GÉANT Innovation Programme

Extended Version of Project Final Report

UCSS

User Controlled SD-WAN Services with Performance Monitoring over GÉANT

Date:	**February 2022**
Authors	**Carmine Scarpitta, Giulio Sidoretti, Paolo Lungaroni, Francesco Lombardo, Andrea Mayer, Stefano Salsano, Marco Bonola**

1. Project Summary

Software Defined Wide Area Networks (**SD-WAN**s) [1] securely build interconnections and VPNs among end-user access networks and applications hosted in the clouds. The SD-WAN architecture is based on edge nodes controlled by an SDN controller. The SD-WAN Edge Nodes can operate in different scenarios: 1) within the provider network (i.e. under the control of the network operator); 2) outside the provider network (i.e. with no control over the transport services); 3) outside the provider network, but with some possibility of interaction with the provider network. Hence it is possible to leverage any combination of transport services, both operator-provided services backed by SLAs and best-effort services.

The UCSS project (User Controlled SD-WAN Services with Performance Monitoring over GEANT) focuses on scenario 2, in which the SD-WAN services are built with no interaction with the provider network. An open source SD-WAN solution, called EveryWAN has been enhanced to provide support for IPv4 over SRv6 (IPv6 Segment Routing) with monitoring of delay performances. The EVeryWAN solution is based on software Edge Nodes controlled by an EveryWAN controller.

The UCSS project has deployed a number of software EveryWAN Edge Nodes, inside a number of Linux VMs in order to demonstrate the feasibility of User Controller Virtual Private Networks based on IPv6 transport (more specifically, Segment Routing over IPv6 transport).

Two Linux Virtual Machines have been deployed inside University Campus Networks connected to NRENs (in Italy and Switzerland), two Virtual Machines have been installed in NREN datacenters (in Italy and Hungary), one Virtual Machine has been installed on CloudLab an academic testbed facility in the US (connected using an IPv6 tunnel broker), and other Virtual Machines for the experiments have been deployed on Personal Computers attached to a second IPv6 tunnel broker and to an italian Internet Service provider (TIM).

The project has analyzed the IPv6 connectivity "transparency" among the different Edge Nodes deployed on the Virtual Machines and has adapted the SRv6 VPN service to the IPv6 "transparency" of the different sites.

We have identified some issues that cause a poor performance (low throughput) of the VPN in some sites.

We have also shown that, when the required IPv6 transparency level is available, it is possible for the end-user to build SD-WAN services using SRv6 transport. We have demonstrated the functionality of the delay monitoring framework for the provided VPN services.

The software components that have been developed in the UCSS project are available under a liberal open source license.

2. Technical description

2.1. Concept and objectives

The objective of the UCSS project is to deploy a service for creating overlay VPNs for end users based on SRv6 with an integrated Performance Monitoring framework. More in general, the VPN services that will be provided fall into the category of SD-WAN (Software Defined - Wide Area Network) services, hence the name of the project is UCSS: User Controlled SD-WAN Services with Performance Monitoring over GEANT.

SRv6 means Segment Routing over IPv6, this architecture has been standardized in the last years by IETF [1], [2], and still there is a lot of ongoing standardization, research and implementation activity around SRv6. The SRv6 architecture offers a lot of flexibility and features both to the network provider and to the end-user to build advanced services, thanks to the "Network Programming" model described in [2]. There is also a large ecosystem of Open Source components for SRv6 (based on the Linux Operating System) which will be the basis for our developments in the UCSS project.

UCSS focuses on providing SD-WAN services over best effort IPv6 connectivity. Therefore, a prerequisite for building an SD-WAN service using SRv6 is that IPv6 transport connectivity is available up to the edge nodes that are at the border of the SD-WAN service. This basic prerequisite is not enough, as further conditions are needed to be able to successfully deploy SRv6 services (in particular, SRv6 tunnels). Moreover, the configuration of the SRv6 tunnels can change depending on some characteristics of the available IPv6 best effort connectivity. Being a relatively new technology, we have not found any operational analysis and discussion on how to build the SRv6 tunnels depending on the connectivity characteristics. Therefore, in the UCSS project we have:

1. experimentally assessed the different characteristics of best effort IPv6 connectivity in a number of sites
2. defined a classification of IPv6 "transparency" based on the results of the assessment
3. designed the configuration of the SRv6 tunnels based on the IPv6 transparency levels of the sites

Then we adapted the EveryWAN Open Source SD-WAN solution [3] to support the configuration of the tunnels based on the results of the IPv6 transparency assessment of the different sites.

As regards the monitoring of the performances seen by the end-users we have:

1. extended the SRv6 tunneling mechanism in Linux by supporting a delay measurement and monitoring solution
2. extended the EverWAN framework and its GUI to integrate the delay measurement and monitoring solution.

2.2. Results and lessons learned

2.2.1. UCSS deployment

In order to run the needed experiments, we have deployed several nodes across Europe. The deployed nodes represent the Edge Nodes of the SD-WAN service and also emulate the user devices in the VPNs. The nodes are actually Virtual Machines (VMs) based on Linux Operating Systems. Some of these VMs are located inside University Campus networks which are connected to the respective NREN. Other VMs are hosted on datacenters of the NRENs. The GEANT network in turn interconnects the NRENs. We also have nodes that are outside the NRENs to test SD-WANs which are also connecting "external" edge nodes. Among the external nodes we included two tunnel brokers: one is operated by Hurricane Electric, one of the major transit providers in the world and located in Paris; another is operated by a small Italian ISP (6project.org) and the tunnel broker is located in Frankfurt. These tunnel brokers provide access to the public IPv6 network to hosts that have only IPv4 connectivity. Finally, we included another external node with public IPv6 connectivity offered by a commercial Italian network provider (TIM).

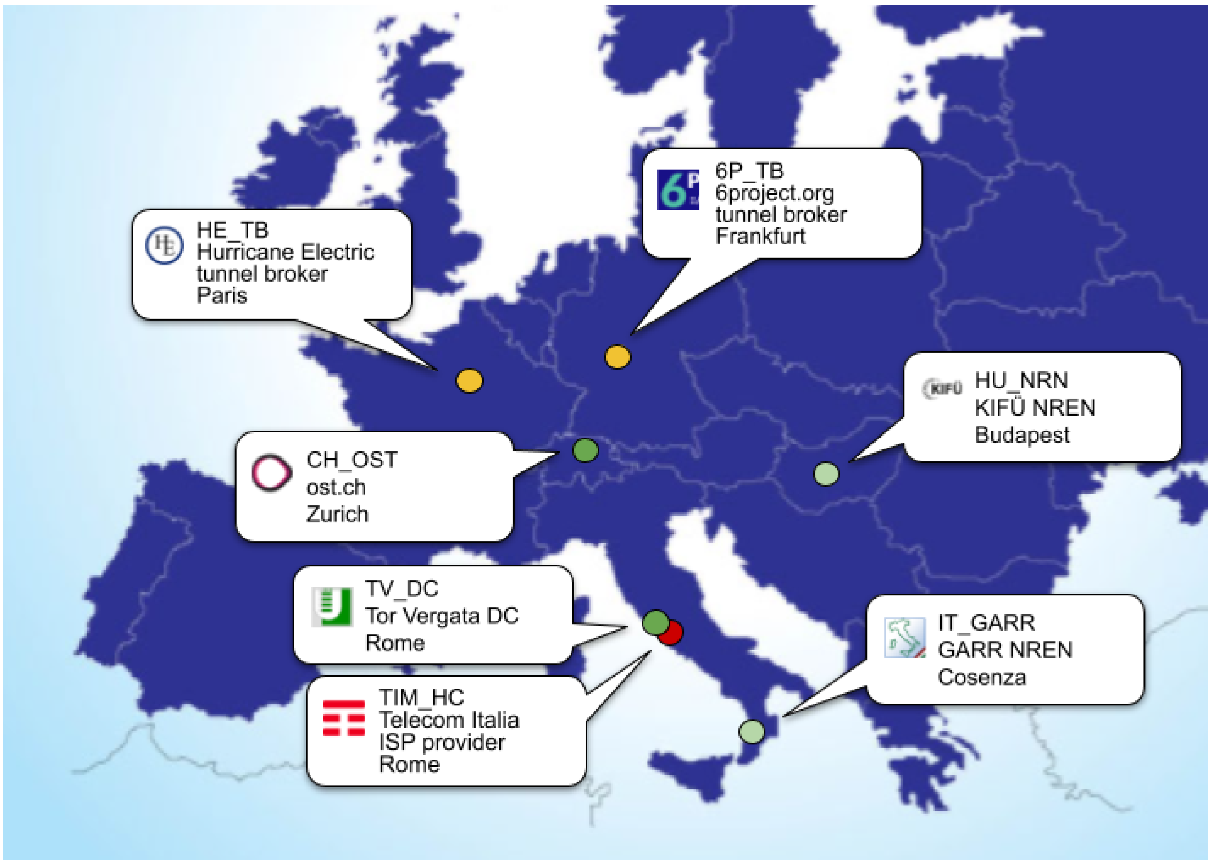


Figure 1 - UCSS deployed nodes

Figure 1 shows the position of the nodes in the European territory. University endpoints are marked in green, while the nodes in the NREN datacenters are marked in light green. The "commercial" connection provided by TIM ISP is shown in red (it is a home gateway located in Rome). The position of the IPv6 tunnel broker gateways is shown in yellow. We can connect "remote" nodes with IPv4 connectivity to the IPv6 tunnel brokers, and a public IPv6 connectivity will be tunneled to the remote node. In particular, the connection to the Hurricane Electric requires a public IPv4 address, while the connection to the tunnel broker 6project.org tunnel broker can be created also with a private IPv4 address connected to the public IPv4 Internet through a NAT. In our experiments, we have connected the Hurricane Electric tunnel broker with a server running in the CloudLab [4] [5] data center, which is a research experimental testbed facility in the US. We have used the 6project.org tunnel broker to opportunistically connect VMs running on our Personal Computers.

The table below shows the list of nodes with their IPv6 addresses and the type of associated connectivity. Subsequently, for each node, a brief description of the hardware and connectivity characteristics.

Name	Location	IPv6 Addresses	Notes
TV_DC	VM in Tor Vergata University Data Center - Rome	2001:760:4016:1200:5054:ff:fe7f:f5a8	
CH_OST	VM in CH OST University near Zurich	2001:620:130:b100:250:56ff:fe8b:a942	
IT_GARR	VM in GARR NREN Data Center - Cosenza	2001:760:ffff:b4:8000::4	
HU_NRN	VM in HUNGARY NREN Budapest	2001:738:0:527:f816:3eff:fedd:c8a4	
HE_TB	Hurricane Electric tunnel broker (ip6-in-ip4) with PoP in Paris (FR)	2001:470:1f12:35f::2	Endpoint placed on CloudLab testbed in the US
6P_TB	6project.org tunnel broker (openVPN), with PoP in Frankfurt (DE)	2a05:dfc7:46:babb:a2a7::2	Openvpn to Frankfurt PoP. Works also behind NAT. We connected VMs running on our PCs
TIM_HC	Commercial connection by Telecom Italia ISP - Rome	2a01:2000:2001:cee7:596e:68ed:9e64:961	IPv6 dynamically assigned

TV_DC This node is hosted in the datacenter of the department of electronic engineering at University of Rome “Tor Vergata”. The node is a VM with 4 vCPU at 2.2 GHz, 8 GB of RAM, 30 GB disk and 1 Gbps network interface. The IPv6 connection is provided by the University's IT department by propagating a /64 subnet directly to our data center. The node address is acquired through stateless address autoconfiguration (SLAAC).

CH_OST This node is a VM located in the datacenter of the OST University in Switzerland. The characteristics of the VM is 2 vCPU at 2.3 GHz, 4 GB of RAM, 12 GB disk and 1 Gbps virtual network interface. The subnet assigned is a /64 of the university prefix by SLAAC.

IT_GARR This node was placed in the OpenStack-based GARR cloud infrastructure. The VM is a 4 vCPU at 2.6 GHz, 4 GB of RAM, 350 GB disk and 1 Gbps virtual network interface. The connection is provided by OpenStack service Neutron and is assigned an /72 IPv6 subnet.

HU_NRN Is located at the Hungarian NREN datacenter in Budapest. The node is a VM with 2 vCPU at 2.2 GHz, 4 GB of RAM, 340 GB disk and 1 Gbps virtual network interface. The subnet assigned is a /64 of the NREN prefix, by stateless address autoconfiguration, but only the /128 IPv6 address assigned by hypervisor is allowed to access on the public network.

HE_TB Is a tunnel broker offered by Hurricane Electric that allows IPv6 connectivity on IPv4 only devices. The system encapsulates the IPv6 packets generated by the node into IPv4 packets which are then sent to the gateway endpoint which decapsulates them and forwards the IPv6 packets to the native IPv6 network. The bare metal machine is a 32 core CPU at 2.40 GHz, 128 GB of RAM and 1 TB HDD. The connectivity is 1 Gbps on the physical interface. The VM is assigned a public IPv4 address with which it is possible to terminate the IPv6-in-IPv4 tunnel. Basic Hurricane Electric provides a /64 subnet, but it is also possible to request a /48.

6P_TB 6project.org is a small Italian ISP that aims to provide IPv6 connectivity to devices that do not have a public IPv4 address, but are located behind a NAT. It is especially useful when located behind a Carrier Grade NAT on which it is not possible to define passthrough rules for IPv6-in-IPv4. To bypass NAT, an OpenVPN connection in IPv4 is first established and IPv6 connectivity is provided within this. The subnet is a /80 and the gateway is located in Frankfurt.

TIM_HC IPv6 connectivity provided by the Italian ISP TIM. Declared as experimental by TIM itself, it is not supplied to customers as standard, but must be requested by activating a special secondary PPPoE connection with specific username and password. The subnet assigned is a /64, but the prefix varies dynamically in the pool of the operator prefix delegated to TIM by the RIPE.

All nodes are managed by SSH connection over IPv4 or IPv6.

2.2.2. Assessment of SRv6 "transparency": methodology and results

All nodes are connected to the public Internet through their IPv6 address. In fact, the first basic test of connectivity was pinging Google's IPv6 DNS (2001:4860:4860::8888). The result was positive in all configurations and we proceeded with the subsequent tests.

```
TV_DC$ ping6 -c 2 2001:4860:4860::8888
PING 2001:4860:4860::8888(2001:4860:4860::8888) 56 data bytes
64 bytes from 2001:4860:4860::8888: icmp_seq=1 ttl=117 time=11.9 ms
64 bytes from 2001:4860:4860::8888: icmp_seq=2 ttl=117 time=12.0 ms

--- 2001:4860:4860::8888 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 2ms
rtt min/avg/max/mdev = 11.930/11.989/12.049/0.124 ms
```

Our goal is to assess the "transparency" of IPv6/SRv6 connectivity among nodes. There can be many factors that reduce the "transparency" of the connectivity: 1) filtering by the firewalls of the infrastructure that is hosting the node/VM; 2) filtering by the firewalls of the campus/corporate/home network in which the node/VM is located; 3) filtering by the transit networks in the path between the campus/corporate/home networks.

Note that in order to assess the "transparency" of IPv6/SRv6 connectivity we always need to consider two endpoints. Having N endpoints, in principle we could assess the transparency among all the possible couples of endpoints, which are N*(N-1)/2. As this would be impractical, we have started to take the node in Tor Vergata University as a reference endpoint and started performing the tests on the 6 couples of endpoints shown in figure 2. We choose the node in Tor Vergata because we know it has an unrestricted IPv6/SRv6 connectivity. In figure 2. we also show the couple 7 between HE_TB and HU_NRN as an example of the other tests that could be performed between couples of endpoints not including the node in Tor Vergata university.

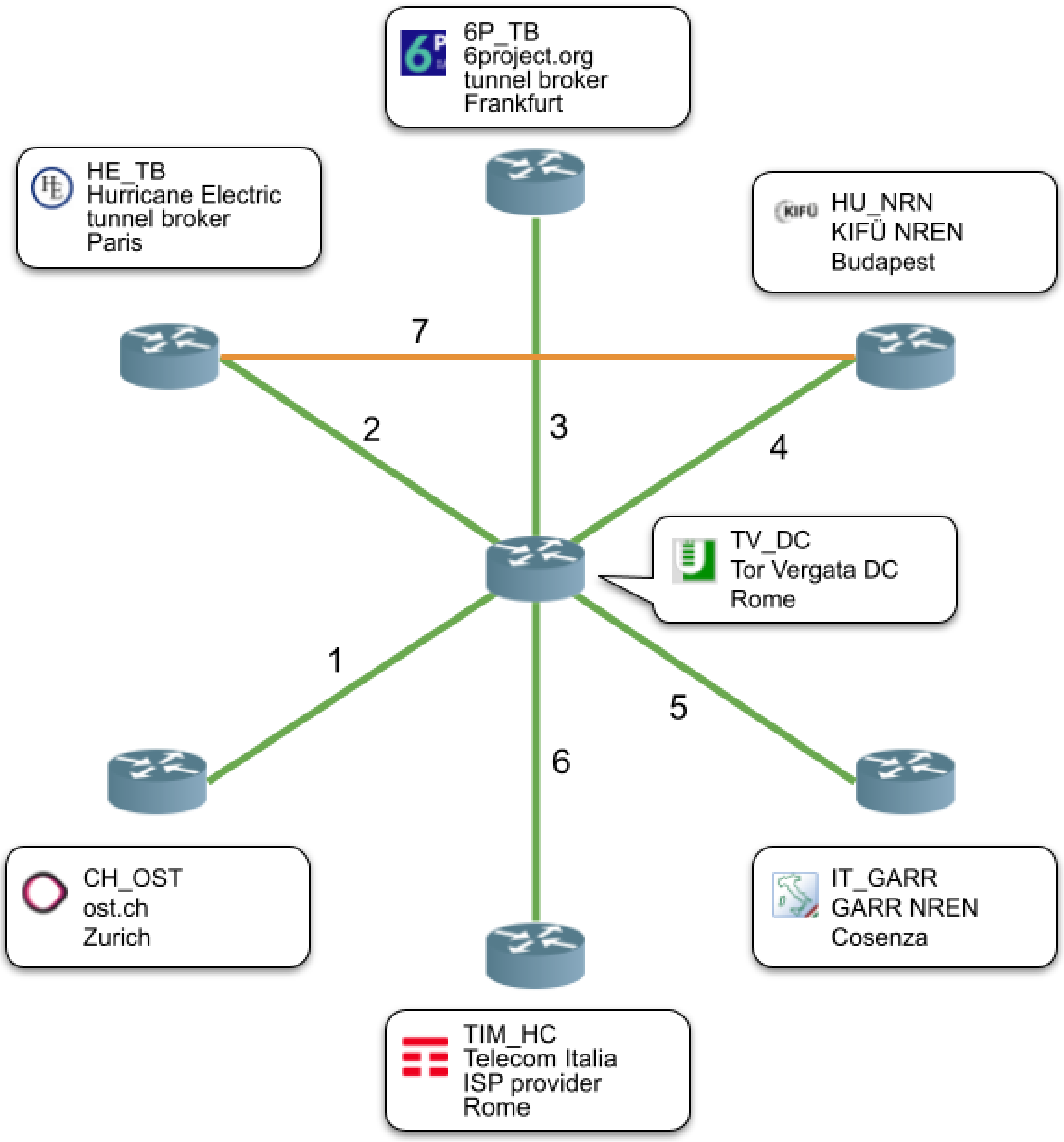


Figure 2 - UCSS deployed nodes

Figure 2 shows the topology considered for the connectivity tests. The topology is mainly a star with TV_DC on the center and other nodes at the edge. The link connections for the star topology are colored in green and the orange link numbered 7 is an example of interconnection between the HE_TB node and HU_NRN. Other links between nodes are numbered from 1 to 6. Ex. 1: TV_DC <–> CH_OS, 2: TV_DC <–>HE_TB, ect.

The connectivity tests between the nodes considered ping (delay) and tcp throughput (bandwidth). These tests have been performed in the following cases

- Plain IPv6;
- SRv6 (encap);
- IPv6-in-IPv6;

- SRv6-in-IPv6.

Figure 3 shows the stratification of the protocols in the different cases mentioned above.

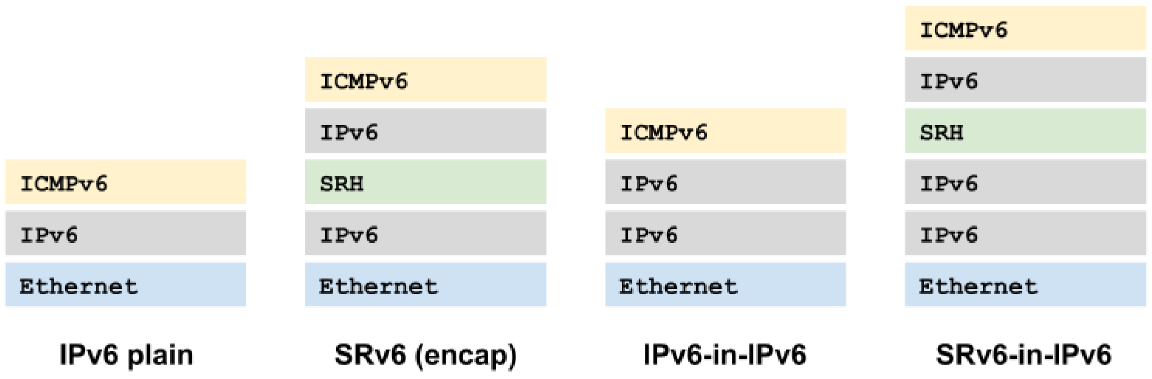


Figure 3 - Protocol layers in the different cases.

The case examined is the classic IPv6 ping which generates an ICMPv6 Echo Request and expects an ICMPv6 Echo Reply as a response. Layer 2 ethernet is common to all and is not subject to discussion, while layer 3 IPv6 changes depending on the case.

We start from the "plain IPv6" case, in which after the IPv6 header there is ICMPv6 as layer 4. The second case is the typical SRv6 encapsulated packet, in which after the IPv6 header we find the routing extension header with subtype 4, i.e. Segment Routing. A subsequent inner IPv6 header plus ICMPv6 follows SRH, because SRv6 provides an encapsulation function. The third case is IPv6-in-IPv6 where an inner IPv6 header plus ICMPv6 follows an outer IPv6 header which is used for forwarding over the transport network. The last case is the combination of SRv6 with IPv6-in-IPv6, where three IPv6 headers are stacked. The outer IPv6 header is used for forwarding on the transport network hop by hop, the middle IPv6 has the Segment Routing header and allows Network Programming operations, finally, being SRv6 an encap mode, the inner IPv6 header has the layer 4 which corresponds to ICMPv6. Case 3 and 4 are of interest if it is not possible to forward packets for case 2.

We have tested the 4 cases for each couple of endpoints considered in figure 2. In addition, we replaced ICMPv6 (ping) with TCP and repeated the tests for the 4 cases. Furthermore, we have considered the SRv6 *inline* mode that adds the SRH extension header between the IPv6 header and the layer 4. This functionality, although removed from the current SRv6 standard, is however implemented in the Linux kernel (actually, only the insertion or push is implemented, while the pop SRH function is intended as a flavor of the End behavior and is not yet implemented). We carried out a case 5 test with insert mode.

The first test performed is the ping between the two nodes. In particular for the star case between the TV_DC node and the node under test. Pinging is done in both directions. As an example we take the case of TV_DC <-> HE_TB (link 2):

```
TV_DC$ ping6 -c 2 2001:470:1f12:35f::2
PING 2001:470:1f12:35f::2(2001:470:1f12:35f::2) 56 data bytes
64 bytes from 2001:470:1f12:35f::2: icmp_seq=1 ttl=52 time=121 ms
64 bytes from 2001:470:1f12:35f::2: icmp_seq=2 ttl=52 time=121 ms

--- 2001:470:1f12:35f::2 ping statistics ---
```

```
2 packets transmitted, 2 received, 0% packet loss, time 3ms
rtt min/avg/max/mdev = 121.347/121.373/121.399/0.026 ms
TV_DC$ iperf3 -c 2001:470:1f12:35f::2
Connecting to host 2001:470:1f12:35f::2, port 5201
[  5] local 2001:760:4016:1200:5054:ff:fe7f:f5a8 port 55810 connected to
2001:470:1f12:35f::2 port 5201
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  72.8 MBytes  61.1 Mbits/sec  3267             sender
[  5]   0.00-10.00  sec  69.6 MBytes  58.4 Mbits/sec                  receiver

HE_TB$ ping6 -c 2 2001:760:4016:1200:5054:ff:fe7f:f5a8
PING 2001:760:4016:1200:5054:ff:fe7f:f5a8(2001:760:4016:1200:5054:ff:fe7f:f5a8) 56 data
bytes
64 bytes from 2001:760:4016:1200:5054:ff:fe7f:f5a8: icmp_seq=1 ttl=53 time=121 ms
64 bytes from 2001:760:4016:1200:5054:ff:fe7f:f5a8: icmp_seq=2 ttl=53 time=121 ms

--- 2001:760:4016:1200:5054:ff:fe7f:f5a8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 121.369/121.416/121.464/0.351 ms
HE_TB$ iperf3 -c 2001:760:4016:1200:5054:ff:fe7f:f5a8
Connecting to host 2001:760:4016:1200:5054:ff:fe7f:f5a8, port 5201
[  4] local 2001:470:1f12:35f::2 port 49796 connected to
2001:760:4016:1200:5054:ff:fe7f:f5a8 port 5201
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec   210 MBytes   176 Mbits/sec    0             sender
[  4]   0.00-10.00  sec   209 MBytes   175 Mbits/sec                  receiver
```

The second step is to establish an SRv6 connection between the two nodes.

In the SRv6 Network Programming model, SIDs are used to define network functions. A SID is composed of a Locator prefix which makes routing possible along the network and an Arguments part where the network functions are encoded. In our case we need to encapsulate an IPv6 overlay connection with two internal IPv6 addresses in an SRv6 packet encoded with a SID indicating the decap function in the receiving node. To form the SID as the locator part we take the public prefix assigned to the nodes and as argument we use the /64 part dedicated to the host assigning a number for the decap function. As an example if we define ::100 as a decap function by combining the prefixes of the nodes we will have the following SIDs:

- Decap on TV_DC: 2001:760:4016:1200::100
- Decap on HE_TB: 2001:470:1f12:35f::100

As a result, below we show the commands needed to establish a bi-directional SRv6 link between the two example nodes (assuming cafe::1 on TV_DC and cafe::2 on HE_TB).

```
TV_DC# ip -6 route add cafe::2 encap seg6 mode encap \
       segs 2001:470:1f12:35f::100 dev ens3
TV_DC# ip -6 route add 2001:760:4016:1200::100 encap seg6local \
       action End.DT6 table 255 dev ens3
TV_DC# sysctl -w net.ipv6.conf.all.proxy_ndp=1
TV_DC# sysctl -w net.ipv6.conf.ens3.proxy_ndp=1
TV_DC# ip neigh add proxy 2001:760:4016:1200::100 dev ens3
```

```
HE_TB# ip -6 route add cafe::1 encap seg6 mode encap \
       segs 2001:760:4016:1200::100 dev he-ipv6
HE_TB# ip -6 route add 2001:470:1f12:35f::100 encap seg6local \
       action End.DT6 table 255 dev he-ipv6
```

The NDP proxy is necessary in TV_DC since the packet comes from an ethernet layer 2 and it is necessary to respond with the interface mac address to the requests sent by the router. In the case of the HE_TB tunnel broker, as the packet comes from an IPv6-in-IPv4 encapsulation, there is no layer 2 and therefore the NDP proxy is not necessary.

From here it is possible to test the connectivity via ping and iperf3 (for tcp):

```
TV_DC# ping6 -c 2 cafe::2
PING cafe::2(cafe::2) 56 data bytes
64 bytes from cafe::2: icmp_seq=1 ttl=64 time=122 ms
64 bytes from cafe::2: icmp_seq=2 ttl=64 time=121 ms

--- cafe::2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 3ms
rtt min/avg/max/mdev = 121.424/121.471/121.519/0.351 ms
TV_DC# iperf3 -c cafe::2
Connecting to host cafe::2, port 5201
[  5] local cafe::1 port 53288 connected to cafe::2 port 5201
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec   105 MBytes  88.3 Mbits/sec  3276             sender
[  5]   0.00-10.00  sec   103 MBytes  86.6 Mbits/sec                  receiver

HE_TB# ping6 -c 2 cafe::1
PING cafe::1(cafe::1) 56 data bytes
64 bytes from cafe::1: icmp_seq=1 ttl=64 time=121 ms
64 bytes from cafe::1: icmp_seq=2 ttl=64 time=121 ms

--- cafe::1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 121.475/121.530/121.586/0.353 ms
HE_TB# iperf3 -c cafe::1
Connecting to host cafe::1, port 5201
[  4] local cafe::2 port 44316 connected to cafe::1 port 5201
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec  63.0 MBytes  52.9 Mbits/sec    0             sender
[  4]   0.00-10.00  sec  62.0 MBytes  52.0 Mbits/sec                  receiver
```

With SRv6 insert mode it is not possible to pop the header, therefore we report only the push command. However, it is possible to observe the arrival or not of the packet at its destination with tools such as tcpdump or wireshark. Assuming the pop function is ::101

```
TV_DC# ip -6 route add cafe::e2 encap seg6 mode inline \
       segs 2001:470:1f12:35f::101 dev ens3

HE_TB# ip -6 route add cafe::e1 encap seg6 mode inline \
       segs 2001:760:4016:1200::101 dev he-ipv6
```

A further step is to set up an IPv6-in-IPv6 tunnel and set an IPv6 address to the tunnel:

```
TV_DC# ip link add name tip6 type ip6tnl mode ip6ip6 \
       remote 2001:470:1f12:35f::2   \
       local 2001:760:4016:1200:5054:ff:fe7f:f5a8
TV_DC# ip link set tip6 up
TV_DC# ip address add 2001:db9::1/64 dev tip6



HE_TB# ip link add name tip6 type ip6tnl mode ip6ip6 \
       remote 2001:760:4016:1200:5054:ff:fe7f:f5a8   \
       local 2001:470:1f12:35f::2
HE_TB# ip link set tip6 up
HE_TB# ip address add 2001:db9::2/64 dev tip6
```

Also in this case it is possible to carry out the ping and iperf3 tests:

```
TV_DC# ping6 -c 2 2001:db9::2
PING 2001:db9::2(2001:db9::2) 56 data bytes
64 bytes from 2001:db9::2: icmp_seq=1 ttl=64 time=122 ms
64 bytes from 2001:db9::2: icmp_seq=2 ttl=64 time=122 ms

--- 2001:db9::2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 3ms
rtt min/avg/max/mdev = 121.626/121.692/121.759/0.355 ms
TV_DC# iperf3 -c  2001:db9::2
Connecting to host 2001:db9::2, port 5201
[  5] local 2001:db9::1 port 39050 connected to 2001:db9::2 port 5201
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  97.2 MBytes  81.6 Mbits/sec  2524             sender
[  5]   0.00-10.00  sec  94.5 MBytes  79.3 Mbits/sec                  receiver

HE_TB# ping6 -c 2 2001:db9::1
PING 2001:db9::1(2001:db9::1) 56 data bytes
64 bytes from 2001:db9::1: icmp_seq=1 ttl=64 time=121 ms
64 bytes from 2001:db9::1: icmp_seq=2 ttl=64 time=133 ms

--- 2001:db9::1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 121.421/127.573/133.725/6.152 ms
HE_TB# iperf3 -c 2001:db9::1
Connecting to host 2001:db9::1, port 5201
[  4] local 2001:db9::2 port 39826 connected to 2001:db9::1 port 5201
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec  51.6 MBytes  43.3 Mbits/sec    0             sender
[  4]   0.00-10.00  sec  50.1 MBytes  42.0 Mbits/sec                  receiver
```

As a last case we apply SRv6 to the previous IPv6-in-IPv6 tunnel just created

```
TV_DC# ip -6 route add cafe::f2 encap seg6 mode encap \
       segs 2001:db9::f100 dev tip6
```

```
TV_DC# ip -6 route add 2001:db9::100 encap seg6local \
       action End.DT6 table 255 dev tip6

HE_TB# ip -6 route add cafe::f1 encap seg6 mode encap \
       segs 2001:db9::100 dev tip6
HE_TB# ip -6 route add 2001:db9::f100 encap seg6local \
       action End.DT6 table 255 dev tip6

TV_DC# ping6 -c 2 cafe::f2
PING cafe::f2(cafe::f2) 56 data bytes
64 bytes from cafe::f2: icmp_seq=1 ttl=64 time=122 ms
64 bytes from cafe::f2: icmp_seq=2 ttl=64 time=122 ms

--- cafe::f2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 3ms
rtt min/avg/max/mdev = 121.531/121.532/121.533/0.001 ms
TV_DC# iperf3 -c cafe::f2
Connecting to host cafe::f2, port 5201
[  5] local cafe::f1 port 43182 connected to cafe::f2 port 5201
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  83.9 MBytes  70.4 Mbits/sec  4962             sender
[  5]   0.00-10.00  sec  81.3 MBytes  68.2 Mbits/sec                  receiver

HE_TB# ping6 -c 2 cafe::f1
PING cafe::f1(cafe::f1) 56 data bytes
64 bytes from cafe::f1: icmp_seq=1 ttl=64 time=121 ms
64 bytes from cafe::f1: icmp_seq=2 ttl=64 time=121 ms

--- cafe::f1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 121.459/121.563/121.667/0.104 ms
HE_TB# iperf3 -c cafe::f1
Connecting to host cafe::f1, port 5201
[  4] local cafe::f2 port 56730 connected to cafe::f1 port 5201
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec   208 MBytes   174 Mbits/sec    1             sender
[  4]   0.00-10.00  sec   207 MBytes   174 Mbits/sec                  receiver
```

The examples of the commands shown above describe all the tests carried out between two different nodes (in the case of TV_DC <-> HE_TB).
The following table summarizes the test results for the other nodes in the topology.
The commands given to the other nodes are almost similar and in any case are available in specific scripts in a dedicated repository.

TV_DC to	CH_OST	IT_GARR	HU_NRN	HE_TB	6P_TB	TIM_HC
ping6	OK	OK	OK	OK	OK	OK
icmpv6 - srh encap	NO	only OUT	OK	OK	OK	OK
icmpv6 - srh insert	NO	only OUT	NO	OK	OK	OK

icmpv6 - ip6-in-ip6	OK	NO	OK	OK	OK	OK
tcp6 iperf	OK	NO	OK	OK	OK	OK
tcp - srh encap	NO	only OUT	OK	OK	OK	OK
tcp - srh insert	NO	NO	NO	OK	OK	OK
tcp - ip6-in-ip6	OK	NO	OK	OK	OK	OK
ip6srh-in-ip6	OK	NO	OK	OK	OK	OK

The table shows that in general the SRv6 flows are supported by the research core network (GEANT) and by commercial networks. It seems that the ISPs in general do not oppose SRv6, allowing transit in any mode. On the other hand, local firewall policies in NRENs severely limit the use of SRv6 in certain scenarios.
Analyzing the case of the CH_OST node we see that the firewall blocks all IPv6 packets with SRH following the outer IPv6. A particular case is SRv6-in-IPv6 where the firewall can be crossed since the SRH it is placed in the middle header, while the outer header is formed by a normal IPv6 which is allowed.
As for the case of the VM in the cloud of GARR NREN, the VM is deployed on OpenStack which uses the Neutron module for the orchestration of the network inside the cloud instance. By default, Neutron is very strict and blocks all possible traffic. To have access to a single port or protocol it is necessary to unblock access with a specific rule in Neutron. The table shows that the Neutron default rules are applied which normally limit incoming connections. This allows, albeit in a limited way, the passage of some outgoing SRv6 packets.
Finally, the node at the Hungarian NREN places different limitations which in some way affect the correct transit of SRv6 flows. In particular, an IPv6 /128 has been assigned which considerably limits the possibility of using SRv6 behaviors. In addition, we experience that the available throughput is limited to a few Kbps.

In general it can be said that the IPv6 Internet is ready to use SRv6 in the WAN environment. The research and commercial core networks allow the transit of SRv6 flows and the ISP operators do not carry out any type of filtering, delivering everything received by the core network.
Instead, particular attention should be paid to the internal firewall and traffic management policies of the NRENs attached to the GEANT network, since for security or technical reasons they apply limitations to SRv6 flows in their networks and data centers.

2.2.2.1. Classification methodology

After performing an initial set of experiments, we understood that there are different possible outcomes from the point of view of the SRv6 transparency when sending packets between two endpoints. If it is possible to send packets with SRv6 encap (case 2) there is SRv6 full transparency. If case 2 test fails but it is possible to send IPv6-in-IPv6 packets (case 3), it is still possible to use Segment Routing with some limitations. Note that we have not found examples in which case 3 and case 4 give a different result. If case 3 and case 4 also fail, there is no possibility to use SRv6 between the endpoints. We have classified the outcomes in the following table.

Segment Routing transparency levels

T0	IPv64 in SRv6	Segment routing header in the outer IPv6 packet is supported. The next header is IPv4 or IPv6.
T1	IPv64 in IPv6	Segment routing header in the outer IPv6 packet is NOT supported. IPv4 in IPv6 and IPv6 in IPv6 are supported.
OP	opaque: no tunneling	IPv4 in IPv6 and IPv6 in IPv6 NOT supported… Useless

Based on the table above, we can classify the SRv6 transparency as T0 when we have full transparency, T1 when it is not possible to use the Segment Routing Header in the outer IPv6 packet, or OP (opaque) when SRv6 cannot be used.

We have also considered another classification dimension, which is based on the available IPv6 addressing range to reach an endpoint. In the nodes that we have deployed we have been given different ranges, from /64 up to a single IPv6 address, i.e. a /128. This is often due to firewall configurations. More in general, we have decided to classify the available IPv6 addressing range as shown in the following table.

Available public IPv6 addressing to reach the EveryEdge node

A0	/48
A1	/56
A2	/64
A3	/80
A4	/128

The combination of the transparency level and of the available addressing range is needed as an input to choose the SRv6 *policies* (or "*Segment Lists*") used for the SRv6 tunnels interconnecting two endpoints.

2.2.3. EveryWAN architecture

The EveryWAN architecture for SD-WAN services has been designed with SDN and NFV principles in mind. The target scenario foresees that Customer Edge (CE) Devices are replaced by Universal Customer Premise Equipment (uCPE) boxes, that integrate compute, storage and networking on COTS hardware giving the possibility of implementing new services as virtual functions to any site and optimizing the provisioning of the existing ones through orchestration. In general, any server providing computing, storage and network interfaces can be used as replacement of a CE equipment. This approach would also overtake the monolithic hardware trend of the legacy CE that has been a barrier for innovation for several years. In our specific solution proposed for UCSS our Customer Edge Device is a Linux Virtual Machine. Hence, we can exploit the SRv6 capabilities provided with the Linux kernel networking to build our SD-WAN services.

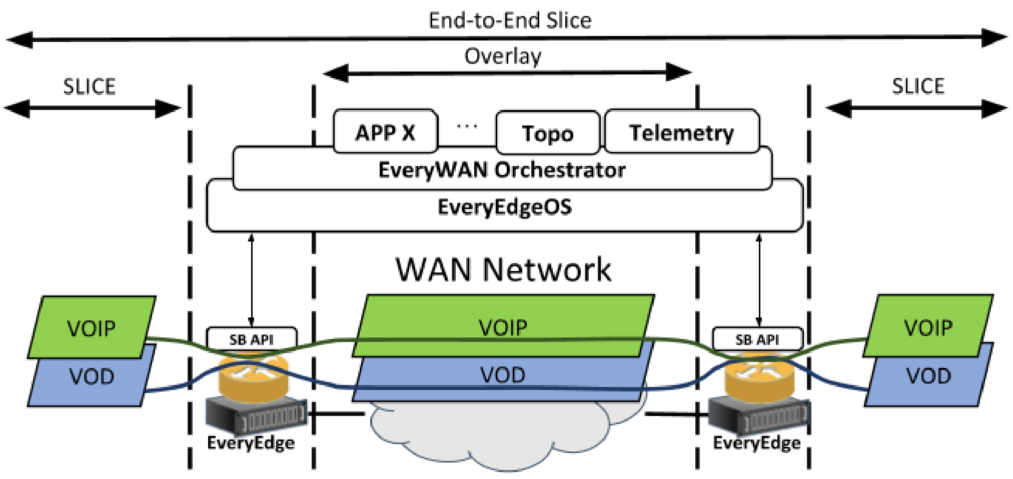


Figure 4 - EveryWAN architecture

Figure 4 shows the building blocks of the EveryWAN Architecture. The Edge Devices are called EveryEdge, they are deployed as virtual network functions (VNF) and are controlled by the EveryEdgeOS SDN controller. The SDN controller deals with many aspects of the device life cycle which include not only the management and the programming but also the initial device registration, authentication and configuration leveraging a Zero Touch Provisioning (ZTP) approach. An overarching orchestrator "EveryWAN Orchestrator" sits on top of this SDN architecture, which orchestrates and automates the deployment of the virtual routers and of the SD-WAN services to any edge site on a network. The orchestrator also offers a GUI through which the tenants can design the network topology, configure the services, manage the SD-WAN interconnections, the virtual devices and the users. The Network Operating System (NOS) and the orchestrator can run in a self-managed server or in a public Cloud. Instead, the tenants will deploy the EveryEdge nodes in all the sites where SD-WAN interconnections need to be established.

We assume that the EveryWAN users will deploy the EveryEdge images in their self-managed uCPEs distributed across the WAN sites. The container/VM images are shipped with a minimal configuration which allows the virtual routers to reach the controller, download the configuration and complete the provisioning process.

As regards the services, the VPNs are realized in a different way with respect to traditional provider based VPNs. Provider Edge (PE) based VPNs are now replaced by CE based VPNs. SRv6 is used as overlay technology in place of MPLS to realize end-to-end connectivity and build "virtual networks" on top of the WAN pipes. In the current version, the communication has been tested with an unsecure connection. In the future, IPSEC will be considered as the technology to secure the communication. With reference to Figure 4, we call *Slice* the edge segments where the applications and users are. The overlays (tunnels) are established between the Edge Devices. Finally, we define End to End Slices (E2E Slice) the composition of Slices and Overlays/Tunnels.

SRv6 as an overlay mechanism provides the ability of building different logical instances of a multipoint-to-multipoint network over the same WAN. This means that different applications can run on different slices and obtain the needed isolation requirements. An overlay approach also has the inherent advantage of abstracting the transport layer and being less dependent on the service providers and their networks. Thanks to this approach, in general several broadband technologies can be used in parallel, even including the legacy MPLS: overlays are not tied to a specific WAN and can use different connections, also in parallel, to implement load balancing policies and guarantee better performance.

We designed and implemented a basic service called EveryWAN Overlay Network (EON) as the basic building block of the EveryWAN architecture. EON can be used to support VPN use cases (e.g. interconnect different branch offices of a company through the ISP WAN) as well as to transport traffic of specific applications. The overlay is realized between end-points in Edge Devices belonging to the same tenant. The end-points are identified as IPv6 addresses, or better, SRv6 Segment IDs (SIDs) in the form of IPv6 addresses. At the time of writing, the connection is established only in a full-mesh fashion, i.e. every "local" Edge Device that has to send packets to another "remote" Edge Device will establish an SRv6 tunnel.

The SRv6 tunneling mechanism allows us to build an even more powerful construct over the EON service that is typically called in the industry as Network Slicing. Network slicing provides the means of building several instances of virtual networks over the same WAN connection. A typical usage of this functionality is the service oriented SD-WAN where tenants can redirect the traffic of a specific application over a defined End-to-end Slice. In this way, different applications can run in isolation and still share the same connectivity. We represent a Routed End-to-end Slice in Figure 5.

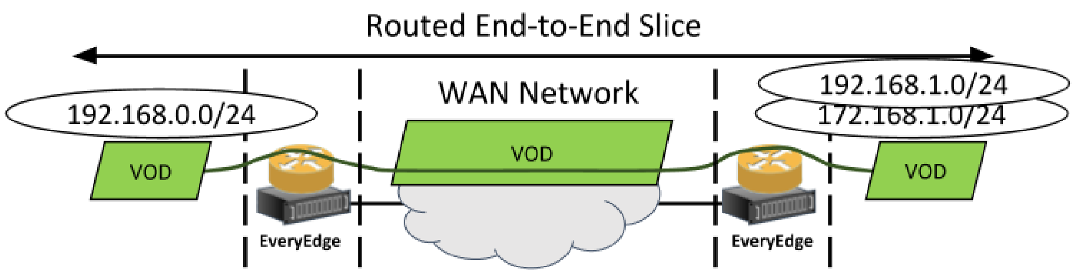


Figure 5 - Routed End-to-End Slice

A Routed End-to-End slice, shown in Figure 5, is a simplified implementation of a L3VPN where the devices attached to the remote sites belong to different broadcast domains (IP subnets) and each site's endpoint acts as gateway for these broadcast domains. It is not meant to allow the served end-points to send packets with arbitrary Ethertype since only Layer 3 traffic will be transported across the remote sites. Moreover, in this scenario it is possible to set up several broadcast domains (IP subnets) behind the slice endpoints using a multi-subnet configuration (Figure 5). A routed End-to-End slice can support IPv4 and IPv6 subnets (and also a combination of IPv4 and IPv6 subnets).

Each Edge Device has access to the Internet and has a public IPv6 address on which it can be reached by the other VPN endpoints. EveryEdge nodes can leverage multiple WAN connections to forward the traffic of the EON local slices. Overlays are not tied to a specific WAN: an EveryEdge node can select which WAN to use for the forwarding of the traffic belonging to a particular slice based on a customer defined policy.

2.2.3.1. EveryEdge node architecture

The EveryEdge node architecture (figure 6) foresees the coexistence of a local control logic based on distributed IP routing and of a classic SDN design in which the node implements a Southbound API towards an SDN controller. The SDN controller, leveraging a Southbound protocol, can instruct the nodes and program the rules in their routing tables. Similar solutions have been already proposed in literature for example [6] and [7]; others have been rolled out in production quite recently [8]. While these solutions, often referred to as hybrid IP/SDN, have been applied in datacenter fabric or in WAN scenarios as replacement of the WAN routers, the novelty of our approach lies in proposing and utilizing such hybrid approach for the commoditization of the Customer Edge routers.

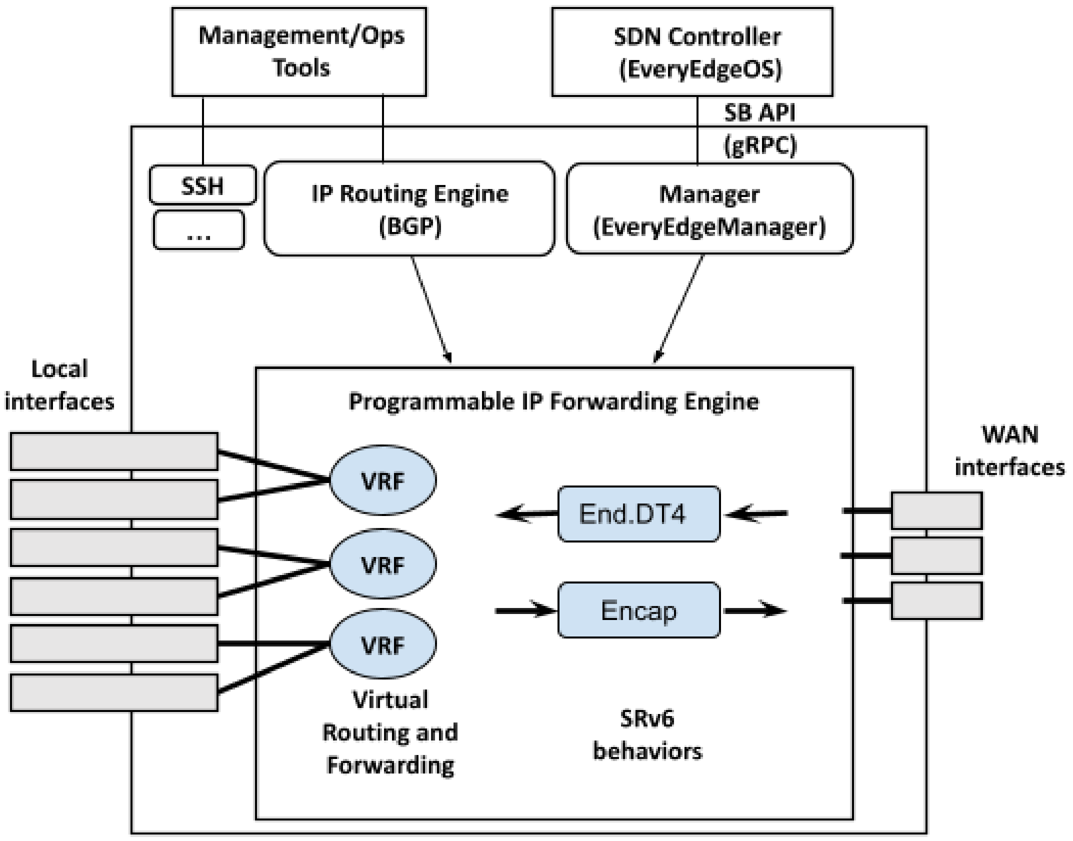


Figure 6 - Architecture of the Edge Device (EveryEdge)

We have realized the EveryEdge node leveraging commodity hardware and open source software. We did not reinvent the wheel but when needed we have built from scratch the missing functionalities. The IP forwarding engine is implemented using Linux networking. We have used a general purpose distribution of the Linux OS, the only requirement is the kernel to be recent (at least 5.11) in order to have native support for SRv6 End.DT4 behavior in the kernel space. The End.DT4 behavior (see [2]) makes it possible to encapsulate IPv4 packets inside SRv6. We have leveraged the approach designed in [7] to make it programmable.

We have also used several virtualization technologies offered by the Linux kernel to build up our slicing mechanisms and multi-tenancy. In particular, the VRFs (Virtual Routing and Forwarding) [9] are used to deploy several instances of logically decoupled slices. The EveryEdge node, with its virtualization mechanisms, guarantees the IP applications running on top of the Slices to be directly interconnected as if they were using their own routers.

Another important component of the architecture of the EveryEdge node is the EveryEdgeManager. It takes care of the initial configuration of the node: implements a ZTP (Zero-Touch Provisioning) approach which includes also the download of the bootstrap configuration of the routing daemon; it authenticates the device with the controller and implements NAT traversal protocols with the help of an additional control plane components. It supports in-band and out-of-band connectivity and can establish insecure or secure channels with the control plane. This wide range of choices avoid the need of setting up a separate out-of-band network and the same WAN interfaces can be used to reach the EveryEdgeController.

The EveryEdgeManager acts as mediator interacting on the southbound direction with the kernel of the EveryEdge router and on the northbound with the SDN controller: it translates the messages received over the Southbound API into actions to be sent to the kernel components. The communication library used to enable the communication with the Linux kernel is based on the open source project pyroute2 [10], a pure python netlink library that has been properly extended to support our needs. Last but not least, a number of management/operational protocols are used for the daily check routines, to keep the sessions alive with the controller and avoid the expiration of the NAT bindings when the Edge Node is connected to the controller through NAT.

The virtual slices are implemented through end-to-end SRv6 tunnels established between Edge routers. The IPv6 Segment Routing (SRv6) encapsulation allows us to interoperate naturally with existing IPv4/IPv6 networks in the customer sites. The only caveat is that SRv6 requires a WAN network supporting IPv6 transport.

2.2.3.2. Controller and Orchestrator architecture

The architecture of the controller, called EveryEdgeOS is shown in figure 7. At the highest level there is the EveryEdgeOS Agent, which acts as a mediator interacting on the south with the controller modules and on the north with the EveryWAN orchestrator. The EveryEdgeOS Agent translates the commands received from the orchestrator into actions to be sent to the EveryEdgeOS modules. In our current design, the controller has five modules: i) the Topology Manager (TM), which is responsible for building and maintaining an updated view of the network topology; ii) the Device Manager (DM), which deals with many aspects of the EveryEdge device life cycle, such as the device registration and authentication; iii) the Overlay Manager (OM), which computes paths and tunnels needed to implement the requested overlay by using the topology graph provided by the TM and the devices information provided by the DM; iv) Tenant Manager (TeM), which is responsible for the tenants registration and configuration; v) Statistics Collector (SC), which collects and reports statistics on the overlay networks and the devices.

The creation of the tunnels and the E2E slices is always initiated by the EveryEdgeOS Controller which receives the configuration on its NorthBound API. The EveryEdgeOS Controller allocates the specific information for the End-to-End slice, like the segment IDs, and then translates the configuration in a set of commands to be performed on the participating Edge Routers. The whole end-to-end process is orchestrated by the EveryEdgeOS controller.

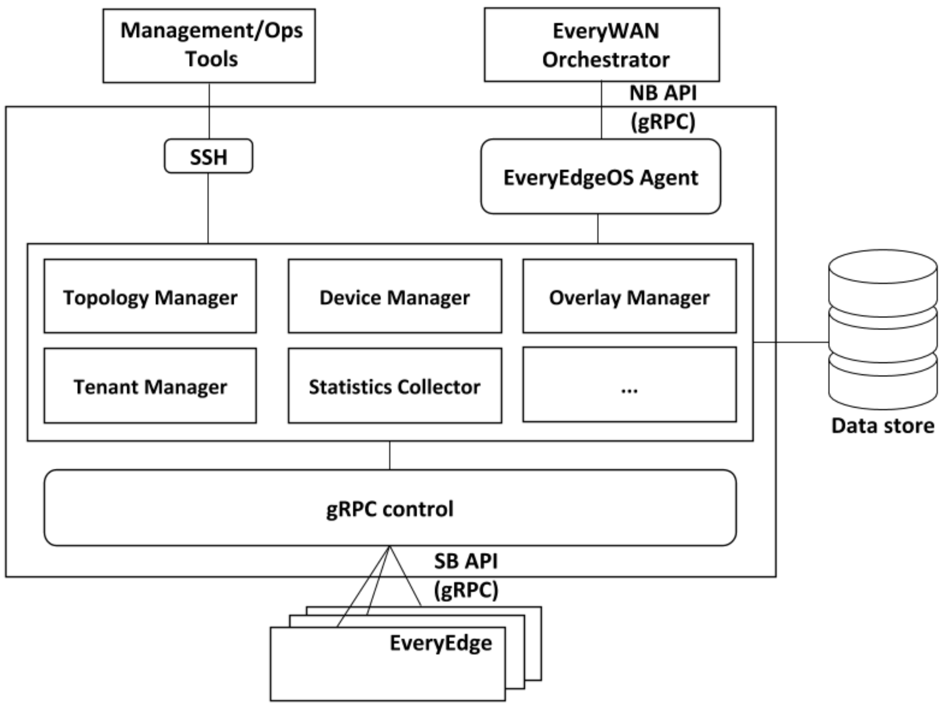


Figure 7 - Architecture of the controller

On top of the EveryEdgeOS controller there is an orchestrator called EveryBOSS, which is shown in Figure 8. The orchestrator offers a GUI called EveryGUI through which the tenants can design the network topology, configure the services, manage the SD-WAN interconnections, the virtual devices and the users. The communication between the EveryBOSS and the EveryGUI is handled via a REST interface. The commands received from the GUI are sent to the SD-WAN controller through the Northbound interface exposed by the SD-WAN controller. Moreover, the EveryBOSS interacts with a Keystone instance to support user registration and authentication functionalities.

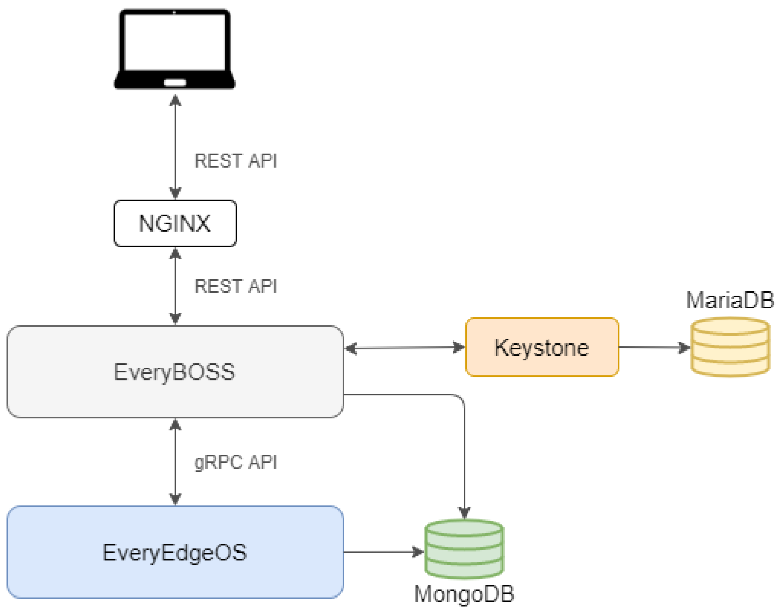


Figure 8 - SD-WAN orchestrator and EveryGUI

2.2.4. Deployment of EveryWAN in the UCSS testbed

We have released a set of deployment scripts to simplify the installation and configuration of the EveryWAN components both on virtual testbeds (e.g. a Mininet emulated networks) and physical testbeds. We used these scripts to deploy EveryWAN on the GÉANT network.

We have deployed the EveryEdgeOS controller and all the management components as Docker containers on the TV_DC node. The management components include: i) Keystone, which is a tool used for user authentication; ii) MariaDB, a database containing users account information; iii) MongoDB where the EveryEdgeOS controller stores all the information about EveryWAN devices and services; iv) the EveryBOSS Orchestrator; v) a NGINX web server which offers the EveryWAN GUI to the users. The GUI can be accessed by contacting the NGINX web server on the TCP port 80 from inside the container itself. To allow the users accessing the GUI and controlling the SD-WAN services from outside the TV_DC node, the TCP port 80 of the NGINX container needs to be published to the outside world. This requires mapping the TCP port 80 of the NGINX container to the TCP port 80 on the Docker host (i.e. the TCP port 80 of the TV_DC node).

Similarly, the gRPC port of the EveryEdgeOS controller needs to be published to allow the EveryEdge devices to reach the controller. The gRPC port of the controller can be configured in the controller configuration file. By default the TCP port 50061 is used. The deployment of the control and management components is shown in Figure 9.

All the steps described above have been automated using the Docker Compose tool, which allows us to easily deploy multi-container Docker applications. Basically, starting from a YAML file describing the configuration of the Docker containers, Docker Compose takes care of building and starting all the containers that are required for the control plane and management plane functionalities.

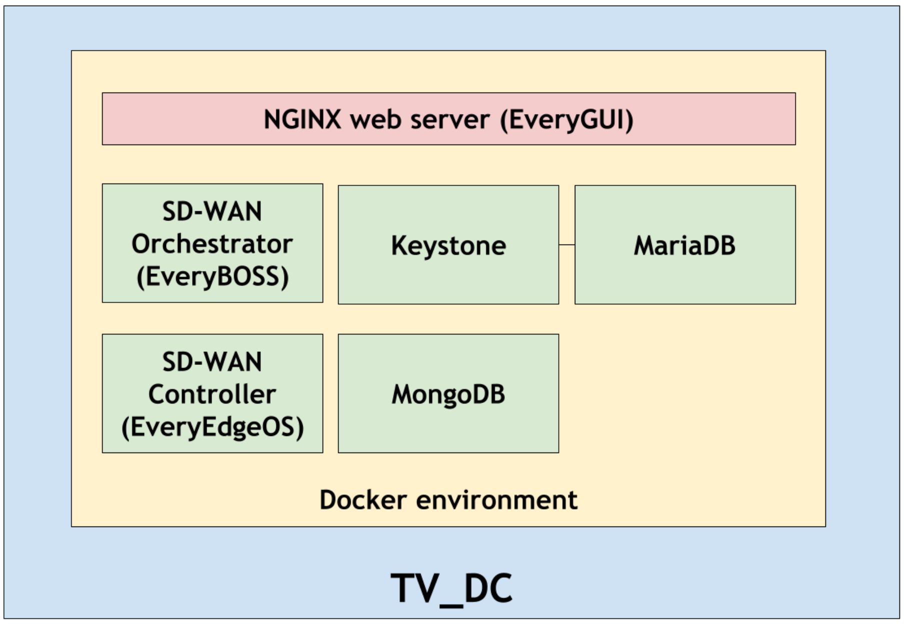

NGINX web server (EveryGUI)
SD-WAN Orchestrator (EveryBOSS)
Keystone
MariaDB
SD-WAN Controller (EveryEdgeOS)
MongoDB
Docker environment
TV_DC

Figure 9 - Deployment of the control and management components

As regards the Edge Devices, we have released a deployment script to simplify the installation of the EveryEdge software in every VM where we want to test the EveryWAN solution. This script creates a Python virtual environment and installs all the dependencies required to run the EveryEdge software. In addition, it is possible to emulate any number of “virtual” hosts attached to the EveryEdge Device. To simulate the hosts we leverage the Linux namespaces. Basically, the EveryEdge software runs in the root network namespace of the Linux kernel. A dedicated network namespace is created for each host we want to emulate. The “virtual” link between the EveryEdge Device and the “virtual” host is realized by connecting the host namespace to the root namespace through a virtual ethernet pair. The deployment schema of the EveryEdge Device is shown in Figure 10.

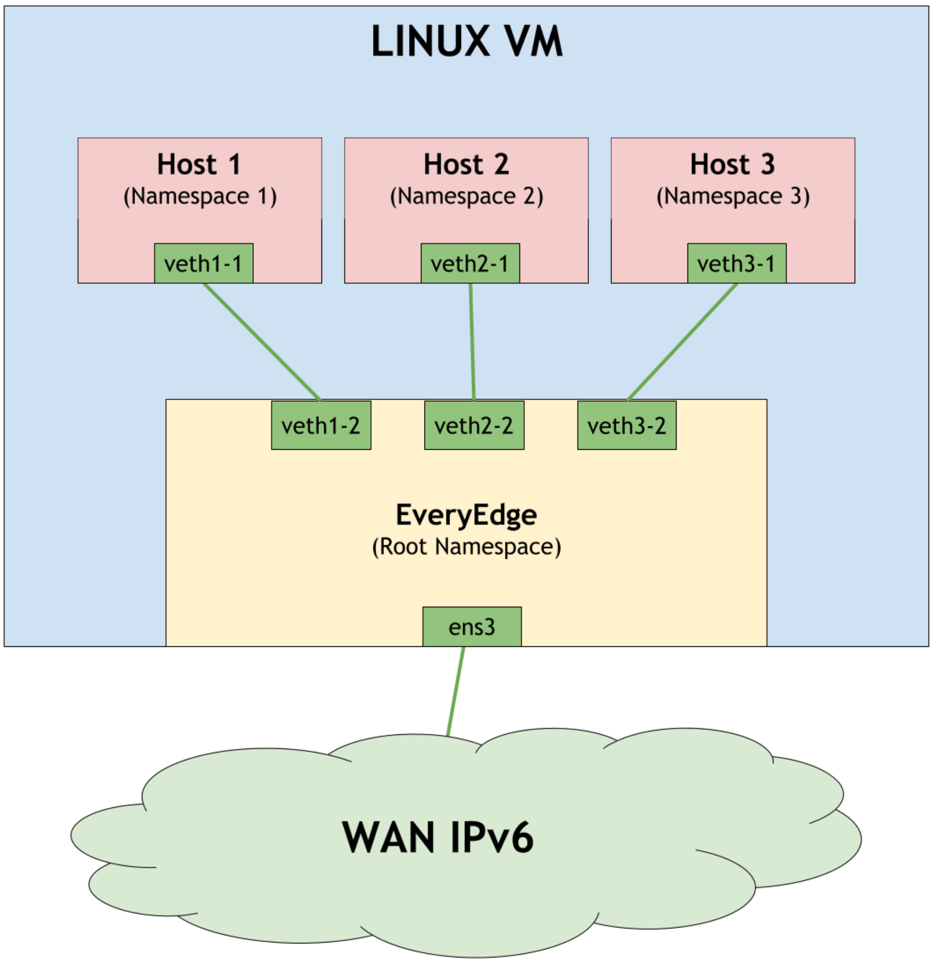


Figure 10 - Deployment of the Edge Device (EveryEdge)

To enable the EveryEdge Device reaching the EveryEdgeOS controller, the IP address and gRPC port of the controller needs to be configured in the EveryEdge. This information together with other settings can be provided through a configuration file.

Using the deployment tools described above, we have deployed the EveryEdge Device together with three attached "virtual" hosts both in the TV_DC and HU_NRN nodes.

2.2.5. SRv6 configuration for the UCSS SD-WAN solution

The EveryWAN solution offers a GUI through which the users can create and manage the overlay VPNs. The GUI allows the users to specify various parameters for the overlay VPNs to be created, including a name, the traffic type to be transported across the remote sites (i.e. IPv4 or IPv6 traffic) and the local slices connected by the overlay network. The EveryBOSS Orchestrator acts as mediator interacting on the north with the NGINX web server that exposes the GUI and on the south with the EveryEdgeOS controller. The controller takes care of the deployment of the overlay networks to any Edge Device. At the time of writing, the connection between the Edge Devices is established only in a full-mesh fashion, i.e. every "local" Edge Device that has to send packets to another "remote" Edge Device will establish an SRv6 tunnel.

The Edge Device leverages the so-called EveryEdgeManager, which acts as mediator interacting on the southbound direction with the kernel of the EveryEdge router and on the northbound with the SDN controller. The EveryEdgeMediator translates the messages received over the Southbound API into actions to be sent to the kernel components. The communication library used to enable the communication with the Linux kernel is based on the open source project pyroute2 [10], a pure python netlink library that has been properly extended to support our needs.

An important mechanism for implementing the slicing service is the ingress classification performed by the Edge Device. In our current implementation, it can be either based on the physical input port or on a virtual input port - we use the so-called VRF lite approach where each interface is individually mapped to a slice and enslaved to the VRF serving that slice. Additionally, if local trunk ports are available in the edge nodes, VLAN interfaces can be used in place of the physical local interfaces.

Hereafter we report the equivalent iproute2 configuration required to setup an overlay network between two Edge Devices named IngressEveryEdge and EgressEveryEdge. For the sake of simplicity, we only describe the steps required to configure a one-way SRv6 tunnel between IngressEveryEdge and EgressEveryEdge. Note that to support communication in both directions, a symmetric configuration needs to be applied for the reverse path (from EgressEveryEdge to IngressEveryEdge).

First, several kernel parameters must be configured both on the IngressEveryEdge and EgressEveryEdge.

Enable the IPv4 forwarding:

```
sysctl -w net.ipv4.conf.all.forwarding=1
```

Enable the IPv6 forwarding:

```
sysctl -w net.ipv6.conf.all.forwarding
```

Enable the SRv6 functionalities both globally and on the WAN interface of the Edge Device:

```
sysctl -w net.ipv6.conf.all.seg6_enabled=1
sysctl -w net.ipv6.conf.ens3.seg6_enabled=1
```

Enable the VRF strict mode, which is required to use the SRv6 End.DT4 behavior:

```
sysctl -w net.vrf.strict_mode=1
```

Disable reverse path filtering for all the interfaces:

```
sysctl -w net.ipv4.conf.all.rp_filter=0
sysctl -w net.ipv4.conf.default.rp_filter=0
```

On IngressEveryEdge we need to instantiate a VRF:

```
IngressEveryEdge# ip link add vrf-100 type vrf table 100
IngressEveryEdge# ip link set vrf-100 up
```

Then, each interface to be connected by the overlay network is enslaved to the VRF:

```
IngressEveryEdge# ip link set eth1 master vrf-100
IngressEveryEdge# ip link set eth2 master vrf-100
IngressEveryEdge# ip link set eth3 master vrf-100
```

Finally, for each remote slice we create a SRv6 explicit path to redirect the traffic destined to the remote slice over the SRv6 tunnel:

```
IngressEveryEdge# ip route add 192.168.4.0/24 encap \
                  seg6  mode  encap  segs  2001:738:0:527:f816:3eff:fedd:c8a4,fc00::100  dev
                  ens3 table 100
IngressEveryEdge# ip route add 192.168.5.0/24 encap \
                  seg6  mode  encap  segs  2001:738:0:527:f816:3eff:fedd:c8a4,fc00::100  dev
                  ens3 table 100
```

In the above commands, we use a Segment List composed of two Segments. The first Segment (`2001:738:0:527:f816:3eff:fedd:c8a4`) is the external IPv6 address of the EgressEveryEdge. It allows the packets to reach the EgressEveryEdge. The EgressEveryEdge uses the second Segment (`fc00::100`) to identify the overlay network (i.e. the VRF) to which the packets should be forwarded to. An alternative solution based on a single Segment is discussed later in this section.

On EgressEveryEdge we instantiate a VRF:

```
EgressEveryEdge# ip link add vrf-100 type vrf table 100
EgressEveryEdge# ip link set vrf-100 up
```

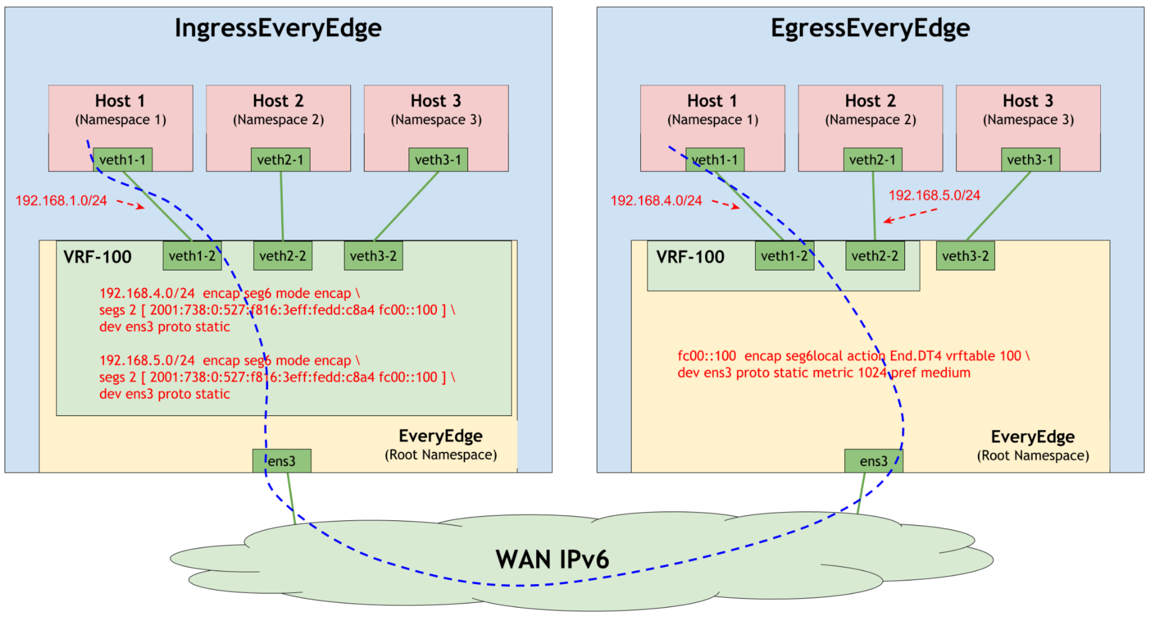
Each interface to be connected by the overlay network is enslaved to the VRF:

```
EgressEveryEdge# ip link set eth1 master vrf-100
EgressEveryEdge# ip link set eth2 master vrf-100
```

Then, we instantiate a SRv6 End.DT4 behavior to extract the traffic from the tunnel and forward it to the appropriate VRF:

```
EgressEveryEdge# ip -6 route add fc00::100 encap seg6local action End.DT4 vrftable 100 dev
                  ens3
```

With the above configuration, the traffic entering the IngressEveryEdge (from the interfaces eth1, eth2 and eth3) and destined to the subnets 192.168.4.0/24 and 192.168.5.0/24 is sent over the SRv6 tunnel. A SRH (Segment Routing Header) is added to all the packets that traverse the tunnel. This SRH carries the Segment List which allows the packets to reach the EgressEveryEdge. The EgressEveryEdge will extract the traffic from the tunnel and will forward it to the appropriate VRF. Finally, it will be delivered to the users according to L2/L3 rules defined for the slice. This scenario is shown in Figure 11.



```
192.168.4.0/24  encap seg6 mode encap \
segs 2 [ 2001:738:0:527:f816:3eff:fedd:c8a4 fc00::100 ] \
dev ens3 proto static

192.168.5.0/24  encap seg6 mode encap \
segs 2 [ 2001:738:0:527:f816:3eff:fedd:c8a4 fc00::100 ] \
dev ens3 proto static
```

```
fc00::100  encap seg6local action End.DT4 vrftable 100 \
dev ens3 proto static metric 1024 pref medium
```

Figure 11 - SRv6 Tunnel example

As regards the Segment List, in general two SIDs are required to implement a SRv6 tunnel. The first SID is the external IPv6 address of the egress device; the second SID is a private SID which uniquely identifies the overlay network. Basically, the IPv6 address of the egress device allows the packets to reach the egress device. Then, the second SID is used to choose the VRF to which the packets should be forwarded to. We found that this implementation can be used for all the addressing range classes described in (i.e. A0, A1, A2, A3 and A4). The only requirement is that the network offers a level of transparency T0, which is required to use the Segment Routing Header in the outer IPv6 packet.

We have found that if the egress device offers an available IPv6 addressing range A0, A1, A2 or A3 the implementation can be optimized to use a single SID instead of two SIDs. This SID should be allocated from the available IPv6 addressing range and uniquely assigned to the overlay network. This implementation allows the packets to reach the egress device and to be forwarded to the appropriate VRF in one shot. In some cases, it is necessary to configure a NDP proxy to enable the egress node to respond to the NDP requests for the SID received by the neighbor devices. If the egress device has an A4 addressing range (i.e. it exports a /128 prefix) only the two-SIDs solution can be used, because it is not possible to allocate SIDs from a /128 network.

2.2.6. Pyroute extensions to support End.DT4

The support of IPv4 VPNs using SRv6 in the Link kernel is possible thanks to a recent patch, which implements the SRv6 End.DT4 behavior. This patch has been developed by our team at CNIT/University of Rome Tor Vergata (before the UCSS project) and it has been integrated in the Linux kernel release 5.11 (14 February 2011).

In the context of the UCSS project, we have contributed to the open source Python tool pyroute2 [10], to support the configuration of the End.DT4 behavior. This was needed to implement the

SD-WAN services based on SRv6. The patch to pyroute2 has been submitted on 2021-11-17. The pull request has been merged on 2021-12-18 [https://github.com/svinota/pyroute2/pull/863].

2.2.7. SRv6 SD-WAN delay monitoring with Twamp

One of our goals was to support user level performance monitoring in our SD-WAN solution. In particular, we focused on delay monitoring. We want to give the user the possibility to monitor the delay between any couple of EveryEdge devices. The monitoring should be offered through the same management GUI used to setup and configure the SD-WAN services.

We note that in order to have a consistent evaluation of one-way delays, the clocks of the two endpoints must be synchronized. The clock synchronization is out of the scope of this work, we assume that the clock can be synchronized with an accuracy in the order of less than 1 ms, so that the delay measurements can be accurate enough for delays in the order of a few milliseconds.

We have chosen to implement the Simple Two-Way Active Measurement Protocol (STAMP) [11] and use it to monitor the delay over the tunnels. STAMP is a generic protocol, based on UDP, that can be used to monitor delays in several different scenarios. We have adapted it to the case of monitoring SRv6 tunnels, as described in [12]. STAMP is based on the sending of Test Packet from a Session-Sender to a Session-Reflector that receives the Test Packet and replies with a Reply Test Packet (figure 12).

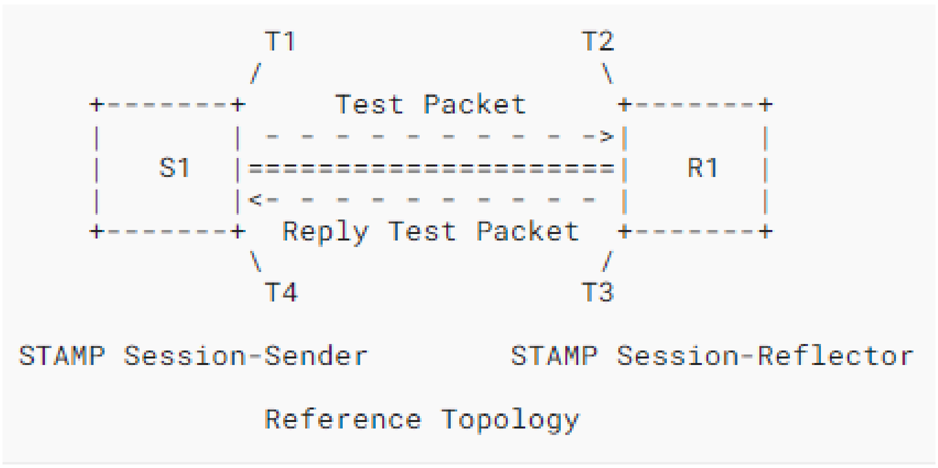


Figure 12 - STAMP reference scenario

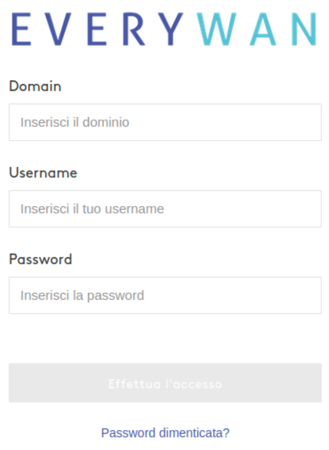
We have implemented both the STAMP Session-Sender and the STAMP Session-Reflector entities. Both entities are dynamically controlled by our EveryEdgeOS controller.

The STAMP packet generated by the Sessione-Sender in the "local" Every Edge node is encapsulated with an IPv6/SRv6 header so that it is sent on the SRv6 tunnel to reach the Session Reflector located in the "remote" EveryEdge node. The Session-Sender is implemented in python using the scapy library. The Encapsulated Test Packet is "captured" in the "remote" EveryEdge node by a filter that intercepts the encapsulated packets with a specific UDP port. In our implementation of the Session Reflector we used the python scapy library to intercept the packet, decode it and prepare the Reply Test packet. The Session Reflector needs to encapsulate the packet with the proper SR policy (SID list) so that the packet can return to the Session Sender located in the "local" EveryEdge node. To this purpose, the Test Packet contains a test session ID, which is used to retrieve the correct SR policy (SID list). The association between test session IDs and SR policies is dynamically configured by the EveryEdgeOS controller.

The EveryEdgeOS controller can start and stop monitoring sessions among the EveryEdge nodes. Moreover, it collects the measurement results that are stored by the Session Senders. The user can use the orchestrator GUI to visualize the results of the measurement sessions.

2.2.8. UCSS experiments with EveryWAN

We deployed all the management components of EveryWAN including the GUI on the TV_DC node. The GUI can be accessed by contacting the TV_DC node on the port 80: https://everywan.netgroup.uniroma2.it.


EVERYWAN

Domain

Inserisci il dominio

Username

Inserisci il tuo username

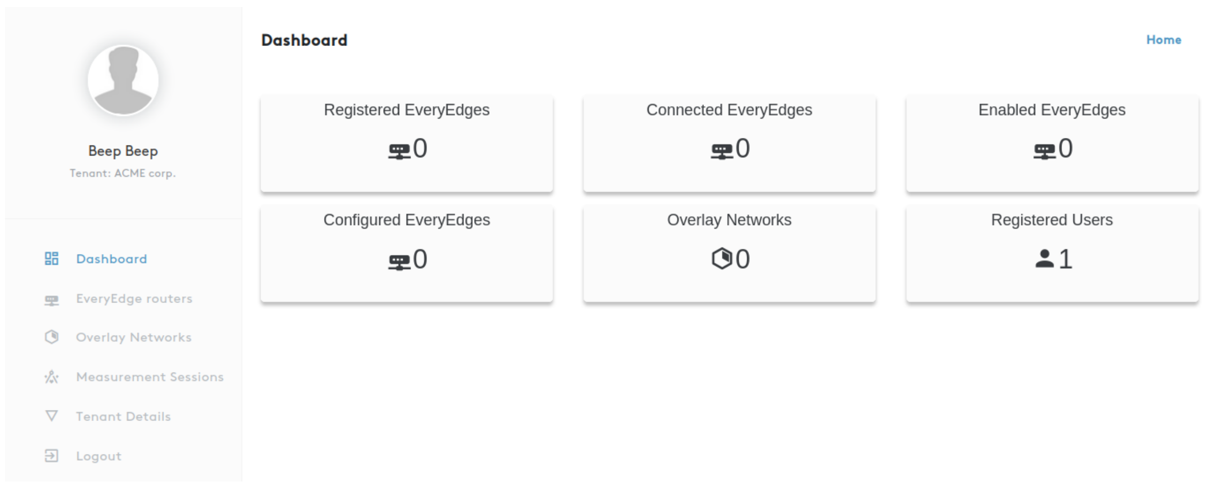
Password

Inserisci la password

Effettua l'accesso

Password dimenticata?

After completing the login, we will be redirected to the dashboard of EveryWAN:


Beep Beep

Tenant: ACME corp.

Dashboard

EveryEdge routers

Overlay Networks

Measurement Sessions

Tenant Details

Logout

Dashboard

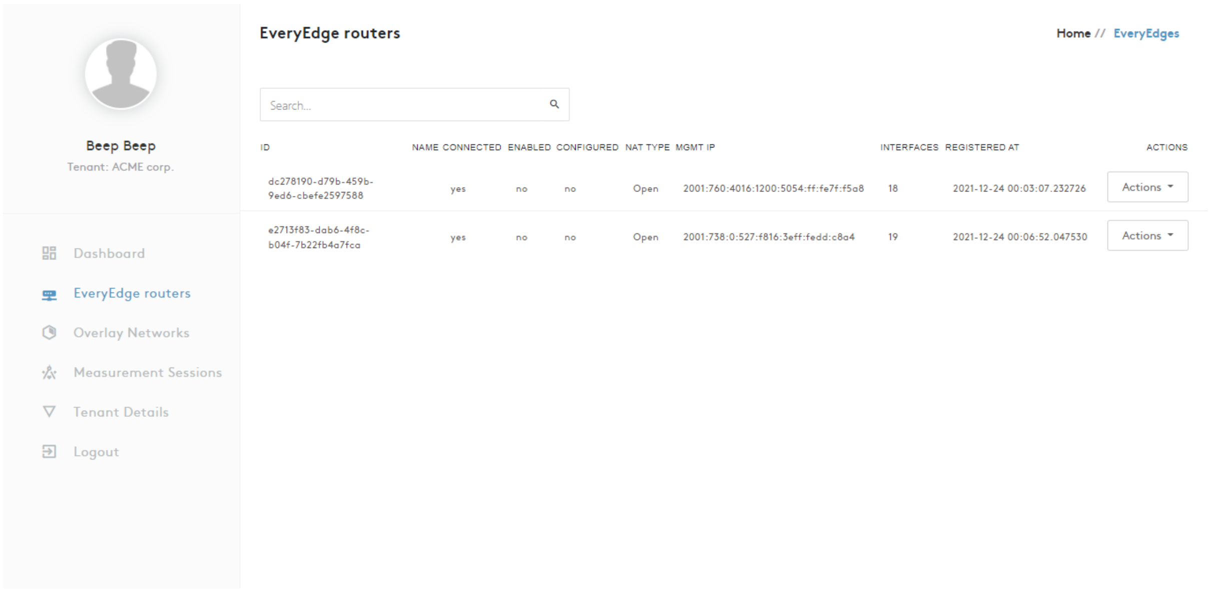
Home

Registered EveryEdges 0	Connected EveryEdges 0	Enabled EveryEdges 0
Configured EveryEdges 0	Overlay Networks 0	Registered Users 1

As we can see from the dashboard, initially there are no EveryEdge Devices registered to the system.

We start the EveryEdge software both in the TV_DC and HU_NRN nodes using the deployment scripts described in the section 2.2.4 of this report. After the initialization phase, the EveryEdge devices automatically register and authenticate to the EveryWAN system. From the EveryEdge routers it is possible to see the new two devices.

We deployed the EveryEdgeOS and the management components on the TV_DC node. The EveryEdge device has been deployed on TV_DC and on other VMs. In the following screenshot we are using the HU_NRN edge node. After starting the EveryEdge nodes, they automatically register and authenticate to the controller. After a while, the two devices appear in the EveryEdge devices list of the GUI:

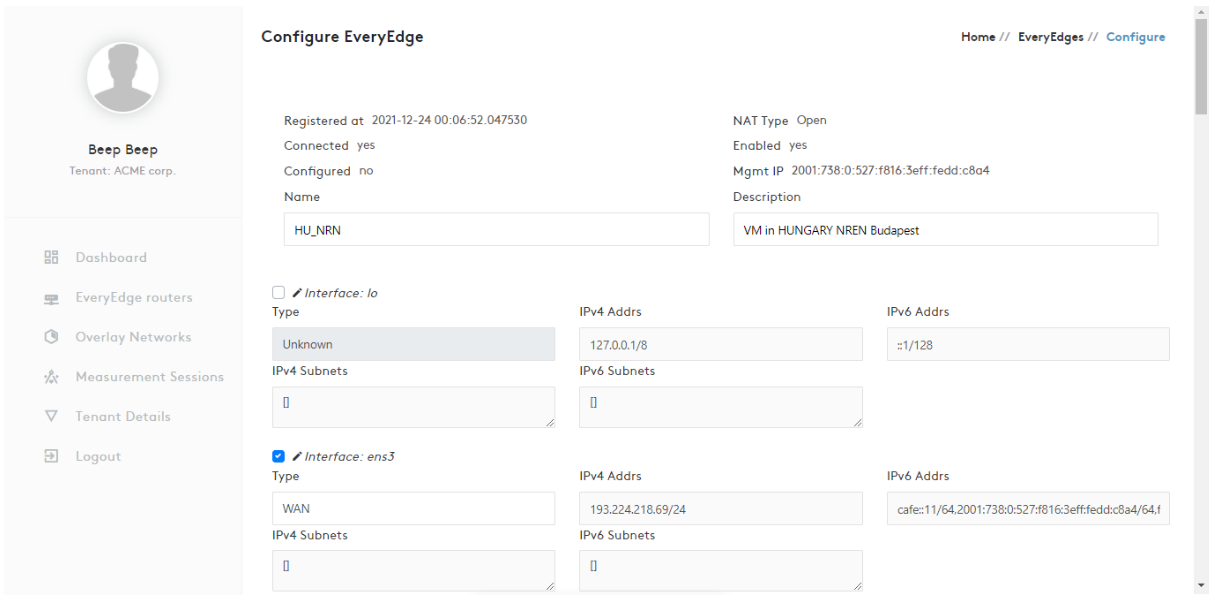


EveryEdge routers

ID	NAME	CONNECTED	ENABLED	CONFIGURED	NAT TYPE	MGMT IP	INTERFACES	REGISTERED AT	ACTIONS
dc278190-d79b-459b-9ed6-cbefe2597588		yes	no	no	Open	2001:760:4016:1200:5054:ff:fe7f:f5a8	18	2021-12-24 00:03:07.232726	Actions
e2713f83-dab6-4f8c-b04f-7b22fb4a7fca		yes	no	no	Open	2001:738:0:527:f816:3eff:fedd:c8a4	19	2021-12-24 00:06:52.047530	Actions

Both the devices are connected but they are not yet configured and enabled. From the GUI we can configure the two devices. We can set the device name and description and we can set the network interfaces to be controlled using EveryWAN. For each interface, we can set the type (i.e. whether it is a WAN interface or a LAN interface) and we can set the IP address.

Hereafter we report the configuration of the HU_NRN node as example:



Configure EveryEdge

Registered at 2021-12-24 00:06:52.047530
Connected yes
Configured no
Name: HU_NRN

NAT Type Open
Enabled yes
Mgmt IP 2001:738:0:527:f816:3eff:fedd:c8a4
Description: VM in HUNGARY NREN Budapest

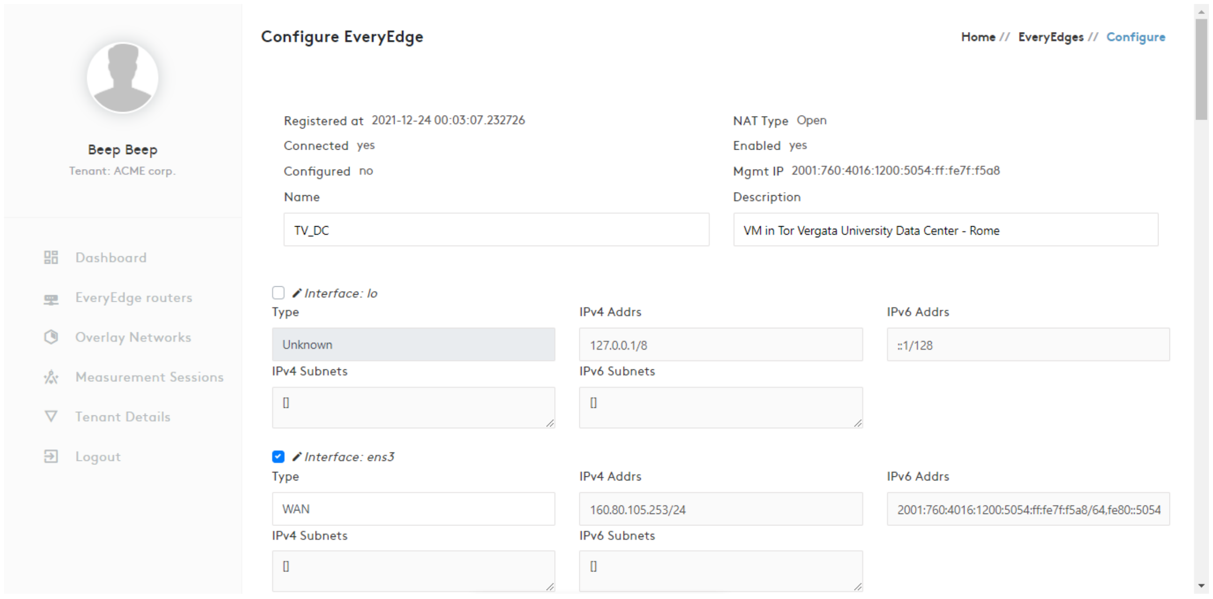
☐ *Interface: lo*

Type	IPv4 Addrs	IPv6 Addrs
Unknown	127.0.0.1/8	::1/128
IPv4 Subnets	**IPv6 Subnets**	
[]	[]	

☑ *Interface: ens3*

Type	IPv4 Addrs	IPv6 Addrs
WAN	193.224.218.69/24	cafe::11/64,2001:738:0:527:f816:3eff:fedd:c8a4/64,f
IPv4 Subnets	**IPv6 Subnets**	
[]	[]	

The configuration of the TV_DC node is shown below:



Configure EveryEdge

Registered at 2021-12-24 00:03:07.232726
Connected yes
Configured no
Name: TV_DC

NAT Type Open
Enabled yes
Mgmt IP 2001:760:4016:1200:5054:ff:fe7f:f5a8
Description: VM in Tor Vergata University Data Center - Rome

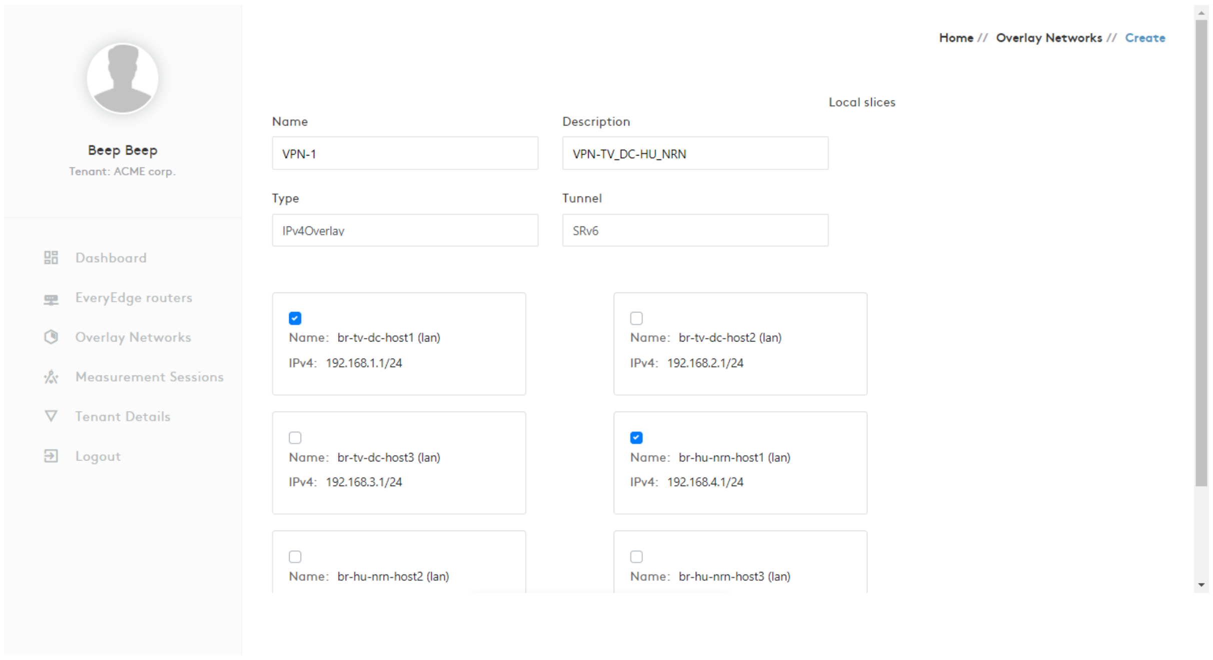
☐ *Interface: lo*

Type	IPv4 Addrs	IPv6 Addrs
Unknown	127.0.0.1/8	::1/128
IPv4 Subnets	**IPv6 Subnets**	
[]	[]	

☑ *Interface: ens3*

Type	IPv4 Addrs	IPv6 Addrs
WAN	160.80.105.253/24	2001:760:4016:1200:5054:ff:fe7f:f5a8/64,fe80::5054
IPv4 Subnets	**IPv6 Subnets**	
[]	[]	

After configuring and enabling the devices, we can create an overlay VPN between TV_DC and HU_NRN:



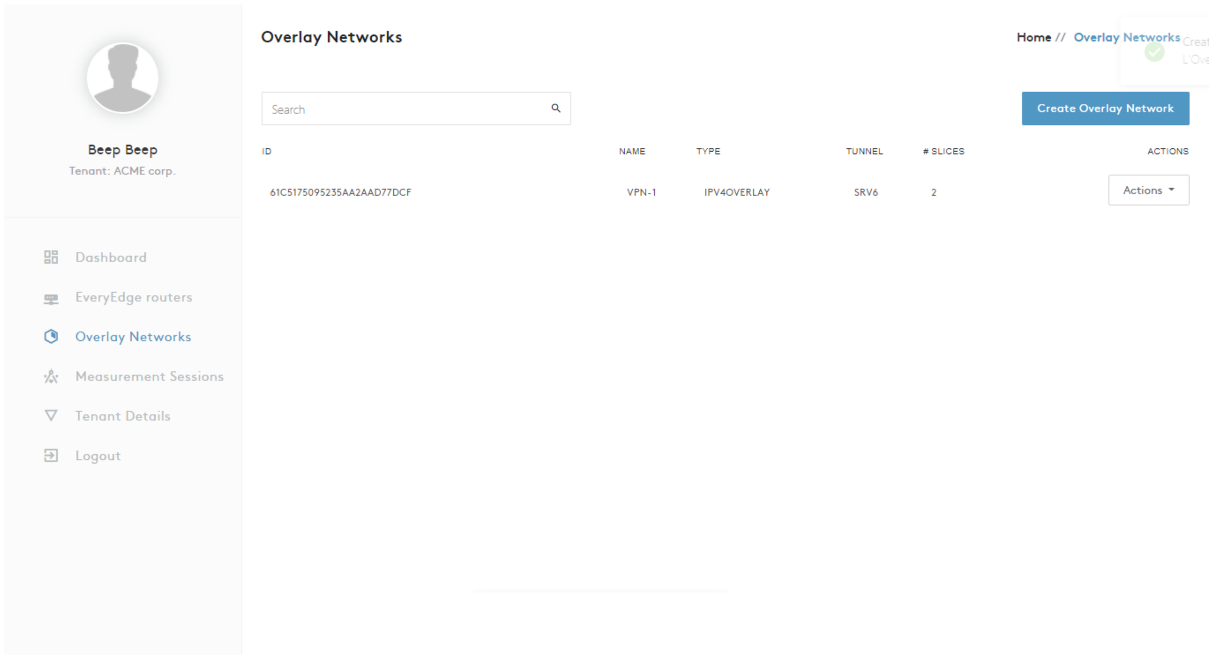
Local slices

Name	VPN-1
Description	VPN-TV_DC-HU_NRN
Type	IPv4Overlay
Tunnel	SRv6

☑ Name: br-tv-dc-host1 (lan) IPv4: 192.168.1.1/24	☐ Name: br-tv-dc-host2 (lan) IPv4: 192.168.2.1/24
☐ Name: br-tv-dc-host3 (lan) IPv4: 192.168.3.1/24	☑ Name: br-hu-nrn-host1 (lan) IPv4: 192.168.4.1/24
☐ Name: br-hu-nrn-host2 (lan)	☐ Name: br-hu-nrn-host3 (lan)

Through the GUI we can also specify the local slices that we want to connect using the overlay VPN. In the scenario shown above we are connecting the slice associated to the interface br-tv-dc-host1 on TV_DC to the slice mapped to the interface br-hu-nrn-host1 on HU_NRN.

The new overlay appears in the Overlay Networks section of the GUI:

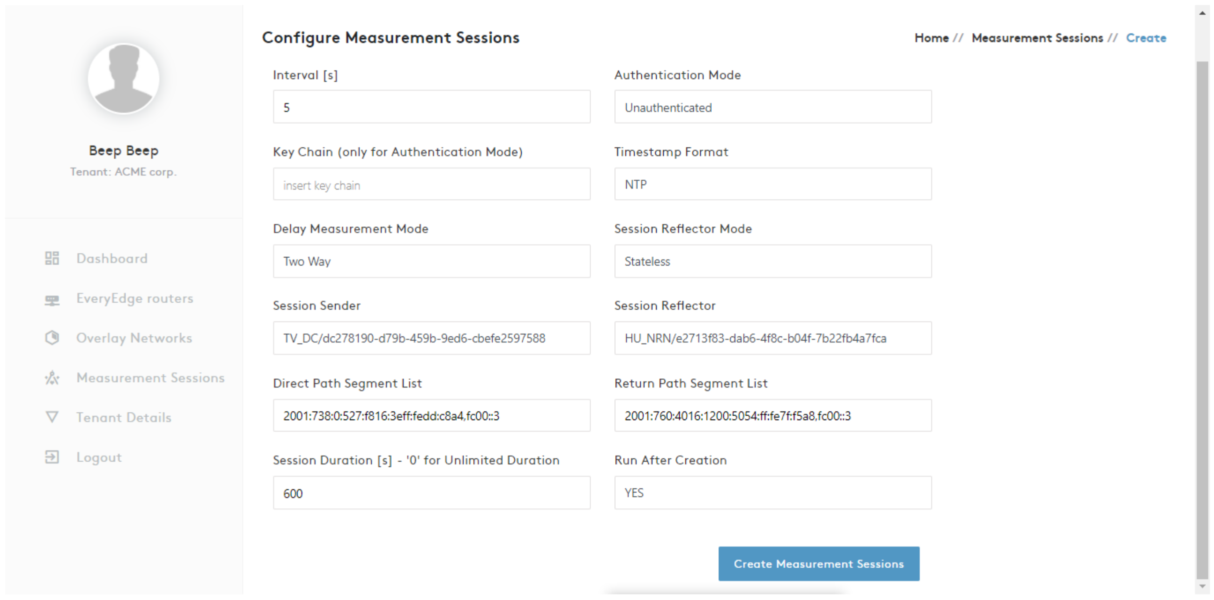


Overlay Networks

ID	NAME	TYPE	TUNNEL	# SLICES	ACTIONS
61C5175095235AA2AAD77DCF	VPN-1	IPV4OVERLAY	SRV6	2	Actions

Finally, we can run a STAMP Session to monitor the delay of the overlay network in real-time.

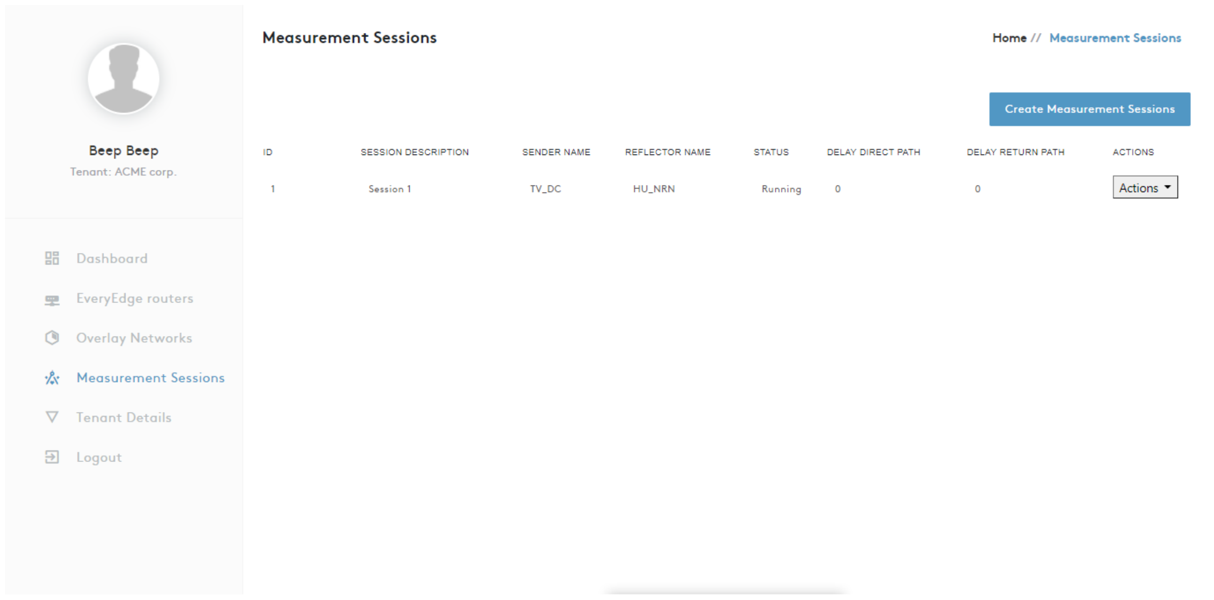
To do this, we access the Measurement Session page of the GUI and we create a STAMP Session between TV_DC and HU_NRN. The GUI allows to set several parameters of the experiment to be run, such as the duration, the interval between two STAMP test packets and the timestamp format to be used:



Configure Measurement Sessions

Interval [s]: 5	Authentication Mode: Unauthenticated
Key Chain (only for Authentication Mode): insert key chain	Timestamp Format: NTP
Delay Measurement Mode: Two Way	Session Reflector Mode: Stateless
Session Sender: TV_DC/dc278190-d79b-459b-9ed6-cbefe2597588	Session Reflector: HU_NRN/e2713f83-dab6-4f8c-b04f-7b22fb4a7fca
Direct Path Segment List: 2001:738:0:527:f816:3eff:fedd:c8a4,fc00::3	Return Path Segment List: 2001:760:4016:1200:5054:ff:fe7f:f5a8,fc00::3
Session Duration [s] - '0' for Unlimited Duration: 600	Run After Creation: YES

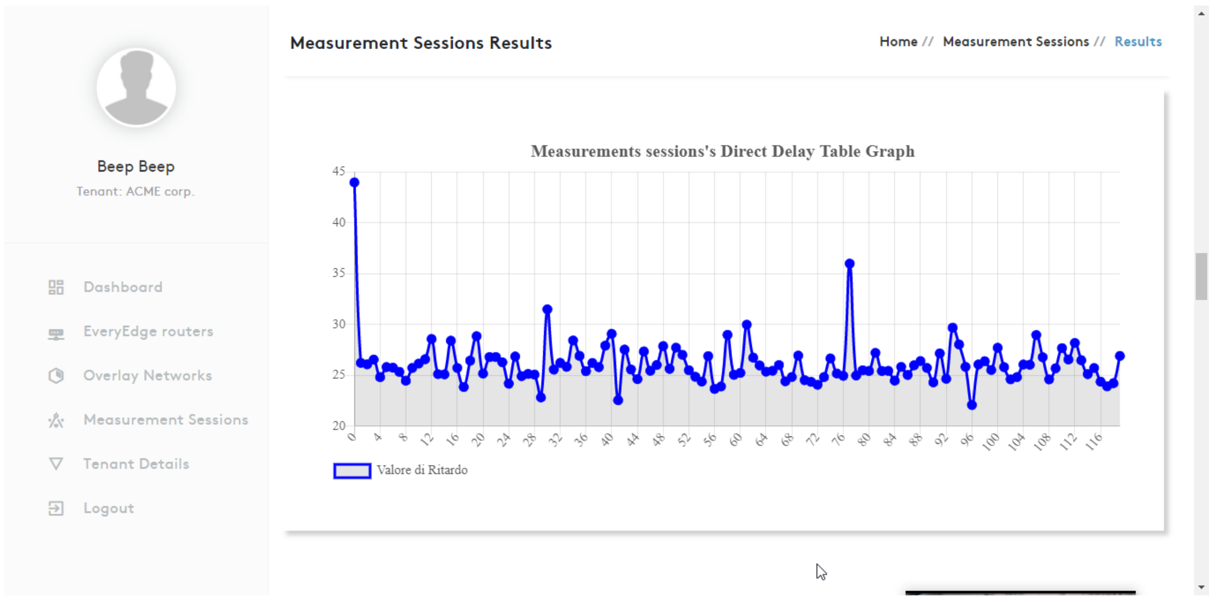
The Measurement Sessions page shows the newly created STAMP Session.



Measurement Sessions

ID	SESSION DESCRIPTION	SENDER NAME	REFLECTOR NAME	STATUS	DELAY DIRECT PATH	DELAY RETURN PATH	ACTIONS
1	Session 1	TV_DC	HU_NRN	Running	0	0	Actions

Finally, we can see the results the measured delays for both the direct and return paths:



2.2.9. Lesson Learned

We have learned a lot during the course of the UCSS project.

The first important learning is that the GÉANT network and the commercial ISP networks that we have used are transparent with respect to SRv6 connectivity, therefore it is possible to implement SD-WAN services based on SRv6 and on the Network Programming model. We have not noticed any filtering of the Segment Routing Header (SRH) in the transport networks of GÉANT and of the commercial ISP networks.

On the other hand, in the NREN datacenters and in the campus networks there are several issues, likely due to the presence of firewalls.

Before discussing the firewall issues, It has also been evident that IPv6 connectivity is not yet a commodity in campus networks. This means that in the majority of the cases it is not possible to get IPv6 connectivity if needed. For example we initially planned to also have a site in London to cooperate with a colleague on this project (as it was written in the proposal), but it turned out that it was not possible to get IPv6 connectivity to the lab.

When it was possible to have IPv6 connectivity, it was often limited to a smaller prefix than /64. We hope that a process to change the mind of network administrators considering the difference between IPv4 and IPv6 will start hopefully soon.

As for the firewall issues, we have identified the requirements to open the firewalls to support our services, but we have not reached good results for two reasons: 1) lack or time (we were able to start the test after finalizing the implementation rather late in December) 2) we were simple guests of the other institutions that hosted our machines, so we could only kindly ask support but clearly our requests were not high priority requests.

We also understood that in order to propose a liberal firewall configuration for SRv6 services, we may need to start a security assessment so that we can propose a solution that is backed by a security analysis. We plan to work on this in the coming months.

From the point of view of the SD-WAN services that we have implemented and experimented with, we have some feedback on the usability, even if we have not exposed the system to real users. By focusing on the needs of the scenario that we have demonstrated, we identified some improvements to the GUI that we plan to implement.

2.3. Impact

Let us first consider the expected impact that was included in the proposal.

Expected impact summary from the proposal:

1. Assessment of the GEANT/NREN support for a SRv6 core network
2. Deployment of user managed VPNs
3. Identification of possible extensions and inefficiencies of the GEANT network
4. Open Source contributions: (i) EveryWAN; (ii) SRv6 Performance Monitor Framework; (iii) SRv6 Linux implementation

We have successfully verified that the GEANT and NRENs networks are transparent to SRv6 services, therefore they can be used to support the User Controlled SD-WAN services. On the other hand we discovered that in general NRENs datacenters have limited support for IPv6 connectivity and that firewalls need to be reconfigured both in NREN datacenters and in campus networks.

The experience gained with the UCSS project can be very useful and have an impact on the development of IPv6 services.

As for the deployment of user managed VPNs, the developed Open Source EveryWAN solution with its GUI is almost mature to be used by external users to autonomously setup user managed VPNs. With the UCSS project we have passed a phase of alpha testing and we have collected some requirements for the GUI and user experience, we estimate that one man month of work is needed to implement the improvements. Then we could start a phase of beta testing, in which the system can be used by external users.

As for the identification of extensions of the GEANT network, we discovered that the logical "bottleneck" in the process at the moment is not in the GEANT network, but in the periphery (NRENs datacenters and in the campus networks and datacenters).

As for the Open Source contributions, we have successfully improved the EveryWAN solution by integrating the SRv6 technology. To the best of our knowledge, this is the first open source SD-WAN solution that includes SRv6. We have also integrated a delay monitoring system. We have extended the pyroute2 Open Source library to support the IPv4 VPNs over SRv6 and our contribution has been accepted in the mainstream.

3. Acknowledgements

The CNIT people that contributed to this activity during the UCSS project:
Carmine Scarpitta, Giulio Sidoretti, Paolo Lungaroni, Francesco Lombardo, Andrea Mayer, Stefano Salsano, Marco Bonola.

People that contributed to earlier versions of EveryWAN:
Pier Luigi Ventre, Bogdan Iatco

The SRv6 activity in the Netgroup of University of Rome Tor Vergata has been supported by CISCO University Research Program.

The EveryWAN initial requirements have been identified in the course of a cooperation with NET Reply.

These students have spent an internship working on the EveryWAN/UCSS web GUI:
Francesco Pinsone, Martino Porceddu, Alessio Rinaldi, Simone Salvatore Vivacqua, Simone Campana

4. Bibliography

Bibliography

[1] S. Previdi, J. Leddy, S. Matsushima, and D. Voyer, “Ipv6 segment routing header (SRH),” RFC Editor, Mar. 2020. doi: 10.17487/RFC8754.

[2] J. Leddy, D. Voyer, S. Matsushima, and Z. Li, “Segment Routing over IPv6 (SRv6) Network Programming,” RFC Editor, Feb. 2021. doi: 10.17487/RFC8986.

[3] C. Scarpitta, P. L. Ventre, F. Lombardo, S. Salsano, and N. Blefari-Melazzi, “EveryWAn- An Open Source SD-WAN solution,” in *2021 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, Oct. 2021, pp. 1–7, doi: 10.1109/ICECCME52200.2021.9590859.

[4] R. Ricci, E. Eide, and C. Team, “Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications.,” ; *login:: the magazine of USENIX & SAGE*, vol. 39, no. 6, pp. 36–38, 2014.

[5] D. Duplyakin *et al.*, “The design and operation of CloudLab.,” *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, p. 1, 2019.

[6] S. Salsano, P. L. Ventre, L. Prete, G. Siracusano, M. Gerola, and E. Salvadori, “OSHI - Open Source Hybrid IP/SDN Networking (and its Emulation on Mininet and on Distributed SDN Testbeds),” in *2014 Third European Workshop on Software Defined Networks*, Sep. 2014, pp. 13–18, doi: 10.1109/EWSDN.2014.38.

[7] P. L. Ventre, M. M. Tajiki, S. Salsano, and C. Filsfils, “SDN architecture and southbound apis for ipv6 segment routing enabled wide area networks,” *IEEE Trans. Netw. Serv. Manage.*, vol. 15, no. 4, pp. 1378–1392, Dec. 2018, doi: 10.1109/TNSM.2018.2876251.

[8] A. Andreyev, “Introducing data center fabric, the next-generation Facebook data center network.,” *Engineering at Meta*, 2014. https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/ (accessed Dec. 24, 2021).

[9] The kernel development community, “Virtual Routing and Forwarding (VRF),” *The Linux Kernel documentation*. https://www.kernel.org/doc/html/latest/networking/vrf.html (accessed Dec. 24, 2021).

[10] “pyroute2 netlink framework.” https://pyroute2.org/ (accessed Dec. 24, 2021).

[11] G. Mirsky, G. Jun, H. Nydell, and R. Foote, “Simple Two-Way Active Measurement Protocol,” RFC Editor, Mar. 2020. doi: 10.17487/RFC8762.

[12] R. Gandhi, Ed., et al., “Simple TWAMP (STAMP) Extensions for Segment Routing Networks,” IPPM Working Group, 2021.