**SEVENTH FRAMEWORK PROGRAMME**

**Theme 3**

**Information and Communication Technologies**

# Deliverable D3.10
# EXPRESS final architecture and design, evaluation plan

**Grant Agreement number:** **287581**

**Project acronym:** **OpenLab**

**Project title:** **OpenLab: Extending FIRE testbeds and tools**

**Funding Scheme:** **Large scale integrating project (IP)**

**Project website address:** **www.ict-openlab.eu**

**Date of preparation of deliverable: 21/02/2014**

| | | |
|---|---|---|
| Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013) | | |
| Dissemination Level | | |
| PU | Public | X |
| PP | Restricted to other programme participants (including the Commission Services) | |
| RE | Restricted to a group specified by the consortium (including the Commission | |
| CO | Confidential, only for members of the consortium (including the Commission Services) | |

Document properties

| | |
|---|---|
| **Document responsible:** | Stefano Salsano (CNIT) |
| **Author(s)/editor(s):** | Stefano Salsano (editor), Nicola Blefari-Melazzi, Giuseppe Bianchi, Luca Veltri, Andrea Detti, Claudio Pisa, Giuseppe Siracusano, Fabio Patriarca, Federico Griscioli<br><br>All authors are affiliated to CNIT |
| **Version:** | 1.0 |


Abstract:

The main objective of EXPRESS is designing an innovative, resilient SDN system capable to extend the SDN applicability domain from fixed networks to intermittently connected network, like wireless mesh networks. The EXPRESS solution will be implemented and then deployed in an experiment involving the three OpenLab testbeds NITOS, W-iLab.t and PlanetLab Europe.

In this document we provide the final architecture and the evaluation plan. This documents extends and refines the high level description of the solution that was provided in deliverable D3.9. EXPRESS uses an SDN approach based on the OpenFlow protocol for the controller-to-switch communication. EXPRESS relies on OLSR distributed routing protocol to setup the control plane for the controller-to-switch communication. A distributed controller solution based on a hierarchy of controllers is designed, capable to handle network partitioning and merging. On the data plane, the SDN/OpenFlow approach is capable to support advanced routing strategies, going beyond the shortest-path routing provided by OLSR.

As for the evaluation, the W-iLab.t and NITOS testbeds will host the wireless mesh nodes and the local controllers, PlanetLab Europe will provide a layer 2 overlay based interconnection of the wireless mesh networks and may host the centralized controller.

# Table of Contents

# 1 Introduction

The main objective of EXPRESS (EXPerimenting and Researching Evolutions of Software-defined networking over federated test-bedS) is designing an innovative, resilient SDN system capable to extend the SDN applicability domain from fixed networks to intermittently connected network, like wireless mesh networks. A more complete introduction to EXPRESS objectives, reference scenario and technical approach can be found in our previous deliverable D3.9 [1]. For reader's convenience, Fig. 1 below reports the reference scenario.

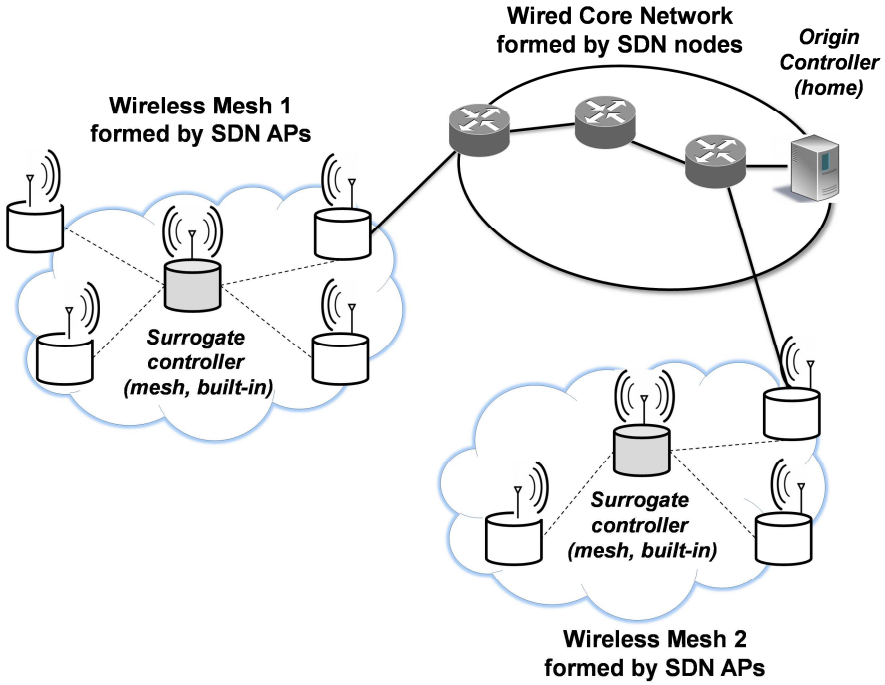


**Fig. 1 – Reference scenario: a provider or community network composed of a set of Wireless Mesh Networks**

Our previous deliverable D3.9 also included the detailed requirement analysis (in section 2), the initial functional specification and high level design (in section 3) and a discussion on the state of the art (in section 4).

In this deliverable we will provide the final architecture, the detailed design and the plans for the system evaluation.

# 2 Final architecture

Taking into account the requirements outlined in section 2 of our deliverable D3.9 [1], the EXPRESS is based on an IP connectivity realized using the OLSR routing protocol. We deploy the SDN mechanisms on top of the basic IP connectivity: the control plane communication between the controllers and the OpenFlow switches uses the IP connectivity setup using OLSR. As for the data plane communications, they can be handled in a flexible way, either using the basic IP connectivity or using SDN/OpenFlow mechanisms. Proper classification mechanism can be configured in the WMR nodes to support this flexibility and to decide which forwarding mechanisms will be used for which flows.

The operation of the Wireless Mesh Routers have been designed in order to support the exchange of OLSR messages and the coexistence of traffic relying on the basic IP connectivity and traffic that is handled using SDN/OpenFlow mechanisms.

## 2.1 Overall architecture

The proposed approach foresees to use an IP ad hoc routing protocol (OLSR) among the nodes of the mesh to establish a basic IP connectivity (see Fig. 2). Such connectivity will constitute the control plane and will support all controller-to-switch OpenFlow messages as well as controller-to-controller messages in case they are needed to coordinate the SDN operations. The use of OLSR ensures the proper reaction to changing topology events, like addition/removals of mesh nodes and wireless links among them. The forwarding on the data plane can rely on the basic IP connectivity or can be based on an SDN/OpenFlow approach, in a flexible way.

As for the wireless channels, we use a single SSID for both the control traffic and the data traffic, therefore we can classify it as an “in-band” control strategy from the OpenFlow protocol perspective.

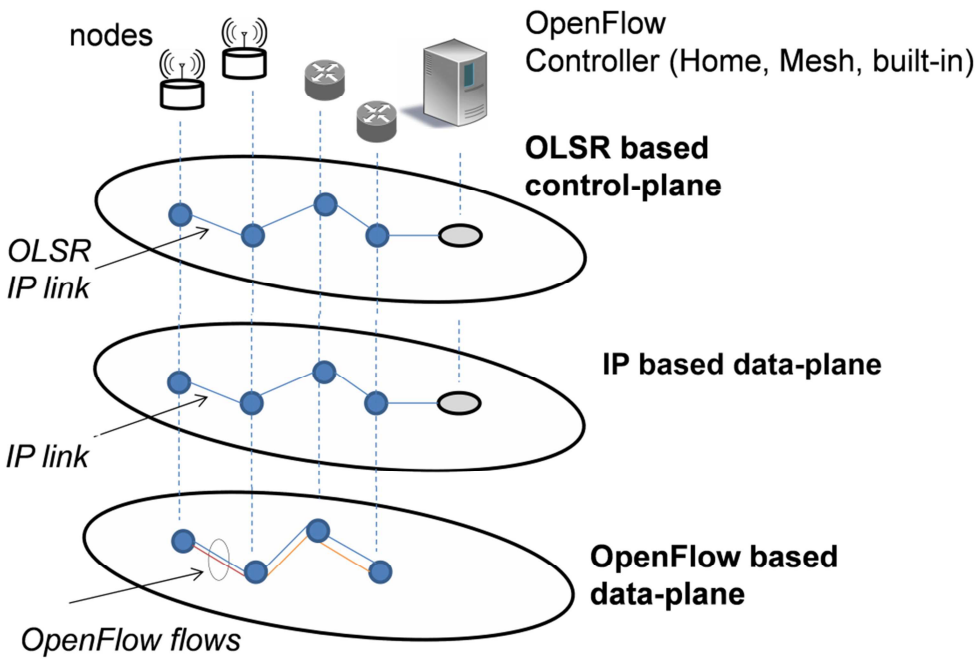


**Fig. 2 – Control and data planes**

The idea is that the control plane will use the basic IP connectivity, while the data plane can use the IP connectivity or an "SDN based connectivity" in a flexible way. By SDN based connectivity we mean that the routing of packet flow is decided by the SDN controller and the forwarding within each node is based on the flow table rules installed using the OpenFlow protocol. In fact, each Wireless Mesh Router will run an OpenFlow switch. Over the control plane, the OpenFlow switch can contact a set of default controllers: the Home Controller (HC), running in the fixed network, and the Mesh Network Main Controller (MMC), i.e. the main controller in the mesh. Moreover the switch can use other "lower priority" Mesh controllers that can take over in case all other controllers are not reachable, these "lower priority" Mesh controllers can implement a subset of the full OpenFlow-based available services, they will be referred as SMC – Secondary Mesh Controllers. A switch node will also have a *built-in* controller located in the switch itself for handling emergency services or, more in general, partitioned networks without a pre-defined Home/Mesh controller. This built-in controller does not need to be a full compliant OpenFlow controller, rather it is a process that is able to inject OpenFlow rules in the local OpenFlow switch. We will call this entity "EFTM - Embedded Flow Table Manager". The hierarchy of controllers and flow table manager is shown in Fig. 3.

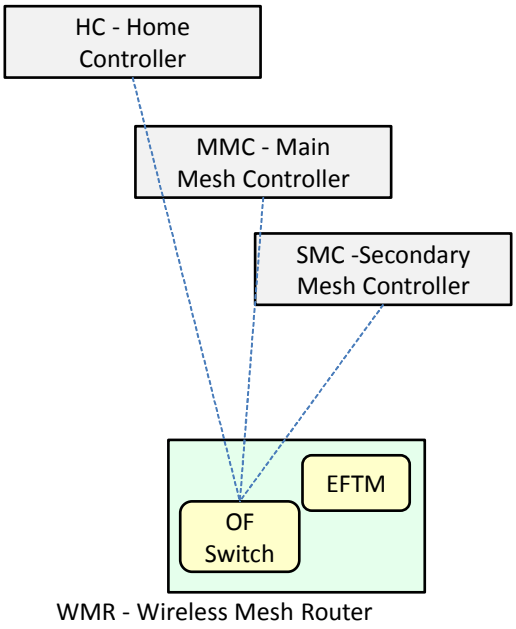


**Fig. 3 – Hierarchy of Controllers and Embedded Flow Table Manager**

The OpenFlow controllers in the hierarchy can be used to engineer the routing of data traffic, forcing an arbitrary subset of the traffic to follow a different route with respect to basic IP routing. In emergency conditions, during which all OpenFlow controller fail or are unreachable, the basic IP routing is always available. It will be a decision of the EFTM which traffic flows are allowed in these emergency conditions: the most restrictive policy will be to allow only control traffic, i.e. directed towards the IP addresses of the control subnet, the most liberal policy will be to allow traffic towards all destinations (including all access networks all Internet destinations that are routed towards the default gateways advertised by OLRS).

The WMR nodes can connect to different controllers, supporting controller failures and dynamic topology modification, including network partitioning and joining.

The high level architecture of a Wireless Mesh Router node, showing the interplay between the OLSR protocol, the IP forwarding and the EFTM entity is shown in Fig. 4. The OpenFlow switch in the WMR is configured by the EFTM so that by default IP packets for the IP control subnet to which the WMR interfaces belong are handled by the IP forwarding modules. This way, the OLSR daemon can send and receive OLSR packets over the wireless interfaces. Once the basic IP routing is established with OSLR, the OpenFlow switch in the WMR interacts with the OpenFlow controllers that can configure the flow table for specific data plane flows.

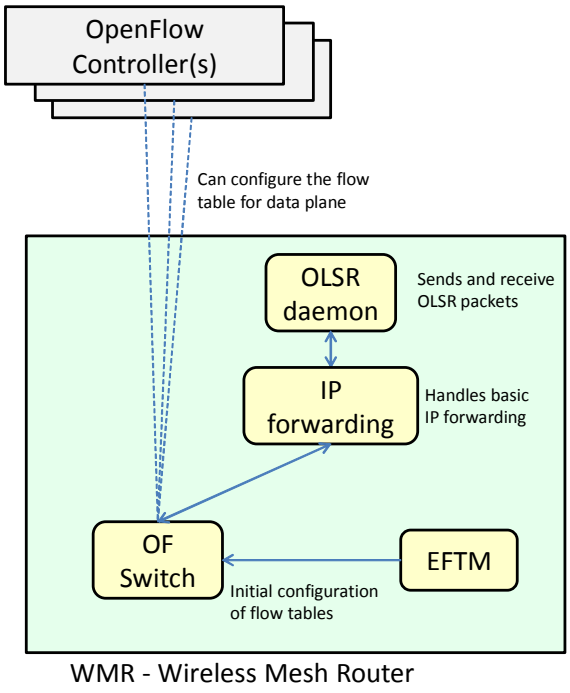


**Fig. 4 – WMR node architecture**

## 2.2 Detailed design of networking and of WMR node internals

The reference network scenario is shown in Fig. 5. A WMN is composed of Wireless Mesh Routers (WMRs) which provide connectivity to a set of Access networks (either offering a wired or wireless interface to user terminals). A subset of the WMRs operate as Gateways and provide connectivity towards the Internet. A set of OpenFlow controllers can operate in the wireless mesh (indicated as Main Mesh Controller and Secondary Mesh controller), connected to a WMR through a wireless/wired connection. The Home Controller is on the wired backbone.

Within the whole control network, each controller is uniquely identified by its IP address in the control network range. It means that a private IP range will be allocated for the control network and each controller and mesh node will be statically given its IP address. Under these assumptions the control plane connectivity can be built by using OLSR, spanning across the mesh network and the wired core network. Control traffic and data traffic use different IP subnets. For instance, the subnet 10.0.0.0/16 can be used for control traffic, while other subnets are used for data traffic. The controllers and the WMR wireless interfaces use addresses of the control subnet, while other interfaces of the network get an IP address belonging to different subnets, e.g. 192.168.x.0/24, each announced in OLSR as an “HNA network” (HNA stands for Host and Network Association).

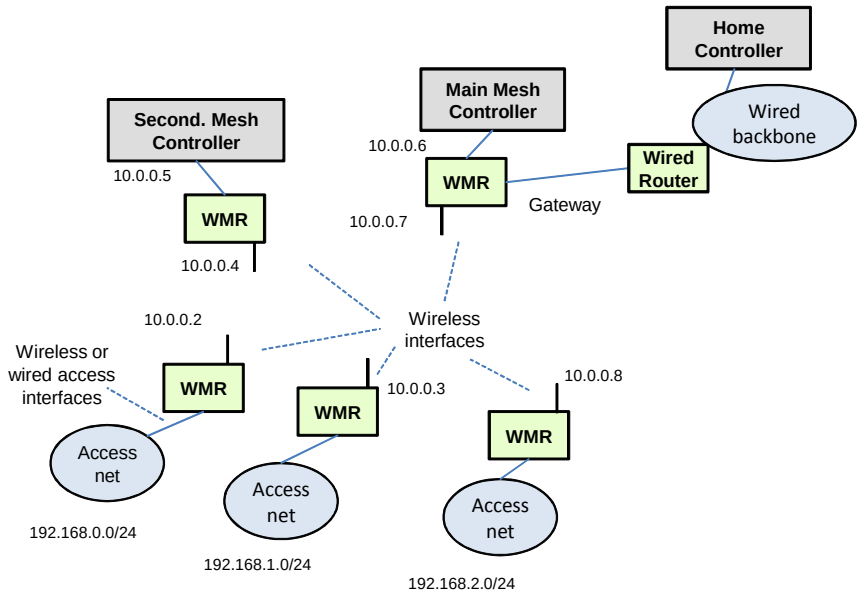


**Fig. 5 – Reference network scenario**

## 2.2.1 Basic IP forwarding/routing

In our WMR node, we assume that the OpenFlow switch will be directly connected to a set of physical wireless (and wired if present) interfaces (see Fig. 6). For each physical interface connected to the switch, a corresponding virtual internal interface is added to the OpenFlow switch. The IP routing and forwarding of the node operates using this set of virtual internal interfaces. Initially, a simple set of rules is configured in the switch so that the packets can flow from the physical interfaces to the virtual internal interfaces and vice versa. A packet that is sent by the IP layer in the WMR over a virtual interface crosses the OpenFlow switch and is sent out over the corresponding physical interface. An incoming packet arriving over a physical interface is forwarded by the OpenFlow switch to the corresponding virtual internal interface[1]. The OLSR routing protocol runs using the virtual internal interfaces and learns the topology of the external links among the WMRs. An OLSR routing instance

[1] This solution will be further detailed in section 2.2.3 and it is different from the solution discussed in previous deliverable D3.9. The solution proposed in deliverable D3.9 was designed for a single networking interface and could not be extended if the node needs to support multiple interfaces within the OpenFlow switch. This configuration with multiple interfaces in the OpenFlow switch is needed to interact with L2 tunnels across PlanetLab and have the possibility to modify the routing using an SDN/OpenFlow approach. With the previous solution described in Deliverable D3.9 we could have crossed PLE only using IP routing in the "border node" of each testbed. We note that this solution with virtual internal interfaces corresponding to the physical interfaces (wireless and wired) of the node has been first proposed in the context of the EU DREAMER/GN3plus project (http://www.geant.net/opencall/Software_Defined_Networking/Pages/Home.aspx#DREAMER). Differently from the DREAMER approach: 1) we consider OLSR instead of OSPF as routing protocol; 2) we add the support for wireless interfaces.

runs on each WMR node and the IP address of the controller is also advertised by OLSR using a Host and Network Association (HNA) messages with /32 mask.

The external OpenFlow controllers can control the rules of the OpenFlow switch. In this way, they can reroute any chosen subset of the traffic over different paths with respect to the shortest paths selected by the OLSR protocols.

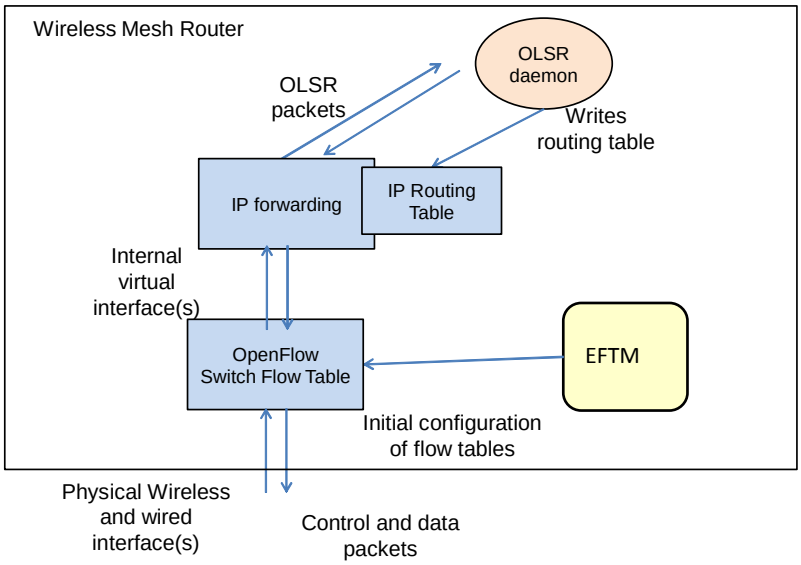


**Fig. 6 – OpenFlow and OLSR interaction**

## 2.2.2 Forwarding/routing of traffic using SDN

Let us now consider how to handle the traffic using SDN approach, allowing traffic flows to follow different paths with respect to the ones decided by OLSR routing. There will be two classes of packets/flows as seen by the OpenFlow switches in the WMR:

1) Packets/flows that are processed using regular IP routing/forwarding (*Basic* class)
2) Packets/flows that will be handled by SDN (*SDN* class)

The OpenFlow rules in the tables of the OpenFlow switches will be used to classify packets as belonging to Basic or SDN classes.

A packet that belongs to the Basic class will be forwarded by the OpenFlow switch from the virtual internal interface to the physical interface (outgoing packets) or vice versa (incoming packets). A packet that belongs to the SDN class will need to find a matching entry in the flow tables of the switch or it will be forwarded to the OpenFlow controller. For this type of traffic within the Wireless Mesh Network a matching entry will specify the outgoing interface and will set as destination MAC address the next hop MAC address and as source MAC address the MAC address of the outgoing interface. In this way the OpenFlow switch will emulate the behavior of a OLSR router in forwarding the packet, but the outgoing interface can be set arbitrarily by the controller without following the routing chosen by OLSR.

For example, a possible approach is to include all traffic that belongs to the control-subnet in the Basic class and all traffic for IP destinations outside the control-subnet (i.e. in the access networks or in the Internet) in the SDN class.

Assume that a packet is generated by a host of the access network and destined to an Internet address outside the wireless mesh network (but the same will also apply to packets destined to a host of the access network as this occurs when packets come back from the Internet or for mesh internal communications). The packet will be received by the WMR on its access network interface. The packet will be processed at IP level and an outgoing virtual internal interface will be selected according to basic IP routing table. The packet will enter the OpenFlow switch. A match is searched in the flow table. If a match is found, the related action is carried out (i.e. set the destination and source MAC address and forward the packet on an outgoing interface). If no match if found, the IP packet is embedded in a OpenFlow packet-in, which is transferred to the controller currently in charge of the WMR using the in-band control network. When the controller receives the packet-in, it can apply the desired routing logic and install data plane entries in the flow table.

To distribute the topology information of the data plane, the IP subnets of the Access Networks are advertised by WMRs and gateway WMRs using OLSR Host and Network Association (HNA) messages. Moreover, gateway WMRs may also advertise the default route 0.0.0.0/0. With this approach, each WMR node knows the full network topology. The controllers inquiry the connected WMR to learn this topology information, which is fundamental to implement traffic engineering logic for data traffic. This approach is different from the traditional OpenFlow topology discover in wired layer 2 network, performed using LLDP messages [1].

### 2.2.3 WMR node design

The architecture of a WMR node is shown in Fig. 7. It includes: one or more wireless interfaces belonging to the Wireless Mesh Network (wlan0, wlan1…); a software bridge using OpenFlow switching logic, e.g. Open vSwitch [3], a set of virtual internal interfaces vi0, vi1…; an optional wired interface used as a gateway to Internet (ethX in the figure); an optional set of wired or wireless interfaces towards client Access Networks (ethY, wlanZ in the figure; an optional virtual interface (tap9 in the figure) corresponding to an Ethernet over UDP tunnel toward a Planet Lab Europe Node.

Each virtual internal interface is logically associated with a physical wireless interface belonging to the Wireless Mesh Network or (where present) with a tunnel toward a Planet Lab Europe Node. All virtual internal interfaces have IP address belonging to the control-subnet, the physical wireless interfaces belonging to the WMN (wlan0, wlan1) and the tap tunnel toward PLE (tap9) do not have an IP address, ethZ and wlanZ have an address of the Access Networks subnet and ethX of the subnet connected toward the Internet.

The OLSR daemon is configured to work on the virtual internal interfaces, therefore it will not operate on the access networks nor on the interface towards Internet.

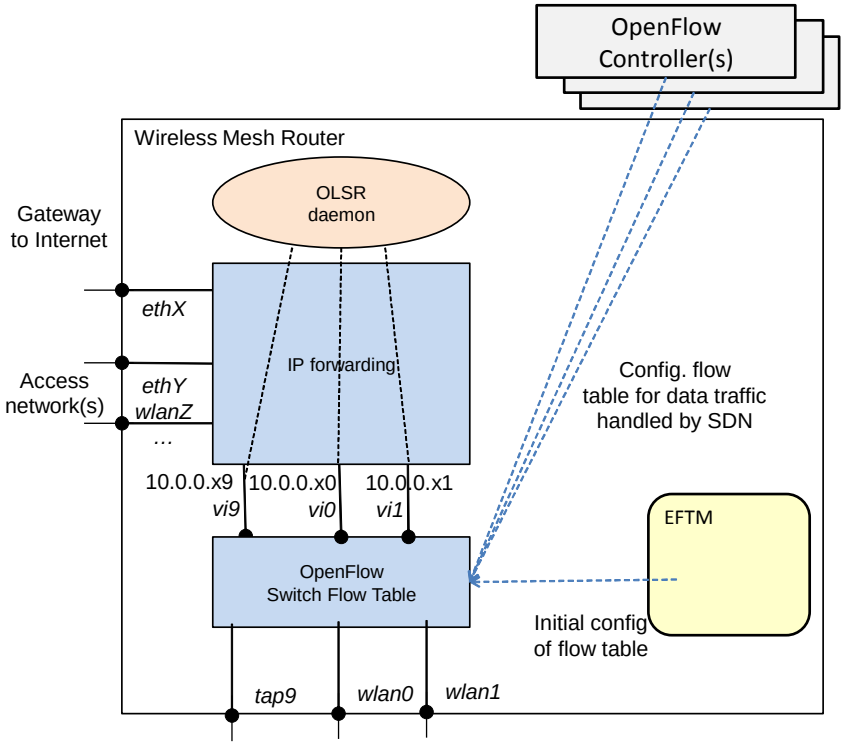


**Fig. 7 – Detailed WMR architecture**

## 2.2.4 Emergency conditions

The EFTM entity in the WMR will continuously check if the WMR is connected to an active controller. In case of controller failures (e.g. due to hardware or communication issue) the ETFM will trigger the start of an “emergency condition”. In this state the EFTM can choose to clear all rules set by the controller, so that the node will only operate at IP level with the routing enforced by OLSR.

In these conditions the EFTM could enforce some policies to handle data traffic under emergency condition. It will be possible to classify the data traffic and allow to forward only a subset of this data traffic which has a higher priority, by setting proper rules in the OpenFlow switch flow table or by operating at the level or IP routing.

When the controller becomes reachable again, the EFTM leaves the emergency status and restores that initial basic rules in the OpenFlow switch (and the default routing policy if some changes to the IP routing have been made).

# 3 Controller distribution architecture

According to our requirements, Express needs a distributed solution that allow different controllers to take control of our WMR nodes.

We considered in our design the following two main aspects:

1) The different controllers need to synchronize about which controller is master for each switch, therefore a “master election” procedure is needed.
2) The different controllers need to share a common view of topology and of the network events that are relevant to take decisions in the controller layer.

In our scenarios, the “master election” procedure needs to be repeated each time that a portion of network become partitioned or when different partitions are joined together in a larger partition. Express adopts a hierarchical approach, as illustrated in Fig. 3. The basic idea is that the control will be taken by the controller connected to the WMR with the highest level in the hierarchy. The “master election” procedure will therefore select for each switch the highest level controller that can control the switch. With this approach, we focus on the problem of partitioning/merging of network portions, while we are deliberately not focusing on load sharing issues. In fact, if the Home Controller will be visible from each switches, it will be selected as the master controller for the whole network.

As for the common view of topology and events, we assume that OLSR topology distribution mechanism can be exploited by OpenFlow controllers. The Express controllers will learn the topology and will receive topology updates using OLSR. For the purpose of Express experiments, the overall map of potential controllers and WMNs can be statically configured in all controllers (e.g. using some configuration file). It is for further study to consider if the OLSR protocol can be extended to support functionality related to our specific scenario, and it is not our priority to design and implement such extensions.

In a traditional OpenFlow environment, it is assumed that communications among controllers are relatively reliable. Therefore the master selection procedure can be executed with information exchange among controllers that cooperatively choose a master to take control of a given switch. Then the controllers send “role request” messages that are able to change controllers status, enforcing for example one “master” and a set of “slave” controllers for a given OpenFlow switch. An example in this line of reasoning can be found in [6]. Considering the requirements of a wireless mesh environment that includes topology changes and links unreliability, there is the risk that a distributed master election procedure produces inconsistent results. For transient periods, controllers could be connected with the WMR but could not be able to communicate each other. Under such circumstances, both controllers would believe they are in charge of controlling the WMR and would try to become “master” using the “role request” messages.

For this reason we designed a procedure in which the WMR itself is in charge of selecting the more appropriate controller given the connectivity status of the network. We note that WMRs and controllers have the same information about the status of the network (excluding transient conditions), because

they share the OLSR vision of the topology. In particular, the WMRs are directly involved in the OLSR topology dissemination while the controller extracts the topology information from a nearby WMR. Therefore, from the topology discovery point of view the WMR acquires topology information even before the controller. Moreover, a WMR can directly check the connectivity with potential controllers trying to establish TCP connections towards them (or monitoring the liveliness of established TCP connections). In the designed procedure a WMR connects only toward a single controller at a given time. This is different from the classical approach where a switch connects in parallel with several controllers. The procedure is performed in the WMR with the help of an external entity with respect to the switch, the EFCM (External Flow table and Controller Manager), which extends the EFTM (External Flow Table Manager) that we have introduced for handling the flow tables. The modified node architecture is shown in Fig. 8. The EFCM is in charge to perform the master election procedure and will instruct the switch to connect to the selected controller at a given time. We can define the proposed mechanism as “master selection” rather than “master election”: it is directly the WMR node that monitors changes in the network topology (split/merging of mesh network, each such change can make unavailable/available a given controller). Following a network topology change, a WMR node takes into account the available (reachable) controllers, selects the best one (the highest in the hierarchy) and setups an OpenFlow control connection with it. From the implementation point of view, a sort of “hard” handover of the controller is performed by the WMR. The OpenFlow switch running in our WMR node (Open vSwitch) does not support the adding of a new target controller at run time. In order to connect to the new controller we have to stop the running switch and to start a new instance of the switch specifying the IP address of the new selected controller.

We note that our mechanism is conceived to work for events of topology changes (network merging/joining) that operate in the time scale of minutes, it may become critical if we want to manage such events in the time scale of few seconds. Hysteresis timers can be added to the solution to prevent too frequent iterations of the procedure and the related instability.

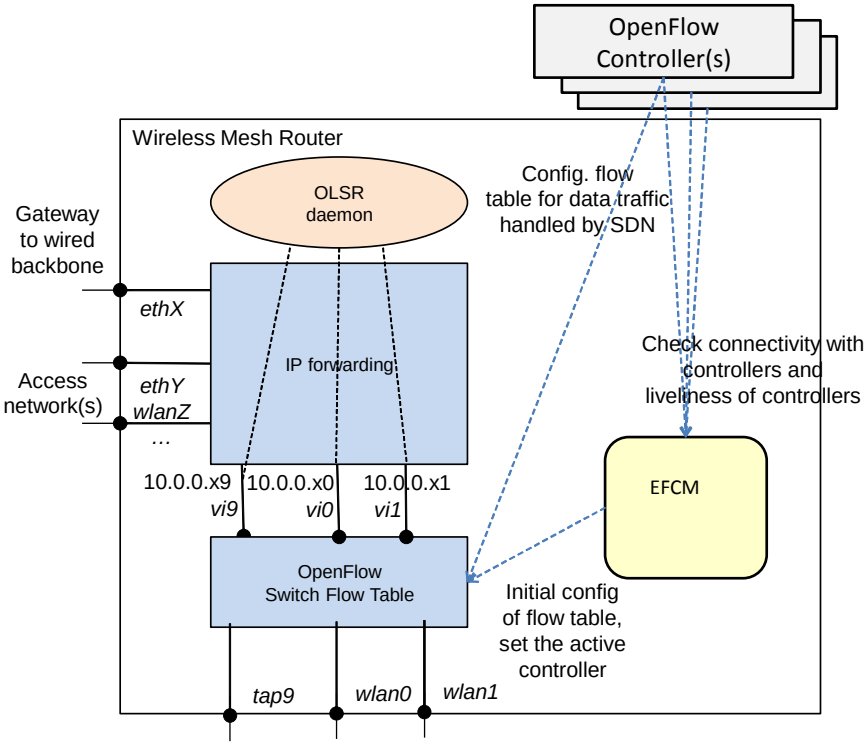


**Fig. 8 – WMR architecture: introduction of EFCM**

Taking the decision on the WMR side with the proposed procedure has some advantages in our scenario:

1. Each OpenFlow switch in the WMR will be connected with a single controller at a time, this provides the guarantee that the switch is receiving rules from a single controller.
2. The OpenFlow switch in the WMR does not have to know in advance the controller or the set of controllers to connect to, but the EFCM entity can discover available controller through dynamic updates. To clarify this point, we recall that the OpenFlow switch that we are using in our demo (Open vSwitch) does not support the adding of a new target controller at run time. Therefore in any case adding a previously unknown controller would imply restarting the OpenFlow switch.
3. A coordination mechanism among controllers is not needed, each controller can operate on its own.

We note that our “hard” handover mechanism between a controller and the other is possible because the basic IP forwarding is handled by the IP routing layer without the need of flow table entries set by the controller. Therefore the switch will restart in a consistent state and will connect to the new target controller.

As for the algorithm to select the master controller, in our demo we simply associate a priority to the controllers based on their IP address. The controller with the highest IP address among the available controllers will be selected. Therefore we will assign the IP addresses to the controllers so that the desired hierarchy is enforced. The available controllers will be announced by OLSR as HNA (Host Network Association) and we will distinguish the controllers assuming that their IP address will belong

to a particular range. This is a simple solution that does not require any enhancement to OLSR, more sophisticated solutions can be adopted extending OLSR so that controllers information can be explicitly advertised in OLSR announcements.

## 3.1 An example of the controller distribution approach

In this section we provide an example of the proposed controller distribution mechanisms.

Fig. 9 shows a Wireless Mesh Network topology divided in three separate partitions (red, blue and green) that do not communicate each other. The red nodes are not connected to any controller but can only rely on IP routing and on the rules configured by the EFCM entity. The blue node is connected to the controller available in its partition, the green node has two available controllers in its partition and is connected with the one with highest priority (the Home Controller, coloured in green).

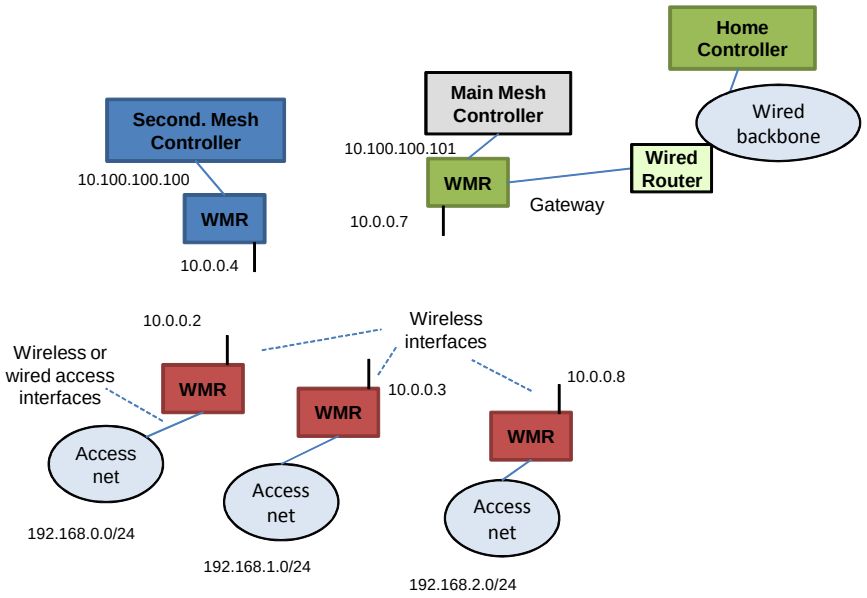


**Fig. 9 – Wireless Mesh Network with 3 partitions**

When the red partition merges into the blue one (Fig. 10), the previously red WMR nodes notices that a controller is available and connect to it.

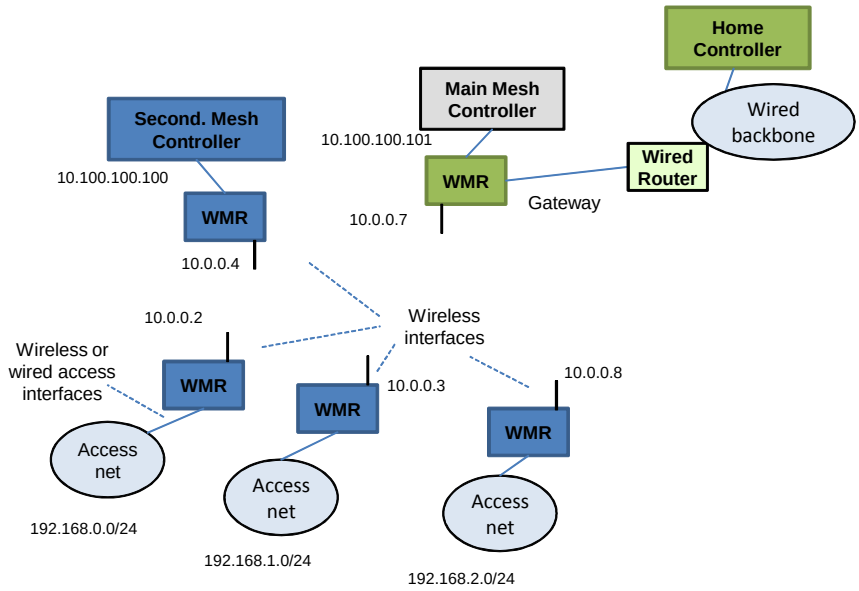


**Fig. 10 – Wireless Mesh Network with 2 partitions**

When the blue partition merges into the green partition and a single partition is formed (Fig. 11) all nodes have potential access to three controllers belonging to three levels in the hierarchy. Among the this set, all nodes select the highest in the hierarchy, that is the Home controller.

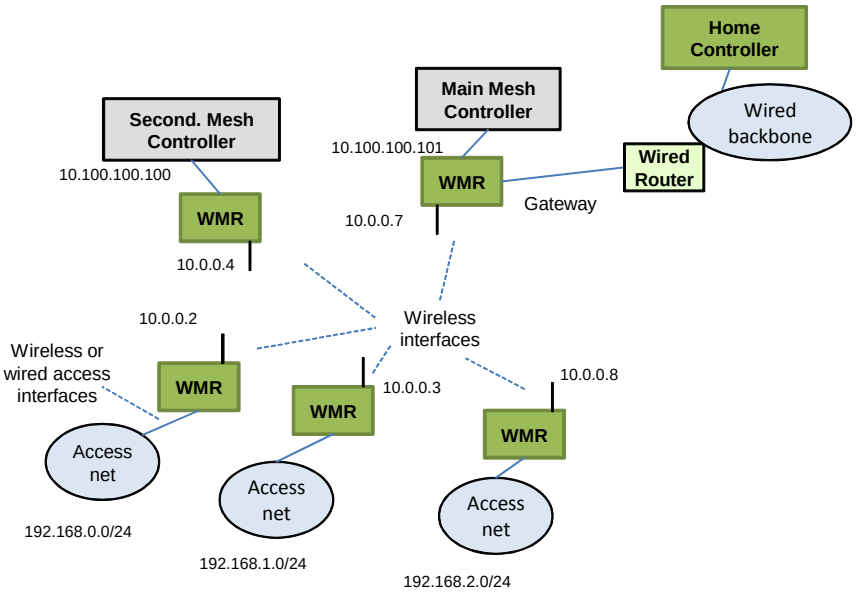


**Fig. 11 – Wireless Mesh Network with a single partition**

# 4 Evaluation plan

We plan to evaluate our architecture using the combination of three OpenLab testbeds: the two wireless testbeds NITOS and W-iLab.t and Planet Lab Europe (PLE). The W-iLab.t and NITOS testbeds will host the wireless mesh nodes and the local controllers, PlanetLab Europe will provide a layer 2 overlay based interconnection of the wireless mesh networks and may host the centralized controller.

## 4.1 Testbed interconnection aspects.

We use Ethernet over UDP tunnels across Planet Lab Europe to interconnect the testbeds. The basic methodology for creating tunnels over Planet Lab Europe has been described in [8][9]. In [10] and [11] the methodology has been extended to support tunnels that terminate outside Planet Lab Europe. In this work we provided support for tunnels terminating in testbed nodes using private IP addresses behind a NAT.

In our testbed setup we use tunnels that terminate outside PLE, i.e. in the NITOS and W-iLab.t testbeds. A node in PLE acts as an Ethernet switch, bridging a tunnel coming from NITOS and a tunnel coming W-iLab.t. Therefore it is possible to provide Layer 2 connectivity between a node in NITOS and a node in W-iLab.t. The interconnection setting is shown in Fig. 12. Note that this figure shows only the two nodes in NITOS and W-iLab.t that terminates the tunnels towards PLE, the experiment topology includes several other WMRs and a set of controllers both in NITOS and W-iLab.t.

As shown in Fig. 12, we had to face the problem of private IP addresses, as both the NITOS and W-iLab.t testbeds are connected to the Internet using NAT. We have designed and implemented 3 different solutions. The first two solutions use a “Port Forwarder” running in the node that interconnects a testbed with the Internet, one is based on kernel level port forwarding using IP tables, the other one is based on application level port forwarding using the socat utility. These solutions require that something is executed (or configured) on the node that interconnects a testbed with the Internet, each time that a new tunnel between a node in the testbed and a node in Planet Lab Europe is created. The third solution does not require the Port Forwarder, but only relies on the regular NAT translation of an outgoing UDP flow originating from a node in the testbed and terminating in the node in Planet Lab Europe that acts as tunnel end-point. This solution is based on the dynamic discovery on the Planet Lab Node of the UDP port that has been assigned by the NAT to the UDP flow that supports the tunnel (this UDP port is needed on the Planet Lab Node to configure the tunnel). A small disadvantage of the NAT solution with respect to the Port Forwarder solution is that the connection needs to be initiated always from the wireless testbed side and it is not possible to initiate it from PLE. A second disadvantage is that the connections through the NAT is closed after a timeout if no traffic is flowing. Typical values of the time out range from 30 seconds to 2 minutes. To mitigate this problem it is desirable that “keep alive” packets are periodically exchanged (e.g. a ping every 10/15 seconds).

For testbed policies we were not allowed to run the Port Forwarder solution on the W-iLab.t testbed, while in NITOS we have successfully tested the kernel/IP tables based solution with the assistance of NITOS network administrators. The NAT solution has the big advantage of not requiring any interaction with the gateway nodes, if the experimental nodes are allowed to initiate UDP connection towards Internet. This advantage overcomes the minor disadvantages that have been mentioned beforehand, therefore the NAT solution will be used by default in our experiments.

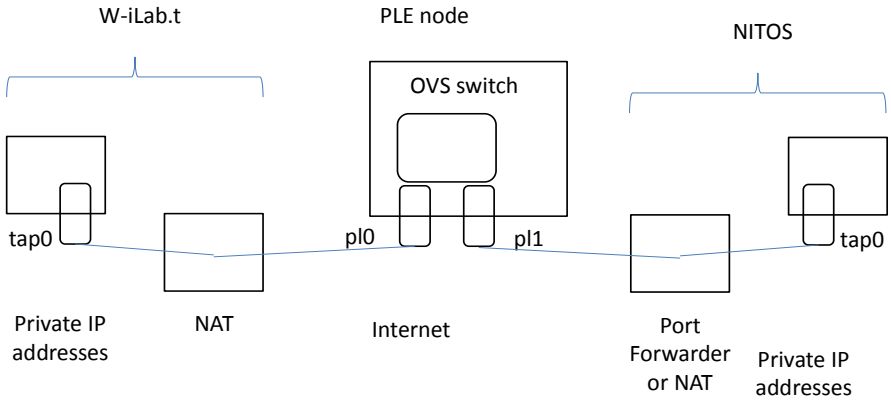


**Fig. 12 – Testbed interconnection**

Further details on the three NAT traversal solutions that have been considered are reported in the appendix.

# 5 Conclusion

In this document we have reported the design of the Express solution for integrating SDN concepts (using the OpenFlow protocol) in a Wireless Mesh Network scenario. We described the Express architecture in terms of the routing and forwarding mechanism and the approach for dynamic selection of the OpenFlow controller assuming a time changing topology with network merging/splitting. Finally we have discussed our evaluation plans and presented the tools needed to support the interconnection of NITOS and W-iLab.t testbeds using PlanetLab Europe nodes.

Our next steps will be to finalize the implementation, deploy of the solution, and run the integrated experiment over NITOS and W-iLab.t interconnected through PlanetLab Europe.

# 6 References


[1] S. Salsano (editor) “EXPRESS requirements, initial functional specification and overall design”, Deliverable D3.9 of EU Project FP7 287581 “OpenLab”

[2] Volkan Yazıcı, “Discovery in Software-Defined Networks”, blog post, http://vlkan.com/blog/post/2013/08/06/sdn-discovery/

[3] Open vSwitch website: http://openvswitch.org/

[4] P. Dely, A. Kassler, N. Bayer, “OpenFlow for Wireless Mesh Networks”, IEEE International Workshop on Wireless Mesh and Ad Hoc Networks (WiMAN 2011), Hawaii, USA, August 2011

[5] T. Koponen et al., “Onix: A Distributed Control Platform for Large-scale Production Networks,” in OSDI, 2010.

[6] Advait Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, Ramana Kompella, “Towards an Elastic Distributed SDN Controller”, HotSDN 2013

[7] A. Tootoocian and Y. Ganjali, “HyperFlow: A distributed control plane for OpenFlow”, in INM/WREN workshop, 2010.

[8] G. Lettieri, L. Rizzo, “OpenFlow enhancements for PLE”, Deliverable D4.3 of EU Project FP7 287581 “OpenLab”

[9] Planet Lab Europe - OpenFlow Overlay Network https://www.planet-lab.eu/doc/guides/user/practices/openflow

[10] A. Gulyás, G. Biczók, F. Németh, B. Sonkoly, “ALLEGRA - Implementation and evaluation of greedy forwarding in PlanetLab”, Deliverable D4.11 of EU Project FP7 287581 “OpenLab”

[11] OpenFlow and PlanetLab, wiki page, http://sb.tmit.bme.hu/mediawiki/index.php/OpenFlow_and_PlanetLab

# 7 APPENDIX A: Details on testbed interconnection and related tools

## 7.1 Existing solution for L2 overlays on Planet Lab Europe

Within Planet Lab Europe (PLE) it is possible to deploy an L2 overlay, as described in [8][9].

It is also possible to create L2 tunnels towards nodes that are external to PLE, so that the L2 overlay can extend outside PLE, see [10] and [11]

Example: an overlay with two planetlab switches and two external hosts:

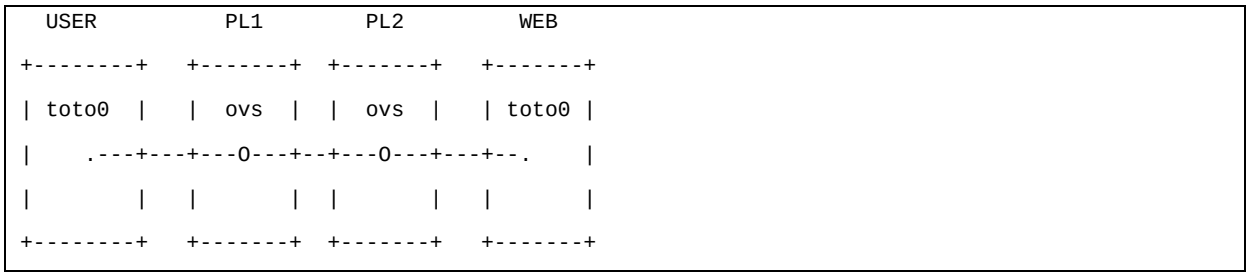


**Figure 1 – A L2 overlay over Planet Lab Europe**

The connection between a PL node and an external node (in the figure above for example between "PL2" and "WEB" is based on a UDP "connection", in which both ends must have a public IP address.

Currently, the IP:port of the external node (e.g. WEB) must be known beforehand and it is configured in the setup script that creates the L2 overlay in PLE (by default the UDP port 2020 is used). When the setup script is run in the experimenter management host, the UDP ports on the PLE nodes are allocated and the startup script outputs the UDP ports on the console. Then the UDP port can be given given as input to the tunnel setup script on the external host (i.e the WEB node In the example above) and the tunnel between the PLE node and the external node can be setup.

## 7.2 Connection to an external testbed: scenario and requirements

The requirement is to dynamically setup L2 tunnels toward nodes in external testbeds.

These nodes are typically behind NAT/Firewall and have private IP addresses.

The solution should support several independent tunnels from Planet Lab Europe toward an external testbed. (In fact, a single experiment can use more than one tunnel and several experiments can be run in parallel).

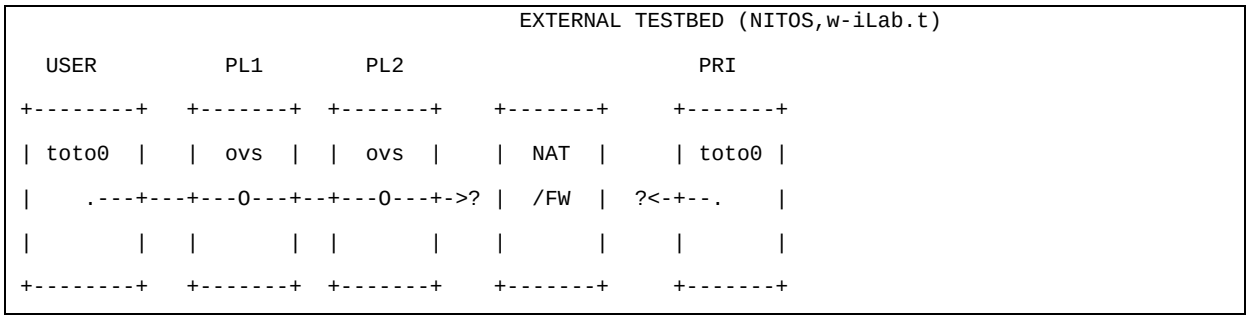


**Figure 2 – A L2 overlay over Planet Lab Europe involving an external tesbed node behind a NAT**

## 7.3 Proposed solution

The simplest and most efficient solution is to have a Port Forwarder (PF) with a public IP address on a server in the external testbed.

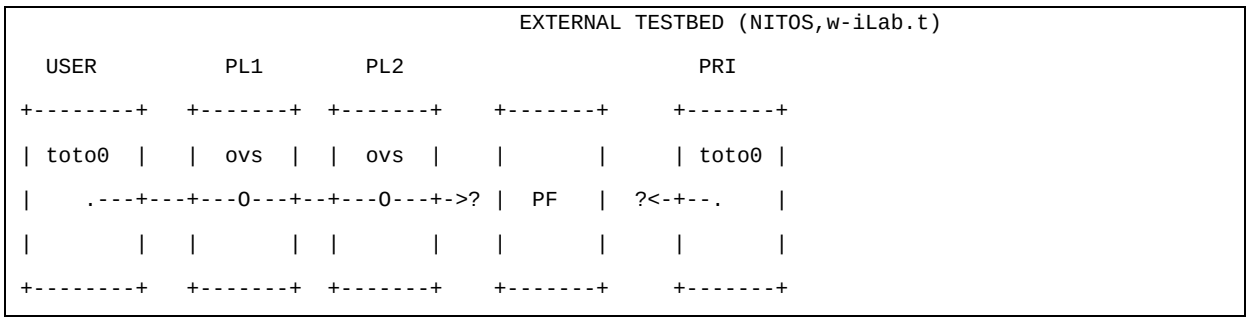


**Figure 3 – Port Forwarder to solve NAT issues**

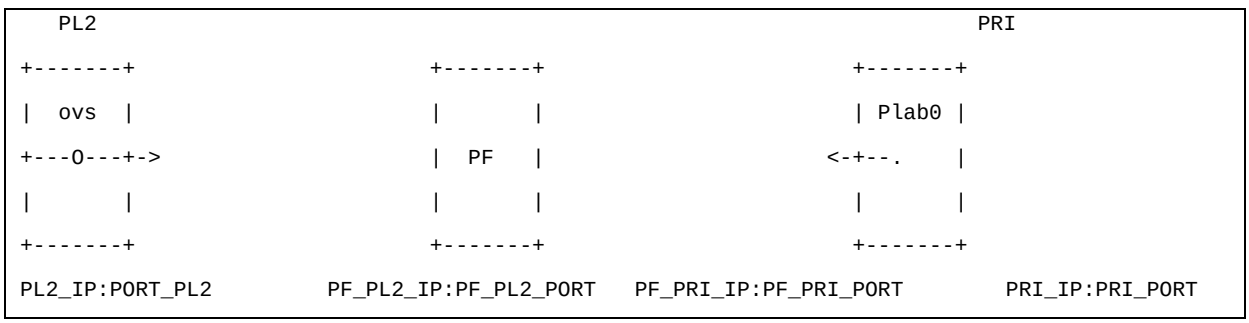


**Figure 4 – Port Forwarder to solve NAT issues: detail with IP addresses and UDP ports**

In the direction PL2 to PRI, the PF will forward packets as follows

destination -> PF_PL2_IP:PF_PL2_PORT source -> PL2_IP:PL2:PORT

remapped into

destination -> PRI_IP:PRI_PORT source -> PF_PRI_IP:PF_PRI_PORT

In the direction PRI to PL2, the PF will forward packets as follows

destination -> PF_PRI_IP:PF_PRI_PORT source -> PRI_IP:PRI:PORT

remapped into

destination -> PL2_IP:PL2_PORT source -> PF_PL2_IP:PF_PL2_PORT

Sequence of operation is as follows:

1) Assign PORT_PF_PUB and PORT_PRI
2) Run the setup script in PLE using PORT_PF_PUB (to be associated to PF_PUB_IP)
3) Read from the console output PORT_PL2 information
4) A) activate PF using with PORT_PL2 information
   B) activate tunnelling in PRI using PORT_PL2

We have made experiment with two different tools for port forwarding: 1) iptables 2) socat

### 7.3.1 Example scenario in our lab

We have created an example scenario with our own PF

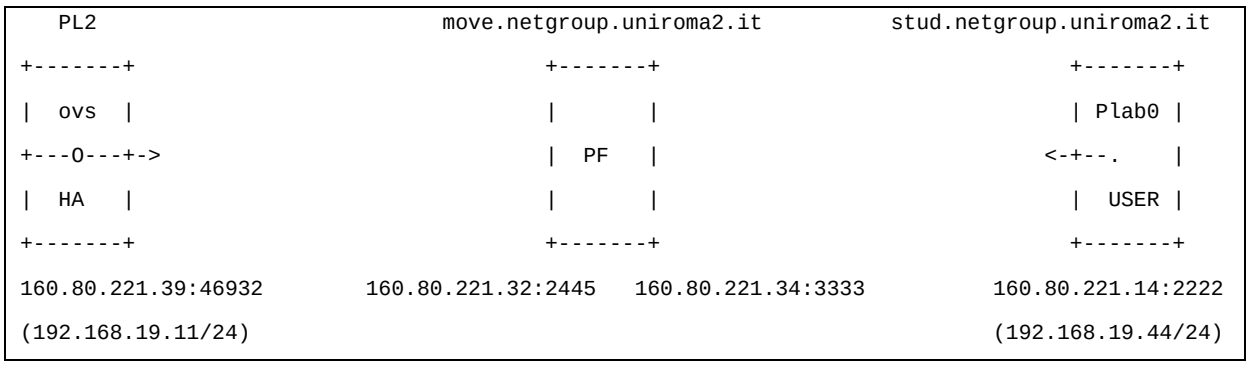


**Figure 5 – Example scenario with external node in our lab**

Note: the PL2 port (46932) is dynamically assigned and shown in “Makefile” script output.

In PL2 we have run script “Makefile” using the “conf.mk” file below in order to create:

```
SLICE=uniroma2_xpress
HOST_HA=planet-lab-node2.netgroup.uniroma2.it
IP_HA=192.168.19.11/24
HOST_USER := move.netgroup.uniroma2.it
LINKS :=
LINKS += HA-USER
EXTERNAL_HOSTS := USER
EXTERNAL_PORT_USER := 2445
```

In “stud.netgroup.uniroma2.it” (USER) we have run (as root) the “tunproxy2.py” python program:

# ./tunproxy2.py  -t 160.80.221.39:46932 -p 2445 -e -d -a 192.168.19.44/24

In PF we have forwarded packets as shown below:

Direction HA > USER

destination -> 160.80.221.32:2445 source -> 160.80.221.39:46932

remapped into

destination -> 160.80.221.14:2222 source -> 160.80.221.34:3333

Direction USER > HA

destination -> 160.80.221.34:3333 source -> 160.80.221.14:2222

remapped into

destination -> 160.80.221.39:46932 source -> 160.80.221.32:2445

In “move.netgroup.uniroma2.it” in order to set the forwarding rules (above) we have created a text file, named “pl_config”, with inside:

```
PL2_IP=160.80.221.39
PL2_PORT=46932
PF_PL2_IP=160.80.221.32
PF_PL2_PORT=2445
PF_PRI_IP=160.80.221.34
PF_PRI_PORT=3333
PRI_IP=160.80.221.14
PRI_PORT=2222
```

The PL2_PORT can be inserted only after the make script on PLE is run.

To use iptables run "iptables_pl" script using the command:

# ./iptables_pl.sh-f pl_config

To use socat tool run “socat_pl” script with command:

# ./socat_pl.sh -f pl_config

NOTE: if socat is used, the tunnel must be setup by sending packets from host USER to HA (e.g. ping 192.168.19.11 from USER).

## 7.3.2 Example scenario NITOS / PLE

Hereafter we describe an example scenario with a L2 tunnel between NITOS and PLE (not yet tested!!)

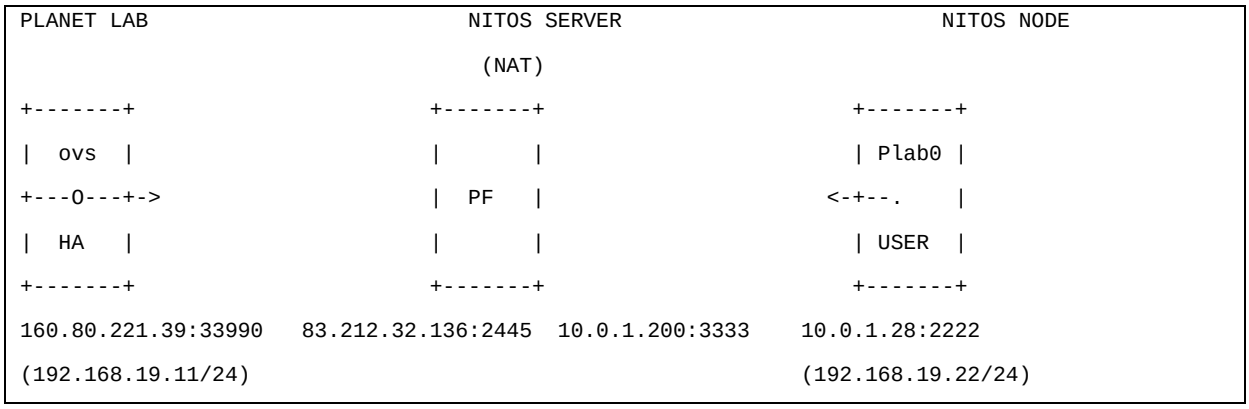


**Figure 6 – Example scenario with external node in NITOS**

Note: the PL2 port (33990) is dynamically assigned and shown in “Makefile” script output.

In PL2 we have run script “Makefile” using the “conf.mk” file below in order to create:

```
SLICE=uniroma2_xpress
HOST_HA=planet-lab-node2.netgroup.uniroma2.it
IP_HA=192.168.19.11/24
HOST_USER := 83.212.32.136
LINKS :=
LINKS += HA-USER
EXTERNAL_HOSTS := USER
EXTERNAL_PORT_USER := 2445
```

In “stud.netgroup.uniroma2.it” (USER) we have run (as root) the “tunproxy2.py” python program:

# ./tunproxy2.py  -t 160.80.221.39:33990 -p 2445 -e -d -a 192.168.19.22/24

In NITOS SERVER is required to forward packets as shown below:

Direction HA > USER

destination -> 83.212.32.136:2445 source -> 160.80.221.39:33990

remapped into

destination -> 10.0.1.200:3333 source -> 10.0.1.28:2222

Direction USER > HA

destination -> 10.0.1.200:3333 source -> 10.0.1.28:2222

remapped into

destination -> 160.80.221.39:33990 source -> 83.212.32.136:2445

In “move.netgroup.uniroma2.it” in order to set the forwarding rules (above) we must create a text file, named “pl_Nitos_config”, with inside:

```
PL2_IP=160.80.221.39
PL2_PORT=33990
PF_PL2_IP=83.212.32.136
PF_PL2_PORT=2445
PF_PRI_IP=10.0.1.200
PF_PRI_PORT=3333
PRI_IP=10.0.1.28
PRI_PORT=2222
```

The PL2_PORT can be inserted only after the make script on PLE is run.

To use iptables run "iptables_pl.sh" script using the command:

# ./iptables_pl.sh -f pl_Nitos_config

To use socat tool run “socat_pl” script with command:

# ./socat_pl.sh  -f pl_Nitos_config

NOTE: if socat is used, the tunnel must be setup by sending packets from host USER to HA (e.g. ping 192.168.19.11 from USER).

## 7.4 Creation of a Tunnel between PlanetLabEurope and an external node behind NAT

Here after we describe ho to connect a PlanetLabEurope (PLE) node with an external node behind a NAT.

Let us consider the scenario depicted below

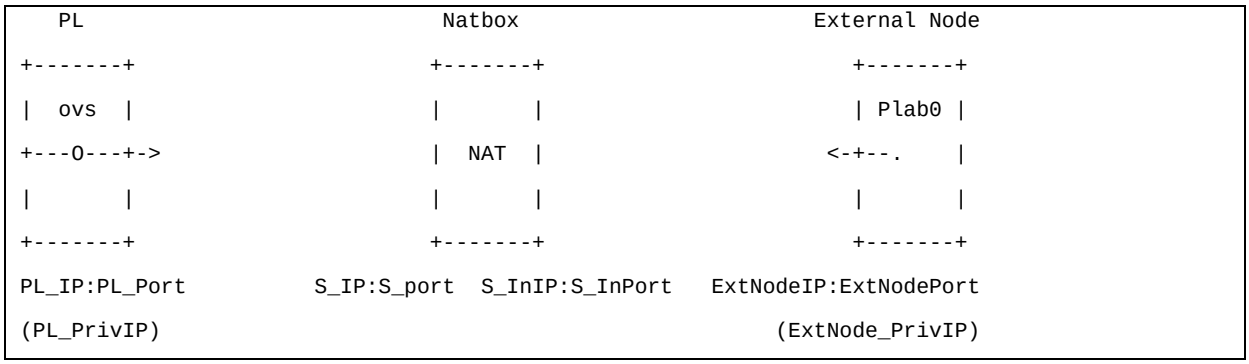


**Figure 7 – Interconnection with an external node through a NAT**

Note: PL_PrivIP and ExtNode_PrivIP are the overlay private IP addresses.

Steps to create a tunnel between a PlanetLab node and the external node:

1. In external node type (in a terminal) the command below in order to discover the public IP address of the Natbox:

   # wget -qO- http://ipecho.net/plain ; echo

   (in output the public IP)

2. Run “Makefile” using the “conf.mk” file in order to create an overlay network in a PLE slice. The overlay network does not need to include any reference to the External Node

   # make -j

   Below is an example of “conf.mk” file to configure PlanetLab (internal) nodes

```
SLICE=uniroma2_xpress
HOST_HONE=planet-lab-node2.netgroup.uniroma2.it
IP_HONE=192.168.19.120/24
HOST_HTWO=merkur.planetlab.haw-hamburg.de
IP_HTWO=192.168.19.121/24
LINKS :=
LINKS += HONE-HTWO
```

3. In a Planetlab node (PL) create a new virtual cable endpoint attached the switch used. This will be the end-point of the tunnel towards the External Node.

# sliver-ovs create-port [SwitchName] [PortName]

(e.g. sliver-ovs create-port uniroma2_xpress HA-EXT )

4. In PL discover the internal port (PL_Port) of the node using:

   # sliver-ovs get-local-endpoint [PLPortName]

   (e.g. sliver-ovs get-local-endpoint HA-EXT)

5. In PL find the public IP (PL_IP) of the node and the interface used

   # ip addr

6. In PL run the “tcpdump” tool to sniff on the public interface [int], filtering by source IP address

   # tcpdump -i  [int] udp src [S_IP] -nn -s0

   (e.g. tcpdump -i eth0  src 83.212.32.136 -nn -s0)

7. In the External node run “tunproxy2.py” using any port and specifying with argument [-a] the overlay private IP address node (ExtNode_PrivIP)

   # ./tunproxy2.py -t [PL_IP:PL_Port] -p [ExtNodePort] -a [ExtNode_PrivIP/SubnetMask] -e -d

   (e.g. ./tunproxy2.py -t 195.148.124.74:48360 -p 2222 -a 192.168.19.122/24 -e –d)

8. In External node run a ping toward the PlanetLab node using the overlay private IP address (setting frequency of ping to 10 seconds)

   # ping -i 10 [PL_PrivIP]

   (e.g. ping -i 192.168.19.122)

9. In PL node read the external server’s port (S_Port) printed on the tcpdump standard output (see step 6)

   Something like below:

   10:50:59.493509 IP 83.212.32.136.4567 > 195.148.124.74.48360: UDP, length 90

   10:50:59.973579 IP 83.212.32.136.4567 > 195.148.124.74.48360: UDP, length 78

   In this case the external port is 4567

10. In PL set the remote endpoint

    # sliver-ovs set-remote-endpoint [PLPortName] [S_IP] [S_Port]

    ( e.g. sliver-ovs set-remote-endpoint HA-EXT 83.212.32.136 4567 )

11. In PL ping the External node using the overlay private IP address in order to check the connection

    # ping -c7 [ExtNode_PrivIP]

    (e.g. ping -c7 192.168.19.122)

## 7.5 Details on the tools

### 7.5.1 IPTABLES_PL

“iptables_pl.sh” scripting allows to set the forwarding rules using iptables unix tool

You can set the “default parameters” editing “iptables_pl.sh” and changing port and ip address in the section “Default Values”, in the top of the script.

When you run “iptables_pl.sh” it generates a text file named “iptables_pl.out”.

It contains the parameters of the configuration set.

You can use it to remove the last configuration (see [-r] option) or configure again the same forwarding rules (see [-f] option).

IMPORTANT: “iptables_pl.out” file is overwritten each time that you run again the script

To show a commands help type:

# ./iptables_pl.sh -h

To set the “default parameters” forwarding rules (i.e. using the address parameters configured in the script)

# ./iptables_pl.sh –d

To manually enter the parameters of the topology

# ./iptables_pl.sh -a [PL2_IP] [PL2_PORT] [PF-PL2_IP] [PF-PL2_PORT] [PF-PRI_IP] [PF-PRI_PORT] [PRI_IP] [PRI_PORT]

To use a configuration file that provides all address parameters:

# ./iptables_pl.sh -f [FileName]

To change only the PLE dynamic port with respect to the “default parameters” forwarding rules (i.e. using the address parameters configured in the script):

# ./iptables_pl.sh –p 34567

To change only the PLE dynamic port with respect to the parameters provided in a file:

# ./iptables_pl.sh –f [FileName] –p 34567

To remove the “default parameters” rules

# ./iptables_pl.sh –r default

To remove the forwarding rules of the specified configuration

# ./iptables_pl.sh -r [ConfigurationFileName]

NOTE: using “iptables_pl.out” file (automatically created) you can remove the last configuration set

### 7.5.2 SOCAT_PL

“socat_pl.sh” scripting allows to forward packets using socat unix tool.

You can set the “default parameters” editing “socat_pl.sh” and changing port and ip address in the section “Default Values”, in the top of the script.

To use “Default Values”

# ./socat_pl.sh -d

To use the “Default Values” and change only the dynamic PlanetLab port (shown in “Makefile” script output)

# ./iptables_pl.sh -p [PortNumber]

To use a configuration text file:

# ./iptables_pl.sh -f [FileName]

To use a configuration text file and change only the dynamic PlanetLab port (shown in “Makefile” script output)

# ./iptables_pl.sh -f [FileName] –p [PortNumber]

## 7.5.3 TUNPROXY2

“tunproxy2.py” is a Python program to run the server of the UDP socket with PlanetLab node. Furthermore it creates virtual TAP interface sets up the overlay private IP address to node

Now a brief list of the useful options:

# ./tunproxy2.py  -t 160.80.221.39:33990 -p 2445 -e -d -a 192.168.19.22/24

To specify IP address and port of the remote PlanetLab node

>> [-t] [PL_IP:PL_PORT]

To set the local port of the node (where is run)

>> [-p] [PortNumber]

To use TAP tunnel

>> [-e]

(if you want a TUN tunnel use [-n])

To set in debug mode

>> [-d]

To set the IP address of the virtual TAP interface created

>> [-a ] [IPaddress/subnetmask]

(specify the subnet mask using the CISCO style)

## 7.5.4 TUNPROXY.C

“tunproxy.c” is a *C* source code to run the server of the UDP socket with PlanetLab node.

To run the program it must be compiled using the command below:

# gcc tunproxy.c -o tunproxy

How to run the program

# ./tunproxy  -t [PL_IP:PL_PORT] -p [LocalPort] -e

(e.g.  ./tunproxy -t 160.80.221.39:33990 -p 2445 -e)

[-e] specifies TAP tunnel.