudstack [7] do not provide automated IaaS provisioning capabilities with flow-level control over network traffic to users. Hence, there is a need for an IaaS provisioning framework to deliver highly customizable virtual networks along with compute resources.

Motivated by these perceptions, we introduce an SDN-based infrastructure management framework for IaaS cloud platforms, as the main contribution of this work. The proposed framework has four broad design considerations. First, we improve user experience by providing a high degree of control over network connectivity and QoS parameters. Second, we

2159-6190/16 $31.00 © 2016 IEEE
DOI 10.1109/CLOUD.2016.106

reduce the management complexities by automating service delivery and explicitly integrating network resources, along with computing and storage resources, into the cloud platform. Third, we reduce management overheads and focus on fast service delivery. Fourth, we improve resource utilization through composed resource allocation.

To meet these objectives, we have designed a data-center resource management architecture to facilitate automated and on-demand provisioning of compute and network infrastructure. We have abstracted and virtualized data-center infrastructure into a pool of virtual resources. A composition algorithm was used to select virtual resources to satisfy on-demand IaaS requests. Finally, we have prototyped the proposed architecture, using OpenFlow as the hardware abstraction layer. The implemented framework makes use of FlowVisor [2], OpenFlow controllers and related tools to virtualize network infrastructure in a data-center. After a successful resource allocation, users gain access to a compute resource controller and a subset of OpenFlow controller functions, allowing them to manage their virtual compute and network resources.

The rest of this paper is organized as follows. In section 2 we examine the main characteristics of recent cloud networking solutions, emphasizing noteworthy contributions to flexibility and controllability in tenant networks. Section 3 deals with the design considerations and system architecture of our framework. In section 4, we discuss our methodology for integrating SDN components into the proposed platform. Section 5 evaluates the prototype's performance and analyzes the results. Section 6 concludes the paper and provides directions for future work.

II. RELATED WORK

In this section, we evaluate some comparable research on cloud network IaaS. We focus on the traffic-flow control features available for users in each approach and the measures taken to guarantee isolation and QoS.

Meridian [8] introduces a network service model that allows cloud users to create and manage virtual topologies in order to execute complex workloads. Users are able to construct logical topologies that suit their infrastructure needs using the Application Programming Interface (API) and GUI tools provided. An underlying network orchestration platform performs logical-to-physical translation and converts API calls into appropriate device configuration commands. It also provides network-wide services to applications, and coordination and mapping requests in the network. However, the authors do not attempt to give users a high degree of control over network traffic. Even simple traffic-flow modification requests need to go through cloud controllers, network controllers and planner modules before they can be physically configured.

Network Reservation System (NRS) [9] uses a central network controller approach to construct on-demand virtual networks. The authors attempt to address both intra-data-center and inter-data-center connectivity requirements by employing OpenFlow and overlay technologies respectively. A layered architecture and a topology model facilitate the construction of automated virtual networks. Isolation between virtual networks within a data-center is achieved using VLANs while VPNs are used for inter-data-center communication. The authors highlight the importance of a QoS guarantee and consider a possible future extension. However, they do not explore a way to grant flow-level control over the virtual networks.

J. A. Wickboldt et al. [10] emphasize that existing cloud management platforms lack the network management support and flexibility to express and control resource allocation requirements. The authors propose HyFS, a solution that explicitly considers network resources to be a part of a cloud platform that includes computing and storage resources. Their proposed framework allows users to demand a complex, feature-rich, virtual infrastructure through a flexible and programmable API. SDN/OpenFlow is used to implement robust networking functionalities within the cloud environment. Compared to existing frameworks, HyFS is distinguished by its tight integration of virtual network support. Nevertheless, HyFS does not grant users with traffic-flow control over their networks or illustrate how traffic isolation is guaranteed.

CloudNaaS [11] highlights the importance of providing users with a high degree of control over their virtual networks. It allows users to deploy applications with rich network functions such as custom addressing, service differentiation and flexible middle-box interposition. The authors introduce a network policy language and a network controller that respectively specify virtual network requirements and configure the virtual networks on a physical substrate. Tenant networks are isolated by creating VLANs. Bandwidth QoS is assured by pre-configuring forwarding queues in switches and dynamically selecting them based on Type-of-Service (ToS) bits in IP packet headers.

While we share similar motives, our work differs from that of CloudNaaS in several ways. First, CloudNaaS only provides users with a set of high-level policy constructs to define network requirements. These high-level policies do not provide capabilities to manipulate traffic flows dynamically between end-points. Second, our proposed architecture employs multiple layers of abstractions. This allows IaaS provider to improve resource utilization using resource management techniques such as virtual machine (VM) migration and oversubscription.

III. INFRASTRUCTURE PROVISIONING FRAMEWORK

Proposed framework employs multiple abstraction layers to facilitate virtualization and mask complexities in physical compute and networking infrastructure as illustrated in Figure 1. Physical Infrastructure Provider (PIP) manages the physical data-center resources and is responsible for maintaining connectivity at the physical level. PIP provides the IaaS provider with access to physical resources through an abstraction layer formed with OpenFlow protocol and hypervisor APIs. With available information, IaaS provider maintains a logical view of the physical infrastructure in a

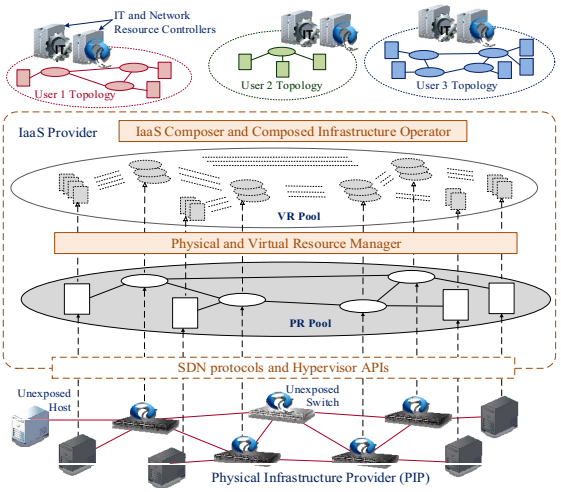

Figure 1: Abstraction of Data-center resources

local ontology known as physical resource pool (PR pool). IaaS provider also manages a second ontology, virtual resource pool (VR pool), by partitioning compute and network resources in the PR pool. The consistency between PR and VR pools are maintained by the physical and virtual resource manager. Finally the IaaS composer and composed infrastructure operator jointly act as a layer of abstraction between user requests and virtual resources. In this section, we illustrate the resource allocation process and functional components of our framework.

A. Virtual Resource Composition

Several existing proposals that address resource allocation for IaaS [12, 13] have paid more attention towards guaranteeing bandwidth than improving underlying resource utilization. Thus they result in less efficient use of datacenter resources and high blocking of IaaS requests. To overcome these limitations, we adopt a composition approach known as 2P-IaaS [14], to allocate resources for IaaS requests. It makes use of a unified ontology model along with reasoning capabilities to semantically represent diverse data-center infrastructure and to discover suitable resources for each request. After discovering suitable resources, individual resource matching values are calculated for each discovered resource using semantic similarity evaluation. Finally closeness centrality values are calculated and used for efficient virtual path composition, ensuring minimum proximity between discovered resources.

B. System Architecture

The main functional components of our system architecture and the ways in which they interact are shown in Figure 2. The framework seeks to improve IaaS request acceptance latency by discovering available physical resources and populating physical and virtual resource pools in advance. Before processing requests, IaaS provider reads available physical infrastructure information from PIP, through *compute virtualizer* and the *network virtualizer* modules. These two modules employ abstract communication protocols and APIs to retrieve limited but sufficient information from PIP. The limits guarantee the privacy of PIP and mask vendor-specific information and other physical layer complexities.

Two ontology-based resource pools maintained by IaaS provider, the PR pool and the VR pool, aids to sustain consistency during physical to virtual resource mapping. Initially, the PR pool is populated with the physical resource information discovered by compute virtualizer and network virtualizer. These two repositories are dynamically updated by (i) the acceptance and allocation of new requests, (ii) the de-allocation of resources when requests are successfully completed, and (iii) the modification or failure of physical infrastructure. The implementation and operation of the two pools are further discussed in the next section. When physical compute and network infrastructure specifications are retrieved, a *resource manager* module processes connectivity details and generates topology, identifying interconnections and relationships in the physical infrastructure. The topology and resource availability is stored in the PR pool. A virtualization algorithm in resource manager then queries the PR pool and populates the VR pool by partitioning physical resources into composable units.

Users can make a request from IaaS provider through a *service access layer*. It allows users to: (i) monitor the operation of the composed infrastructure, (ii) interact with compute resource controllers to start, suspend and shut down VMs and initiate virtual network computing (VNC) sessions, (iii) communicate with network resource controllers to install, change or delete flow rules and controller applications, and (iv) request VM migration and modify resource allocations.

When a new IaaS request arrives, the service access layer forwards the extracted information to the *infrastructure composer*. The infrastructure composer then queries the VR pool and searches for resources that match the request criteria. The composition algorithm is then executed, connecting resources that satisfy the IaaS request. Once the request has been successfully composed, the composer forwards it to

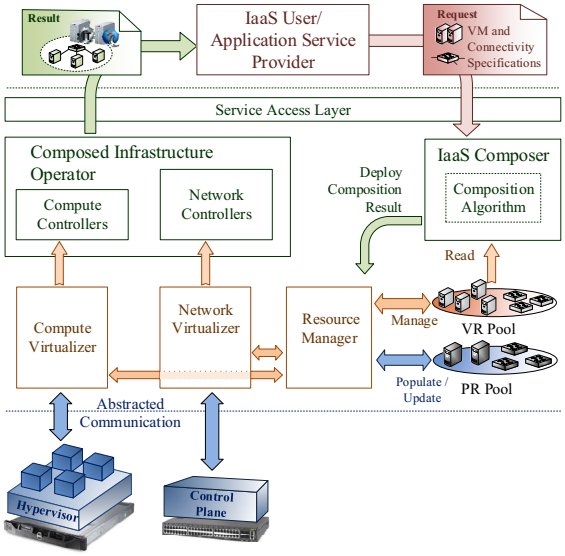

Figure 2: System Architecture

resource manager for deployment. To commence the deployment, it performs three distinct, parallel tasks: (i) changing the state of corresponding composable units in the VR pool to represent reservation, (ii) requesting the compute virtualizer to create VMs, and (iii) requesting network virtualizer to create interconnections between the VMs.

Resource manager verifies the deployment by signaling compute virtualizer and network virtualizer to update the PR pool. If the deployment is successful, it delivers accessibility information to the *composed infrastructure operator*. Subsequently, the composed infrastructure operator creates a *compute controller* and a *network controller* to provide user applications with a set of API methods through the service access layer in order to manage the infrastructure. These API methods are translated into traffic-flow modification rules and VM management commands at the network controller and compute controller respectively. Thus the end result of a successful allocation is a set of API methods that allows the user applications to manage VMs and dynamically manipulate traffic flows between them. One instance of network controller and one instance of compute controller is assigned to each accepted IaaS request. If the request is rejected, the user is given a response through the service access layer.

IV. OpenFlow-Based Implementation

To verify the operation and evaluate the performance of the proposed framework, we implemented a prototype. We used the Java programming language for the development of most of the management framework. Network virtualizer was implemented as a java application module that uses FlowVisor API. Each network controller instance allocated to each request is implemented as a python application running atop dedicated POX [15] controller. The framework was tested on both a physical test-bed and Mininet[16] emulated environment. Each VM host in the system was configured with a Kernel-based Virtual Machine (KVM) [17] hypervisor and an open virtual switch (OVS) [18] over a Linux platform. An open source API toolkit known as Libvirt [19] was used for management. Libvirt is also appropriate as an abstraction layer since it conceals the implementation complexities and sensitive security information.

The operation of the framework can be divided into five phases: (i) resource discovery and construction of the PR pool, (ii) virtualization and construction of the VR pool, (iii) VM mapping and connectivity composition, (iv) resource reservation and deployment, (v) composed infrastructure operation. This section outlines the techniques and tools used to carry out these phases.

A. Resource Discovery and Construction of PR Pool

During resource discovery, resource manager makes two parallel requests to network virtualizer and compute virtualizer, inquiring about the current state and availability of network and compute resources respectively. Switches are uniquely identified from their data-path identifiers and links are identified by end-point switches. Each physical VM host contains an OVS to allow internal VMs to connect with the external data network, thereby forming virtual access layer of switches in the topology. Thus OVSs act as network endpoints and are abstracted in the same way as physical switches.

The PR pool is designed to maintain a logical replica of the data-center topology constructed by resource manager. When an OVS is connected to a physical network interface card (NIC) of a host, OVS adopts MAC of the NIC as its DPID. Resource manager uses this property of OVS to map network information obtained from network virtualizer to host information obtained from compute virtualizer.

To construct the PR pool, we employed W3C standard OWL-2 ontology since its semantic graph structure is suitable for describing computing infrastructure. We have used Infrastructure Description Language (INDL) [20], type schema with several modifications to meet our requirements. Specifically, we added port attributes with appropriate relations to allow OpenFlow port-level information to be stored. We also abstracted processor and memory components into a single ontology class and identified it as a compute component, to facilitate cloud integration.

B. Virtualization and Construction of VR Pool

After the PR pool is updated with physical topology information, the virtualization algorithm in resource manager queries the PR pool and creates composable VR units through partitioning. The VR pool therefore maintains a set of virtual machines, virtual links and virtual switches to be used during VM mapping and composition. These virtual resources were created with the intention of satisfying the input requirements of the composition algorithm. Physical VM host resources are logically partitioned into VMs with three classes (*A*, *B* and *C*). This ensures that all VMs in the same class contain an equal amount of resources. Class *A* VMs have most resources and those in class *C* have fewest. For this work, we employed only static partitioning to decide the number of VMs and resources in each category. Partitioning and aggregation to optimize resource use is a subject for future work. We consider link bandwidth, switch CPU and switch flow table as critical network resources. Each resource in the VR pool contains a state, together with functional and non-functional characteristics associated with computing, network or storage services. Interconnecting network link information, including bandwidth and delay parameters, is also stored, along with other network device properties.

C. VM Mapping and Connectivity Composition

Once an IaaS request has been received, the composer module extracts the requested end-point and QoS information and executes the composition algorithm. The 2P-IaaS composition algorithm works in two phases: (i) a VM mapping phase, and (ii) a connectivity composition phase. During the VM mapping phase, VMs that satisfy the request are discovered from the VR pool. The objective of the mapping algorithm is to improve the data-center host's resource utilization by minimizing excessive network use when VMs belonging to the same request are distributed. If insufficient VMs are discovered, the request is rejected

without proceeding to the connectivity composition phase. Otherwise, the connectivity composition algorithm is executed. If suitable switches and links are available and sufficient QoS can be ensured, the request is accepted and resources are reserved through resource manager.

D. Resource Reservation and Deployment

When an IaaS request is accepted, resource manager changes the state of reserved VMs from *available* to *reserved*. But, for each virtual link reserved, a new virtual link is created with the required bandwidth and deducted from the original bandwidth. Similarly, new virtual switch are created on the original virtual switches by reserving switch CPU and flow table resources.

After updating the VR pool, resource manager constructs a deployment plan by defining VMs and FlowVisor slices. Each slice created per request is assigned a VLAN identifier so that the traffic is isolated. Packets generated from VMs belonging to a particular request are tagged with a unique VLAN-ID and are assigned to a corresponding slice.

Bandwidth isolation is achieved by creating QoS queues for switch ports with rate limiters. When slices are created, flow-spaces are formed assigning queue numbers using *force-enqueue* in the *add-flowspace* method in the FlowVisor API. We also isolated switch resources through the corresponding allocation of switch CPU and flow-table to each slice. For this, we initially populated the PR pool with flow-table and management plane CPU information of each switch. During slice creation, the percentage of control plane CPU was assigned using *rate-limit* and the percentage of flow table using *flowmod-limit*.

E. Composed Infrastructure Operation

Once the deployment has commenced, the system architecture allows the PR pool to be updated to verify the deployment and to deliver accessibility information to composed infrastructure operator. It in turn allows compute controller and network controller to manage VMs and traffic flows respectively. However, in our prototype, since network controller information must be given when slices are created, we maintained a pool of network controller and compute controller instances in composed infrastructure operator. The final output of a successful IaaS request is therefore a VM management interface and a set of network virtualizer API methods that manages traffic flows in the virtual network.

V. PERFORMANCE EVALUATION

To evaluate the framework's performance, we carried out a set of experiments using both a physical test-bed and Mininet. For the physical experiments, our test-bed consisted of 9 physical VM hosts, 3 physical access layer switches, 2 core layer switches and a gateway. These servers and switches were interconnected to form a fat-tree topology with 1Gbps Ethernet connections. For physical core and access switches, we used OpenFlow-supported Pica8 P-3290 switches. Our test-bed consisted of physical compute nodes with a range of processing, memory and storage resources. We used Ubuntu-server 14.04 LTE as the host operating system. For testing purposes, VMs created from the framework were booted from live versions of Lubuntu 14.04 and Tiny-core version 6.3 operating systems.

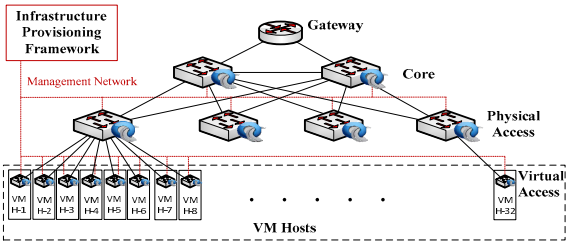

Figure 3: Topology of prototype test-bed

In the initial design, we categorized physical VM hosts into three resource classes loosely based on resource availability. When constructing the PR pool, these classes were used to partition physical resource into respective classes of VMs, to match composer requirements. To implement the management components of the framework, we used six work stations. One was used to execute a Java-based cloud management application and another to run FlowVisor. The remaining four machines ran POX controllers. A hub-connected out-of-band management network was used for all configurations, monitoring and other management operations.

We also implemented an emulated test environment using Mininet to compare the performance of the physical test-bed and analyze the scalability of our framework. It consisted of 32 virtual hosts created with VirtualBox [21], with 6 switches, as shown in Figure 3. Mininet and the VirtualBox were both executed on a Dell PowerEdge R720 server equipped with 2 Intel Xeon E5-5910 processors and 196 GB of RAM. Performances of the main operational phases were evaluated using: (i) Management overhead, (ii) Scalability and (iii) Request service time. The performance of the resource discovery and request deployment phases was evaluated on both physical and Mininet test-beds. The performance of the virtualization and composition phases was analyzed using data collected for Mininet test-bed in order to satisfy scalability requirements.

A. Resource Discovery and Construction of PR Pool

We first quantified the scalability of the approach by measuring the management overhead of the PR pool population phase, according to the number of VM Hosts connected to the system. In this phase, resource manager signals compute virtualizer to discover VM host resources and FlowVisor to discover network resources over the management network. We physically connected and disconnected VM hosts to access switch ports and obtained traffic measurements for 10 iterations. The traffic volume was measured using the IPTraf tool [22] and separated into FlowVisor and hypervisor IP addresses. The average values appear in Figure 4(a).

Management network overhead varies linearly with the number of VM hosts. When a new host is added, an increase in traffic can be anticipated. An increase in overhead can also

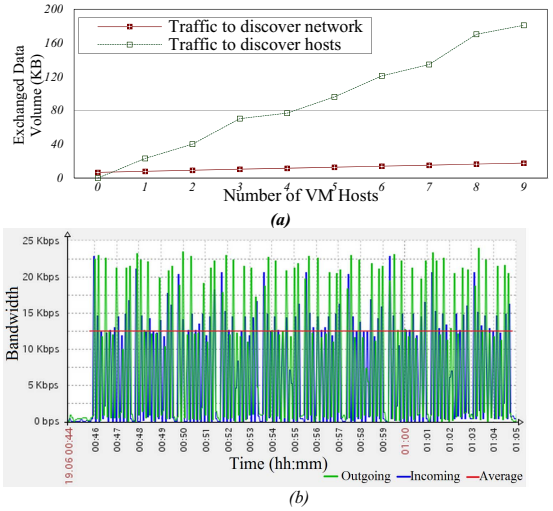

Figure 4: Management traffic (a) volume, (b) bandwidth, during discovery

be expected since, with the new VM host, a hypervisor virtual switch is also added to the system. However, VM host discovery overhead varies in a higher gradient than the network overhead. This variation is also predictable since host resources have higher granularity than switch resources.

We also measured bandwidth use with a standard open-source resource monitoring toolkit known as Zabbix [23]. The bandwidth used to obtain resource information about the physical topology (9 hosts and 5 physical switches) was measured for 10 iterations. The incoming and outgoing bandwidth use, including average use, is shown in Figure 4(b). This graph shows that the framework needs on average less than 15 Kbps to discover physical resources. The results in Figures 4(a) and 4(b) demonstrate that management network overhead during the resource discovery phase is minimal.

PR pool construction involves three aspects: (i) compute resource discovery, (ii) network resource discovery, and (iii) ontology update. We measured the time taken to obtain compute resource and network resource information in the discovery process for both physical and Mininet topologies. Results are shown in Figure 5, in terms of the number of hypervisors added to the test-bed. The graphs also show the times taken to update PR pool ontology for both test-beds. Since we have limited servers in physical test-bed (i.e. 9 VM hosts), physical result graphs are displayed for 9 VM hosts. Both topologies take a similar amount of time to obtain network resources and ontology update times are comparable in both cases. However, the physical resource update consumed relatively more time than Mininet update. We felt that the difference was mainly due to delays added by the physical network interface cards. This was verified by an experiment comparing the time taken for a small Libvirt call between two virtual and physical hosts.

B. Virtualization and Construction of VR Pool

For compute resources, the virtualization process involves partitioning physical host resources and creating VMs. For network resources, we created one virtual resource for each

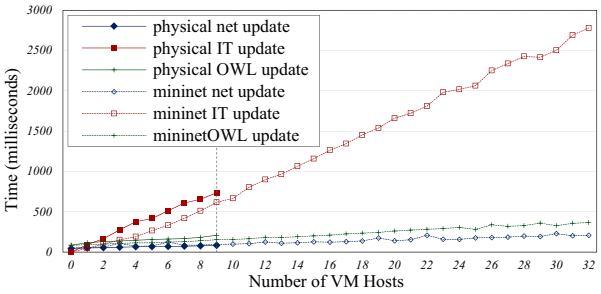

Figure 5: Physical resource Discovery time

physical resource, to include all available resources. We measured the time needed to virtualize both compute and network resources in terms of the number of virtual resources created. The results are shown in Figure 6, together with the time to update the VR pool according to its size. The total time to construct and update the VR pool can therefore be calculated from the graph by adding virtualization and OWL update times. For a large data-set, we used a PR pool populated by resources from Mininet topology. This allowed us to analyze the virtualization and composition algorithms.

The graphs show that the time to virtualize resources varies linearly with a small gradient. However, ontology update time increases in polynomial time with respect to the number of elements in the VR pool. By extrapolation, we found that around 600 milliseconds were required to update 1000 virtual resources into VR pool. Hence for massive data-centers with a large amount of operational free resources, the ontology update will add considerable overhead. However, under normal operating conditions in small-scale data-centers, with moderate use of data-center resources, the total time needed to update the VR pool will be tolerable. Moreover, since both the PR pool and the VR pool were updated after successful deployment of a user request, the time required to update the VR pool does not directly affect the user experience.

C. VM Mapping and Connectivity Composition

Once an IaaS request arrives at the composer, the 2P-IaaS algorithm initially discovers and maps host resources (end-point VMs). Connectivity composition follows. We evaluated the VM mapping function and measured the time taken to calculate semantic similarity and closeness centrality scores for each requested resource. In order to analyze the performance of the VM mapping function under different load conditions, we made continuous high-life-time requests in order to saturate the system. Three sets of requests were

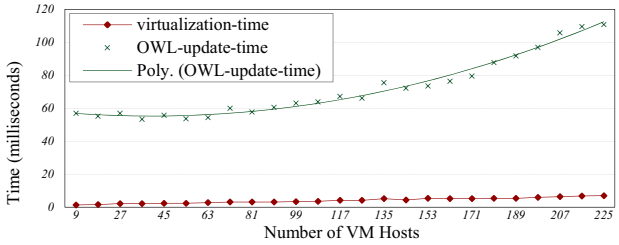

Figure 6: Virtualization and VR Pool Update time

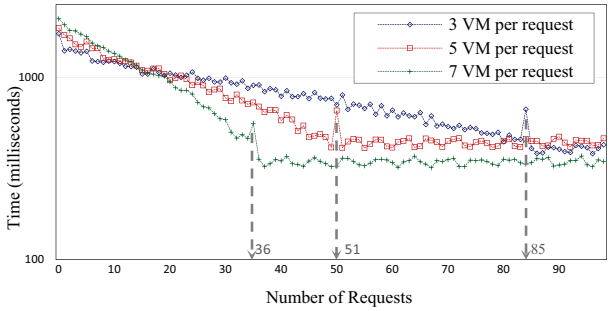

Figure 7: Time to discover suitable end-point VMs

created with 3, 5 and 9 VMs per request with 100 requests in each set. The measurements are shown in Figure 7, according to the sequential arrival of requests.

The vertical axis of the graph is represented in logarithmic scale to highlight important observations in the results. It can be seen that the performance of the VM mapping function varies significantly with the number of available resources in the VR pool. Since the experiment was started with a large number of resources in the VR pool, the mapping function initially takes more time to find the most suitable VMs. This is due to the processing time taken by large data-sets returned from ontology queries. When the number of available resources decreased, the time taken by the mapping function decreased logarithmically. This demonstrates the theoretical complexity values presented in [14], i.e. *O(n log n)* for querying and sorting resources, and *O(n)* for the evaluation of semantic similarity and closeness.

Once suitable resources are exhausted, the mapping function starts to reject incoming requests. Rejections were noticed for requests with 3 VMs from the 85th request and for requests with 5 VMs from the 51st request. We saw a sudden increase in the time consumed by the discovery algorithm at the point the rejections started (marked by a dotted-line arrow in the graph). This is due to additional exhaustive search queries made by the program with relaxed variables. The results of these additional search queries are kept in data structures to prevent repetition in all following requests with similar requirements. With this improvement, the time taken by the mapping function was minimal and roughly constant after the rejections started.

After mapping VMs, the 2P-IaaS algorithm builds interconnections between VMs, a process also known as connectivity composition. We measured the time taken by the

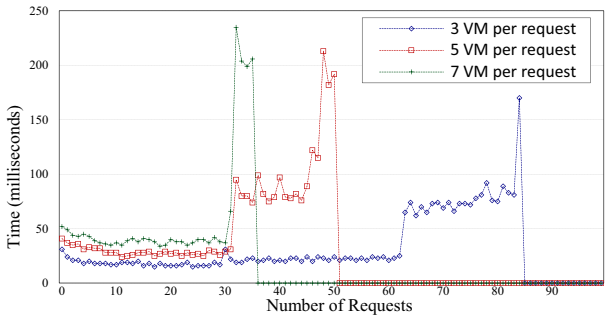

Figure 8: Time for connectivity composition

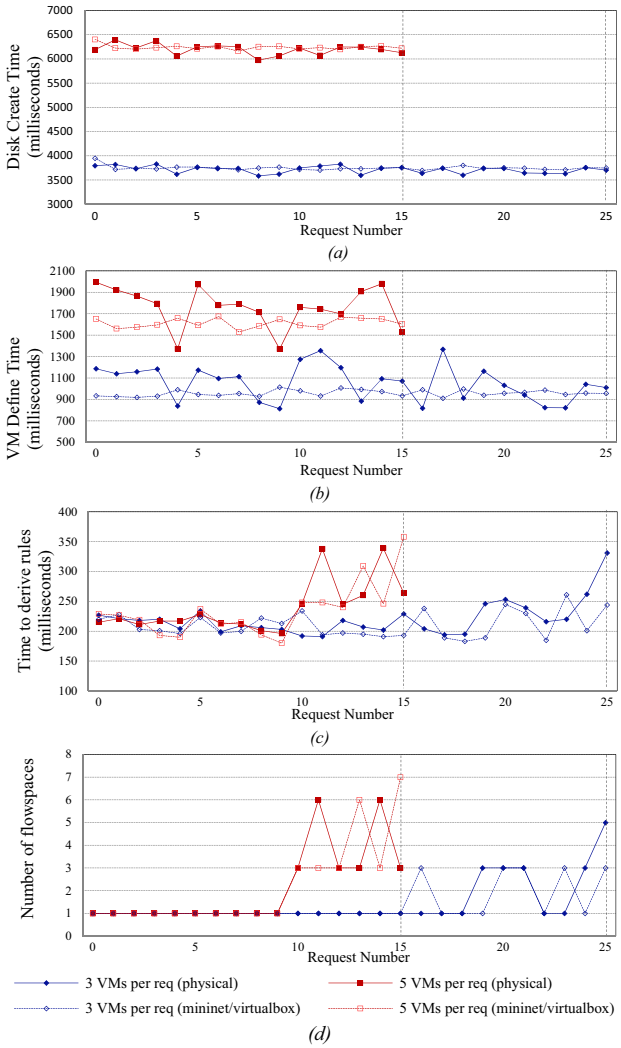

Figure 9: Deployment Results; (a) Disk create time, (b) VM instantiation time, (c) Slice create time, (d) number of flow-spaces created

connectivity composition algorithm to progressively discover and compose network paths. The results are shown in Figure 8. As we discussed previously, the objective of the VM mapping function is to improve host resource use while minimizing the resource use caused by the distribution of VMs belonging to the same request in different hosts. However, when sufficient VMs are not available in the same host, a topologically closer host is selected, resulting in an increased complexity of path computation. This can be seen in the graph as sudden increases in the composition time. Particularly for 5VM requests, initial composition time is constant around 28 milliseconds for the first 32 requests. Afterwards, composition time increments were observed in distinct stages roughly corresponding to 80mS, 12mS and 200ms until the rejections starting from the 51st request. Composition time was seen to increase when the number of VMs per request increased. This can be explained by the increase in processing requirements needed to interconnect a higher number of end-points.

D. Resource Reservation and Deployment

The deployment of successfully composed requests involves creating disk images, defining VMs and deriving flow rules to create slices and flow-spaces through FlowVisor. We compared the time required for each of these sub-tasks on both our physical and simulated test-beds. The simulated environment was constructed using virtual-box VMs with resources equivalent to the physical test-bed. These virtual hosts were interconnected through Mininet, which was topologically equivalent to the physical test-bed. The results were plotted according to the number of requests (Figure 9).

A disk image for a VM is created by making an image creation request to the corresponding host's Libvirt agent over a secure shell. We measured the time between request to response for each VM in accepted 3VM and 5VM requests. The results (Figure 9 (a)) show that the time taken to create disk image files for both physical hosts and virtual hosts was equivalent. Although disk creation time is relatively independent of the underlying hardware platform, we discovered that VM creation times are highly hardware-dependent, as shown in Figure 9 (b). As mentioned before, our physical test-bed consists of heterogeneous computing platforms with uneven processing power. Factors such as CPU technologies (Intel Xeon, i7 etc.) and memory speeds play a major role in the results. However, although the simulated platform represents an equivalent number of cores and an equivalent amount of memory to the physical platform, all virtual hosts share the same CPU technology and memory speed. Thus, we only noticed minor variations in VM creation time for the simulated environment.

After creating all end-point VMs in a request, a virtual network interconnecting end-points is created by deriving and making FlowVisor API calls for slice and flow-space creation. Time measurements for these operations are shown in Figure 9 (c) and the number of flow-spaces required for each slice is shown in Figure 9 (d). From these results, we see that slice creation time is correlated to the number of flow-spaces. We can conclude that allocating VMs close to each other for the same request during composition reduces the load over FlowVisor and minimizes delays in slice creation. It also improves resource use and decreases request service time.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have focused on fully automated service provisioning and on providing users with control over allocated infrastructure. We have illustrated our approach through our cloud IaaS framework. We have shown that the framework is capable of delivering cloud IaaS services to users with a high degree of control. The operation was demonstrated through a series of prototype experiments with both physical and Mininet test-beds. By comparing the results of these two environments, we demonstrated the efficiency and scalability of our approach. We also showed experimentally that, in an OpenFlow environment, selecting end-point VMs topologically close to each other can potentially reduce request service times and load.

As for our future work, we will investigate different composition and resource allocation techniques. In addition, we are planning to extend this work towards distributed data-centers and address cloud elasticity requirements.

REFERENCES

[1] P. Mell and T. Grance, "The NIST definition of cloud computing," Tech. Rep., 2011.

[2] R. Sherwood, et al., "FlowVisor: A network virtualization layer." OpenFlow Switch Consortium, Tech. Rep (2009).

[3] M. Casado, N. Foster, and A. Guha, "Abstractions for Software-Defined Networks," Communications of the ACM, vol. 57 no. 10, October 2014, pp 86-95.

[4] N. McEeown et al., "OpenFlow: Enabling innovation in campus networks," ACM SIGCOMM Computer Communications Review, vol 38, no 2, pp. 69-74, April 2008.

[5] Amazone AWS, "Amazone Web Services," February 2016, [Online]. Available: https://aws.amazon.com/.

[6] Openstack, "Openstack Cloud Software," February 2016, [Online]. Available: https://wiki.openstack.org/.

[7] Cloudstack, " Apache CloudStack: Open Source Cloud Computing," February 2016, [Online]. Available: https://cloudstack.apache.org/.

[8] M. Banikazemi, D. Olshefski, A. Shaikh, J. Tracey and G. Wang, "Meridian: an SDN platform for cloud network services," IEEE Commun. Mag., vol. 51, no. 2, Feb. 2013, pp. 120 127.

[9] D. Theodorou et al., "NRS: A System for Automated Network Virtualization in IaaS Cloud Infrastructures," ACM VTDC, June 2013.

[10] J. A. Wickboldt et al. , "Retinking Cloud Platforms: Network-aware Flexible Resource Allocation in IaaSClouds," in Proceedings of 13th IFIP/IEEE International Symposium of Integrated Network Management (IM 2013), 27-31 May 2013, Ghent, Belgium, pp. 450-456.

[11] T. Benson et al., "CloudNaaS: a cloud networking platform for enterprise applications," in Proceedings of ACM SOCC, 2011.

[12] 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 CoNEXT, J. C. de Oliveira, M. Ott, T. G. Griffin, and M. Mdard, Eds. ACM, 2010, p. 15

[13] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in Proceedings of the ACM SIGCOMM 2011 Conference, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 242–253

[14] K. M. Metwally et al., "Two-phase ontology-based resource allocation approach for IaaS cloud service," in 12th Annual IEEE Consumer Communications and Networking Conference (CCNC), 2012.

[15] POX, "POX: Python based DSN Controller," January 2016 [Online]. Available: https://github.com/noxrepo/pox.

[16] Mininet, "An Instant Virtual Network on your Laptop (or other PC)," April 2016 [Online]. Available: http://mininet.org

[17] KVM, "KVM: Kernel Virtual Machine - Version 2.6.20 kvm-12," January 2016 [Online]. Available: http://www.linux-kvm.org/.

[18] Open vSwitch, "OVS: Production Quality, Multilayer Open Virtual Switch - Version 2.3.0," August 2015 [Online]. Available: http://openvswitch.org/.

[19] Libvirt, "Libvirt: The Virtualization API-Version 1.2.7," August 2015, [Online]. Available: http://www.libvirt.org.

[20] M. Ghijsen et al., "Towards an Infrastructure Description Language for Modeling Computing Infrastructures," in The 10th IEEE International Symposium on Parallel and Distributed Processing, 2012.

[21] VirtualBox, "VirtualBox: general-purpose full virtualizer for x86 hardware - Version 4.3.22," January 2016 [Online]. Available: https://www.virtualbox.org/.

[22] IPTraf, "IPTraf: IP Network Monitoring Software - Version 3.0.0," September 2015 [Online]. Available: http://iptraf.seul.org/.

[23] Zabbix, "Zabbix: The Enterprise-class Monitoring Solution - Version2.2 LTS," January 2016, [Online]. Available: http://www.zabbix.com/.