Software Defined Networking: Meeting Carrier Grade Requirements

(invited paper)

Dimitri Staessens, Sachin Sharma, Didier Colle, Mario Pickavet and Piet Demeester
Department of Information Technology
Ghent University - IBBT
Ghent, Belgium B-9050
Email: firstname.lastname@intec.ugent.be

***Abstract*—Software Defined Networking is a networking paradigm which allows network operators to manage networking elements using software running on an external server. This is accomplished by a split in the architecture between the forwarding element and the control element. Two technologies which allow this split for packet networks are ForCES and Openflow. We present energy efficiency and resilience aspects of carrier grade networks which can be met by Openflow. We implement flow restoration and run extensive experiments in an emulated carrier grade network. We show that Openflow can restore traffic quite fast, but its dependency on a centralized controller means that it will be hard to achieve 50 ms restoration in large networks serving many flows. In order to achieve 50 ms recovery, protection will be required in carrier grade networks.**

***Index Terms*—Openflow, Restoration, Energy Efficiency**

I. Introduction

The goal of Software Defined Networking is to provide open user-controlled management of the forwarding hardware of a network element. Openflow was designed particularly to deploy and test experimental protocols in the production quality campus network Stanford uses every day, instead of in a separated lab environment [1]. If operators want to be able to program the behaviour of high speed networking elements such as IP routers or Ethernet switches for their custom needs, they require direct programming of the forwarding hardware. Modern routers/switches contain a proprietary FIB (Forwarding Information Base), which is implemented in hardware using TCAMs (Ternary Content Addressable Memory).

Openflow provides control of forwarding hardware by providing a standardized abstraction of it called a Flowtable. An Openflow switch is a network element implementing an instance of the (abstract) Flowtable, and has a secure channel to the Openflow controller, which manages the Openflow switches using this Openflow protocol. The Openflow protocol supports messages to add, delete and modify flow entries in the Flowtable. A flow entry consists of (1) a *packet header* which defines the flow, (2) an *action* which defines how the packet should be processed, and (3) *statistics* which keep track of the number of packets per flow, the number of bytes per flow, and the time since the last packet matched per flow.

The controller installs these flow entries in the Flowtables of the Openflow switches. Incoming packets processed by the Openflow switches are compared against the flow entries in the Flowtable. If a matching flow entry is found, the predefined actions for that entry are performed on the matched packet. If no match is found, the packet is forwarded to the controller over the secure channel (PACKET_IN message). The controller is responsible to determine how the packet should be handled; either by returning this specific packet to the switch and stating which port it should be forwarded to (PACKET_OUT message) or by adding valid flow entries in the Openflow switches (FLOW_MOD message).

In the access/aggregation domains of most carriers, an Ethernet-based aggregation is used to provide services for residential and business customers. Implementing a split architecture between control and forwarding in these areas creates the opportunity for network operators to use commodity hardware under the control of centralized software to perform the intricate functions of specialized aggregation domain network elements at much a lower hardware cost. Moreover, Openflow allows virtualization and thereby supports service separation. As mobility is essential, the infrastructure to provide wireless services is a must and the aggregation network is also used to guarantee mobile backhauling.

The term *carrier grade* [2] describes a set of functionalities and requirements that architectures should support in order to fulfill the operational part of network operators. The requirements are (1) Scalability (2) Reliability (3) Quality of Service (QoS) and (4) Service Management. In order to be applied to carrier grade networks, Openflow must be able to meet these requirements. In this paper, we focus on reliability, and we also show how Openflow can enable energy-efficiency improving strategies to be deployed in the network. Reducing network power consumption can lead to improved hardware scalability and reduces the carbon footprint.

II. Energy efficiency in Openflow networks

An important goal for future networking is the reduction of its carbon footprint. The attention for climate change is influencing the ICT sector. ICT accounts for 2 to 4% of

the worldwide carbon emissions [3]. About 40 to 60% of these emissions can be attributed to energy consumption in the user phase, whereas the remainder originates in other life cycle phases (material extraction, production, transport, end-of-life). By 2020 the share of ICT in the worldwide carbon emissions is estimated to double in a business as usual scenario. Since optical signals consume less power than electrical signals, optical technologies could enable higher energy efficiency. There is a worldwide increase in research efforts in the last years, with initiatives such as the current EU funded Network of Excellence TREND [4], COST action 804 [5] , the GreenTouch consortium [6] and the CANARIE funded GreenStar Network [7] which also has European members.

III. Energy saving strategies

When it comes to solutions to save power in carrier networks, different strategies are possible. On the highest level one can investigate if optimizations are possible in the network topology. Currently, networks are designed to handle peak loads. This means that when the loads are lower an overcapacity is present in the network. At night time the traffic load can be 25% to 50% of the load during day time. This lower load could allow a more simplified network topology at night which in turn allows certain links to be switched off. Additionally, the switching off of these links allows for line cards to be switched off and thus leads to reduced node power consumption. An example which implements this principle is multilayer traffic engineering. The MLTE (Fig. 1 [8]) approach can lead to power savings of 50% during low load periods.

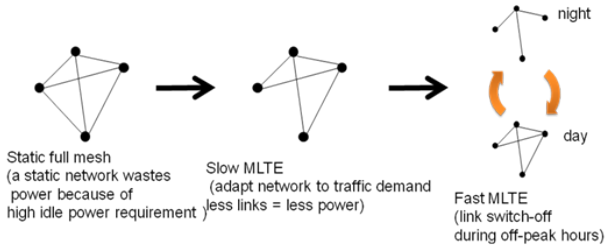

Fig. 1. Multilayer Traffing Engineering (MLTE)

The Openflow architecture allows us to implement MLTE operations as an application in the Openflow controller. In order to have the maximum benefit, the controller should be able to power up/down parts of the switch on demand, as a function of the energy-efficient algorithms in the application. Openflow currently has limited support for the control of power management in the switches. For enabling efficient power management, we should allow the burst and adaptive link modes in the switches and advertise them to the controller. On the controller side, it should be extended to allow control of such energy efficient features. Since the Openflow architecture removes the control software part from the switches and moves it to a central location, we could expect a reduction in power consumption in the switches at the cost of the power consumption of the controller.

Since access networks are organized in tree structures, shutting down links is not a feasible option, which means dynamic topology optimization cannot be applied in an access network. On a given topology further optimizations can be achieved by using adaptive link rates and burst mode operation [9]. Adaptive link rate is based on the principle that lower link rates lead to lower power consumption in the network equipment. By estimating the average link rate required on a line and adapting the link rate to this level power saving becomes possible. Another possibility is burst mode operation where packets are buffered in a network node and then sent over the link at the maximal rate. In between the bursts the line can be powered down. These strategies can be mainly useful in access networks due to the burstiness of the traffic. However, the difference in power consumption between different link rates is mainly manifested at the higher bit rates. Secondly, burst mode operation works on very small time scales so the number of components which can be switched off is limited. Finally, both approaches require larger packet buffers which also need powering. Hence it is yet unclear whether the strategies in reality can lead to significant power optimization.

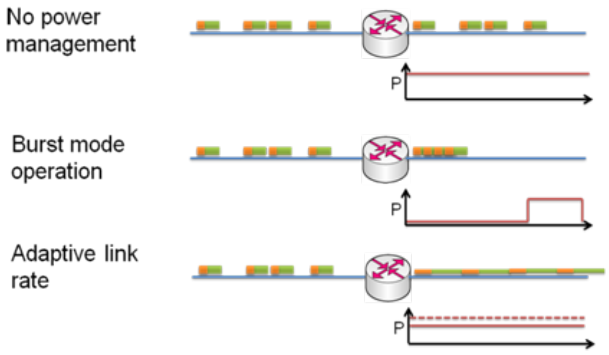

Fig. 2. Power management

To enable energy efficient networking applications to run on the Openflow controller, some extra messages should be added to the Openflow specification which not only indicate the status of a port on the switch, but also allows us to control the individual ports.

The following features should be controllable: port power up/down, burst mode, adaptive line rate. First of all the controller needs to be aware of the capabilities of the switch. So we would add the energy-efficiency features to the OFPT_FEATURES_REQUEST and OFPT_FEATURES_REPLY messages. In order to control the functionality, we need some extra protocol messages. These can be done by adding the capabilities to the OFPT_PORT_MOD message. All these messages can use the ofp_port structure to advertise features and to change them as required. Currently the ofp_port structure has a bit OFPPC_PORT_DOWN in the config field, which indicates whether a port is administratively down. This can also be used to power on/off a specific port on the switch using the OFPT_PORT_MOD message. We can implement adaptive line

rate using the curr_speed and max_speed fields. For the other features, it has to be investigated where they must be added to the ofp_port structure in the config field, the features field and/or the advertised fields. We would also need to add the OFPC_BURST_MODE.

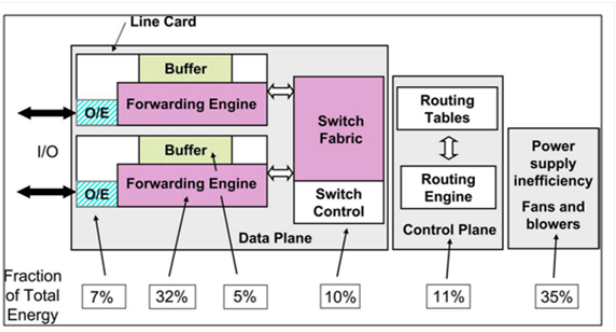

Fig. 3. Energy consumption of a core router [10]

Consolidation of the control software and hardware outside of the switches means a reduction in switch power consumption offset by the power consumption of a new element, the controller. As we can see from Fig. 3, most of the power consumption stems from the Forwarding Engine (32%) and the Cooling/Power Suply (35%). Only a small fraction (11%) of the power is consumed by the control plane, of which only 5% in the routing engine [11]. Openflow allows us to reduce this part by moving the routing tables (RIB)/routing engine and control plane functionality to the controller and keeping only the forwarding engine (FIB) in the switch with a smaller Openflow software component and extra hardware for performing communication with the controller. The controller will consume more power due to power supply inefficiency than the reduction in power in the switches, so we think the general Openflow architecture will be similar in power consumption compared to conventional network architectures.

IV. Data plane resiliency

Carrier grade networks should be able to detect and recover from incidents without impacting users. Hence a requirement is added in the carrier grade network so that it should recover from failure within 50 ms sub interval [12]. Resilience mechanisms [13] can be classified as restoration and protection. In case of protection, the paths are preplanned and reserved before a failure occurs. When a failure occurs, no additional signaling is needed to establish the protection path. In case of restoration, the recovery paths can be either preplanned or dynamically allocated, but resources are not reserved until failure occurs. When a failure occurs additional signaling is needed to establish the restoration path. Protection is a proactive strategy while restoration is a reactive strategy.

Data plane recovery in Openflow networks can be done in two essentially different ways. One is to support the recovery mechanisms of a specific implemented protocol into an Openflow application. An example is supporting MPLS-TE [14] [15] which has restoration functionality in its own control plane. The other approach is to build resilience into Openflow, supporting the recovery of arbitrary flows, regardless of the type of traffic they are carrying. This is the option we explore in this section.

A. Data plane restoration

Fast flow restoration in the Openflow data network requires an immediate action of the controller after notification of a link status change event thrown by the Openflow switch detecting the failure. The recovery is performed by removing the incorrect flow entries and installing new entries in all affected Openflow switches as fast as possible following the notification of the link failure. We use the following algorithm to restore traffic: after the controller gets notification of a link failure, a list is made of all affected paths. For all these affected paths, a restoration path is calculated using a shortest path algorithm on the remaining topology. For affected switches which are on both the working and the restoration path, the flow entry is modified. For the other switches, there are 2 possibilities. If the switches are only on the failed path, the entries are delete. If they are only on the restoration path, new entries are added.

B. Data plane protection

In order to further reduce packet loss resulting from restoration actions in an Openflow network, we can turn to protection. Protection removes the need of the Openflow switches to contact the controller for the deletion, modification and addition operations required to to establishment the restoration path. This is accomplished by precomputing the protection path and establishing it together with the original (i.e. working) path. This means adding the flow entries for the protection path in the switches. Advantages of this are faster recovery, but the disadvantage is a larger Flowtable, which increases costs (expensive TCAM equipment) and may slightly impede forwarding performance.

Path protection requires end-to-end monitoring of the path to enable the quick switchover. The recovery mechanism depends on Bidirectional Forwarding Detection (BFD) session to declare the fault in path. BFD is a network protocol used to detect faults between two end-points. It provides low-overhead detection of faults even on physical media that don't support failure detection of any kind, such as Ethernet, virtual circuits, tunnels and MPLS Label Switched Paths.

V. Control plane resiliency

Because Openflow is a centralized architecture, relying on the controller to take action when a new flow is introduced in the network, reliability of the control plane is of very high importance. The controller should also be resilient against targeted attacks. There are multiple options for control plane resiliency. One can provide two controllers, each in a separate control network and when connection to one controller is lost, the switch switches over to the backup network. This is a very expensive solution. Another option is to try to restore the connection to the controller by routing the control

traffic over the data network. When a switch loses connection to the Openflow controller, it sends its control traffic to a neighboring switch, which will require the controller to detect such messages and establish flow entries for routing the control traffic through this neighbour switch. This through-the-data-plane solution is an intermediate step towards full in-band control. An effective scheme for carrier grade networks may be to implement out-of-band control in the failure free scenario, switching to in-band control for switches who lose the controller connection after a failure. In-band control is supported in the Openflow specification.

VI. Case study: restoration in a nationwide topology

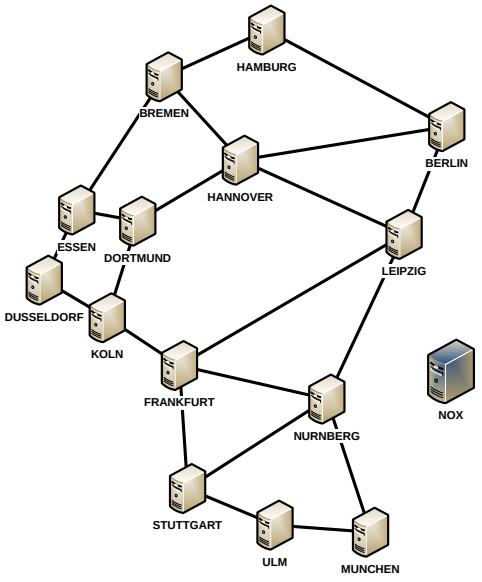

Fig. 4. German backbone network topology

In this section we evaluate flow restoration in Openflow networks. The topology of the experiment is depicted in Fig. 4. It contains 14 nodes and 21 links.

We emulated this topology on our iLab.t Virtual Wall testbed. The iLab.t Virtual Wall facility is a generic test environment for advanced network, distributed software and service evaluation, based on emulab [16]. The virtual wall facilities consist of 100 (linux) nodes (dual processor, dual core servers, 6x1 Gb/s interfaces per node) interconnected via a non-blocking 1.5 Tb/s VLAN Ethernet switch, and connected to a display wall (20 monitors) for experiment visualization. Each node is connected with 4 or 6 Gigabit Ethernet links to the switch.

Each of the 14 Openflow nodes is connected to a server node (not shown) and also has a dedicated interface to a switched ethernet LAN which establishes connection to the NOX controller (i.e. out-of-band control). All links are 1Gb Ethernet links. We implemented the restoration algorithm in the NOX Openflow controller [17]. Switches are running Open vSwitch [18]. To evaluate the restoration time, we generate packets using the linux kernel module pktgen with 3 ms intervals. In total there are 182 flows in the network. We also loop traffic from each server through its connected switch to verify pktgen generation consistency. All server nodes have manually configured routing tables which send all traffic through the interface connected to the Openflow network.

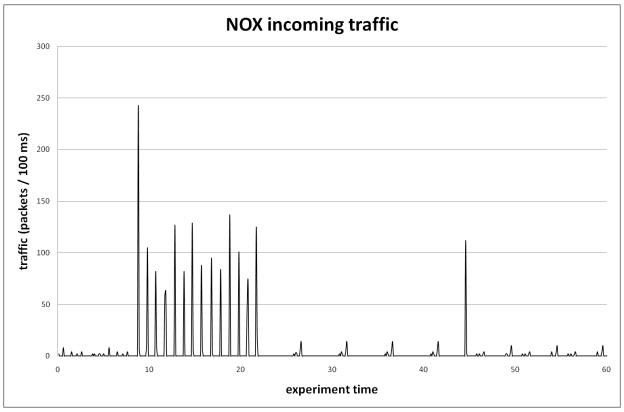

Fig. 5. Traffic received at NOX

We will use the incoming traffic intensity at the NOX controller as captured using tcpdump to illustrate the experiment setup. This is shown in Fig. 5. At the beginning of the experiment, the Openflow switches connect to the controller which generate the low spikes in the first seconds of the experiment. Then we start sending the pktgen traffic, waiting one second between each server to allow the NOX to install the flow entries. These are the 14 large spikes 9 s to 23 s in the experiment. The number of packets depends on the path lengths as each switch requests flow entries to the controller. The one second interval is to make sure we don't accidentally overload the NOX by trying to establish too many flows in a short timespan. After the flows are installed, we see small spikes at 5s intervals. These are ECHO_REQUEST messages which are sent to check liveness of the controller link. At roughly 44.5 s into the experiment we break the link Berlin-Hamburg by giving an eth down command in the Berlin switch, which will trigger a PORT_STATUS message sent to the controller when the Berlin switch detects its Ethernet port is down. At this point, the controller starts recovery (large spike at around 44.5 s) and all traffic is restored.

Fig. 6 shows traffic on the link Berlin-Hamburg as captured in Hamburg. Capturing with tcpdump started roughly 8 seconds into the experiment. We show both the total traffic on the link, as the (unidirectional) traffic from Berlin to Hamburg. At the start we see a stepwise increase in traffic as each pktgen is started on each node. The large step at approximately 15 s is when the server at Hamburg starts sending traffic to all other clients. By 23 s all traffic is set up in the network. Each flow is roughly 300 packets per second, total traffic on the link is around 6000 packets per second. At around 44.5 s, the link is broken and all traffic on the link is lost.

Fig. 7 shows the traffic on the link Bremen-Hamburg. After the link Berlin-Hamburg is taken down, this is the only link

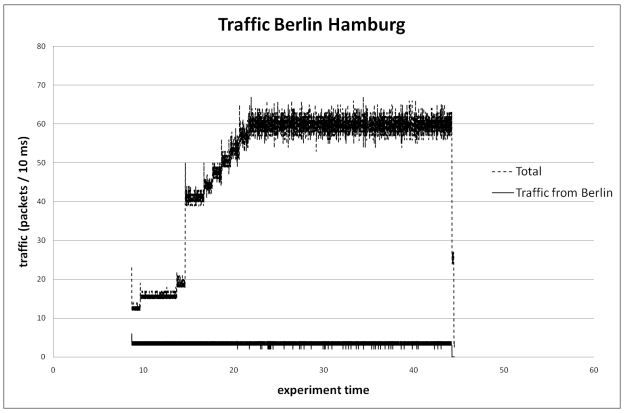

Fig. 6. Traffic on affected link

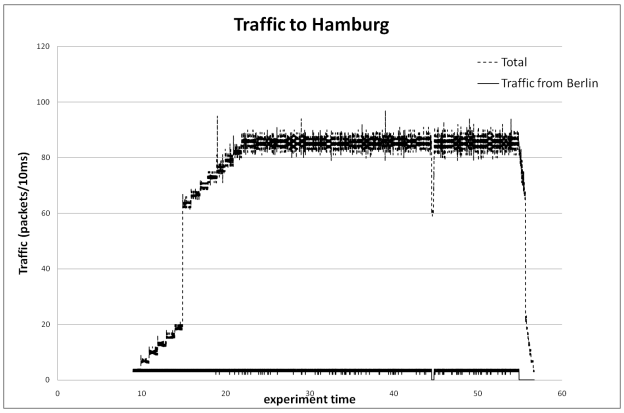

Fig. 8. Traffic at Hamburg

Fig. 7. Traffic on restoration link

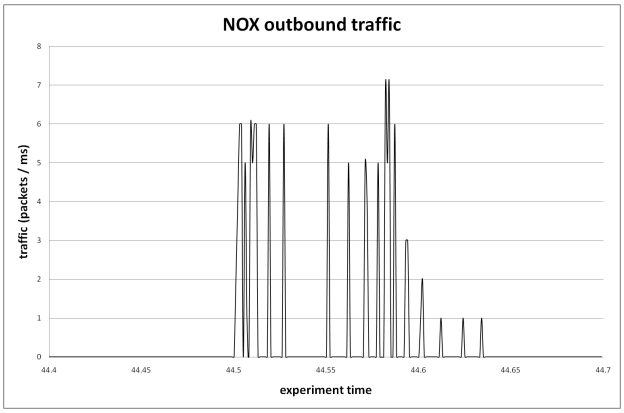

Fig. 9. NOX outbound traffic during recovery

connecting Hamburg, so all traffic from and towards Hamburg must now follow this link. Before the link failure, no traffic from Berlin to Hamburg is on this link. At the time of the link failure, we see a small drop in total traffic on the link. This is the traffic which was coming from the link Berlin-Hamburg over Hamburg-Bremen which is now lost. The restoration on the controller reroutes the affected traffic, and we see an increase in the traffic on this link. After restoration, the link serves all traffic to Hamburg, so it also contains the traffic from Berlin.

Fig. 8 shows the traffic from the viewpoint of the client/server at Hamburg. After the initial setup of the pktgen in all servers, the traffic is stable until the link failure at 44.5 s. Hamburg sees a drop in received traffic as some flows are lost until the controller restores the traffic. The traffic flow from Berlin shows that its traffic is completely lost over a short interval, inspection of the tcpdump traces shows this to be 289.75 ms.

Fig. 9 shows a detail of the outgoing traffic from the NOX controller during the recovery phase, captured in 1 ms intervals. At approximately 44.498 s, NOX starts sending the FLOW_MOD (modify/delete/add) messages required to install the new Flow entries in the affected switches. The tcpdump trace shows that in total 118 FLOW_MOD messages are sent to restore 19 affected flows in 102 ms, including the path calculation times. The 3 (last) packets in the graph at times $45.61 - 45.63$ s are TCP ACK messages and need not be included, recovery is finised at time 45.6 s.

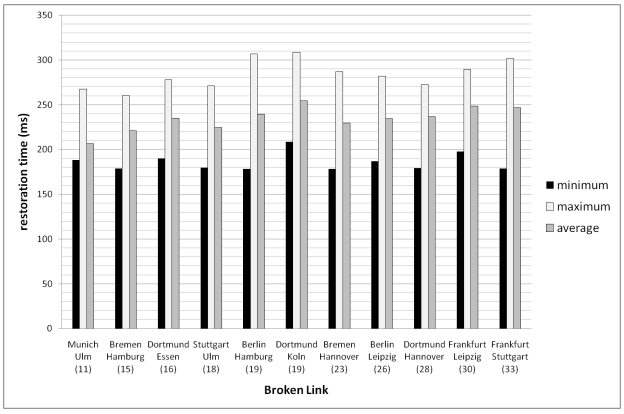

Fig. 10. Restoration times

We did this experiment for a number of link failures in the given experimental topology. The results are depicted in Fig. 10. The x-axis shows the broken link, the y-axis shows, in milliseconds, the minimum restoration time (or the time it

took to restore one connection), the maximum restoration time (or the time it took to restore all connections) and the average restoration time (the expected time for any flow to be restored after a failure). The links are ordered from left to right in the number of flows that are affected if this link fails, shown in brackets in the axis labels.

The figure shows that the first flows are restored roughly 180 ms after the failure, while total recovery takes anywhere between 260 and 310 ms. Also, there seems to be only minor dependence on the number of flows which had to be restored. In fact, recovery time will depend on the number of flows to be restored, the average path length of each flow, the average path length of the restoration paths and the average number of nodes which are on both the working and the restoration path. Furthermore, there are some unknown random factors, such as momentary load of the NOX cpu and traffic bursts in the control network. Such factors make drawing conclusions from a single experiment difficult. Therefore we will perform more experiments to allow statistical analysis, which will allow us to see how the recovery time scales with the number of flows to be restored.

The recovery times in Fig. 10 are substantially longer than the 100 to 105 milliseconds it takes for the NOX to calculate the new paths and send out the FLOW_MODS to the switches. Actually, the tcpdump traces show that the first paths are restored within 1 to 6 ms after the NOX starts sending the correct flow entries. The reason for the difference in delay is the failure detection time, i.e. the time it takes the linux kernel to detect/declare that the ethernet link is down after the "eth down" command effectively took the interface down. This seems to be roughly 170 ms in our experiments. So a significant part of the recovery time in Fig. 10 is currently in detecting the failure. However, and this is an important point, even with instant detection, while the recovery of a single flow would take on the order of 10 - 20 milliseconds [19], to recover all flows in the network would take at least the time needed by NOX to perform all recovery actions. For our experiments this was roughly $80 - 130$ ms. In carrier grade networks, the number of flows to restore will be orders of magnitude higher, requiring many more flows to be restored, which will severely stress the control network and controller hardware for the short timespan of recovery. In normal operation, the control network load is generally orders of magnitude lower. Implementing a high speed control network only for restoration will probably not make sense. Implementing protection mechanisms in the switches will be more cost-efficient, slightly increasing the bandwidth requirement at flow setup time due to extra protection information to be sent to the switch, but highly decreasing the bandwidth requirements during failures by allowing the switch to perform the protection switching without controller interference.

VII. Conclusions and future work

We have presented two aspects of carrier grade networks which can be met by Openflow, being improved scalability by reducing the energy consumption and performing recovery in case of network failures. While the Openflow architecture may not be able to significantly reduce energy consumption by consolidating the control hardware/software in a single machine, it shows significant promise by facilitating network-wide energy efficiency solutions such as MLTE in combination with local power saving options such as controlled adaptive line rates in the Openflow switches. Second, we give an indication how Openflow can handle both data plane and control plane failures. We have implemented a flow restoration scheme in an open source controller (NOX) and ran extensive experiments in an emulated carrier grade topology. We show that Openflow can restore traffic, but its dependency on a centralized controller means that it will be hard to achieve 50 ms restoration in large networks.

In future work, we will implement flow protection to be able to recover under 50 ms. Also we will experiment with hybrid in-band and out-band control for dealing with failures in the controller network.

Acknowledgments

This work was partially funded by the European Commission under the 7th Framework ICT research Programme projects SPARC [20], OFELIA [21] and Network of Excellence TREND [4].

References

[1] N. McKeown et al., *Openflow: Enabling innovation in campus networks*, SIGCOMM, Rev. 38(2), 69-74, 2008.
[2] http://metroethernetforum.org/index.php
[3] M. Pickavet et al., *Worldwide energy needs for ICT: The rise of power-aware networking*, in Proc. 2nd International Symposium on Advanced Networks and Telecommunication Systems (ANTS), 2008.
[4] http://www.fp7-trend.eu/, Towards Real Energy-efficient Network Design
[5] http://www.irit.fr/cost804/, Energy Efficiency in Large Scale Distributed Systems
[6] http://www.greentouch.org/
[7] http://www.greenstarnetwork.com/
[8] B. Puype et al., *Multilayer traffic engineering for energy efficiency*, Photonic Network Communications, vol. 21, no. 2, 127-140, 2011.
[9] B. Nordman et al., *Reducing the Energy Consumption of Networked Devices*, IEEE 802.3 tutorial, 2005
[10] R. Tucker et al., *Evolution of WDM Optical IP Networks: A Cost and Energy Perspective*, Journal of Lightwave Technology, vol. 27, no. 3, 243-251, 2009.
[11] W. Vereecken et al., *Power consumption in telecommunication networks: overview and reduction strategies*, IEEE Communications Magazine, vol. 49, no 6, 62-69, 2011.
[12] B. Niven-Jenkins et al., *MPLS-TP requirements*, RFC 5654, IETF 2009
[13] J. P. Vasseur et al., *Network recovery: protection and restoration of optical, SONET-SDH, IP and MPLS*, Morgan Kaufmann, 2004.
[14] A. R. Sharafat et al., *MPLS-TE and MPLS VPNs with Openflow*, SIGCOMM 2011.
[15] A. Kern et al., *MPLS-Openflow based access/aggregation network*, iPOP 2011.
[16] http://www.emulab.net/, Emulab Network Emulation Testbed
[17] N. Gude et al., *NOX: Towards an Operating System for networks*, SIGCOMM 2008.
[18] http://openvswitch.org/, An Open Virtual Switch.
[19] S. Sharma et al., *Enabling Fast Failure Recovery in Openflow Networks*, DRCN 2011.
[20] http://www.fp7-sparc.eu/, Split Architecture Carrier Grade Networks.
[21] http://www.fp7-ofelia.eu/, OpenFlow in Europe, Linking Infrastructure and Applications.