VeRTIGO: Network Virtualization and Beyond

Roberto Doriguzzi Corin, Matteo Gerola, Roberto Riggio, Francesco De Pellegrini, Elio Salvadori
CREATE-NET
via Alla Cascata 56/D, 38123 Povo - Trento - ITALY
Email: {name.surname}@create-net.org

***Abstract*—In this paper we present VeRTIGO (ViRtual TopologIes Generalization in OpenFlow networks), a Software–defined networking platform designed for network virtualization. Based on the OpenFlow original network slicing system FlowVisor, the VeRTIGO platform aims at covering all flavors of network virtualization: in particular, it is able to expose a simple abstract node on one extreme, and to deliver a logically fully connected network at the very opposite end.**

In this work, we first introduce the VeRTIGO system architecture and its design choices, then we report on a prototypical implementation deployed over an OpenFlow–enabled testbed. Experimental results show that VeRTIGO can deliver flexible and reliable network virtualization services to a wide range of use cases in spite of failure and/or congestion at the underlying physical network.

I. Introduction

The recent interest on Network Virtualization (NV) is motivated by two main reasons: first, NV enables the deployment of different architectures and protocols over a shared physical infrastructure, and for such a reason it is considered the most concrete tool to validate novel approaches coping with the current Internet "ossification". Second, NV aims at letting different virtual network instances to coexist and a clean separation between services and infrastructures, so that NV can enable new business models beyond basic connectivity, thus offering infrastructure providers new potential income sources.

As highlighted in [1], [2], most works on NV pivoted around the idea of slicing existing network resources (at link and node level) to instantiate several logical instances of networks composed of virtual nodes. This approach is based on the infrastructure as a Service (IaaS) model, where physical resources are shared among different users. In this model, those who operate virtual networks face all the customary operational complexity of managing a network: this includes dealing with network congestion, node/link failures, limited automation in the configuration of the devices, etc. This option, worth indeed for users who want direct control of the network they run, sounds costly to those focused on providing added-value services on top. To these users, eventually, the most interesting option is to be given access to the network as if it was an "abstract node", thus hiding all details of the underlying physical or virtual topology. In this way, specific services can be simply offered by properly configuring such an abstract node.

In this paper we introduce a novel architecture leveraging the Software defined networking (SDN) paradigm: the aim is precisely to broaden the "virtualization offering" that an infrastructure provider can expose to its customers. Hence, depending on customers' needs, the proposed architecture can enable the instantiation of either (i) a virtual network composed of virtual links and virtual nodes or (ii) an abstract node collapsing the whole network into a single router or switch instance. In fact, in our view, the customer can choose among these two extreme options:

1) *full virtual network*: the customer has full control of the network (e.g. traffic engineering techniques, failure handling policy, etc). The topology should be fully customisable according to the needs of the customer, which sometimes may look for network configurations that depart from the physical one;
2) *single (abstract) node*: the customer concentrates on routing policies, while leaving the management of the underlying geographically distributed network layer to the infrastructure provider. The infrastructure provider can differentiate its offering according to the service level required by its customers (e.g. maximum latency or packet loss between two node ports).

In either case, the core requirement for the infrastructure provider is to ensure that the service level is acceptable to its customers, even in case of network congestion and/or failures.

At present, FlowVisor [3] is the most popular SDN based implementation to instantiate virtual networks. FlowVisor leverages on the capability of OpenFlow [4] of abstracting the underlying hardware and can operate logically between control and forwarding paths on a network device effectively exposing different view (slices) of a physical infrastructure to different controllers.

In a recent work [5] the authors have proposed a mechanism to overcome one of the main limitations of FlowVisor, i.e., the fact that virtual topologies in FlowVisor are restricted to subsets of the physical topology. However, the solution proposed in [5] leverages on the VLAN tag to differentiate between virtual links and virtual network. The price to pay is an hard limitation on the usage of VLAN headers.

Authors in [1], [2] have described in detail the advantages of exposing a single abstract node. However, little details have been provided about the corresponding architecture and related performance figures. In [6] the author explicitly refers to Keller's work; the paper focuses on algorithms to optimally embed "virtual router services" on top of a physical network. Authors of [7] elaborated on their previous work (RouteFlow) to generalize the instantiation of purely IP-based virtual routers towards more flexible resource mapping (e.g., by multiplexing

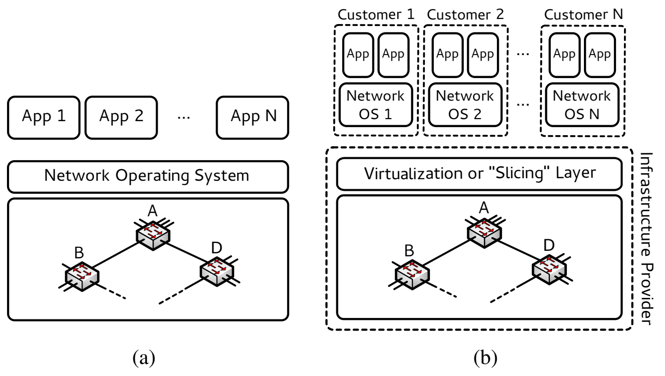

Fig. 1. (a) A Software-defined network architecture; (b) an SDN network where a virtualization layer allows to share the physical infrastructure among several customers

1:N or aggregating M:1/M:N routers) running over detached VMs, thus not really exploiting the full potential of virtualizing network resources. Most of these proposals generally restrict to pure L3 routing services, and lack implementation details or performance results.

In this paper we present VeRTIGO, a NV architecture that extends FlowVisor to deal with virtualization scenarios described before. The basic idea is to extend FlowVisor with additional intelligence able to expose different views of the network to different controllers, depending on customer's specific needs. Compared to previous contributions on this topic, this paper includes a set of preliminary results obtained on a real network in production within the OFELIA facility [8]. Results show that our solution supports failover capabilities in case of network congestion or link failures; we could also identify relevant research directions to improve robustness.

The structure of the paper is as follows. Sect. II discusses motivations for the proposed architecture; it also clarifies the need for a new NV solution truly able to fit all the potential requirements from the customers. The architecture of VeRTIGO is then presented in Sect. III and some preliminary results discussed in Sect. IV. Finally, Sect. V draws conclusions and indicates directions to upgrade VeRTIGO.

II. Motivation

Software defined networking (SDN) is a recent architectural networking framework aimed at decoupling the network control plane from the physical topology. Eventually, the data plane is to be controlled through a uniform vendor-agnostic interface. One of the most important features of such a framework is a network-level operating system able to expose a logical map of the entire network to services or control applications implemented on top of a *logically centralized* control plane. This network architecture lets a network administrator (or a researcher) to easily introduce new network functionalities by writing a software program that handles the network; as seen in Figure 1(a) this resembles closely what happens on a single computer scenario. One of the most known implementations of SDN is OpenFlow [4]; we observe that, however, there is no requirement for the use of OpenFlow within a software-defined network.

An interesting feature of SDN is the possibility to slice available network resources; within the set of available OpenFlow based tools, FlowVisor is the one acting as Network Virtualization layer. Leveraging on the hardware abstraction provided by OpenFlow, FlowVisor sits between the physical hardware (network element) and the software that controls it (controller). FlowVisor uses OpenFlow protocol to slice the underlying physical network. FlowVisor generally hosts multiple guest controllers, one controller per slice: it ensures that a controller can observe and control its own slice only, while isolating one slice from another (see Figure 1(b)).

While the advantages for an infrastructure provider to adopt or to leverage on NV techniques are pretty clear (see [9] for an interesting summary), some potential drawbacks exist [1]; this holds especially for customers whose focus is on providing added-value services to the end users. In fact, a customer renting a virtual network still has to manage a network; in order to have their services delivered, they have to deal with critical scenarios like network congestion or failure conditions. Moreover, NV may introduce additional complexity to the infrastructure provider as well: in order to deliver to the customer the service level he/she is paying for, the infrastructure provider must allocate enough resources to provide such guarantee; in turn, this limits dramatically the opportunity for optimizing its infrastructure usage. Furthermore, the business model behind this service is based (again) on "connectivity" thus limiting the chance to differentiate from other infrastructure providers.

Customers willing to focus on their service portfolio offering would greatly benefit from a virtualization architecture that expose the network as if it was a single node, strongly decoupling the two worlds and letting the customer to focus on its services only while leaving the infrastructure provider to manage the complexity of a physical network. At the same time, there could be customers willing to extend their market geographically (e.g. infrastructure owners) that would instead prefer to rent a "full-fledged" virtual network in order to keep a direct control on its operations.

We argue that most probably there is no solution that fits all the potential requirements from the customer but instead it is more promising to let the customer choose among the two extremes according to its needs. By leveraging on SDN capability to abstract network resources and control them through primitives, VeRTIGO can enable such a vision. As shown in Figure 2, through VeRTIGO it is possible to instantiate either a single node interconnecting the edge locations or a virtual network with a fully customisable topology to guarantee maximum flexibility to the customer who rents it. Further details are provided in the following section.

III. Architecture

The main building blocks of the VeRTIGO architecture are shown in Figure 3. As seen there, VeRTIGO interacts with the underlying network and the controllers via the control channel,

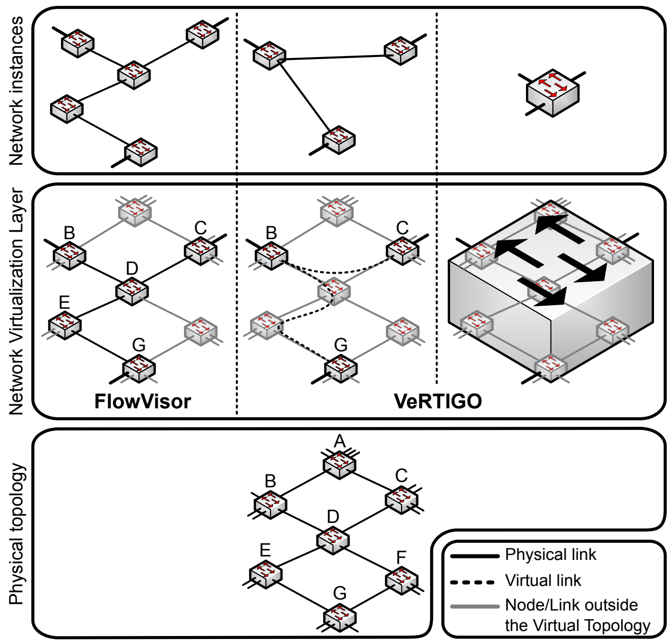

Fig. 2. Network virtualizing through SDN: FlowVisor Vs. VeRTIGO.

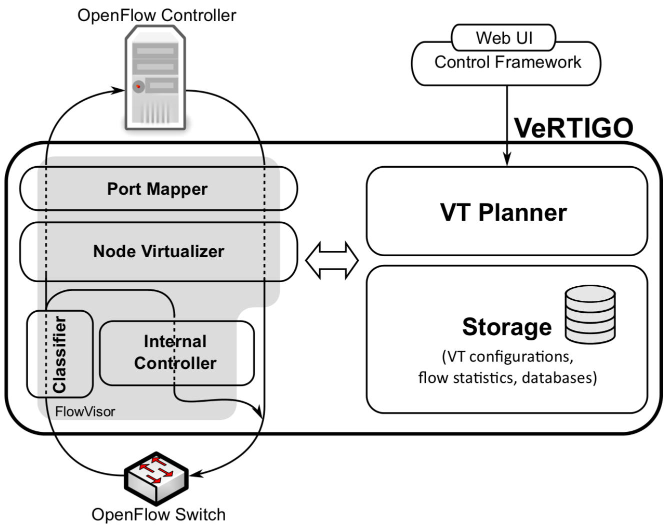

Fig. 3. The main building blocks of VeRTIGO.

based on OpenFlow protocol. In particular, it can be noticed that VeRTIGO is implemented around the latest release of FlowVisor: it adds a set of key additional modules that we describe in this Section.

VeRTIGO's capability to provide abstract instances of the physical network (namely, either virtual networks or abstract nodes) is grounded on two basic virtual elements: *Virtual Links* and *Virtual Ports*. These two elements are used both to instantiate arbitrary network topologies including virtual links between not adjacent switches, as well as to interconnect remote access ports of an abstract node. In the first scenario virtual links and ports are exposed to the OpenFlow controller as part of the network. In the latter, the controller only sees an abstract OpenFlow switch.

As in [5], virtual links aggregate logically sets of physical links and OpenFlow nodes while virtual ports are simply physical ports with virtual port numbers. A single physical port can be mapped onto multiple virtual ports, depending on the number of virtual links instantiated on the physical link connected to that physical port. However compared to that work, VeRTIGO does not tamper with the flow headers to implement virtual links but, instead, makes use of a database to store the header sequence of each flow crossing a specific virtual link.

Classifier: the Classifier module classifies OpenFlow messages received from a switch leveraging on the information contained in the configuration files. In the abstract node scenario, only messages related to access ports are forwarded to the OpenFlow controller while the other messages are handled by VeRTIGO's internal controller. In the virtual network scenario, only end points of a virtual link are enabled to communicate with the controller. All the other nodes are hidden by VeRTIGO which directly controls their flow tables.

Node virtualizer: the OpenFlow control channel is the interface that connects each OpenFlow switch to an OpenFlow controller. The controller identifies the switches and their features through this interface and builds a switch/topology map. When VeRTIGO instantiates an abstract node, it also needs to implement a single control channel between the nodes and the controller. The purpose of the Node Virtualizer is to multiplex multiple channels between the physical network and VeRTIGO into a single virtual channel between VeRTIGO and the controller. This multiplexing process is performed along with the process of remapping the datapath identifiers of network nodes.

Port Mapper: port mapper operations are needed in the virtual network scenario where a single physical port can be the termination of multiple virtual links. In this situation, each OpenFlow message related to this physical port must be modified with virtual port numbers consistent with the virtual links before being forwarded to the controller. The port mapping process is also necessary in the abstract node scenario to remap the access port numbers.

Internal controller: this module controls those switches that are hidden to the controller. This is the case for all switches composing an abstract node and for switches that are not end points of virtual links in the virtual network scenario. These nodes never communicate with the remote controller as their flow tables are managed by the Internal Controller.

Storage: the storage module provides a set of operations and functions to store configurations for abstract nodes and virtual networks and to handle databases of flow headers and flow statistics.

VT Planner: this component implements a path selection algorithm to efficiently aggregate physical nodes and links into virtual links. The task of the VT planner is to associate network instances (as per customer request) to physical resources: it receives as input traffic load statistics and provides, as output, the description of virtual links that implement either an abstract node or a virtual network. Heuristics for the allocation of virtual links rely on the characterization of physical links based on a set of metrics such as throughput and latency.

A dedicated monitoring module, not shown in Figure 3, is in charge of collecting flows statistics to be input to the VT Planner. Information gathered from network devices by the monitoring module includes: (i) nominal capacity of physical ports and (ii) statistics on received and transmitted bytes and packets on each physical port.

UI and Control Framework: in order to be effectively used by an infrastructure provider, the proposed architecture needs an external Control Framework[1]; the aim is to simplify the configuration of virtual networks or abstract nodes by specifying requirements in terms of, e.g., flowspaces, nodes, virtual links etc. The configuration is passed to the VT Planner.

IV. Evaluation results

The experimental evaluation described in this Section has two main purposes: to evaluate the overhead introduced by our initial prototype and to show how VeRTIGO transparently handles changes in the physical network substrate. In particular, we use two reference events: link failure and network congestion. The focus of all the tests is on the "abstract node" virtualization scenario, this choice has two main reasons: (i) being based on experiments performed on a real network, it provides concrete evidence on the capability of VeRTIGO to support practical and relevant use cases; and (ii) the preliminary results obtained provide hints on where architectural enhancements could significantly improve performance.

A. Evaluation setup

The experiments described in this section were conducted on the OpenFlow 1.0 devices located in our premises and that are part of the CREATE-NET island in production within the OFELIA facility [8]. The island currently consists of:

- 9 OpenFlow switches: 3 NEC IP8800, 2 HP ProCurve 3500 and 4 commodity PCs hosting NetFPGA 1 Gbit modules
- 6 server-class PCs: Intel Xeon 4 core processors

The tests in Sect. IV-C have been performed using 3 NEC switches, all 4 NetFPGA hosting PCs and 3 servers, connected as shown in Figure 6. The tests in Sect. IV-B have been performed using one of the NEC switches and a commodity PC equipped with a quad-core 2.7 GHz processor.

B. Latency overhead

VeRTIGO's operations have a cost in terms of additional latency on the control channel. In this experiment we measured the overhead introduced by VeRTIGO on the control channel for common OpenFlow messages such as: packet_in (new flow) and port statistics requests/replies.

New flow. In this experiment we connected two Ethernet interfaces of the commodity PC to the switch. One interface was used to force the switch to generate new flow messages for the controller by generating 50 flows per second.The second one connected a modified version of the NOX controller [11] to the switch through the OpenFlow control channel.

In OpenFlow, for each incoming flow which does not have a flow entry in the switch flow table, a packet_in message is sent to the controller through the control channel. We measured the "new flow time", i.e., the latency between sending a new flow through one interface and receiving the corresponding packet_in message from the second interface. We analyzed three different scenarios: i) direct connection between switch and controller, ii) FlowVisor v0.8.1 logically placed between switch and controller, iii) VeRTIGO logically placed between switch and controller.

Figure 4 shows that VeRTIGO's operations increase the new flow time from switch to controller by 1.065 ms on average.

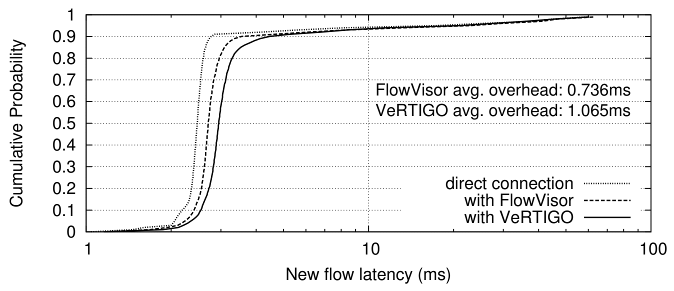

Fig. 4. Cumulative probability of the latency for *new flow* messages.

Port statistics request/reply. In this experiment we connected only one interface of the PC to the switch. On the PC we run an instance of the NOX controller configured to send approximately 200 port statistics requests per second and to record the round-trip delay between the request and the statistic reply sent by the switch.

Results of the experiment are reported in Figure 5.

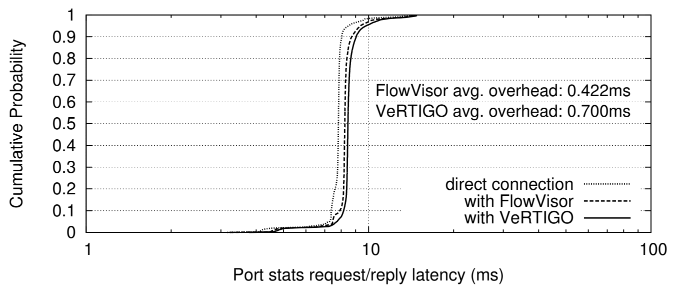

Fig. 5. Cumulative probability of the latency for *port stats* requests.

Although these are only preliminary results, they clearly show that VeRTIGO operations do not significantly degrade the performance of the control channel. However, as a future work, we plan to perform additional tests using more powerful controllers like Beacon [12], FloodLight [13] and Maestro [14]. Furthermore, we will also evaluate the scaling limits of VeRTIGO with respect of the rate of generated messages and the number of physical nodes.

[1]Within the OFELIA project, we are planning to leverage on an extended version of the GENI–based FOAM (FlowVisor OpenFlow Aggregate Manager) [10].

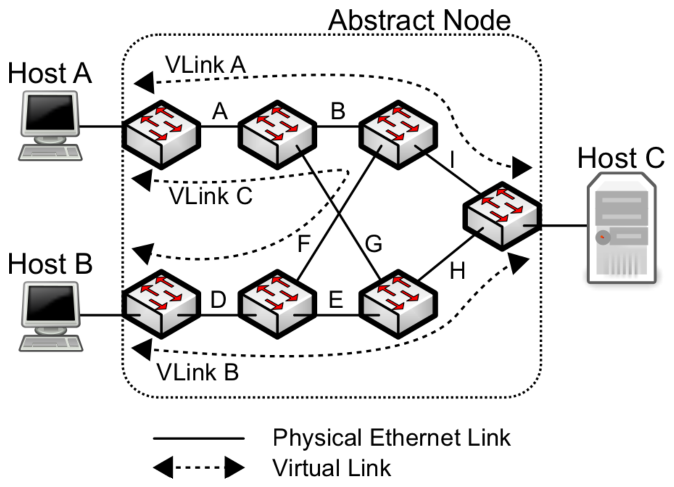

Fig. 6. Representation of the abstract node instantiated for the traffic congestion/link failure tests.

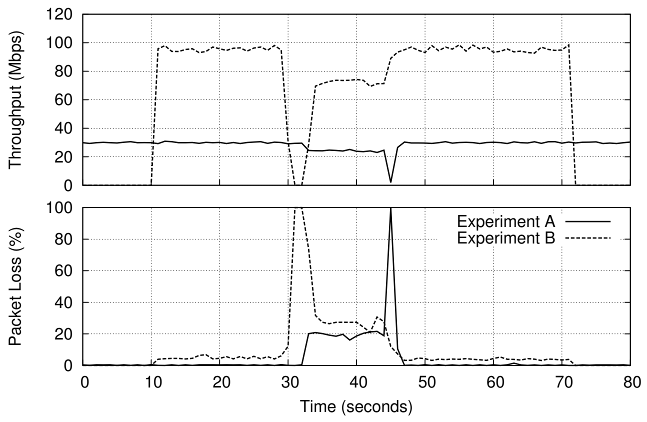

Fig. 7. Throughput and packet loss measured during the traffic congestion/link failure test.

C. Network congestion/Link failure tests

The objective of this test is to prove the ability of VeRTIGO to react to events such as network congestion and link failures. In such cases, VeRTIGO should be able to automatically reconfigure virtual links while, at the same time, minimizing the connectivity downtime caused by the reconfiguration process.

In order to do so, the current implementation of the VT Planner is based on a series of pre–defined cached virtual topologies for which it enforces the corresponding embedding into the physical topology. We remark that the general problem of virtual topology embeddings has received some attention in literature [15], [16]. We are planning to implement more complex virtual topology embedding algorithms (with no need for pre-caching) in future releases of this component. Finally, it is worth noticing that, the dynamic virtual link reconfiguration feature can be disabled in order to support use cases where experiment reproducibility is a requirement.

In our test, we instantiated an abstract node as represented in Figure 6. The underlying virtual topology to interconnect its three access ports has been computed by the VT Planner by instantiating virtual links VLink A, B and C.[2] Each of the three access ports was also connected to one of the server-class PCs configured for the experimentation as *Host A*, *Host B* and *Host C*.
All internal links were limited to 100Mbps, while links connecting the hosts were left to the nominal speed of 1Gbps. VeRTIGO detects link failure events by using port-status messages from switches. Also, it detects network congestion events by polling and querying switches for the tx/rx statistics every 5 seconds: congestion is detected on a virtual link when the throughput on any of its physical links exceeded 80% of the nominal capacity 3 times in a row. On either congestion or link failure, the VT Planner recomputes the best path algorithm for the involved virtual links.
The test started at time 0 (see plots in Figure 7) when *Experiment A* generated a 30 Mbits/sec flow from Host A to Host C through the VLink A. After 10 secs, a second flow at 100 Mbits/sec was generated by *Experiment B* from Host B to Host C through the VLink B. At time 30 secs, we simulated a link failure by disconnecting physical link E. As a result, the throughput of Experiment B decreased to 0 while the loss rate increased to 100%. As reported by plots in Figure 7, after about 2 secs, VLink B was restored with a new sequence of physical links, i.e., D-F-I.
Starting at second 33, virtual links VLink A and VLink B were sharing the physical link I (see plots in Figure 7): hence, both experiments started experiencing a reduction of the throughput and higher packet loss rate.
About 11 secs later, the VT Planner detected congestion on the link and computed again the best path algorithm: virtual link reconfiguration took about 1 second (see the spike in the loss rate plot at time 44 sec) and produced a new configuration for VLink A with the following aggregation of physical links: A-G-H. As shown in Figure 7, after re-computation, both experiments were running at full rate again.

This experimentation demonstrates how critical the virtual link reconfiguration process is, due to the long connectivity interruptions it can introduce. In our implementation the process consists of the following operations: (i) recomputation of the best path excluding any possible failed link and using rx/tx statistics as a metric, (ii) deletion of all flow entries previously installed to set up the virtual link to be restored/re-configured and (iii) set up of a new virtual link by installing new flow entries on all switches on the path. Our experimentation also proved that connectivity issues are less impacting when reconfiguring congested links. In this case, in fact, recomputing an alternate best path for a congested virtual link leaves the virtual link operational until the new path is determined. Of course, latency can dramatically improve by either instantiating more meshed underlying topologies or by leveraging on

[2]Of course, richer meshed topologies can be used to interconnect access ports, provided that one configures the Control Framework parameters accordingly; we preferred to confine to the simple one reported for the sake of clarity.

MPLS fast restoration mechanisms as soon as OpenFlow 1.2 compliant devices will be available on the market. It is worth noticing that even though node failures scenarios have not been considered, they could potentially leverage on the same re-configuration process: actually, a node failure corresponds to a multiple links failure.

V. Conclusions and next steps

This paper presents VeRTIGO, a novel architecture based on the SDN paradigm. VeRTIGO generalizes the "virtualization offering" that an infrastructure provider can deliver to customers. It supports the instantiation of either (i) a virtual network composed of virtual links and nodes with arbitrary topology or (ii) an abstract node collapsing the whole virtual network into a single router or switch instance.

VeRTIGO extends FlowVisor by additional intelligence: it can expose different views of the network to different controllers in order to meet customer's specific needs. In particular, in the "abstract node" scenario, previous contributions [1], [2] have not fully elaborated on the corresponding architecture and evidence of the related performance was still pending; this paper includes a set of results obtained on a real network in production within the OFELIA facility [8]. We have shown not only the feasibility of the proposed approach, but outlined also its performance when dealing with events like network congestion or link failures. Our ultimate goal indeed is to deliver a NV protocol suite reliable enough to let infrastructure providers offer NV-based services on top of their asset. To this respect, these preliminary outcomes provide valuable insight to identify the critical aspects to be strengthened in order to make such systems competitive.

As part of future work, we intend to improve the VT Planner by implementing a virtual topology embedding algorithm. This is meant to provide VeRTIGO with higher flexibility when allocating physical resources to virtual networks and abstract nodes instances. Also, another key element to improve system performance would be the availability of OpenFlow v.1.2 enabled network devices, including support for MPLS and QnQ techniques.

References

[1] E. Keller and J. Rexford, "The "Platform as a Service" model for networking," in *Proc. of USENIX INM/WREN*, San Jose, California, 27 Apr. 2010.

[2] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker, "Virtualizing the network forwarding plane," in *Proc. of ACM PRESTO*, Philadelphia, USA, 30 Nov. 2010.

[3] R. Sherwood, G. Gibb, K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" in *Proc. of USENIX OSDI*, Vancouver, Canada, 4-6 Oct. 2010.

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 2, pp. 69–74, April 2008.

[5] E. Salvadori, R. Doriguzzi Corin, A. Broglio, and M. Gerola, "Generalizing virtual network topologies in OpenFlow-based networks," in *Proc. of IEEE GLOBECOM*, Houston, TX, USA, 5-9 Dec. 2011.

[6] Z. Bozakov, "Architecture and algorithms for virtual routers as a service," in *Proc. of IEEE IWQoS*, San Jose, California, 6-7 June 2011, pp. 1–3.

[7] M. Nascimento and C. R. et al., "Virtual routers as a service: the routeflow approach leveraging software-defined networks," in *Proc. of ACM CFI*, Seoul, Korea, 13-15 June 2011, pp. 34–37.

[8] OFELIA project. Online: http://www. fp7-ofelia.eu.

[9] J. Carapinha and J. Jiménez, "Network virtualization: a view from the bottom," in *Proc. of ACM VISA*, Barcelona, Spain, 17 Aug. 2009, pp. 73–80.

[10] GENI Flowvisor OpenFlow Aggregate Manager (FOAM). Online: https://openflow.stanford.edu/ display/FOAM/Home.

[11] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards and operating system for networks," *ACM SIGCOMM Computer Communication Review*, July 2008.

[12] Beacon. Online: https://openflow.stanford.edu/display/Beacon/Home.

[13] Floodlight. Online: http://floodlight.openflowhub.org/.

[14] Z. Cai, A. Cox, and T. Ng, "Maestro: A System for Scalable OpenFlow Control," *Tech. Rep. TR10-11, Rice University - Department of Computer Science*, December 2010.

[15] M. Chowdhury, M. R. Rahman, and R. Boutaba, "ViNEYard: Virtual network embedding algorithms with coordinated node and link mapping," *IEEE/ACM Trans. on Networking*, vol. 20, no. 1, pp. 206 –219, Feb. 2012.

[16] J. Lischka and H. Karl, "A virtual network mapping algorithm based on subgraph isomorphism detection," in *Proc. of ACM VISA*, Barcelona, Spain, 17 Aug. 2009, pp. 81–88.