OSHI - Open Source Hybrid IP/SDN networking
(and its emulation on Mininet and on distributed SDN testbeds)

Available at: http://netgroup.uniroma2.it/OSHI

Editors and main authors: P.L. Ventre(1), S. Salsano(2)
Contributions by: G. Siracusano(2), L. Prete(3), M. Gerola(4)

(1) Consortium Garr / University of Rome “Tor Vergata”
(2) CNIT / University of Rome “Tor Vergata”
(3) Consortium Garr
(4) CREATE-NET

DISCLAIMER
This work has been partly funded by the EU in the context of GN3plus project (http://www.geant.net/), but it does not necessarily represent the official position of the project itself and of its partners. Authors are solely responsible for the views, results and conclusions contained in this work.

COPYRIGHT
"Copyright and all rights therein are retained by authors. This work may not be modified or reposted without the explicit permission of the copyright holder"
# Table of Contents

1  Introduction: the Dreamer Project ................................................................. 3
2  Need for an Open Source Hybrid IP/SDN node ............................................... 5
3  Definition of a Hybrid IP/SDN network ........................................................... 8
   3.1  Implementation and testbed aspects. .......................................................... 14
4  OSHI requirements and analysis ................................................................. 15
5  OSHI deployment .......................................................................................... 18
6  OSHI Node - Architecture and implementation details ................................. 19
7  Coexistence mechanisms : Tagged and Untagged approach .......................... 25
8  Ingress classification functionalities .............................................................. 30
9  Ethernet Virtual Leased Line service .............................................................. 34
10 OSHI emulation tools ................................................................................... 37
   10.1 Experiments on Virtual BOX run time ..................................................... 38
   10.2 Mininet environment – Deployer and experiments .................................. 39
   10.3 OFELIA testbed – Deployer and experiments .......................................... 46
11 Performance evaluation experiments ............................................................ 52
12 VXLAN solution to cope with the scalability issues of OpenVPN ............... 57
13 Improved experiments with xentop ............................................................... 60
14 The NeST testbed ......................................................................................... 65
References ......................................................................................................... 67
### 1 Introduction: the Dreamer Project

The objective of the DREAMER project is to investigate and prototype how a network based on OpenFlow/SDN control plane can provide the same functionalities of an IP/MPLS control plane, offering a carrier grade approach to resiliency and fault management. Three architectural aspects, essential for large distributed networks, will be specifically targeted: i) the control plane Controller resilience and its distribution; ii) the data plane resilience, with carrier grade fast restoration mechanism; iii) compatibility with existing IP network through a migration strategy, because current IP networks are not replaceable at once. The DREAMER architecture will be prototyped mostly starting from available Open Source platforms, the prototype will be tested on the GÉANT OpenFlow testbed. The results aim to contribute to GÉANT and NRENs network infrastructures development roadmap. A re-usable “Topology Deployer” SDN application will be released, supporting the deployment of arbitrary virtual layer 2 topologies over the GÉANT OpenFlow testbed facility, that can contribute to extend the “Testbed as a Service” capability of the GÉANT OpenFlow testbed. Another important release will be the OSHI node (an open source hybrid IP/SDN network node), which will represent a novelty in this field. In Figure 1 we report the Dreamer high-level architecture.

![Figure 1 - Dreamer high level architecture](image)

In the low level of the Dreamer architecture shown in Figure 1 we can find a set of Hybrid IP/SDN nodes, which are interconnected with layer 2 links (e.g. Ethernet). Incoming packets are processed as regular IP packets (including IP routing protocols) or as SDN tunneled traffic depending on information in the packets. Likewise it is possible for an IP/SDN router to forward an outgoing packet using regular IP or using an SDN Based Path. The identification of SDN tunneled traffic can be based on different approaches, for example we could simply reuse MPLS labels, or we could...
use other tagging approaches like VLANs, Q in Q, Ethernet PBB or we could even use layer 2 protocol type identification. This choice will also depend on the different layer 2 technologies supported. For the objective that we have stated, each of these mechanisms is basically equivalent. In the DREAMER prototype we will select the simplest solution that will satisfy our experimentation needs. At the middle level of Figure 1 we find the Controller instances, they can provide a quick reaction to fault events, and together they take care of the network-wide global view. We choose as network OS the open source ONOS, developed by ONLAB. At the time of writing, it is the only open source project that supports as baseline the distribution and coordination of different Controller instances. In the upper level of Figure 1 there is the management server, useful to manage in suitable way the IP/SDN network.

This technical report addresses: i) the realization of the Hybrid IP/SDN node called OSHI; ii) the definition of all mechanisms necessary to deploy a hybrid IP/SDN network (coexistence and ingress/egress classification); iii) the complete deployment of the hybrid IP/SDN network on three environments: Virtual BOX run time, Mininet emulation environment and OFELIA/GOFF testbed facility; iv) the realization (and deployment) of a new network service called “Virtual Leased Line”, built leveraging the capacity of our hybrid node), and finally the development of the application called “Topology Deployer”. We have focused our attention on the complete development of the node and the services: from the requirements to the deployment (executed in Virtual Box, Mininet environment and OFELIA facility). We have followed the same steps for the Topology Deployer, moreover we have used it to accelerate the deployment on the OFELIA facility. At the end of this report we present also some performance results about the network node, and the deployed services.
2 Need for an Open Source Hybrid IP/SDN node

Let us consider a single provider domain (see Figure 2), interconnected with other providers using BGP. Within the provider network, an intra-domain routing protocol like OSPF is used. The provider offers Internet access to its customers, as well as other transport services (e.g. layer 2 connectivity services or more in general VPNs). We reuse the terminology used in the IP/MPLS context, identifying Core Routers (CR), Provider Edge routers (PE) and Customer Edge routers (CE). The provider network includes a set of geographically distributed core and edge routers interconnected by point to point links (POS, GBE, 10GBE…) or by legacy switched LAN (including VLANs). Most often, a provider network integrates IP and MPLS technologies. MPLS creates tunnels (LSP – Label Switched Path) among routers. On one hand, this can be used to improve the forwarding of regular IP traffic with: i) traffic engineering, ii) fault protection and iii) avoiding the distribution of the full BGP routing table to intra-domain transit routers; with performance similar to the one provide by SDH technologies. On the other hand, MPLS tunnels are used to offer VPNs and layer 2 connectivity services to customers.

Let us consider the migration of a provider network to SDN. IP core and access routers could be replaced by SDN Capable Switches (SCS), giving the possibility of realizing advanced and innovative
services and/or optimizing the provisioning of the existing ones. The introduction of SDN in IP backbones requires the coexistence of regular IP forwarding/routing and SDN based forwarding for different types of advanced services (VPNs, Virtual Leased Lines, Traffic Engineering...). The migration paths should foresee the coexistence of IP and SDN based services in a hybrid IP/SDN scenario (resembling the current co-existence of IP and MPLS). In this migration scenario a set of hybrid IP/SDN nodes are capable of acting as plain IP routers (running the legacy IP routing protocols), as well as SDN capable nodes, under the control of SDN controllers. We observe that IP/MPLS control and forwarding plane have been optimized in the years and are capable to operate on large scale carrier networks, while SDN technology has not reached the same maturity level. The reason to migrate towards SDN is not related to performances (for now), but rather to the openness of the SDN approach which could ease the innovation.

now we want consider similar effort in this field (presented in literature), and compare these solutions with our vision.

Some geographically distributed research networks, such as OFELIA [3] [4] in EU, GENI and Internet2 in US, various campus networks and few real deployments (e.g. Google B4 WAN [13]) are currently based on OpenFlow (OF). The initial approach, focused on campus networks, has slightly become more and more datacenter and WAN oriented. However, while in small infrastructures a fully OF-based solution can be feasible and scalable, an ISP networks requires a more sophisticated approach that leverages also on distributed protocols and is interconnected with L2/L3 “standard” networks. Google B4 WAN can be considered as a “hybrid” effort for the coexistence of IP routing and SDN forwarding and the “fall back” to IP forwarding in some circumstances.

With OpenLSR [9] the authors presented an open-source Label Switching Router that generates OSPF and LDP packets using Quagga and thus computes the MPLS labels that are installed in the switches using the OF protocol. This effort want to realize a open source MPLS router, leveraging COTS (commercial off the shelf) hardware (NetFPGA) and open source software suitable modified. This architecture provides a distributed “standard” control plane, while it uses OF only locally in a node to synchronize the FIBs and to program the data plane. The paper is only focused on MPLS services, considering the SDN approach as means to realize a label switch router.

Instead Routeflow [10] leverages on Quagga to compute the routing that is eventually installed into the hardware switches via the OF protocol. To achieve this effort, Routeflow creates a
simulated network, copy of the physical one, in the OF controller machine, and exploits distributed protocols, such as OSPF, BGP, IS-IS between the virtual routers. Eventually, the OF controller analyzes the virtual routing tables, translates them into OF rules that are pushed toward the switches. With this solution the IP routing logic is extracted from each node and executed in the centralized controller.

![Figure 3 – Routeflow approach](image)

Authors in [11] have described in detail a solution to expose OF network to Internet using BGP. In the exposed scenario, the whole set of OF switches is seen as a single IP router from the perspective of external peers and intra-domain IP routing is fully replaced with OF rules.

Compared with these works, our approach does not remove IP routing protocol processing from backbone nodes, but it considers hybrid IP/SDN nodes, also capable of dealing with IP routing, so our solution represents a novelty in the field of hybrid node. With our approach we achieve easier interoperability with non-OF devices in the core of the network and native fault-tolerance based on the regular IP routing. Moreover we can deploy a new set of services unfeasible with other approach (otherwise feasible with a higher level of difficulty), leveraging the abstraction provided by SDN approach.

The idea of Hybrid IP/SDN nodes is obviously not new. According to the OF1.X specifications [12], two types of switches are supported: OF-only and OF-hybrid. While in the first type all the packets are processed only by the OF pipeline, the second supports both the OF processing and the standard L2/L3 functionalities, such as Ethernet switching, ACLs, QoS, L3 routing, MPLS. Currently, only proprietary hardware switches implement the hybrid approach, with L3 “standard” routing capabilities; the work realized in the context of this technical report analyzes and implements a fully open-source OF-hybrid solution designed to be flexible and scalable.
3 Definition of a Hybrid IP/SDN network

In current IP/MPLS scenario, there is a clear notion of what is a MPLS tunnels, called in jargon LSPs (Label Switched Paths). In a SDN network several types of tunnels or more generically network paths can be created, exploiting the deep inspection capability of the network equipment and thus the various fields of different protocols (TCP/UDP, IP, VLANs, Ethernet, MPLS, ...). There is not a standard established terminology for such concept, we will refer to these network paths as: “SDN Based Paths”, in short SBP. A SBP is a “virtual circuit” which is setup using SDN technology to forward a packets flow between two SBP end-points across a set of SDN capable nodes. The notion of packets flow is very broad and it can range from a “mouse”-flow i.e. a specific TCP connection between two hosts, to an “elephant”-flow e.g. a collection of traffic among different subnets. A flow can be identified looking at headers at different protocol levels.

A Hybrid network includes of a set of Hybrid IP/SDN Nodes and may also include regular IP routers, traditional layer 2 switches and non-Hybrid SDN nodes. These nodes interwork in order to provide a set of services. We address the definition of this Hybrid IP/SDN (H-IP/SDN) network by considering: i) the mechanisms for co-existence of regular IP traffic and SBPs; ii) the set of services (realized and realizable in short term) that can be offered using the SBPs; iii) the ingress classification and tunneling mechanisms. We analyze these three elements first in a high level perspective, then in the following paragraphs we focus our attention (giving also implementation details) on the subset of functionalities that we have already implemented and demonstrated in our Open Source experimental setups.

Let us first consider the mechanisms necessary to assure the coexistence of regular IP traffic and SBP on the links between Hybrid IP/SDN nodes. Considering for example the IP/MPLS model, two mechanisms have been defined in order to assure the co-existence of IP routed traffic and of MPLS label based forwarding: 1) leverage the protocol field of the L2 protocol to distinguish whether the L3 protocol is “IP” of “MPLS”; 2) all traffic is forwarded using MPLS labels, with a special null label for traffic that needs to be handled at IP level in the router. The first solution does not make sense in an IP/SDN architecture because there is not a Layer 2.5 “SDN protocol” comparable to MPLS, the second one could be re-usable in our approach, assuming to use MPLS encapsulation for all SBPs and for IP basic traffic, and using the null label to identify the IP packets. On the other hand, a SDN approach offers a great flexibility and can classify packets with a “cross-layer” approach, by considering packet headers at different protocol levels (MPLS, VLANs, Q-in-Q, Mac-in-Mac and so
Therefore, leveraging SDN approach, it is possible to specify a set of conditions regarding flows that have to be handled at IP level and the ones to be handled using SDN. These conditions can be in the form of white lists / black lists and can change dynamically, interface by interface. On the other hand this flexibility may turn into high complexity, therefore the risk of misconfigurations and routing errors should be properly taken into account (see for example [14]). This could suggest to limit the possible options in the operational practice, selecting a specific tunneling mechanism to create the SDN Based Paths.

Hereafter we identify a set of possible coexistence approaches that can be taken singularly or properly combined.

In the first table we report major details about the VLAN based approaches.

<table>
<thead>
<tr>
<th>#</th>
<th>VLAN based approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coex-A</td>
<td>A specific VLAN ID is used for IP traffic, the others are used for SDN Based Paths.</td>
</tr>
<tr>
<td>Coex-B</td>
<td>Traffic with no VLAN (i.e. layer 2 protocol field &lt;&gt; 0x8100) is IP, all VLAN ids are used for identify traffic belonging to the SBPs.</td>
</tr>
<tr>
<td>Coex-C</td>
<td>Traffic of a given set of layer 2 protocol field is IP (i.e. layer 2 protocol is IP 0x800 or is ARP 0x806), all the rest is considered as SDN traffic belonging to SBPs.</td>
</tr>
</tbody>
</table>

**Table 1 – Coexistence mechanism: VLAN approach**

These solutions are well supported by the early versions of OpenFlow (the ones that are reliably implemented in switches and controllers at the time of writing). A practical disadvantage is that number of the SBPs that can be supported per each link is limited by the number of VLAN IDs (4096). These solutions may interfere with VLANs used in the links between Hybrid IP/SDN nodes. This is not a problem if such links are point-to-point links, where VLANs are not used. It can be a problem if two Hybrid IP/SDN routers are interconnected over a switched LAN using VLANs. In this case a set of VLANs should be reserved for SBPs and cannot be used for regular VLANs. Note also that in these solutions, the SDN Based Path cannot be used to support a Pseudo-wire service i.e. a fully transparent Layer 2 tunnel between two end-points. The tunnel end points cannot use VLANs but only plain Ethernet, as the VLAN tag would be re-written.

Now we take into account the solutions using a Q in Q encapsulation for the Ethernet packets (details in Table 2):
### Q in Q approaches

<table>
<thead>
<tr>
<th>#</th>
<th><strong>Coex-D</strong></th>
<th>Double VLAN ID, the external Tag is used for distinguishing between IP traffic and SDN Based Paths and to create the SBPs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td><strong>Coex-E</strong></td>
<td>The external and internal VLAN IDs are used jointly to create the SBP.</td>
</tr>
</tbody>
</table>

**Table 2 – Coexistence mechanism: Q in Q approach**

The former approach (Coex-D) allows to offer a Layer 2 Virtual Leased Line to the SBP end-points that can be used to transport VLANs (but does not assure full layer 2 transparency as Q-in-Q approach cannot be used by the SBP end point). The number of SBPs that can be used on a given link is limited to 4096. The Coex-E approach overcomes this limitation, allowing to setup $2^{24}$ SBPs on each link. On the other hand in this case VLANs cannot be used by the end-points of the SBT.

A MPLS approach is another viable solution, as shown in the following table:

<table>
<thead>
<tr>
<th>#</th>
<th><strong>MPLS based approaches</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td><strong>Coex-F</strong></td>
</tr>
<tr>
<td>#</td>
<td><strong>Coex-G</strong></td>
</tr>
<tr>
<td>#</td>
<td><strong>Coex-H</strong></td>
</tr>
</tbody>
</table>

**Table 3 – Coexistence mechanism: MPLS approach**

In these solutions, all the traffic needs to be encapsulated in MPLS, an immediate practical disadvantage is that MPLS is not supported in early releases of OpenFlow (the ones that are currently supported by switches and controllers). These solutions offer the great advantage that a transparent “pseudo-wire” layer 2 service can be offered to SBP end-points.

Ethernet based approach are reported in **Table 4**.

<table>
<thead>
<tr>
<th>#</th>
<th><strong>Ethernet based approaches</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td><strong>Coex-I</strong></td>
</tr>
<tr>
<td>#</td>
<td><strong>Coex-L</strong></td>
</tr>
</tbody>
</table>

**Table 4 – Coexistence mechanism: Ethernet based approach**

There is a practical difficulty with the first approaches, which is not supported in the early releases of OpenFlow. There are operational problems with the second approach, because this works for a network with only point-to-point Ethernet links. It does not work well if there are legacy switches among two hybrid IP/SDN nodes, because also the broadcast should be handled taking into account the classes of source MACs.
For the definition of the concrete architecture to be implemented in our demonstrator, we took into account the limitations of the available open source tools (in our case Open vSwitch and Quagga) in terms of tunnels handling and of compliance to OpenFlow protocol releases. While we are writing there is not a stable release that implements the MPLS action both in available SDN capable switch (SCS) suite and in Routing suite. Therefore we have decided first to use VLAN tags as IP/SDN coexistence mechanisms. We have selected two solutions: i) Coex-A, in which the IP traffic travels in the network with a specific VLAN tag (for example the VLAN ID “1”) while SBPs use other VLAN tags (Tagged coexistence); ii) Coex-B, in which the IP traffic travels “untagged” and SBPs use VLAN tags (Untagged coexistence). Note that the reference scenario considers an IP wide area network which is not meant to work globally as a switched Ethernet network: the VLAN tags are used “locally” on the layer 2 links between IP routers and make sense only between them. This information should be changed hop by hop as it happens with MPLS and the LSPs.

Let us now consider the services and features that can be offered by the Hybrid IP/SDN network. As of writing, we only designed and implemented an “Ethernet Virtual Leased Line” (VLL), leveraging our hybrid approach. This service guarantees to the served end-points to be directly interconnected as if they were in the same Ethernet LAN. Note that we do not offer a fully transparent pseudo-wire service: the served SBP end-points cannot use VLANs on the VLL link. Our implementation offers services at an edge router, the end-point can be a physical port of the edge router or a logical port, identified by a VLAN tag. Two arbitrary end-points in edge routers can be bridged by the offered VLL service. The interconnection is realized in our Hybrid IP/SDN network with a SBP using VLAN tags switching in the “core” network (see Figure 4). As future work we are now considering the offering of an Ethernet Virtual Switch, in which several end-points can be transparently bridged into a virtual switch offered by the Hybrid IP/SDN network.

Let us finally consider the ingress classification and tunneling functionality, which need to be performed in an “Edge” Hybrid IP/SDN node at the border of an Hybrid IP/SDN network (shown as
The PE – Provider Edge in Figure 2). The PE needs to perform “ingress” and “egress” functionality in order to map regular IP or layer 2 traffic coming from external networks (e.g. access networks) that use legacy technology and are not capable of working in Hybrid IP/SDN mode into the services offered by the Hybrid IP/SDN network. In a general perspective, the ingress edge router will need to classify incoming traffic considering a large set of parameters (among these, input ports, VLAN tags, MPLS-like Forwarding Equivalence Classes on IP source and destination addresses, transport level ports...). When talking about the “ingress functionalities” is useful to distinguish these functionalities in two distinct parts: the ingress classification mechanism, and the ingress/egress tunneling. The former one is the component which really classifies the traffic and distinguishes for example regular IP from traffic to be tunneled into SBP. The tunneling component, considering the output of the classification component, executes the proper operations on the traffic (for example VLAN tag pushing/popping or MPLS tag pushing/popping) depending on the coexistence mechanism deployed in the Hybrid network. We envisaged mechanisms that can be used in order to build the “ingress” and “egress” functions, the key difference among them is the role played by the routing component and the switching component in the PE node and the services that can be realized. Hereafter we report a set of feasible approaches:

The first ingress classification solution is called “IP based” approach. In this solution the traffic from the incoming ports of the PE is connected to the routing engine of OSHI PE node and the classification is performed during the IP routing process. This approach is similar to the FEC (Forwarding Equivalence Class) processing in an IP/MPLS node. In fact, the incoming IP traffic is classified in FECs by the routing engine, looking at information in the IP header (mostly the IP destination address). The traffic that is classified as belonging to some FECs can be mapped into LSPs.

<table>
<thead>
<tr>
<th>#</th>
<th>IP based approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ingr-A</td>
<td>The traffic is collected through a plain incoming IP interface and then processed by the routing component of the hybrid node. The classification is done during or after the routing.</td>
</tr>
</tbody>
</table>

**Table 5 – Ingress classification: Access IP interface approach**

A possible implementation of this solution considering our node architecture is to perform the classification after the routing, in the SDN Capable Switch. The traffic is forwarded towards an interface according to the routing decision and enters the SDN Capable Switch. If the traffic belongs to a SBP, it needs to be “intercepted” by some rules in the SDN Capable Switch. The traffic
can be classified using parameters of the upper layers (starting from the layer 3). In this case, it will be redirected to the correct interface (according to the active SDN service). On the other hand if it is IP traffic, it will be simply forwarded according to the routing decision. For both IP and SBP traffic, depending on the coexistence and tunnelling mechanisms used in the hybrid IP/SDN core some tunnelling actions can be performed on the packets (e.g. pushing/popping of tags).

Depending on the design choices the classification based on upper layers (layer 3 and above) can become difficult to manage due to the large number of the micro flows to be handled. Another side effect that derives from this approach is the impossibility to deploy layer 2 services.

The second set of solutions is called “Port/VLAN based” approach as show in Table 6. These solutions resemble the classification approach in a VLAN aware switch. A VLAN aware switch can have access and trunk ports: the traffic on an access ports is automatically mapped in a VLAN, the traffic over a trunk port is differentiated by its VLAN tag.

<table>
<thead>
<tr>
<th>#</th>
<th>Port/VLAN based approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ingr-B</td>
<td>The access traffic is classified considering the physical access port of the PE.</td>
</tr>
<tr>
<td>Ingr-C</td>
<td>The access traffic is classified considering the VLAN tag.</td>
</tr>
</tbody>
</table>

*Table 6 – Ingress classification: Port/VLAN based approach*

The Ingr-B solution resembles the “access port” approach of a VLAN switch. All traffic on a physical port will be either best effort IP traffic or it will belong to a VLL. If it is IP traffic, it will be forwarded towards a logical IP interface of the PE for the L3 processing and then redirected towards another interface according to the routing functionality. If it is necessary, some tunnelling actions can be performed on the packets for forwarding them in the Hybrid IP/SDN network, like for example pushing/popping tags or rewriting them. If the incoming port is associated to a VLL its traffic will be handled by the SDN Capable Switch in the PE and will be forwarded using the proper SDN Based Path (SBP).

In the Ingr-C solution the traffic on a given PE port is classified based on the VLAN tag. A given VLAN tag can be associated to IP best effort traffic, in this case the traffic will be forwarded towards a logical IP interface of the PE for the L3 procession. The other VLAN tags can be associated to different VLLs. The traffic using these other VLAN tags will be handled by the SDN Capable Switch in the PE and will be forwarded using the proper SDN Based Paths (SBPs). This approach is conceptually equivalent to the one used in a “trunk port” of a VLAN aware switch. As
an extension we could have that untagged traffic can be mapped into the IP best effort traffic (like in a hybrid port of a VLAN aware switch).

In both Ingr-B and Ingr-C solutions after that the traffic has been classified some tunnelling actions can be performed on the packets (e.g. pushing/popping of tags) depending on the coexistence and tunnelling mechanisms used in the hybrid IP/SDN core.

The Port/VLAN based solutions (Ingr-B and Ingr-C) gives the possibility to deploy L2 services (like the VLLs).

The major difference among the IP based approach (Ingr-A) and the port/VLAN based approach (Ingr-B and Ingr-C) is the relative importance given to the two components of the OSHI PE node, and the services that we are able to deploy. Indeed in the IP based approach the routing function has greater importance respect to the switching component (that is the OpenFlow Capable Switch component), and at the same time we are not able to provide L2 service like VLLs. On the other hand in the port/VLAN based approach we move “into the background” the routing function, this also means that the classification mechanisms are delegated to the SDN Capable Switch component. Using this approach we can deploy L2 services like VLL.

Note that the choice of the classification approach used in the access is orthogonal to the coexistence mechanism chosen for the Hybrid IP/SDN core. As a results of this processing (classification and tunneling) the access OSHI will encapsulate traffic in the SBPs. The egress edge router will extract the traffic from the SBP and forward it to the appropriate destination (first executing the classification and eventually de-capsulation).

### 3.1 Implementation and testbed aspects.

We designed the ingress classification mechanisms in our testbed taking into account the requirement of VLL service and also the capability of the available tools in terms of tunneling handling and compliance to OF protocol releases.

Coming to our open source implementation, let us consider the user traffic entering in a port of a Provider Edge OSHI router. Untagged traffic can be classified as regular IP traffic or as belonging to the end-point of a VLL. For VLAN tagged traffic, each VLAN tag can be linked to a VLL end point and one of the VLAN tags can be assigned to regular IP traffic.
4 OSHI requirements and analysis

Obliviously, the starting point is to provide services meeting carrier grade requirements. In the Telco jargon “carrier grade” or “carrier class” refers to a system or a hardware or a software that is extremely reliable, well tested and proven in its capabilities. Carrier grade systems are tested and engineered to meet or exceed "five nines" high availability standards, and provide very fast fault recovery through redundancy (normally less than 50 milliseconds). To give an example we should obtain performance and mechanisms similar the one provided by the SDH technologies.

The implementation of the prototype node needs to leverage the available Open Source tools for routing engines and OpenFlow enabled software switches. Examples are Quagga [7] or Bird [15] for the routing functionality and Open vSwitch for the SDN Capable Switch. This choice could not provide the best performance compared with the specialized equipment, but at this point we are not interested in the absolute performance, a problem which could be dealt with in the future.

Another important requirement is the incremental deployment and the backward compatibility, we want to obtain a seamless integration with the existing solutions. At the same time there is the need for a migration strategy that does not create a closed ecosystem, but one capable to interwork with the current technologies. The solution has to start from the current deployed architecture, does not require a “zero-day” in which the old infrastructure needs to be replaced with a new one based on layer 2 SDN based equipment. In our vision SDN is seen as an enabler for value-added services, for example it could be used to provide a higher level of protection for the most important traffic with fast recovery mechanisms. The “compatibility” is a very important requirement, because nowadays with the ossification of the modern networks it is very difficult to introduce changes.

Now we take account operational and services requirements of our hybrid node, this requirements derive directly from the needs of the Italian NREN GARR, that represents our stakeholder. As we have stated in the previous sections, our OSHI Node fits in a Hybrid network scenario, where the current solutions have to be replaced by SDN functionalities, giving the possibility to realize advanced and innovative services and/or to optimize the provisioning of existing ones. In our view, there is the coexistence of current IP based services and new services based on SDN concepts, in an integrated IP/SDN scenario. So in this environment our node has the arduous task to be the means with which to make real this “hybrid” vision, as it happens today with the MPLS enabled router. Our target solution must support the coexistence of IP best effort
services and SDN Based Paths (SBPs). BGP is used as inter-domain routing, OSPF (or the like) is used as intra-domain routing for the IP services in the IP/SDN transport network. IP best effort connectivity services are offered towards customer ASs, exporting BGP routing. OSHI must support the creation and deletion of the SBPs, that can be used to hide the full routing table in internal nodes and to implement innovative services (like the Virtual Leased Lines or Virtual Switches). OSHI can be used also for traffic engineering purposes to carry the IP best effort traffic in a SBP; the idea is to switch dynamically the IP packets in the SBP over a different path with respect to the one chosen by the intra-domain protocol (in our solution it will be OSPF). SBPs will need to provide fast restoration mechanisms (with performance requirements in order of tens millisecond) for regular IP best effort traffic or special traffic (identified in some way, in this sense we have great flexibility thanks to the SDN approach). OSHI has to offer also the support for VPNs and L2 transport services, in particular: SBP can be used to offer VPNs or L2 transport service to a third party, moreover it will be possible to setup end-to-end SBP (SDN Based Paths) to offer VPNs and L2 transport services, both with default routing and with TE routing; obviously we have to guarantee also protection mechanism of SBP (SDN Based Paths) used to offer VPNs and L2 transport services. Another aspect that we have to cover also the monitoring: traffic monitoring of SBP (SDN Based Paths) has to be supported.

Hereafter, we report an images showing high level architecture envisaged for our hybrid node:
The architecture is very simple and straightforward, we analyze its components starting from the low level: we introduce a classifier that receives all the traffic L2, this classifier influences only the in packets, indeed it includes the rules useful to classify the incoming packets as IP or SDN traffic. If a packet matches the IP rule, it will be redirected towards the routing function (green components), and finally forwarded through an interface using the routing table, built by the routing daemon. On the other hand if the packet does not match the rule for IP traffic, it will be redirected towards the switching components (OVS), and from there forwarded either through a rule in the OVS table, or through a command sent by Controller. In this design the IP traffic outgoing from the IP routing engine is not affected by the classifier, only the incoming packets. In order to provide a fault tolerant solution, we have envisaged at least two connections towards different OF controllers: the first active (master connection) and the second as backup (dashed line).
5 OSHI deployment

We have planned and realized the complete deployment of the OSHI node in four phases, during which we have covered all the aspects of the OSHI lifecycle and the complete deployment of an Hybrid network (in small scale):

- Phase 1: we have explored the possible solutions for the correct interworking between the routing and switching component; we have realized an early prototype and we have tested it on VBOX and on Mininet;

- Phase 2: we have refined the design of the early prototype and we have shifted our attention on the design and the deployment of the OSHI Core Node, moreover we have also designed and realized the coexistence mechanism previously presented. We have tested them on VBOX and on Mininet;

- Phase 3: we have taken care of the AOSHI deployment, we have also realized the envisaged classification function. At the end of this task first we have tested the functionality of the access node, then the complete solution on more complex topologies. Once we have completed this task, we have focused on the realization and deploy of the VLL service. We have validated the VLL service on the same topology used previously.

- Phase 4: In the last phase we have shifted our attention on the deploy in the OFELIA facility; we have realized different complete deploys on OFELIA Testbed, the last is the one that we have used to run several tests, whose results will be presented in the section 11
6 OSHI Node - Architecture and implementation details

We designed our Open Source Hybrid IP/SDN (OSHI) node combining a SDN Capable Switch (SCS), an IP forwarding engine and an IP routing daemon. The SDN Capable Switch is connected to the set of physical network interfaces belonging to the integrated IP/SDN network, while the IP forwarding engine is connected to a set of virtual ports of the SCS, as shown in Figure 6. In our OSHI node, the SCS component is implemented using Open vSwitch, the IP forwarding engine is the Linux kernel IP networking and Quagga acts as the routing daemon (in particular we have activated the OSPF daemon).

![Figure 6 – OSHI Hybrid IP/SDN node meta architecture (data plane)](image)

The internal virtual ports that interconnect the SCS with the IP forwarding engine are realized using the “Virtual Port” feature offered by Open vSwitch, these are not real port and they are created on demand. Each virtual port is connected one-to-one to a physical port (for example eth interface) of the IP/SDN network, so that the IP routing engine can reason in term of the virtual ports, ignoring the physical ones. The SCS differentiates among regular IP packets and packets to be processed by SDN. By default, it forwards the regular IP packets from the physical ports to the virtual ports, so that they can be processed by the IP forwarding engine, controlled by the IP routing daemon. This approach avoids the need of translating the IP routing table into SDN rules.
to be pushed in the SCS table. A small performance degradation is introduced because a packet to be forwarded at IP level will cross the SCS switch twice. It is possible to extend our implementation to consider mirroring of the IP routing table into the SCS table, but this is left for further work.

An initial configuration of the SCS tables is needed to connect the physical interfaces and the virtual interfaces, to support the SCS-to-SDN-controller communication, or for specific SDN procedures (for example to perform Layer 2 topology discovery in the SDN controller). Periodically Controller like Floodlight, sends in the network special traffic (LLDP and BLDP) in order to discover the topology of the network, this traffic requires special rules and when it reaches a OVS, it must be sent towards the controller, without these special rules we would break the Topology service of the Controller and we could not use service like Forwarding module. For example in our implementation without a special care this traffic would be sent on the OSHI’s internal interface and would never be sent to the Controller. In order to fix this problem we set special rule on the OSHI, we report in the box below this couple of rule:

```
1 // Fix LLDP and BLDP Issue – OSHI configuration (2)
2 IF ETH.TYPE == 0x8CC // LLDP Proto
3  THEN Send To Controller With Priority=301
   ovs-ofctl add-flow br-one hard_timeout=0,priority=301,dl_type=0x88cc,action=controller
4 IF ETH.TYPE == 0x8942 // Special Packet Send Together With LLDP Used by BigSwitchNetworks
5  THEN Send To Controller With Priority=301
   ovs-ofctl add-flow br-one hard_timeout=0,priority=301,dl_type=0x8942,action=controller
```

**Box 1 - OSHI configuration (2) – Rules necessary to fix LLDP and BLDP issue**

We have reported in the previous box both the rules and the associated OVS commands, in our case the priority fixes our problem because other matching rules have a lower priority value (300), as it will be clearer after. Now we show as the above architecture can be realized using the aforementioned component:

```
1 // OSHI configuration (1)
2 ADD bridge
   ovs-vsctl --no-wait add-br br-one // OVS Command that create a SCS
3 FOR each physical interface (ethi)
4   bridge ethi to br-one
   ovs-vsctl --no-wait add-port br-one ethi // OVS Command that bridge a physical interface
5   create and bridge the internal virtual interface vii
   ovs-vsctl --no-wait add-port br-one vii -- set Interface vi type=internal //OVS command
```

**Box 2 - OSHI configuration (1) – Bridge and vii creation**

After showing the creation of the internal port and the bridge of the ETHi one, now we present as the physical interfaces can be “connected” directly to the virtual interfaces (i.e. we show the initial
configuration of the OF table), note that this rules will be “overridden” by the rules necessary to deploy coexistence mechanisms and ingress classifications. Using this initial configuration we deploy only a simple IP service. In each node we have a double forwarding: IP forwarding and the SCS forwarding. The IP plane forwards towards a Viol (i-th virtual internal) interface according to the routing decision, on the other hand the “second” forwarding takes the packet that comes from a Viol interface and forwards towards the correspondence ETHi. In the “reverse path” when the packet comes from ETHi interface (that is it arrives from Network), the SCS forwards it towards the correspondence Viol, and finally the packet will be processed from the IP plane. We report now in following box the necessary modifies:

```plaintext
1 // OSHI configuration (3)
2 FOR each couple of interface (ETHi, Viol)
3   IF in_port == ETHi
4     THEN forward the packet on the correspondent Viol interface with priority 300
5       ovs-ofctl add-flow br-one hard_timeout=0, priority=300, in_port=ETHi_port, action=output:Viol_port
6   IF in_port == Viol
7     THEN forward the packet on the ETHi interface with priority 300
8       ovs-ofctl add-flow br-one hard_timeout=0, priority=300, in_port=Viol_port, action=output:ETHi_port
```

Box 3 - OSHI configuration (3) – Initial configuration

The last configuration task is the properly configuration of the routing component. This step is very simple and straightforward, on the network there are a lot of tutorial that explains in detail the right configuration of the Quagga software. To give an example in [16] there is a detailed tutorial that drive the user in all the configuration aspects from the simplest one to the more advanced (like loopback configuration). After this configuration need to be adapted according to the approach used for the coexistence of the regular IP traffic and SDN one.

A local management entity in the OSHI node takes care of these management tasks (as shown in Figure 7), in particular: provides the initial configuration of the SCS table, inserts the configuration necessary to implement the coexistence mechanisms, and finally if the node is an access node insert the rules needed to perform the ingress/egress classification mechanisms. Regarding the configuration of the VLLs, this task is completely demanded to the OF Controller (see VLL section):
Figure 7 – OSHI Hybrid IP/SDN node architecture (control/management plane)

On the “control plane” side, the SDN Capable Switch is meant to be controlled by an external SDN controller. The communication between the SCS and the controller must be properly supported. Let us consider the OpenFlow protocol. A TCP connection is needed to the SCS to communicate with the Controller. This can happen “in-band” or “out-band”. In-band means using the same IP network and Layer 2 network that is used for data traffic handled by the switch. Out-band means the control network is separated (virtually or physically) from the data plane network. A common solution for Data Centre SDN and production network is to use a separated VLAN for the control network. The underlying mechanism needed to ensure the connectivity and the routing in the control network impact on the performance of the system, especially on those related to handling of faults. In our setup, it is possible to use an “in-band” approach for SCS-to-SDN-controller, i.e. using the regular IP routing/forwarding, avoiding the need of setting up a separate out-of-band network. Our approach is not a simplification rather in this case represents the worst case.

The in-band approach, for the communication between OSHI and Controllers, needs a special care, and the introduction of changes on the basic behaviour of the controller. Controllers like Floodlight, once they complete the handshake with the OVS, clear all flow mod on the switch, but this action deletes also our flow mod. These are necessary to the proper functioning of our solution, without the aforementioned rule the OSHI cannot be forward in a suitable way all the
traffic, but the worst things is that they cannot longer reaches the controller, hereupon they cannot be forward too all the SDN traffic. All goes down.

The OSHI nodes can be classified into core (COSHI) and access (AOSHI). Coming to our implementation, the Architecture of Core Node can be built from the meta architecture shown in Figure 6, the main differences is the lack of the physical and virtual interfaces directly connected to the forwarding engine and the virtual interfaces connected directly to the SCS. On the other hand the AOSHI architecture is equal to the one of a COSHI, but we can distinguish from a COSHI due to additional functionalities for ingress traffic classification (as we have already explained).

Using the OSHI-LME we complete other tasks not directly related to the routing and the SCS component but to the regular IP networking implementation in Linux kernel. We need to configure carefully other option in each OSHI node in particular: first we need to enable in each OSHI node the IP forwarding behavior (this is necessary for the correct operation of the IP part), finally we need to disable on each Node interface the “RP filter”. We should disable the reverse path filter in all the situation where we could have an equal cost multipath, between the source and the destination. Basically, if the reply to a packet would not go out the interface this packet came in, then this is a “bogus” packet and should be ignored. We can see the packet on TCPdump or in program like Wireshark, but it never reached the socket. For clarify we report this situation in the following image:

![Figure 8 – Bogus packet](image)

The result is a dropped packet and the response will never reach the source. The main functionality of a router is to route packets from one place to another. Linux machine can be used as router on network and it will route substantial amount of traffic without any issues, if configured correctly. Due to the increasing amount of malicious and attack traffic on the internet,
it has become very much necessary to take some extra care while configuring routes on a Linux machine or physical router's. One of the major problem that internet security people are dealing with today, is spoofing. Spoofing can be controlled to a certain extent by using Reverse Path filtering (not fully although). Reverse path filtering is a mechanism adopted by the Linux kernel, as well as most of the networking devices out there to check whether a receiving packet source address is routable. So in other words, when a machine with reverse path filtering enabled receives a packet, the machine will first check whether the source of the received packet is reachable through the interface it came in: if it is routable through the interface which it came, then the machine will accept the packet, otherwise if it is not routable through the interface, which it came, then the machine will drop that packet. In situation shown in the Figure 8 the destination could be reached with both interface. The RP filter, probably considering the path through the destination, check only the first entry (the interface used to send the request) in the Kernel table. As proof of this we experiment in all tests executed the 50 % of losses, because in some cases the response take the same route of the requests. Disabling the RP filter in the interfaces connected to the NET A we resolve our problem, so at the end for our experiments we disable it in all interfaces. In this scenario, we do not take care security issue, so disabling this further control does not influence our implementation.
7 Coexistence mechanisms: Tagged and Untagged approach

Let us discuss some details about coexistence mechanisms, targeting our open source implementation. As for the coexistence mechanisms, we have leveraged on the multiple tables functionality recently implemented in Open vSwitch. The OpenFlow pipeline of every OpenFlow switch contains multiple flow tables, each flow table containing multiple flow entries. The OpenFlow pipeline processing defines how packets interact with those flow tables. The of tables of an OpenFlow switch are sequentially numbered, starting at 0. Pipeline processing always starts at the flow “Table 0”: the packet is first matched against flow entries of flow “Table 0”. Other flow tables may be used depending on the outcome of the match in the first table. When processed by a flow table, the packet is matched against the flow entries of the flow table to select a flow entry. If a flow entry is found, the instruction set included in that flow entry is executed. These instructions may explicitly direct the packet to another flow table (using the GOTO Instruction, or Resubmit), there the same process is repeated again. A flow entry can only direct a packet to a flow table number which is greater than its own flow table number, if the matching flow entry does not direct packets to another flow table, pipeline processing stops at this table. When pipeline processing stops, the packet is processed with its associated action set and usually forwarded. If a packet does not match a OF entry in a OF table, there is a table miss. For more details see OF specification document 1.4.0 [18]. Thus The idea is to have in “Table 0” one rule matching the regular IP and submitting the packets to “Table 1”.

For Tagged coexistence, the rule in “Table 0” matches the specific VLAN tag of regular IP. For Untagged coexistence, this rule matches the untagged traffic (using the OVS command we need to insert a match for tag equal “0xffff”). The rules for SBPs are always in “Table 0”, this choice has been crucial for the correct instauration of the SBP, because as of writing there is not an implementation of Floodlight that supports the Multiple tables functionalities. Instead “Table 1” contains the rules to forward the traffic from the incoming ports to the corresponding virtual ports (and vice-versa). These rules are similar to the one that we provide in the initial configuration to allow the communication with Controller. In case of Tagged coexistence, the virtual interface used by the IP forwarding engine and routing daemon includes the VLAN tag, therefore the SCS does not need to push/pop the VLAN tag. In the following images we report a flow chart that shows
step by step of the packet processing in the SCS flow tables both Tagged approach and Untagged one:

![Flowchart of packet processing in SCS flow tables](image)

**Figure 9 – Coexistence mechanisms: packet processing in the SCS flow tables**

We use a multiple tables approach to realize negative matches, omitting an aesthetic point of view, there are not great differences with our solution and one that uses a single table. As during the lookup in the table the OVS uses a hash function in order to find a match. This applies to rules based on VLAN tag, because for this field there is the possibility of the negative match (match on VLAN id equal “Oxffff”), but in future we can leverage this structure to shift towards solution based on different fields and gain an advantage to realize a negative match (with other fields there is not the possibility to express a match like field <> value, a concrete example is the classificatory 1C).

As we have already explained, in the realization of these mechanism we leverage the capability of the SCS component (we have chosen to give more importance to the OVS), indeed the packet are first processed by the SCS, which one determines the traffic belonging, using the rules added by the LME. These rules implement the behaviour shown in the Figure 9. Now we give more implementation details about this mechanisms. The first approach (1A) is very simple and straightforward, we distinguish the IP traffic from SDN traffic though different VLAN ID and we have to “send” the packet to the “Table 1”: 

---

26
1 // VLAN based approach 1A
2 IF packet contains the VLAN_IP
3 Then resubmit the PIN to the Table 1 with priority 300
   ovs-ofctl add-flow br-one "table=1, hard_timeout=0, priority=300, dl_vlan=VLAN_IP, actions=resubmit(,1)"
4 For each couple of interface (ethi, vii) in the Table 1:
5 IF in_port == ethi && dl_vlan == VLAN_IP
6 THEN forward the packet on the vii interface
   ovs-ofctl add-flow br-one "table=1, hard_timeout=0, priority=300, in_port=ethi_port, action=output:vii_port"
7 IF in_port == vii && dl_vlan == VLAN_IP
8 THEN forward the packet on the ethi interface
   ovs-ofctl add-flow br-one "table=1,hard_timeout=0, priority=300, in_port=vii_port, action=output:ethi_port"

Box 4 - Coexistence Mechanism (1) – Tagged coexistence

In order to complete the configuration we have to set up properly the Quagga software: we bridge the vii interface on the SCS, the coexistence of regular IP traffic from SDN has been has been guaranteed using non overlapping VLAN. So the last step is to create from vii interfaces the Vii.VLAN_IP ones and set as listening interfaces the latter in the configuration files of the routing daemon. This foresight give us the security that only IP traffic reaches the routing daemon, otherwise if the controller sends flood command the SDN traffic could reach routing daemon.

Now we consider the VLAN based approaches 1B, these coexistence mechanism does not request special care for the routing daemon, because the IP traffic does not belong to a VLAN. In this case the routing daemon is in “listening” on the Vii interface. The classification function in these case is similar to the one that we have previously shown, the greater difference is the match of the “resubmitting” rules:
// VLAN based approach 1B
2 IF packet does not contain a VLAN TAG
3  Then resubmit the Packet In to the Table 1 with priority 300
   ovs-ofctl add-flow br-one "table=0, hard_timeout=0, priority=300, dl_vlan=0xffff, actions=resubmit(,1)" // Note that dl_vlan=0xFFFF match all packet that does not contain VLAN tag
4 For each couple of interface (ethi, vii) in the Table 1:
   IF in_port == ethi && dl_vlan == VLAN_IP
5  THEN forward the packet on the vii interface
   ovs-ofctl add-flow br-one "table=1, hard_timeout=0, priority=300, in_port=ethi_port, action=output:vii_port"
6 IF in_port == vii && dl_vlan == VLAN_IP
7  THEN forward the packet on the ethi interface
   ovs-ofctl add-flow br-one "table=1,hard_timeout=0, priority=300, in_port=vii_port, action=output:ethi_port"

Box 5 - Coexistence Mechanism (2) – Untagged coexistence

The same schema can be used to realize the mechanism 1C, note that this schema is not currently implemented in our open source solution, but its realization is simple and straightforward. We have to substitute the first rule with a set of rule that match the Ethernet protocols that we consider IP traffic, we report in this box a little example (here you can see also the advantages of the using multiple tables):

   ovs-ofctl add-flow br-one "table=0, hard_timeout=0, priority=300, eth_proto=0x806, actions=resubmit(,1)" // This OVS command send all arp traffic on to Table=1
   ovs-ofctl add-flow br-one "table=0, hard_timeout=0, priority=300, eth_proto=0x800, actions=resubmit(,1)" // This OVS command send all IP traffic on to Table=1

Box 6 – Coexistence Mechanism (3) – VLAN based approach 1C

We have to take special care of the routing daemon, with this approach, because the 1C schema does not guarantee the separation of the traffic like the approaches 1A and 1B, the problem occurs when arrive SDN traffic on the OSHI node and it does not contain any matching rule. The OVS sends the traffic as PIN towards the Controller (using IP services). When the controller receives this Packet IN takes a decision for the forwarding, and it could send a flood command, this action causes the forwarding of the SDN traffic towards the internal interface (vii), so we have to modify the controller’s basic behavior, when it decides for packet flooding, the switch must flood only on the ETHi interfaces (the physical interfaces, so we can guarantee that the SDN traffic does not reach routing daemon). Otherwise we can take a drastic choice, disabling in the Controller side the forwarding’s module.

The last missing piece is the fix for the Topology discovery traffic, we need to adapt the fix shown in the previous paragraph to the coexistence mechanism used. In the case of Tagged coexistence
we have to put those rules in the “Table 0”, because the LLDP and the BLPD flows without VLAN tag. For the Untagged Coexistence we have to insert the “fixing” rules in the “Table 1” for the same reason.
8 Ingress classification functionalities

The implementation of the ingress classification is configured within the SCS of Access OSHIs using the Local Management Entity present in each OSHI node. By configuring rules in the SCS it is possible to map the traffic on an ingress physical port as follows: i) untagged traffic to a virtual port (for regular IP); ii) untagged traffic to a SBP (for a VLL end-point); iii) VLAN tagged traffic to a virtual port (for regular IP); iv) VLAN tagged traffic to a SBP.

Taking into account the envisaged mechanism for the ingress classification (presented in the section 3) we can associate them to i), ii), iii) and iv) as follows. With the “Port based SCS classification” (2B) the only feasible approaches are i) and ii), indeed with this mechanism the only way to classify the traffic is the ingress port (we do not have other field to leverage for our purpose). The input ports can be associated to IP traffic or a SDN tunnel. On the other hand, considering the “VLAN based SCS classification” (2C), the AOSHI node can map the traffic on an ingress port according to the procedures i), iii) and iv). The approach ii) is not feasible because in the same AOSHI port can arrive both IP traffic and more than one SBP (because the VLAN L2 network multiplexes the different flows on the same port), thus if in the access network the regular traffic “travels” without tag the untagged traffic in the AOSHI node goes always towards a virtual port. Otherwise this traffic is blocked in the L2 network.

In our vision the access and the core networks one are two distinct parts completely independent, both for regular IP and SBP traffic. Indeed if it is possible to have an access network where the regular IP traffic flows without tag, while in the Core network with a VLAN tag (Tagged coexistence). Otherwise it is possible the contrary: the access network uses Tagged coexistence, while in the core network the IP traffic travels “naked”. This characteristics influences the ingress classification mechanisms, indeed in all those cases we have a difference in the coexistence mechanism among access and core network, the ingress mechanism (we have separated the ingress classification in two distinct part, the ingress function and the tunneling function) is not a simple “forward to” but can be something more complex like for example pushing/popping tags in the packet header. This results in a complex configuration of the node, because we have taken into account more than one case. We start to illustrate the cases associate to the regular IP and “Port based SCS classification”:
Considering the first image, the ETH2 interface is the access port, VI2 is the correspondent virtual internal interface, the regular IP is associated to the VLAN 1 (T1 stands for VLAN tag 1). The packets arrive to the AOSHI without tag, they are modified through a “push” VLAN tag action before to be forwarded towards VI2. On the other hand an IP packet designed for the access port is modified with the action “STRIPE_VLAN” before to be sent to ETH2 port. The case represented in the second images is simple and straightforward, we have a correspondence between access and core and the rules are the same of the coexistence mechanism 1B (this is the simple case of an action “forward to”). Now we report in the following box some implementation details of the ingress classification shown in Figure 10:

```plaintext
ovs-ofctl add-flow br-one "table=0, hard_timeout=0, priority=300, in_port=ETH2, actions=
mod_vlan_id:1, resubmit(,1) // This rule matches the traffic from ETH2
ovs-ofctl add-flow br-one "table=1, hard_timeout=0, priority=300, in_port=VI2, actions=
strip_vlan, output:ETH2 // This rule matches the traffic from VI2
```

**Box 7 - Port based SCS classification and Tagged coexistence**

This rules are inserted after the one of the coexistence mechanism and “overridden” in part (only for the access port) their behavior. Now we consider the cases associated to the approach L2 network VLAN aware and regular IP traffic:
Considering the above images, the ETH2 is the access port of the AOSHI node, VLAN 1 (T1 stands for VLAN tag 1) is assigned for the IP traffic (when IP travels with tag) while the other VLAN (Tags 2 and 3, T2 and T3 in the images) are assigned to SBPs. The ports with a single tag associated of L2 switch are Access ports (push/pop VLAN tag), on the other hand the port connected to ETH2 is a Trunk port. When IP travels without tag in the L2 network (ports without tag), we set for the access port the TAG 0 (it represents in Jargon the default VLAN, see Figure 14 and Figure 15), in this case the port does not push/pop tag. We do a similar configuration for the Trunk port that will be associated to T2, T3 and default VLAN (this means that traffic T2, T3 or without tag is not blocked from this port). Let us consider the cases 1 and 4 represented in the Figure 12 and in the Figure 15. These are simple and straightforward, we have a correspondence between access and core and the rules are the same of the coexistence mechanism 1A and 1B (these are the simple case of an action “forward to”). The case 2 is a little bit more complex because in the access network the regular IP traffic travels with a TAG while in the core it should not contain any tag. Thus in the AOSHI, before to map this traffic to the Vii port, we have to strip the VLAN tag from packets that arrive from access network, on the other hand the packets that come from the core
have to be modified (push action) with the TAG chosen in the access network. Now we report some implementation details of the operation executed in the AOSHI node:

```
ovs-ofctl add-flow br-one "table=0, hard_timeout=0, priority=300, in_port=ETH2, dl_vlan= 1,
actions= strip_vlan, resubmit(1) // This rule matches the access traffic
```

```
mod_vlan_vid:1, output:ETH2 // This rule matches the traffic from VI2
```

| Box 8 - VLAN based SCS classification and Untagged coexistence (1) |

Let us consider the case 3, the access network treats the IP traffic as belonging to the default VLAN (indeed the access port and the trunk port are both WHITE), on the other hand the core network uses as coexistence mechanism the Tagged approach. As a result of these choices, before to map this traffic to the VII port, we have to push the VLAN tag in the packets that arrive from access network, on the other hand the packets that come from the core have to be modified (strip action) because in the access network this traffic should travel without tag. Now we report some implementation details of the operation executed in the AOSHI node:

```
ovs-ofctl add-flow br-one "table=0, hard_timeout=0, priority=300, in_port=ETH2, dl_vlan= 0xffff,
actions=mod_vlan_vid:1, resubmit(1) // This rule matches the access traffic
```

```
ovs-ofctl add-flow br-one "table=1, hard_timeout=0, priority=300, in_port=VI2, dl_vlan= 1,
actions=strip_vlan, output:ETH2 // This rule matches the traffic from VI2
```

| Box 9 - VLAN based SCS classification and Tagged coexistence (2) |

As example we report also the setting of the L2 switch in the access network. Let us consider the case 1, we identify as X,Y,Z the Access ports (monochrome ports) while the Trunk port is the W. In our open source implementation we use as L2 bridge Open vSwitch, configured to work in the standalone mode.

```
ovs-vsctl set-fail-mode L2Bridge standalone // This is a work around
```

```
ovs-vsctl set port X tag=1 // Access port configuration
```

```
ovs-vsctl set port Y tag=2 // Access port configuration
```

```
ovs-vsctl set port Z tag=3 // Access port configuration
```

```
ovs-vsctl set port W trunk=1, 2, 3 // Trunk port configuration
```

| Box 10 - L2 switch configuration steps |

Considering the above configuration, there is a source (or more than one) of regular IP traffic attached to the port X, one side of the SBP GREEN and BLUE respectively attached to the port Y and Z.
Ethernet Virtual Leased Line service

The Ethernet Virtual Leased Line (VLL) service is the first service that we have designed leveraging the functionalities offered by a SDN controller. We plan to design other services in the near future, for example an Ethernet Virtual Switch. The VLL service is implemented with a SBP that switches VLAN tags between two end-points (in both directions). We have to distinguish two cases in the creation of the SBP: i) the endpoints of the SBP are directly interconnected to the AOSHI node, ii) the endpoints are interconnected to the access node through a VLAN aware L2 network. In the former case the ingress classification of the AOSHI maps the untagged traffic to the SBP using the input port. In the latter case the mapping is performed leveraging the VLAN tag pushed in the L2 network. Thus for the correct “instauration” of this virtual circuit we need first properly configure the access network, as we have already explained in the previous section. The creation of the SBP is performed using a python script called VLL Pusher, it “pushes” through the Controller all the rules necessary (also the ingress classification ones) to build the end to end VLL. It reads from a configuration file (vll_pusher.cfg) the information needed for the VLL construction: the AOSHI nodes, the access interfaces, and finally the access VLAN tag. Obtained this information it translates the interface names in port number (this is necessary because the OF rules accepts only the port number) using the REST API “/wm/core/contrroller/switches/json” of the Controller. This API provides a JSON object containing a complete description of all network equipment. Controllers like Floodlight maintains a topological database of the network (built using the LLDP and BLDP traffic). This can be queried from the internal or from the northbound API to obtain for example a route between a two endpoint (the endpoint is always expressed as pair of switch and port number). The VLL pusher uses the Topology REST API /wm/topology/route/X/Y/Z/W/ of the Floodlight controller in order to retrieve the route that interconnects the VLL end-points. The variables X and Z are the access node, while Y and Z the relative access port. Obtained the route allocates the VLAN tags hop by hop (following the route) and then uses the Static Flow Pusher REST API to set the rules for packet forwarding and VLAN tag switching. Now we report an image that try to resume the above operations:
Considering the above images, T1, T2, T3, T4 stand respectively for VLAN tag 1, VLAN tag 2, VLAN tag 3 and VLAN tag 4. The first and last tag are 0 (equals no VLAN tag, i.e. default VLAN), if the endpoint is directly attached to AOSHI (no layer 2 network VLAN aware).

Let us consider the label switch operation executed hop by hop, we have emulated the behavior of MPLS and ATM. The norm is that the inserted label is the one associated to the input port of the next hop, then there are two special case that we have take into account: one hop route, and the ingress/egress node. The former is the case of two endpoints connected to a different access port of the same AOSHI. The OSHI node switches the incoming label with the one associated to the output port. The latter is the situation of a route with at least two hops, considering the case of the egress node (for the ingress it is symmetric): i) if it receives a packet (belonging to SBP) from the core network, it switches the label with the one associated to the output port; ii) if it receives a SBP packet from the access network, it modifies the label with the one of the next hop input port. In the following images we show the above situation and the label switching operation performed in the AOSHI node (T1, T2, T3 stand respectively for VLAN tag 1, VLAN tag 2, and VLAN tag 3):
As we have shown in the flow chart of the Figure 16, the AOSHI’s rules are inserted out of the loop, because we have to insert in the AOSHI node the ingress classification rules, moreover we have to treat as special case the “one hop” route. On the other hand COSHI nodes needs only the rules necessary to complete the coexistence mechanisms. Now we report in the following image the label switching operation performed by a COSHI node (as before T1, T2, T3, T4 stand for VLAN tag 1, VLAN tag 2, VLAN tag 3 and VLAN tag 4):
10 OSHI emulation tools

We realized our OSHI node for three target environments: Virtual Box, Mininet emulator and the OFELIA testbed. We used the Virtual Box deployment to emulate small setups with two or three OSHI nodes. We have developed also a graphical editor tool in JavaScript TopoDesigner, which allows to design a network topology and to configure the services. TopoDesigner exports the created topology in JSON format. It is also possible to synthetically generate a topology using Networkx [8], a Python package for the creation/manipulation of complex networks. A set of python scripts (Topology Deployer) parse the topology file and deploy the experiment over Mininet or OFELIA. This includes the automatic configuration of IP addresses and of dynamic routing (OSPF daemons) in all nodes, therefore it relieves the experimenter from a huge configuration effort. In the below image we show the emulation workflow previously presented:

![Emulation workflow diagram](image)

**Figure 20 – Emulation workflow**

In this scenario the Virtual Box does not fits, as we have already explained, we are able to setup only small experiments with this environment, the aim of the experiment on Virtual Box is only to test in the first instance the properly operation of the components in a controlled environment. On the other hand we are able to set experiments of arbitrary dimension (on OFELIA the dimension depends on the available VM, while on Mininet depends on the available resources on the PC) with the deployers and it make sense to integrate them into this framework.
10.1 Experiments on Virtual BOX run time

The experiments on Virtual Box do not scale well, because the overhead, introduced by a single virtual machine, is very high, and this limits the “dimension” of an experiment in a regular personal computer,. For “dimension” we intend to the size of the topology, that is the number of the OSHI nodes deployed. However these experiments for us have been very important for the fast prototyping of our solution and first validation test.

At first, we have deployed our open source implementation in this environment and we have performed several tests in order to verify the correctness of our early prototypes. Only after we have shifted our attention on the other environments. In the experiments based on Virtual Box we do not have the same flexibility of a simulation environment, but we can have the same level of control, and this is very important for debugging and the troubleshooting. Moreover these experiments are preliminary for the deployment in the testbeds like Géant or CREATE-NET, where the complexity reached is very high (as we have reported in the section 10.3). In a distributed testbed environment a simple mistake can become a big problem and the troubleshooting may take long time, so the Virtual Box has been used not only for fast prototyping but also for resolve all possible mistakes done in design level or possible misconfigurations.

We have performed several tests on this environment, we have reported here the two most meaningful (we show them in the next images): i) we have tested the Core OSHI operation and the coexistence mechanism; ii) we have introduced the AOSHI node and we have tested its correct operation, finally we have validated the proper operation of the ingress classification.

![Figure 21 – Experiment 1 on Virtual Box](image1)
![Figure 22 – Experiment 2 on Virtual Box](image2)

The configuration is very simple and straightforward, we start from scratch using Virtual Box with the same image loaded, based on Ubuntu 13.04. First, we need to install all the software needed,
that is the Open vSwitch and the Quagga Software, then it is necessary the proper configuration of all the stuff previously installed. To simplify the running of experiments we have developed a bash script that accepts as input a set of parameters (like ETHi, the IP, OSPF network etc..) and then configure automatically both the routing daemon and the SCS part of the VMs deployed. In order to realize the previous “topologies”, we need to configure carefully some option in the Virtual Box Run Time, we need to create two internal network using the VBOX’s option, so we able to emulate the 10.0.3.0/24 and the 10.0.4.0/24 networks. For each machine we have at least one interface connected to this internal networks (also this aspect has to be configured in the VMs setting). On the other hand the network 10.0.2.0/24 has been created by VBOX, because we have set in the first VM a “NAT” interface in order to obtain connectivity towards the public network. The VMs deployed have the following roles: VM1 acts as COSHI, VM2 as AOSHI and finally VM3 is the traffic generator.

In order to test the properly operation and the correct interworking we have verified this set of condition:

• Correct execution of the OSPF protocol: after a period of one second immediately we have seen the exchanges of the OSPF traffic, following a period of ten second there is the convergence of the routing protocol and the proper installation of the routes;
• Establishment of the Controller connection: as soon as the convergence is reached all the OSHI deployed one to one can connect to the Controller;
• Coexistence and separation of the traffic: we have generated different traffic (IP and SDN) from sources deployed in VM3 and verified the correct forwarding. Moreover we have verified the correct separation of the traffic belonging to the different services.

In Virtual Box environment we have not deployed any VLLs.

**10.2 Mininet environment – Deployer and experiments**

Mininet is considered as one of the best dev/testing tool, leveraging the capability of this fantastic software, we can obtain a machine-local virtual network. It founds its operation on Linux virtual network features (network namespace etc..). Use this approach is cheaper than VMs (with minimal hardware requirements), this makes it very attractive. At the same time we can have a high level of control (since we can monitor all the links in the virtual network). Using this tool, we can experiment with arbitrary topologies and different nodes, cutting off the configuration time. We can rapidly prototype, develop and test, it is possible start up in a few seconds interestingly-
sized networks. Typically designs that work on Mininet, can transfer seamlessly to hardware in real network for full speed operation. It is easy to replicate the experimental and test results. We can examine effects of code or network changes before testing/deploying on hardware, moreover it allows automated system-level tests and experiments. In our open source implementation Mininet is one of the most important building blocks, because it represents the last step before the deploy in real testbed like Create-Net and Géant one. Deploying a new solution is relative simple in Mininet, however we have to pay attention in starting same processes in a machine, we have to guarantee the separation between them. On Mininet this procedure is simplified because every emulated host is an isolated process (Linux process) and has its own network namespace, its private proc file and etc.

The realization of OSHI should be simple and straightforward in Mininet, because it provides a set of high level API (python based), that allows to create a topology of “Open vSwitches”, which one provide connectivity to the Hosts, thus we should only create the Vii interface on every switch, bridge them, and finally start the Quagga software. Here the problems start: the default behaviour of Mininet is to execute all the switches in the Root namespace, but this breaks our needed of separation. Every Vii interfaces (or Vii.VLAN_IP interfaces) can see the other Vii interface, if we want to emulate a Layer 3 network, we also need to enable the IP forwarding. Hereupon we cannot emulate properly this behaviour because every vii can directly forward the packet instead to respect the designed topology. Mininet does not support the approach OVS kernel implementation in its private network namespace, if we want to make the interfaces “private”, we have to use the User Space implementation for the SCS. However the problems do not end: this solution lacks of performance and does not permit to create virtual internal interfaces, so for our design it is not a viable strategy. The problem encountered previously (OVS kernel in its private namespace) is due to the typical configuration of Open vSwitch and Mininet in a host machine; typically when we install the OVS (in the recent version of Linux Kernel we do not need to install the software), we have all the components in execution on the root name space, in the following image we report the architectural component of OVS:
We cannot have the switches in their private namespace and the ovsdb-server/ovs-vswitchd in the root namespace; so the only solution is to extend the Mininet functionality and to build “our” class of OSHI node. This can be obtained in a simple way, we have to create a Host (by default it has its private network namespace), and we have to launch in each one the User space component of the OVS, that is the ovsdb-server and the ovs-vswitchd. However there is still a problem to be solved: the communication between the user components. The software OVS in order to enable this communication uses the Unix sockets. One is the control socket on which ovsdb-server listens for runtime management commands (for example sent by ovs-appctl or vlog). Instead the other is necessary for the communication between ovs-vswitchd and ovsdb-server. The problem arises because Mininet is an emulation software with shared file system, so except for some folder each host sees the same file system, and different OVS will use the same sockets, creating locking problem on it. The problem can be solved specifying different paths for the ovsdb-server and ovs-vswitchd (using special options in the relative launching commands).

Once we have resolved this problem, the last step is the launch of the Routing software. We have to start in each host the Zebra daemon and the OSPF daemon. The architecture of the Quagga software is a client/server, the routing daemons are the client while the Zebra daemon is the server; the latter is the component designed to “talk” with the kernel and to update the routing table of the host, it receives the routing information by the client, that is by the routing daemons.

We report in the following image the architecture, hoping that it will clarify the interaction:
First we have to launch in “our” Hosts, the aforementioned daemons, so each Zebra daemon uses its private Routing Table. Both daemons have to be properly configured, and this can do using different configuration file for each host. The routing daemons can communicate with the Zebra one using a Unix socket, thus we have to specify different paths for the sockets used by each couple OSPF-Zebra. As we have mentioned earlier, Mininet uses a shared file system, and for our purposes we have to run different Zebra and OSPF daemons in the same machine. We have the same problem encountered with the OVS software, the daemons will try to use the same socket, creating locking problem on this. However in this case is more difficult to resolve, because the Quagga software does not provide any means to change the socket path, the latter is hardcoded and written by the “configure” before the compile action. We have to take an unorthodox way, that is: for each host we create private directories (only the necessary) and fake roots (that replicate the root of the host machine) and in each host we execute the command “chroot” in order to change its root with the newly created. Once we have resolved also this problems, we have all the building block necessary to run our experiments on Mininet. We report now a recap image, that shows the run time environment and the internal details of the OSHI node in Mininet:
Once resolved all the problem encountered during the deployment of OSHI on Mininet, we have written a python script called mininet_deployer, which extends the Mininet functionalities and automatically prepares for us the local “testbed”. The complete configuration is executed in 14 phases:

1. **Topology creation**: the Deployer builds from scratch a virtual topology creating the OSHI nodes, the L2 switches, the Controllers, and the CE routers. It can use as input different sources: built-in topologies (hardcoded in python), topologies codified in JSON (created by TopoDesigner) and finally Networkx topologies (random generation of graph with arbitrary dimension through Erdos-Renyi model [19]).

2. **OSPF network generation**: the script generates for each point to point link and layer 2 network an OSPF network and assigns the network address (the generation is simplified using all /24 networks).

3. **Fix network manager**: the script generates a fix for each interface in the root namespace. The network manager try to manage automatically all active interfaces in the root namespace. In order to prevent this the Deployer adds to the file used by network manager (/etc/network/interfaces) the manual management of these interfaces. In order to make effective the changes it restarts also the network manager.

4. **VLAN tag generation**: the script generates for each layer 2 network, the association among VLAN tag and ports of the layer 2 switch that compose the network. This is executed for the regular IP traffic and the SBP traffic.

5. **Loopback generation**: the program generates for each OSHI in the network the loopback address (/32);
6. Unwanted Traffic stopping: the script stops Avahi daemon, Dhclient and old Zebra and 
OSPF daemon (if it is necessary);

7. Environment configuration: the Deployer creates for each OSHI and Controllers a private 
folder in /tmp/, where it inserts all the configuration file.

8. Open vSwitch configuration: i) the script starts in each node all the process necessary to 
the properly execution of Open vSwitch; ii) it creates the SCSs; iii) it “bridges” all the 
“physical” interfaces and for each one generates the correspondent virtual internal 
interface (Vii); iv) the Deployer sets the controllers; v) the program configures the 
coexistence mechanism (only IP part); vi) it sets up in the AOSHI node the ingress 
classification mechanisms, using the information about VLAN tag previously generated;

9. Quagga configuration: the script, using the information about OSPF network previously 
generated, configure the routing component of each OSHI node. For each interface active 
it configures the associated IP, the hello interval and then OSPF cost. Finally it configures 
the announced network: all the one directly connected and the Loopback interface (we 
announce a /32 prefix). It inserts the configuration file of Zebra and OSPF daemon in the 
private folder of the nodes and finally the script starts the routing daemons passing as 
input the configuration files previously generated.

10. Linux networking configuration: the script enables in each OSHI the IP forwarding and 
finally it disable the Reverse path filter.

11. Host configuration: the Deployer configure the networking part of each CE router in the 
network (IP address and the default via);

12. Layer 2 access network configuration: the script set up properly each L2 switch deployed. 
In particular it configures the access ports and the trunk ports using the information about 
VLAN tag previously generated.

13. VLL Pusher CFG generation: the Deployer build the configuration file of the VLL pusher 
using the information about the VLAN tags, AOSHI nodes and access ports associated to 
the SBPs to deploy.

14. Environment cleaning: At the end of the emulation the script cleans the Host Machine 
environment: i) it deletes all the OSHI’s and Host’s folder previously created; ii) the 
Deployer stops the processes associated to the Nodes; iii) it deletes links and interfaces; iv) 
it restarts the network manager and Avahi daemon, finally kills all the OVS process and the 
Quagga daemons created; iv) it unmounts all the fake root created.
The scripts is completely unaware of the topology, we can change without any problem the topology of the network and the only effort is due to creation of this particular topology with the Mininet API or with the Topology Designer. We have introduced a new python class called OSPFNetwork, it is necessary for the automatic generation of the configuration file, indeed it maintains the association among interfaces and the OSPF network. Thus at the end of Deployer’s execution the nodes have been properly configured and we have to generate only the traffic for our tests.

We have performed three different validation tests on this environment, in the following images we report the topologies used: i) we have tested the Core OSHI operation and the coexistence mechanism; ii) we have introduced the AOSHI node and we have tested its correct operation, finally we have validated the proper operation of the ingress classification; iii) we have used the topology of the previously test, the main differences is the introduction of the VLLs that interconnect the CEs:

![Figure 26 – Experiment 1 on Mininet](image)

![Figure 27 – Experiment 2 and 3 on Mininet](image)

In order to test the properly operation and the correct interworking we have verified this set of condition:

- Correct execution of the OSPF protocol: after a period of one second immediately we have seen the exchanges of the OSPF traffic, following a period of 15 second there is the convergence of the routing protocol and the proper installation of the routes;
- Establishment of the Controller connection: as soon as the convergence is reached all the OSHI deployed one to one can connect to the Controller;
• Coexistence and separation of the traffic: we have generated different traffic (IP and SDN) from sources deployed in VM3 and verified the correct forwarding. Moreover we have verified the correct separation of the traffic belonging to the different services.
• Correct operation of the VLLs: we have verified the proper functioning of the VLLs deployed and the coexistence with the regular IP traffic. We have also run some performance test that have shown an average gain around 10% (in the section 11 we will show more detailed results);

10.3 OFELIA testbed – Deployer and experiments

As for the OFELIA deployment, we have run our experiments on the CREATE-NET testbed based on the OCF (OFELIA Control Framework) developed in the context of the OFELIA project [3]. The testbed is composed by a set of 8 OpenFlow capable switches and 3 Virtualization Servers that can host experimental Virtual Machines controlled by the testbed experimenters. Using SDN mechanisms (and in particular the Flowvisor network virtualization tool [1]) the testbed resources can be “sliced” among different experiments and each experiment can be controlled by a different OpenFlow controller. Our Deployer can actually operate on any OCF compliant testbed (several testbed “islands” based on OCF are currently available as a result of the OFELIA project).

Considering our goal to experiment with the Hybrid IP/SDN nodes, we would need to operate on the OpenFlow switches at the core of the testbeds, replacing them with OSHI nodes or integrating the OSHI features in the existing switches. Unfortunately, such operations are not permitted in the existing testbeds. Therefore we have resorted to an overlay approach. We have deployed a number of OSHI nodes over a corresponding number of VMs and we have created an overlay network topology of Ethernet over UDP tunnels (using OpenVPN without encryption) among the VMs. Each link in the overlay topology, meant to emulate a physical L2 link among OSHI nodes and between OSHI nodes and CEs. Within an OSHI node the tunnels corresponds to virtual interfaces that are bridged into the Open vSwitch switch. We have deployed also one or more “overlay” controllers on other VMs that will take control of the overlay switches in the OSHI nodes.

In this approach, we can identify two different networking levels: the first level provides the connectivity among VMs across the testbed OpenFlow switches (we call it “testbed level”) and the “overlay level” in which the OSHI nodes operate. Within each level an SDN controller provides the logic to control the switches of the given level. As an example we deploy an overlay topology of 3 OSHI nodes and 2 hosts, according to the topology shown in the upper bound of the Figure 28. In
the example 7 physical VMs are spread over 2 Virtualization Servers. Within the “testbed level” a single switched Ethernet domain is realized and single IP subnet is used for all interfaces of all VMs. Therefore each interface of a VM will have a different IP address in this subnet. Using the IP subnet 192.168.0.0/16, we follow this convention for the interface addresses 192.168.x.yz where x represents the Virtualization Server, y a VM in the Virtualization Server, z the interface of the VM. Note that the Testbed Controller, i.e. the controller in the Testbed level does not need an IP address in the experimental range, as it is reachable over a different VLAN (called management VLAN) and a specific Ethernet interface (eth0) is reserved in each VM for the management traffic.

Figure 28 – Testbed and Overlay networking levels

In the previous figure we report in red the overlay links (tunnels) between the VMs, the dashed lines blue represent the management network, instead the ones continuous the data network.

At the initial testbed setup and later on when needed we create new VMs in the testbeds (CREATE-NET or GOFF) we run our “setup script” that performs the basic setup of the VM installing all software. The setup procedure creates dedicated local folders, containing both the configuration files and the utilities needed. This script is run once on a newly created VM and could be replaced by the capability to install a custom VM on CREATE-NET and/or GOFF testbed. The same VM will be used for emulating different experimental topologies for making different experiments. Here comes into play the “configuration script” that creates the set of tunnels among the VMs as needed to emulate the wide area network links. This configuration script is also able to “clean up” everything and restart from scratch in creating an experimental topology.
In order to make it easy to modify the required experimental topology, a separate configuration file is used to provide the topology parameter for one experiment (e.g. how many virtual links, the remote end-point addresses of this virtual links...) Assume we have \( K \) OSHI nodes. Each OSHI node \( i(1 \leq i \leq K) \) has \( M_i \) Ethernet interfaces (it depends on which Virtualization Server it is mapped) and \( N_i \) virtual interfaces /tap (one for each link in the Overlay topology) In this way, the “OFELIA Deployer” tool, given an experimental topology to be realized, can produce as output the set of configuration files for each VM. Running the configuration script with the provided configuration file in each VM will deploy our experimental topology over a testbed (CREATE-NET / GOFF).

Different mechanisms have been used to automate and facilitate both the setup and the configuration processes. A management server coordinates all operations, communicating with the rest of the experiment machines through the testbed management network and using Distributed Shell (DSH) for distributing and executing remote scripts. Through the setup procedure, the needed scripts are copied on the experiment VMs and run locally. It also possible to execute cleanup procedure that activates the cleaning process and reset the VMs configuration to their default state, as after the software setup. Experiment users can opt to use a centralized configuration as well or just keep it locally in the configuration folder. The described operations can be run either locally after login or with the help of the management server. A more detailed overview of the scripts and a typical use-case can be found at [17]. To simplify the running of experiments we have developed the management console, a CLI (Command Line Interface) that can run remote commands, open different SSH terminals, put and get files to/from either a single machine or a group of machines, providing a convenient framework to run and control the experiments.

As shown in Figure 20, the OFELIA Deployer python script automatically produces the configuration scripts for emulating a given topology, composed of access and core OSHI nodes and of Customer Edge routers (CE). Each OSHI node and each CE is mapped into a different VM running in one of the Virtualization Server of the testbed. In order to produce the configuration file Deployer needs some basically information about the testbed, this information are contained in the Topology to testbed mapping file. It contains for example the association of OSHI nodes and CEs to testbed VMs, the experimental VLAN or the available physical interfaces (Ethernet interfaces).

The Testbed Deployer operations can be automatic or manual: in the first case using for example JSON topology it generates automatically the configuration file of the topology to deploy. On the
other case the generation is manual and driven by the User (in a few words the user can “program” all the aspects). About we note that the Testbed Deployer is not a simple python script (that automatically generates all), but it is something more, we have to distinguish it in two part: a set of python classes and the actual Deployer script. The latter one can generate, using the python classes, the configuration file of each machine to be deployed, using different sources as input. In our Deployer implementation we have provided the generation of built-in topology (programmed in python using the Deployer library), JSON topology, and Networkx topology. Instead, using the framework provided by the classes of the Testbed Deployer library, an Experimenter User can build his/her Deployer and “program” in a simple way the configuration file of the topologies. We report in the following box a fragment of code that shows as we can create in a simple way the configuration files of a parametric mesh topology using the Deployer framework:

```python
1 //Deploy of Full Mesh Topology
2 ... // Initialization section of temporary variable
3 testbed = TestbedOFELIA("ofelia.map") // We create Testbed object passing as parameter the mapping file
4 for i in range (0, size): // the size parameter is the dimension of the mesh topology
5 oshi = testbed.addOshi(name) // Add a new COSHI to the experiment. Return an Oshi object
6 for lhs in oshis:
7 testbed.addPPLink(lhs_name, oshi_name) // Add an overlay link among lhs and oshi. Return a composite object, that represents in our case an overlay link.
8 oshis.append(oshi)
9 ctrl = testbed.addController(name, OF_tcp_port) // Add a controller to the experiment. Return a Controller object
10 testbed.addPPLink(oshi, ctrl) // Connect the Controller to the overlay network
11 testbed.configure() // Generate the configuration file for the overlay topology to deploy
```

Box 11 - Deploy of Full Mesh Topology with Testbed Deployer API

As we have already explained the Deployer frameworks offers python APIs to “literally” program the configuration file (used by the machines to “realize” the desired topology), we have several advantages using this approach: the API are very similar to the one introduced by Mininet (we believe that are very simple to learn such as those of Mininet); smooth learning curve (above all for the Mininet users); “almost” automatic conversion of Mininet scripts. Now we report more implementation details about the Deployer framework, showing the diagram of the classes that compose it:
The generation of the configuration file is executed through the TestbedOFELIA function configure(): the routine creates the header of the configuration file, moreover inserts the shared (among the nodes) option MGMTNET, that stores the testbed management subnet (this information is necessary in order to avoid overlap with our addresses); as second step the routine calls the procedure configure() for each host. The Host.configure() procedure is a little bit complex, because it writes on the setup file all the necessary information to the properly configuration of the machine. As an example we consider the OSHI.configure(): i) the procedure writes on files general information like hostname, DPID and bridge name; ii) it serializes on the setup file information about controllers, i.e. their numbers and the IP address; iii) the routine writes the information about the interface, i.e. Ethernet interfaces, Tap interfaces and VI interfaces; iv) as last step it saves in the file the announced network. At the end of the TestbedOFELIA.configure() procedure the configuration file is ready to be processed by the VMs.

We have performed several validation tests and deployment on the OFELIA facility: triangular mesh, triangular mesh and hosts (the previous topology with the addition of three hosts), and finally the topology represented in the Figure 27 (with only three hosts). The available resources are not the major lack in these testbeds, the impossibility to reuse the same VM image on different machine is the biggest problem (from our point of view). Thus every time we want to add a new machine we have to performed the setup procedure, and this step is very long and tedious.
This is the main problem that affects the dimension of our experiment. In order to automate the procedure we have used our OFELIA Deployer (thus we have tested also it). At the end of the configuration, for each topology deployed we have verified: the correct execution of the routing protocol, the establishment of the Controller connections, the correct coexistence and the separation of the IP traffic with the SDN one and finally the properly functioning of the VLLs deployed.
11 Performance evaluation experiments

As a first performance validation of our solution we have deployed the topology represented in Figure 31 (i.e. two core OSHI, three access OSHI and three CEs) over the physical testbed topology in Figure 30 (showing 3 OpenFlow switches and 3 virtualization servers). Considering the Figure 31 we represent with blue and red background respectively the core OSHI (COSHI) and access OSHI (AOSHI) nodes. All the VMs deployed have 1 GB of RAM, instead for OSHI controller we have an amount of RAM halved. We have realized four experiments in order to validate our open source implementation: i) CPU load vs. OSPF hello interval (no synthetic traffic); ii) CPU load vs. packet rate; iii) CPU load vs. packet rate without OpenVPN; iv) TCP max throughput.

The management console assists in the orchestration of the experiments, allowing to easily deploy monitoring and traffic generation scripts in all target VMs. The monitoring script takes statistics using top, traffic generation scripts uses the iperf traffic generator tool [22]. The scripts can be programmed to run at a given time (we sync all VMs using NTP protocol). At the end of the execution the management console allows to recover the collected statistics from all VMs. The goal of this section is not to provide a thorough performance evaluation of OSHI nodes, but mainly to provide a first hint about the performance of our open source implementation and an idea of the available tools and of the experiments that can be run. We are well aware that the results shown are preliminary.

Let us analyze the first experiment, we aim to evaluate the processing cost of packet forwarding in OSHI nodes, considering as sources of traffic the OSHI nodes themselves (in this case we do not have any iperf sources). We have considered the processing load with different OSPF hello
intervals: 30, 10, 5, 1 seconds. At these rates, we were not able to measure the variation of CPU load due to the processing of OSPF hello messages sent over OSHI nodes. The most CPU-consuming process was the top command itself. Therefore no specific report is provided for this experiment.

In the second experiment we aim to evaluate the processing cost of packet forwarding in OSHI nodes, in three cases: i) OSHI IP forwarding, in which the packets also cross the Open vSwitch two times; ii) OSHI VLL forwarding, in which the packets are directly forwarded by the Open vSwitch (swapping the VLAN tag, and pushing/popping the VLAN tag in the ingress/egress nodes); iii) ROUTER IP forwarding, in which the Open vSwitch is removed and the OSHI node interfaces are directly connected to IP forwarding engine. Using iperf tool as traffic source, we generate an UDP packet flow of: 1500, 2000 and 2500 packet/s (datagram size is set at 1000 Byte). We report more details about the traffic generated in Table 7.

<table>
<thead>
<tr>
<th>Rate (p/s)</th>
<th>Rate (Mb/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1500</td>
<td>12</td>
</tr>
<tr>
<td>2000</td>
<td>16</td>
</tr>
<tr>
<td>2500</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 7 – Iperf packet and data rates

At the chosen rate, we verified that the iperf server has not reported any loss and all the system works well. Using the top tool, we evaluate the CPU load, sampling each second and averaging over 60 seconds. The results are shown in Figure 32, which reports the data collected in the first hop (AOS3 in Figure 31). Each value in the chart represents the average of 5 runs (as anticipated, a complete statistical analysis with confidence intervals is out of scope here).

As expected we notice that VLL forwarding (i.e. using an Open vSwitch rule) is better than IP forwarding, we can obtain this results, because the routing component does not intervene, thus we are able to reduce the context switches between user processes, among the user processes and the kernel. It is interesting to note that the overhead introduced by the Open vSwitch processing in the OSHI IP with respect to ROUTER IP is almost negligible. This happens because only the first time the rules are managed by the open-vswitchd in the user space, instead the other time will be handled directly by the Open vSwitch kernel module, through a fast cache (as it is explained in [21]).

With the help of the top tool, we looked at the processes with the highest utilization, in search for the system bottleneck. In this scenario the tunneling mechanism based on OpenVPN represents the bottleneck. We performed a further experiment with a rate of 3000 p/s. At this rate iperf
reported some packet loss. We have noticed a CPU load percentage near 80% in all three cases. We will not report the results there because the measurement of CPU load in these conditions are not reliable due to the XEN virtualization environment (we are taking the CPU load measurements from within the VMs running in a XEN virtualization server). The performance limitations of OpenVPN have been analyzed in [20] and our results are comparable. The main reason for this bottleneck is that OpenVPN implements Ethernet in UDP tunneling in user space (there is a lot overhead spent in the CPU context switch). In order to have a more scalable experimental platform we have recently replaced the OpenVPN tunneling with VXLAN tunnels implemented by Open vSwitch.

In a second experiment we have evaluated the IP forwarding performance of an OSHI node vs. a plain Linux IP router, removing the OpenVPN tunneling mechanism and directly operating on VLAN interfaces. This type of experiment cannot be automatically deployed using the topology designer and deployer, and we could setup only a simple topology with two CE routers and two OSHI nodes. The results are shown in Figure 33. They confirm that a much higher packet forwarding rate can be achieved, likely in the order of 100K pack/s, as the CPU load stays in the order of 10% for a rate higher than 10K pack/s. It is also shown that the performance penalty of an OSHI node vs. a plain IP router remains relatively small.
The packet rates 7500 p/s, 10000 p/s, 12500 p/s and 15000 p/s correspond respectively to a rates of 60 Mb/s, 80 Mb/s, 100 Mb/s and 120 Mb/s. Considering the low CPU load in all cases, probably we have reached the limit of the physical net with the last experiment performed (120 Mb/s).

The third presented test has been run over the Mininet emulator. We have deployed the topology shown in Figure 34, which reports the image from the GUI of the TopoDesigner. On this topology, we evaluate the TCP throughput between two CE routers comparing the OSHI IP solution with the OSHI VLL service. We argue that the throughput will be limited by the sum of CPU processing load on all nodes (the Mininet emulation is run on a single machine that emulates the whole network. Therefore we expect that the throughput using the OSHI VLL service will be higher than the throughput using the OSHI IP routing. This is confirmed by the results shown in Table 8.
Figure 34 – Mininet topology and deployed VLL

<table>
<thead>
<tr>
<th>#</th>
<th>VLL (Mb/s)</th>
<th>IP (Mb/s)</th>
<th>% GAIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>1555</td>
<td>1150</td>
<td>26,04 %</td>
</tr>
<tr>
<td>STD DEV</td>
<td>21,8</td>
<td>20</td>
<td>#</td>
</tr>
</tbody>
</table>

Table 8 – TCP max throughput for OSHI IP and VLL
12 VXLAN solution to cope with the scalability issues of OpenVPN

In our first design we created the tunnels using the user space OpenVPN tool (we used tunnels with no encryption). The achieved performance was poor (we have shown preliminary results in the previous section), as the encapsulation and decapsulation performed in user space is very CPU intensive (a large overhead is spent in the continuous “context switches” between kernel and user spaces during packet processing). Therefore we considered a second design approach that uses VXLAN [23] tunnels provided by Open vSwitch (OVS).

A tutorial on VXLAN is available in [25]. We cite literally:

| VXLAN technology is meant to provide the same services to Ethernet connected end systems, that VLANs do today and also provide a means to stretch L2 network over a L3 network. Traditionally, all data centers use VLANs to enforce Layer2 isolation. As data centers grow and needs arise for extending Layer2 networks across data center or may be beyond a data center, the shortcomings of VLANs are evident. These shortcomings are: The current limit of 4096 VLANs (some are reserved) is not enough. In a data center, there are requirements of thousands of VLANs to partition traffic in a multi-tenant environment sharing the same L2/L3 infrastructure for a Cloud Service Provider; Due to Server virtualization, each Virtual Machine (VM) requires a unique MAC address and an IP address. So, there are thousands of MAC table entries on upstream switches. This places much larger demand on table capacity of the switches. VLANs are too restrictive in terms of distance and deployment. VTP can be used to deploy VLANs across the L2 switches but most people prefer to disable VTP due to its destructive nature. Using STP to provide L2 loop free topology disables most redundant links. Hence, Equal-Cost Multi-Path (ECMP) is hard to achieve. However, ECMP is easy to achieve in IP network. VXLAN addresses above challenges, VXLAN ID (called VXLAN Network Identifier or VNI) is 24-bits long compared to 12-bits of VLAN ID. Hence, it provides over 16 million unique IDs. VXLAN is defined as draft, further details about this solution can be found in [24].

A tutorial on configuring VXLAN for Open vSwitch is available at [26]. Figure 35 is extracted from this tutorial and provides a visual representation of the issue of stretching a L2 network over a L3 network.
As explained in [21], OVS Kernel module implements VXLAN tunnels, and this allows to dramatically improve performance with respect to the user space tunneling approach of OpenVPN. The design of the VXLAN tunneling solution for OSHI is reported in Figure 36.

Figure 36 Implementing VXLAN tunnels in OVS
Tunnels in OVS are just virtual ports with own OpenFlow port number. The SDN Capable OVS is also able to perform encapsulation and decapsulation of VXLAN tunnels, each tunnel corresponds to a port in the switch. The VXLAN tunnel ports can be connected with the virtual ports exposed to the IP forwarding engine. The switch also includes the “physical” port toward the IP/SDN network (which is a VLAN port in the OFELIA testbed) and the local br0 switch port that can be assigned the
IP address of the physical interface. At the time of writing, the VXLAN tunneling solution has been implemented and tested and we have executed also its performance evaluation (we will report performance results of the VXLAN solution in the following section).
13 Improved experiments with xentop

The fidelity of the first two experiments described in section 11 is limited by the fact that the CPU load measurements are taken using the top tool from within the VMs running in a XEN virtualization environment. This does not provide stable and reproducible results. On the other hand the xentop tool, which should be run as root in the hosting XEN server can provide the correct information about the resource usage of each single VM. Therefore we have developed a module to gather CPU load information for each VM in the XEN host. This module makes the relevant results available to the VMs through a simple based on a TCP socket that returns a JSON text file with the needed information At the time of writing we have tested the tool to collect the first results as described below.

In order to leverage the capability of xentop we have updated our Measurement Tools, adding a new Thread that executes periodic polling and gathers the CPU load of the monitored VMs. In each run we collect 20 CPU load samples with polling interval in the order of two seconds, the first 10 samples are discarded and the last 10 are averaged to get a single CPU load value. Then we evaluate the mean (AVG) and standard deviation (DEV) over 20 runs. Using the xentop based approach, we have repeated the first two experiments described in section 11.

Table 9 to Table 11 report the CPU load using Open VPN as tunneling mechanism, for the OSHI VLL, OSHI IP and plain IP router. At low packet rates (500 and 1000 p/s) it can be appreciated that plain IP router is less CPU demanding, then we have OSHI VLL and then OSHI IP. The differences among the three solutions are not big because the CPU load is dominated by the OpenVPN tunneling, but we can safely infer that the penalty introduced by the OSHI solution is minimal. At higher rates the standard deviation increases and it is not possible to confirm the results (longer experiments with a higher number of runs are needed to reduce the variability of the evaluated CPU load average, this is in our plans). Figure 37 and Figure 38 reports the experiment results in a chart, adding the linear regression lines. The theoretical CPU saturation rate turns out to be in the order of 3500 p/s for the three solutions.

<table>
<thead>
<tr>
<th></th>
<th>500 p/s</th>
<th>1000 p/s</th>
<th>1500 p/s</th>
<th>2000 p/s</th>
<th>2500 p/s</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AVG</strong></td>
<td>26,73</td>
<td>41,82</td>
<td>60,86</td>
<td>67,17</td>
<td>79,49</td>
</tr>
<tr>
<td><strong>DEV</strong></td>
<td>0,79</td>
<td>4,36</td>
<td>2,71</td>
<td>14,63</td>
<td>8,19</td>
</tr>
</tbody>
</table>

Table 9 CPU load (%) for OSHI IP (OpenVPN)
Table 10 CPU load (%) for OSHI VLL (OpenVPN)

<table>
<thead>
<tr>
<th></th>
<th>500 p/s</th>
<th>1000 p/s</th>
<th>1500 p/s</th>
<th>2000 p/s</th>
<th>2500 p/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>25,83</td>
<td>43,54</td>
<td>56,62</td>
<td>72,9</td>
<td>73,98</td>
</tr>
<tr>
<td>DEV</td>
<td>0,64</td>
<td>1,85</td>
<td>4,49</td>
<td>2,94</td>
<td>10,71</td>
</tr>
</tbody>
</table>

Table 11 CPU load (%) for ROUTER IP (OpenVPN)

<table>
<thead>
<tr>
<th></th>
<th>500 p/s</th>
<th>1000 p/s</th>
<th>1500 p/s</th>
<th>2000 p/s</th>
<th>2500 p/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>25,11</td>
<td>39,23</td>
<td>55,15</td>
<td>72,69</td>
<td>75,30</td>
</tr>
<tr>
<td>DEV</td>
<td>0,84</td>
<td>10,07</td>
<td>8,39</td>
<td>5,09</td>
<td>17,42</td>
</tr>
</tbody>
</table>

Table 12 CPU load (%) for OSHI IP (plain VLAN)

<table>
<thead>
<tr>
<th></th>
<th>500 p/s</th>
<th>1000 p/s</th>
<th>1500 p/s</th>
<th>2000 p/s</th>
<th>2500 p/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>6,92</td>
<td>12,19</td>
<td>14,71</td>
<td>17,2</td>
<td>19,02</td>
</tr>
<tr>
<td>DEV</td>
<td>0,26</td>
<td>0,36</td>
<td>0,4</td>
<td>1,49</td>
<td>3,72</td>
</tr>
</tbody>
</table>

Table 13 CPU load (%) for ROUTER IP (plain VLAN)

<table>
<thead>
<tr>
<th></th>
<th>500 p/s</th>
<th>1000 p/s</th>
<th>1500 p/s</th>
<th>2000 p/s</th>
<th>2500 p/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>5,98</td>
<td>10,36</td>
<td>12,89</td>
<td>15,56</td>
<td>17,59</td>
</tr>
<tr>
<td>DEV</td>
<td>0,21</td>
<td>0,73</td>
<td>0,87</td>
<td>1,48</td>
<td>3,04</td>
</tr>
</tbody>
</table>

Table 12 and Table 13 report the experiments using the plain VLAN networking, without any tunneling. In this case it is not possible to deploy the VLL service and only plain IP router and OSHI IP have been compared. We can appreciate a CPU load penalty for OSHI IP forwarding with respect to plain IP forwarding in the order of 10%-20%. Apparently, the CPU load penalty is decreasing in relative terms at higher CPU load, but this is subject to further evaluation in future experiments.

Note that the theoretical CPU saturation rate (see regression lines in the following images.) for plain IP router is in the order of 14000 p/s, that is 20 times higher with respect to the OpenVPN tunneling case. Adding OSHI-IP forwarding would reduce the theoretical CPU saturation rate to something in the order of 12500 p/s.
We report in the following images the above results in a chart and we show also the regression lines of the OSHI IP and ROUTER IP solution:

Using the xentop based approach, we executed the performance tests of the VXLAN solution that was introduced to overcome the limitations of OpenVPN. Table 14 to Table 16 reports the experiment results and Figure 41 and Figure 42 provides the chart with the regression line. In the VXLAN based solution the plain IP router has to go through Open vSwitch in any case, therefore as expected it has no advantage with respect to OSHI IP. The OSHI VLL solution is the less CPU intensive and its theoretical CPU saturation rate is in the order of 13000 p/s. The OSHI IP solution increases CPU load of less than 10%, and its theoretical CPU saturation rate is in the order of 12000 p/s.

<table>
<thead>
<tr>
<th></th>
<th>500 p/s</th>
<th>1000 p/s</th>
<th>1500 p/s</th>
<th>2000 p/s</th>
<th>2500 p/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>8,51</td>
<td>14,4</td>
<td>17,92</td>
<td>20,04</td>
<td>23,365</td>
</tr>
<tr>
<td>DEV</td>
<td>0,55</td>
<td>0,68</td>
<td>0,99</td>
<td>2,1</td>
<td>3,26</td>
</tr>
</tbody>
</table>

Table 14 CPU load for OSHI IP (VXLAN)

<table>
<thead>
<tr>
<th></th>
<th>500 p/s</th>
<th>1000 p/s</th>
<th>1500 p/s</th>
<th>2000 p/s</th>
<th>2500 p/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>8,19</td>
<td>13,36</td>
<td>14</td>
<td>16,95</td>
<td>22,88</td>
</tr>
<tr>
<td>DEV</td>
<td>0,4</td>
<td>1,67</td>
<td>4,15</td>
<td>4,15</td>
<td>5,48</td>
</tr>
</tbody>
</table>

Table 15 CPU load for OSHI VLL (VXLAN)

<table>
<thead>
<tr>
<th></th>
<th>500 p/s</th>
<th>1000 p/s</th>
<th>1500 p/s</th>
<th>2000 p/s</th>
<th>2500 p/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>8,45</td>
<td>14,03</td>
<td>17,43</td>
<td>20,74</td>
<td>24,59</td>
</tr>
<tr>
<td>DEV</td>
<td>0,35</td>
<td>1,04</td>
<td>0,61</td>
<td>1,96</td>
<td>2,09</td>
</tr>
</tbody>
</table>

Table 16 CPU load for ROUTER IP (VXLAN)
In order to show the effectiveness of the VXLAN solution, in Figure 43 we compare the CPU load for OSHI IP solution in the OpenVPN, VXLAN and VLAN scenarios. It can be appreciated that VXLAN tunneling adds a reasonably low processing overhead (close to the plain VLAN scenario, that represents the best case in terms of processing overhead), while OpenVPN tunneling would dramatically reduce the forwarding capability of an OSHI node in the testbeds.

In the last images we compare the CPU load for OSHI VLL and ROUTER IP solution in the OpenVPN, VXLAN and VLAN scenarios. The results confirm as we said earlier, i.e. low processing overhead introduced by VXLAN solution.
Figure 44 – OSHI VLL CPU load

Figure 45 – ROUTER IP CPU load
14 The NeST testbed

The Netgroup SDN Testbed (NeST) is a private testbed made up by three server in a small data center at the University of Rome Tor Vergata. It has been used to support different research activities, among the others:

- the joint SDN-IP demo with ONLAB
- the OSHI experiments after the ends of the DREAMER project

The physical layout of the testbed is shown in Figure 46

For the Joint SDN-IP demo with ONLAB and for the OSHI experiments after the end of the DREAMER project, the KVM virtualization platform has been installed in each serve. The logical configurations of the testbed in the two experiments are respectively shown in Figure 47 and in Figure 48. Note that the IP numbering in the setup for the Joint SDN-IP demo with ONLAB was different and that the interconnection with the Uniroma2 network was through a switched LAN. On the other hand a direct connection on a router port is used in the setup shown in Figure 48.
Figure 47 – Logical setup of the tesbed for the ONOS SDN-IP experiment

Figure 48 – Logical setup of the tesbed for the OSHI experiments after the end of DREAMER
References

[19] Erdos-Renyi model - http://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93R%C3%A9nyi_model
[22] iperf traffic generation tool http://iperf.sourceforge.net/