The Virtual Puppet Master: Adaptive Streaming on Top of an SDN-Enabled Virtual Infrastructure

Roberto Canonico, Enrico De Maio, Pasquale Di Rienzo, and Simon Pietro Romano(✉)

Università degli Studi di Napoli Federico II, Napoli, Italy
{roberto.canonico,spromano}@unina.it,
{enr.demaio,p.dirienzo}@studenti.unina.it

Abstract. In this paper we present the Virtual Puppet Master, an orchestration framework for the dynamic deployment of real-time services. We show how to leverage both virtualization and Software Defined Networking in order to seamlessly implement a scalable live streaming architecture supporting effective, on demand deployment of a hierarchical distribution tree, as well as live migration of network nodes. The architecture is illustrated both from the design and from the implementation perspective. A first qualitative assessment of the overall functionality of the framework is also introduced.

1 Introduction

This paper aims to demonstrate how the combined use of virtualization and Software Defined Networking (SDN) can significantly ease the deployment of advanced network services in cloud-based scenarios. We will focus on live streaming over the Internet, which is by no doubt one of the killer applications for today's OTT (Over The Top) operators, due to its stringent requirements in terms of both scalability and dynamic deployment of the delivery infrastructure.

The framework we present has been called *Virtual Puppet Master* (VPM) to indicate its capability to dynamically create, configure, monitor and modify a hierarchical distribution tree made of a number of nodes, each playing a specific role in the overall streaming chain connecting a single source to a distributed set of clients. The architecture is entirely built around the concept of virtualization and naturally lends itself to a cloud-based deployment. The SDN paradigm comes into play for all functions associated with the creation and dynamic management of an overlay network of content delivery nodes. It also proves fundamental when implementing support for the so-called 'live migration' of a subset (i.e., both nodes and links) of the overall distribution tree.

The paper is organized as follows. In Sect. 2 the SDN paradigm will be briefly introduced. In Sect. 3 we will discuss the design of our framework, called *Virtual Puppet Master*, from a design perspective. Section 4 will dig into our implementation choices. Section 5 will present some qualitative results obtained

© Springer International Publishing Switzerland 2015
S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 825–836, 2015.
DOI: 10.1007/978-3-319-27308-2_66

after deploying a real-world test scenario through the VPM approach. Finally, Sect. 6 will conclude the paper, by also discussing directions of our future work.

2 Software Defined Networking

The acronym SDN stands for "Software Defined Networks" and it represents an innovative networking approach aimed to give network administrators methodologies to design, build and manage networks within dynamic and fast-paced environments. The basic trait of SDN consists in decoupling the part of a switch that handles decisions, called control plane, from the part that just refers to the forwarding table, named data plane. In an SDN-compliant switch the forwarding table is updated by obeying to directives coming from external entities. The entity in charge of instructing an SDN-based switch is named controller, and it is basically a software component that, on a side, interacts directly with one or more switches through a well-defined protocol (at the time of this writing, the *OpenFlow* protocol [4]) and, on the other side, checks requests from business applications. In other words, an SDN controller has two interfaces: (i) a southbound API, used by OpenFlow to communicate with SDN switches; (ii) a northbound API, used for interactions with business applications. Most controllers typically implement this last API as a REST service.

SDN offers an abstraction of the underlying network. By using a controller, its northbound API in particular, the real network is abstracted from the business application. This feature makes programmers' job easier as well, because they just have to learn one API for all switches.

OpenFlow is, at the moment, the SDN-defined controller-switch protocol and is implemented by all known SDN switches implementations.

3 The Virtual Puppet Master: Design Considerations

In this section we will examine the design of the VPM infrastructure. We will introduce the technologies used, motivating their choice and showing the final architecture. The code name we chose for the research project is *Virtual Puppet Master* because of its ability to alter, from behind the scenes, the network in which an application is deployed. Hosts belonging to the network are, in fact, just like puppets in a show: they think they have total control over their actions and choices, while in reality they are manipulated by an orchestrating external entity taking decisions on their behalf.

Adaptive streaming is such an exhaustive example that we decided to build our architecture with this type of scenario in mind. The defined architecture had to comply to a number of both functional and non-functional requirements. Among such requirements, we herein cite scalability, reconfiguration, fault tolerance, application agnosticism, user friendliness, customization and monitoring.

Given the agility and scalability requirements, we opted for a hierarchical design, in particular a tree. Due to the absence of loops, this topology allows for simpler routing and management algorithms, as just one path connecting two nodes will always exist. However, the availability of just one path also means that, in case of failures, some nodes will not be reachable, which conflicts with the fault tolerance requirement. For this reason, we chose to give the user the opportunity to specify redundancy links among switches, so that in case a switch crashes, an alternative path will be found.

We opted for an overlay network architecture. As hosts are virtual, it seems just appropriate to make the network they are plugged in virtual as well. Virtual machines have the illusion of belonging to the same network, while host nodes reside on different local networks. The fact that VMs belong to the same network means that, for any application running on them, there is no need to worry about network details like addresses translations and port forwarding. This makes guest cooperation immediate, a very desirable requirement on a cloud computing infrastructure.

When an SDN controller is present into the network, the traffic has to be explicitly allowed by flow rules. This means that just creating connections between switches does not automatically make a stream flow. We decided to build our network logic around this concept: in order to make a stream flow from the root to a leaf, the user has to explicitly ask for it. In our logic, we will define a path as a sequence of switches and links starting from the root and terminating in a leaf. Each switch has flow rules installed allowing a packet to jump to the next hop. In other words, a path can be defined as such only if the flow rules allow packets to travel across it. About the tree, we will distinguish among three types of nodes: (i) Root node (unique); (ii) Relay node(s); (iii) Leaf node(s). A VM present on one of these nodes will become respectively a root, a relay or a leaf VM. The user will thus be asked to mark a node with one of these types. This approach leads to the creation of what in the literature is called a *role-based tree*. According to the node type and depending on whether a VM belonging to the installed application is present or not, the flow installation procedure will slightly change. If a node is marked as root or lies along the path but does not host any VM, an application packet will have to cross it and go directly to the next hop. On the contrary, if a VM is present and the node is not a root, application packets will have to be forwarded first to the VM and eventually from the VM to the next hop.

Migrating a VM from a hypervisor to another can be a very difficult operation in this kind of infrastructure. This is due to the fact that a guest might be in the middle of a communication with an external client or another VM and, by migrating it, this communication might be lost. So, a full ‘network’ migration is needed. Based on the above considerations, for what concerns the migration algorithm, we took inspiration form a recent work about the live migration of ensembles (shortened in LIME) whose details are explained in [2]. The referenced paper illustrates in detail how to effectively perform this operation by preliminarily cloning the network instead of just moving it right away. This approach leads to the best performance as the VM, once migrated, can immediately start its services, since packets are already flowing to its new location. So, when a migration request is received, we first check if there is any flow from the root to the leaf where that VM is currently hosted. If the answer is yes, we check if there is already a flow towards the new switch that this VM will be attached to (this is done because another VM might be already running there). If the answer is yes also in this case, we do nothing as there is already a path towards that leaf. On the contrary, if there is no path, we first compute it like if the VM were already there and install flow rules on switches to enforce it. After this operation, we start the migration process. Note that, during migration, packets will be replicated on the two paths (or cloned, in the original LIME terminology). Once migration is successfully completed, the old path is removed. The migrated VM will receive data right away because packets will be already flowing: in this way, latency is minimized.

4 VPM Implementation

The VPM has been designed as a management application capable to provide the network administrator with means to: (i) alter the state of the network by creating topologies and redundancy links, or by issuing specific flow rules; (ii) monitor hypervisors, VMs and network state; (iii) migrate a VM from a hypervisor to another; (iv) add/remove resources (like hypervisors) on demand.

We decided to adopt a Model-View-Controller design pattern, implemented as a web application. The fact that the application is accessible by a common browser makes it portable across different operating systems, with no particular installation or technology, besides standard JavaScript and HTML5.

In the following of this section we will briefly introduce the most important choices we took when dealing with the implementation of the VPM.

For the virtualization part of our architecture, we chose KVM (Kernel-based Virtual Machine) [5], mainly because of its robustness to hardware heterogeneity when it comes to VM migration, as well as for the support of full hardware virtualization. KVM is also supported by *OpenStack* [6], our first-choice cloud platform. As KVM is just for the virtualization part, we have used *Qemu*[1] to emulate network and storage. Qemu was chosen as it is the de facto user-level application for KVM. In particular, network emulation is needed to implement the overlay network we talked about in the previous section. Storage emulation is done as well, to make a shared ISCSI disk available to VMs as a local disk. We also relied on *libvirt*[2], a universal API developed by the RedHat Corporation that allows applications to interface with a lot of different virtualization solutions, by also providing monitoring tools to check both guests and hosts status.

In order to migrate a VM, a disk sharing technology is needed. Here we opted for ISCSI, a particular type of Storage Area Network (SAN) and block-level disk-sharing technique.

Openvswitch is the OpenFlow switch implementation of our choice. It is a software implementation of an OpenFlow switch, released under the Apache 2.0 license and available for all Linux distributions as a kernel module. Openvswitch

[1] http://www.qemu.org.
[2] http://www.libvirt.org.

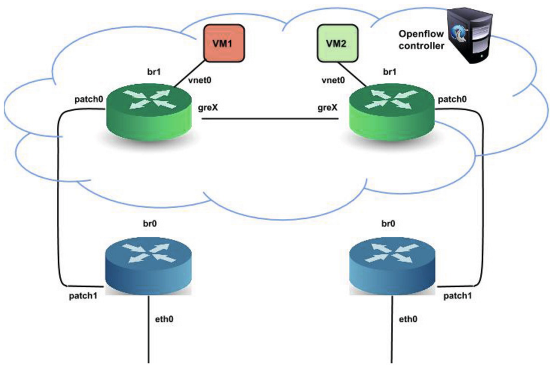


Fig. 1. Openvswitch setup

obviously supports OpenFlow. Though, it implements its own management protocol as well. By management protocol we mean that Openvswitch offers the user a way to interact with its internal structure, thus allowing to create switches, ports and links between switches. This feature has nothing to do with OpenFlow, but it is needed as, according to our requirements, we need to provide the user with a way to create networks by connecting switches.

In order to properly isolate the overlay from the physical network, we opted for the design illustrated in Fig. 1, making use of the Openvswitch capability of creating multiple bridges. Our implementation specifies 'br0' as default gateway so that the Linux network stack can implement GRE (Generic Routing Encapsulation) tunneling by making use of the 'eth0' interface, inaccessible without this workaround.

As the figure indicates, each node has two local bridges, br0 and br1, connected with patch ports. While br0 has a working NIC attached (eth0) and can exchange packets with the external network, br1 is isolated, with GRE tunnels as its only way to communicate. We opted for this setup because it is a very convenient and clear way to separate business traffic from VMs traffic. Bridges br1 are the only ones belonging to the overlay network and, because of that, they are also the only ones connected to the OpenFlow controller. In this way we make sure to examine and handle just the packets addressed to our application, hence saving a lot of overhead. Patch ports will be used on leaf nodes of the distribution tree, when a packet will need to 'jump' from the overlay network to the physical network. Just in this specific case, a special flow rule will be issued, allowing that packet to flow across the patch port, thus arriving at the bridge br0 (and eventually at eth0). This setup, commonly known as *isolated bridge*, is widely used in production environments.

Fro what concerns the control layer, we opted for Floodlight, an Apache licensed Java-based OpenFlow controller. It is fully extensible, based on a plugin system. This means that if there is some unavailable feature, it is possible to create it. We extensively used this system for VPM as we needed custom

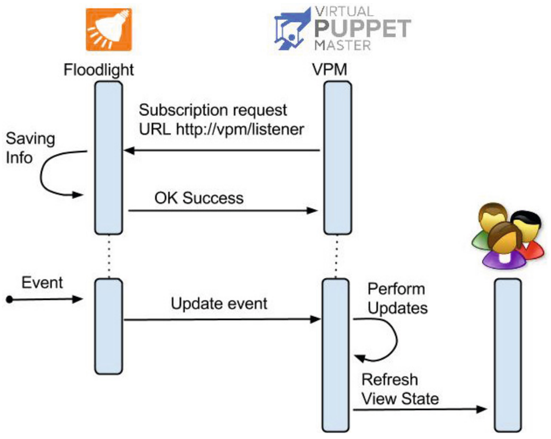


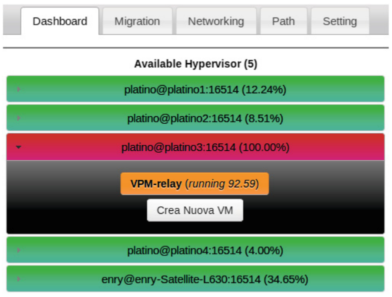
Fig. 2. VPM-Floodlight Notification service

behavior for some features. We implemented our own Floodlight module to perform some management operations. Our communication protocol makes use of the Observer design pattern (see Fig. 2): at start, the management application asks the controller to be notified of events and communicates a callback URL. Each time a significant event happens, the plugin makes a POST request to the specified URL containing data about that particular event. This solution decouples the management application from the plugin as the latter does not need to contain any static reference to the former.

In order for our management application to create links between OpenFlow switches, we needed a Java implementation of OVSDB, the *Open vSwitch Database Management Protocol*. Instead of creating it, we took the implementation done by the 'opendaylight-ovsdb' group and modified it. Opendaylight [1] is a collaborative SDN project which includes a Java-based OpenFlow controller with a rich set of plugins. One of them, in particular, interacts with a remote OVSDB database. We decided to make a stand-alone library version of the plugin by removing all the dependencies from the controller and taking just the protocol implementation part.

The management application is the core of the whole system. The user interface, as already anticipated, is a mix of HTML and JavaScript controls, while the backend is implemented in Java, through servlets. Exchanged messages are instead formatted as JSON objects.

The *Dashboard* tab, as the name suggests, is aimed to give the user an overview of the network nodes, here represented just for what concerns the hypervisor part (for the whole network there is in fact a dedicated tab). A list of registered hypervisors is shown (see Fig. 3). Offline ones are marked gray while online ones can be marked with different colors according to their current CPU utilization percentage (depending on two dynamically configurable threshold values).



Available Hypervisor (5)

platino@platino1:16514 (12.24%)

platino@platino2:16514 (8.51%)

platino@platino3:16514 (100.00%)

VPM-relay *(running 92.59)*

Crea Nuova VM

platino@platino4:16514 (4.00%)

enry@enry-Satellite-L630:16514 (34.65%)

Fig. 3. The VPM dashboard tab

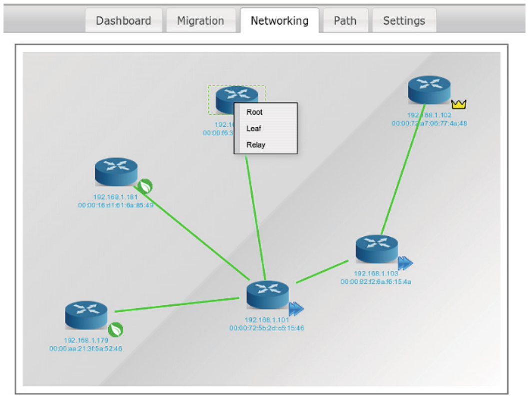


Fig. 4. The VPM network tab

Inside the *Network* tab, the user can create connections between switches in order to build a tree network topology with some redundancy links. To allow users to draw topologies, we made use of a JavaScript library called *mxGraph*, coupled to a server component capable to receive and process graphs sent by the client. This tab will show every switch currently attached to the controller (see Fig. 4), which are candidates for the final topology. Note that, being a VPM node both a hypervisor and a switch, these switches represent hypervisors as well and in fact, by hovering the mouse on a generic device, a list of installed VMs will appear. In order to create a connection, the user has only to drag the mouse from one switch to the other, while to set the role a right click on a switch will open a context menu on which it is possible to choose if that switch is a root, a relay or a leaf. We decided to let the user draw a generic topology, be it a tree or not; anyhow, when the "send tree" button is pressed a server side algorithm will compute the topology to actually make a tree. Spare links will be identified as redundancy connections.

With reference to tree creation, we used Kruskal's Minimum Spanning Tree algorithm [3] combined with some application logic. In fact, this algorithm is unaware of a node type and, by just giving it a generic topology, we would have no guarantees that the role of a certain node is respected. Moreover, the user could have made errors in the topology creation, like relays not connected to any leaf, which would inevitably bring to an erroneous topology. So, before giving the topology to the algorithm we first remove any dangling branches, which are branches ending with a relay or a node whose role was not set. Then, to make sure that the computed tree is the one the user wanted, we appropriately set link weights according to the type of its endpoints. In particular, each type is associated with a value, as follows: (i) Root: 0; (ii) Relay: 1; (iii) Leaf: 2. A link weight is set to the sum of its edge values. Finally, we provide such a topology as an input to the actual Kruskal's algorithm. This process will compute the correct tree. Only links which belong to the tree will be actually translated into GRE tunnels (by using our custom library), because they are the only links needed for our traffic to flow. Redundant links will be stored and used upon necessity.

On the *Path* tab the user can establish a path starting from the root and ending in a specific leaf, or delete an existing path. After selecting a root, the user is allowed to also select a leaf. In a dedicated panel, any existing path between these nodes will be shown. In order to create a path it is also required to set an external IP of a receiving node belonging to the physical network relative to the leaf node. This field is required in order to make the packets of a flow "jump" from the overlay to the physical network. Any kind of IP, be it unicast, multicast or broadcast, is allowed. When a new path is defined, a backend servlet will install specific flows on the selected switches along the path.

On the *Migration* tab users are able to migrate a virtual machine from one host to another. Since only guests, situated on leaves, are allowed to be migrated, they will be the only ones to be shown. In order to perform such an operation, the user has to simply drag and drop the desired VM from the source to the destination hypervisors. In the backend, the `Libvirt` API is leveraged to perform the migration process, while calls to a dedicated path manager component implement the LIME-inspired migration algorithm we explained in Sect. 3. A further servlet can be periodically polled by the client to retrieve information about the progress status of a specific migration.

5 Qualitative Tests

In this section we will first present the testing environment and then describe the streaming demo application. Finally, by using screenshots, we will give an overview of the final result. As VPM is still in a beta release state, please note that testing is just functional. Quantitative evaluations, involving throughput and latency measurements, are the subject of our future work.

We wanted our testing platform to be portable, so we simulated a 5 host virtual environment inside of an HP Proliant N54L 708245-425 micro server. This server is equipped with an AMD Turion II Neo N54L dual core processor, with 8 GB RAM, 2 network interface cards, and 2 Hard Disks (80 and 150 GB respectively). In order to achieve the best performance, we installed a bare metal hypervisor, VMWare vSphere ESXi 5.5, to simulate "physical hosts" through VMs. Due to the absence of any local user interface, this hypervisor can only be remote-controlled by its dedicated client software. This client-server architecture, which is common to most type I hypervisors, is necessary to make the virtualization as more efficient as possible. In Fig. 5 you can see an overview of our testbed.

HDs are organized this way: 160 GB are shared across VMs and used to simulate their virtual hard disks, while 80 GB are allocated to a specific VM with a raw device mapping technique (as this VM will represent our ISCSI target, it needs dedicated storage). Hosts are divided in 5 VMs, numbered from platino0 to platino5, having 1 GB RAM and 20 GB HD each. Among them, platino0 is designed to be an ISCSI target as well, and it mounts a dedicated

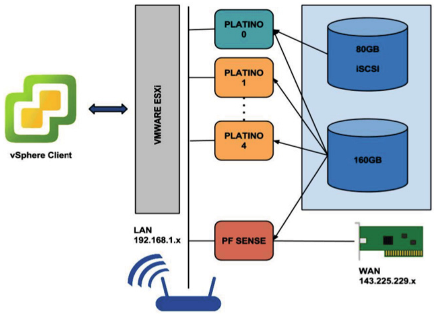


Fig. 5. The VPM Testbedb

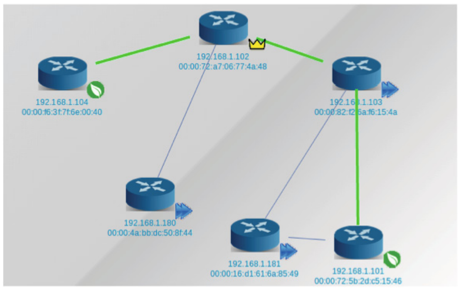


Fig. 6. Streaming demo topology

80 GB HD as an external peripheral. There is an additional VM mounting an open source firewall called *pfSense*[3], which provides NAT and DHCP functions as well. This VM creates a 192.168.1.x/24 LAN while providing external access through its physical WAN interface. A second NIC, not displayed in the figure, is also allocated to pfSense. This NIC is attached to an external router so that it becomes part of the virtualized LAN as well, along with any physical host currently attached to it. With this setup we allow for more complex scenarios, where any number of devices can become part of our testbed.

Each virtualized node is an Ubuntu server 13.10 (Saucy Salamander). We have chosen this distribution because of its low minimum requirements and also due to the fact that almost all needed software is already present in its default repositories, making the setup process significantly faster. Two bridges, br0 and br1, are set up like we saw in Sect. 3. In order to create the overlay network inside a Qemu instance, a proper XML file is processed through the libvirt API. By using the software `open-iscsi`, each node mounts the ISCSI target located at platino0, so 5 LUNs (Logical Unit Numbers), each one representing an already configured VM, are available.

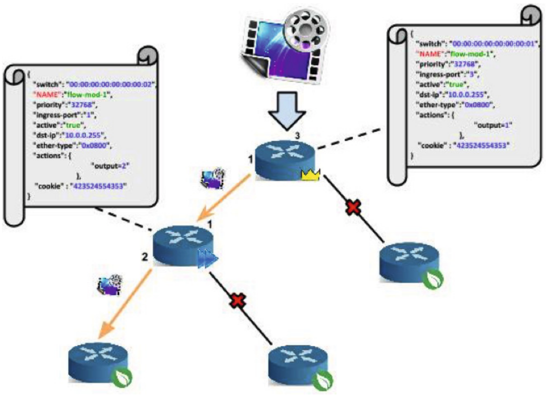
Guest VMs come with an Ubuntu server 13.10 OS installed as well. The streaming technology supported is GStreamer[4], an Open Source multimedia framework already present in the Ubuntu default repositories. In particular, the `gst-launch` utility was used to create a live stream of a sample video file. Two bash scripts have been created, the former of which is used on the root VM, while the latter works for both relay and leaf nodes.

By using our web GUI, we created the topology shown in Fig. 6. Figure 6 shows that the created topology presents some redundant links, colored in black. Two leaves are specified, one with a direct connection to the root node (platino2) and another one using an additional node as relay.

The demo streaming application works as follows: the root VM will just broadcast the live stream to the network broadcast address, 10.0.0.255. Under

[3] http://www.pfsense.org.
[4] http://gstreamer.freedesktop.org.

normal circumstances, this would imply the flooding of the entire network and would render the relay job of VMs useless, because the stream would just naturally flow to every switch ignoring any halfway entity. For our testing purposes this is just ideal, because without explicit flow rules these packets would not travel at all. What we have implemented is a logic that we called *sink or swim*, which can be expressed as follows: (i) if no flow rule is issued on the switch, the flow will just stop at the current hop, and will not be propagated through the device network interfaces ('sink'); (ii) if a flow rule is issued with an output action for specific interfaces, the flow will travel just across said interfaces even though it should be broadcast ('swim'). Figure 7 shows an example. Even though the root guest specified the network broadcast as destination, you can note how the stream just flows towards the leftmost leaf, like it would do in a unicast scenario. This is also a very good demonstration of VPMś ability to 'fool' virtual hosts.



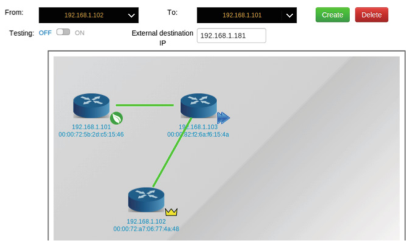
```
{
  "switch": "00:00:00:00:00:00:00:02",
  "NAME":"flow-mod-1",
  "priority":"32768",
  "ingress-port":"1",
  "active":"true",
  "dst-ip":"10.0.0.255",
  "ether-type":"0x0800",
  "actions": {
                    "output=2"
             },
  "cookie" : "423524554353"
}
```

```
{
  "switch": "00:00:00:00:00:00:00:01",
  "NAME":"flow-mod-1",
  "priority":"32768",
  "ingress-port":"3",
  "active":"true",
  "dst-ip":"10.0.0.255",
  "ether-type":"0x0800",
  "actions": {
                    "output=1"
             },
  "cookie" : "423524554353"
}
```

Fig. 7. "Sink or Swim" paradigm

As a preliminary test, we decided to boot only the VMs on the root and leaf nodes. This means that relay nodes will act as a "pass-through" nodes, by just sending packets to the next hop along the path. We hence started a stream and, after verifying that no external user could receive it, we activated a path to the rightmost leaf, specifying 192.168.1.181 as external destination IP address (Fig. 8).

On a leaf node, which contains a VM, we instead find two rules: one to forward packets to the VM and another to relay them from the VM to the next hop. The "action" field in the last flow rule enforces the rewriting of the destination IP to 192.168.1.181 before relaying the packet to the external network. After this operation, we can verify that the stream correctly arrives at destination, as illustrated in Fig. 9.



From: 192.168.1.102 To: 192.168.1.101
Testing: OFF ON External destination IP 192.168.1.181

Fig. 8. Creating a path

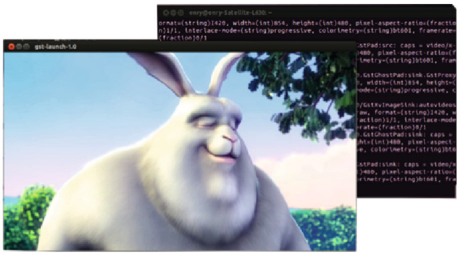
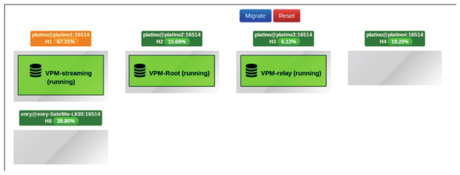


Fig. 9. Stream correctly received

As a further test, we booted up the relay VM on the relay node, to verify the correct operation of the Notification System. This action triggered the "add VM" event, with a subsequent alert message destined to the VPM.

We also made sure that the VPM had successfully changed flows belonging to the switch on which the new VM was booted. This test was a success, also leading to the same flow configuration we described for the leaf switch (except for the recipient address translation rule).


Migrate
Reset
platino@platino1:16514 H1 67.31%
platino@platino2:16514 H2 15.69%
platino@platino3:16514 H3 6.12%
platino@platino4:16514 H4 10.20%
VPM-streaming (running)
VPM-Root (running)
VPM-relay (running)
enry@enry-Satellite-L630:16514 H8 39.80%

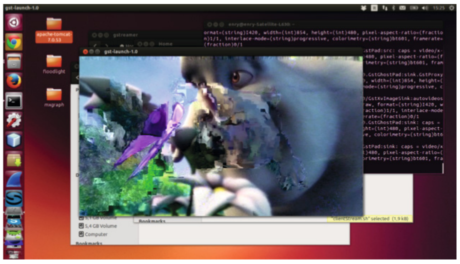


Fig. 10. VPM migration testing

Fig. 11. Video degradation effects

We finally forced the migration of the VM situated on the rightmost leaf to the leftmost one, while the user on the external node was watching the live stream (see Fig. 10).

Migration was successful and, looking at each involved switch, we noticed that flows were not present anymore on the right path and a new path, towards the new leaf, had been automatically created. About downtime, it was estimated, over a mean on 10 migration attempts with the same video source, to be about 2 s, which is perfectly reasonable for a live streaming application. During these 2 s, the user just experienced some frame losses (as it can be appreciated in Fig. 11), but continuity of service was preserved and the overall impact was acceptable.

6 Conclusion

In this paper we presented the Virtual Puppet Master, an architecture for the orchestration of advanced services on top of SDN-enabled clouds. We discussed both design and implementation of the architecture and provided information about the results of preliminary tests we conducted on a small-scale testbed.

Our future work on the VPM framework will be aimed at quantitatively assessing the performance attainable by the system in a large-scale deployment.

Acknowledgments. This work was partially funded by the Italian Ministry of Education, University and Research (MIUR) within the framework of projects PON01 01007 "PLATform for INnOative services in future internet" (PLATINO) and PON04a2_C (SMART HEALTH).

References

1. Feamster, N., Rexford, J., Zegura, E.: The road to sdn. Queue **11**(12), 20:20–20:40 (2013)
2. Keller, E., Ghorbani, S., Caesar, M., Rexford, J.: Live migration of an entire network (and its hosts). In: Proceedings of the 11th ACM Workshop on Hot Topics in Networks, pp. 109–114. HotNets-XI, ACM, New York (2012)
3. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proc. Am. Math. Soc. **7**, 48–50 (1956)
4. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. **38**(2), 69–74 (2008)
5. Medina, V., García, J.M.: A survey of migration mechanisms of virtual machines. ACM Comput. Surv. **46**(3), 30:1–30:33 (2014)
6. Sefraoui, O., Aissaoui, M., Eleuldj, M.: Openstack: toward an open-source solution for cloud computing. Int. J. Comput. Appl. **55**(3), 38–42 (2012)