Wireless Mesh Software Defined Networks (wmSDN)

Andrea Detti, Claudio Pisa, Stefano Salsano, Nicola Blefari-Melazzi
Electronic Engineering Dept.
University of Rome "Tor Vergata"
Italy
{andrea.detti, claudio.pisa, stefano.salsano, blefari}@uniroma2.it

***Abstract*—In this paper we propose to integrate Software Defined Networking (SDN) principles in Wireless Mesh Networks (WMN) formed by OpenFlow switches. The use of a centralized network controller and the ability to setup arbitrary paths for data flows make SDN a handy tool to deploy fine-grained traffic engineering algorithms in WMNs. However, centralized control may be harmful in multi-hop radio networks formed by commodity devices (e.g. Wireless Community Networks), in which node isolation and network fragmentation are not rare events. To exploit the pros and mitigate the cons, our framework uses the traditional OpenFlow centralized controller to engineer the routing of data traffic, while it uses a distributed controller based on OLSR to route: i) OpenFlow control traffic, ii) data traffic, in case of central controller failure. We implemented and tested our Wireless Mesh Software Defined Network (wmSDN) showing its applicability to a traffic engineering use-case, in which the controller logic balances outgoing traffic among the Internet gateways of the mesh. Albeit simple, this use case allows showing a possible usage of SDN that improves user performance with respect to the case of a traditional mesh with IP forwarding and OLSR routing. The wmSDN software toolkit is formed by Open vSwitch, POX controller, OLSR daemon and our own Bash and Python scripts. The tests have been carried out in an emulation environment based on Linux Containers, NS3 and CORE tools.**

Keywords— Software Defined Networking; Wireless Mesh Networks; Community Networks; Traffic engineering

I. INTRODUCTION

A Wireless Mesh Network (WMN) is a multi-hop radio network, whose nodes are IP routers with one or multiple wireless interfaces, typically based on IEEE 802.11 WiFi. WiFi interfaces are usually configured in ad-hoc mode and use omnidirectional antennas. However, the increasing availability of wireless routers with multiple interfaces allows configuring the interfaces in the more reliable infrastructure mode, in which some hub routers are configured as Access Point (AP) and collect traffic from spoke routers that are configured as STAtion (STA). A hub router is interconnected to other parts of the network through another WiFi interface, which may also operate in the same frequency, in case of directional antennas.

Nowadays the greatest mesh networks are the so called Wireless Community Networks (WCNs) (e.g. [1][2][3][4]). WCNs are used to share the cost of Internet access, but also to support the distribution of community information and services. Current wireless community networks may have more than 20.000 nodes [4].

Software Defined Networking and its OpenFlow implementation [5] have been recently proposed for application in Wireless Mesh Networks (WMNs). The work in [6] describes the definition and implementation of a solution for OpenFlow-based routing in WMNs and its applicability to the mobility management of mobile clients. The works in [7] and [8] provide an analysis of opportunities and research challenges arising from the application of SDN in wireless heterogeneous scenarios, including WMNs.

Realizing a mesh of OpenFlow switches, rather than of IP routers, provides the flexibility of implementing packet processing functions, such as forwarding or filtering, which may operate on a multi-protocol base, up to the transport layer headers. This flexibility can foster innovation in mobility management, advanced routing and traffic engineering, and, more in general, in the optimization of the use of the scarce communication resources of WMNs. Moreover, an OpenFlow WMN simplifies network management, since the network control logic runs on a centralized server, which has the task of pushing matching criteria and processing actions to the OpenFlow switches of the network.

In practical terms, switching from current WMNs based on IP routers to WMNs based on OpenFlow switches requires a software update only. Indeed, most WMN nodes are based on Linux OS (e.g. OpenWRT distribution) and OpenFlow tools like the OpenFlow Reference Implementation [11] or Open vSwitch [15] are available for Linux-based systems (including OpenWRT).

To achieve the pros described above we must face some cons and the peculiar characteristics of the WMN environment, such as the unreliability of radio channels that may temporarily prevent the communications with the controller; or the unavailability of layer 2 routing mechanisms such as Spanning Tree or Auto Learning, which are instead commonly used in wired deployment to support communications between switches and controller.

To exploit the pros and mitigate the cons, the contribution of this work is proposing and implementing a Wireless Mesh Software Defined Network (wmSDN) that uses OpenFlow to route data traffic, and exploits the OLSR routing protocol [12] to: i) route OpenFlow control traffic; ii) route data traffic in the emergency case of controller unreachability. The wmSDN software toolkit implementation is composed by Open vSwitch [15], the POX controller [14], OLSRd [13] and our own Bash and Python scripts.

We test our wmSDN architecture in a concrete traffic engineering use-case, for which we devise and implement the logic for the POX controller [14]. The tests are carried out by using an emulation environment based on Linux Containers [16], NS3 and CORE tool [17]. All related software is released as open-source [18].

II. OpenFlow Background

An OpenFlow based SDN is characterized by a standardized programming interface, namely the OpenFlow protocol, which separates forwarding and control functionalities. As shown in Figure 1, an OpenFlow network is formed by *switches* that forward data packets and communicate with one or more *controllers* using the OpenFlow protocol.

A controller configures the forwarding behavior of the switches by setting *rules* in their *flow tables*. A rule is composed of *match* criteria and *actions*. The match criteria are multi-protocol classifiers that identify the set of packets that should be affected by the rule; the actions specify the packet processing to be applied by the switch (e.g. output forwarding port, header rewriting instructions, etc.).

The controller can install rules in a proactive way, or it can react to events coming from the OpenFlow switches and be notified through the OpenFlow protocol. The typical example of the latter case is that of an OpenFlow switch unable to make a forwarding decision, since the incoming packet does not match any rules in the flow table. In this case the switch can be configured to encapsulate the packet in a OpenFlow control packet called *packet-in*, and send it to the controller. The controller may carry out different actions, depending on its *software defined* logic. For instance, the controller may compute the route for this packet, push the related packet forwarding rules down to the requesting switch and send back the encapsulated packet. In turn, the switch will apply the new forwarding rule to the returned packet, as well as to next packets that match the same rule.

Note that in general an OpenFlow switch does not only operate at layer 2 like an Ethernet switch, but it can implement match criteria and actions at different protocol layers, up to the transport layer in the current 1.1.0 version of the OpenFlow specification [11]. This multi-protocol flexibility makes possible to customize an OpenFlow switch to realize traditional network appliances (e.g. an Ethernet switch, an IP router or a NAT/firewall, etc) or a hybrid of them. In our WMN use-case (Section IV), we use OpenFlow switches to carry out IP forwarding on a flow basis and for this reason, in what follows we remind the difference between the processing of an IP packet in a traditional router and in an OpenFlow switch.

When a traditional router receives a packet, its routing table associates the packet destination IP address to the next hop IP address and to an outgoing interface. Assuming an Ethernet like interface, the router resolves the next hop IP address into a MAC address and forwards the packet by rewriting the MAC addresses as follows: the destination MAC address becomes the resolved next hop MAC address; and the source MAC address become the router outgoing interface MAC address.

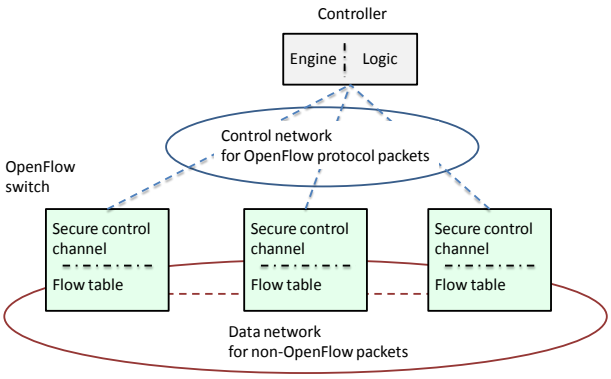

Figure 1. OpenFlow network

In an OpenFlow switch, the IP routing table is replaced by the flow table. For each rule of the table, the match will operate on the IP destination address and mask. The rule actions instruct the switch to forward a packet on the proper outgoing interface, and to rewrite the source and destination MAC addresses to behave as described above. We point out an important difference among the regular IP and the OpenFlow based approach: in the OpenFlow approach, the destination MAC address of the IP next hop must be known when setting the rule (and the IP next hop address is not even used in the rule). In the IP approach, the IP next hop address is indicated in the routing table, and the IP-to-MAC resolution can be performed at packet forwarding time using ARP.

A fundamental requirement of an OpenFlow deployment is that IP connectivity needs to be assured for the communication between the OpenFlow switches and controller, which runs over TCP (or SSL). Using the OpenFlow terminology, this control can happen "in-band", if the same network is used to transfer both data and OpenFlow control traffic, or "out-of-band" if different networks are used.

In a wired layer 2 network controlled with OpenFlow, out-of-band signaling is mostly used. To deploy an out-of-band control a dedicated Virtual LAN (VLAN) may be used to transfer the data units of the OpenFlow protocol through traditional layer-2 switching mechanisms like Spanning Tree algorithm or auto learning of MAC addresses. In addition to the control network, another VLAN is used to create a data network for transferring data traffic, in which forwarding is carried out using OpenFlow mechanisms and routing policies are managed by the OpenFlow controller.

In case of in-band control both data and control traffic are handled by OpenFlow mechanisms. Accordingly, locally configured *control-rules* specify the actions to forward control traffic, i.e. packets going to or coming from the controller. In this wired case, it is possible to use a forwarding action (aka OFPP_NORMAL) that merely enforces the use of traditional layer-2 switching mechanisms. This assumes that the switch is also able to operate with these traditional mechanisms and to support the co-existence of them with OpenFlow. The forwarding of data traffic takes place as previously described for the out-of-band case.

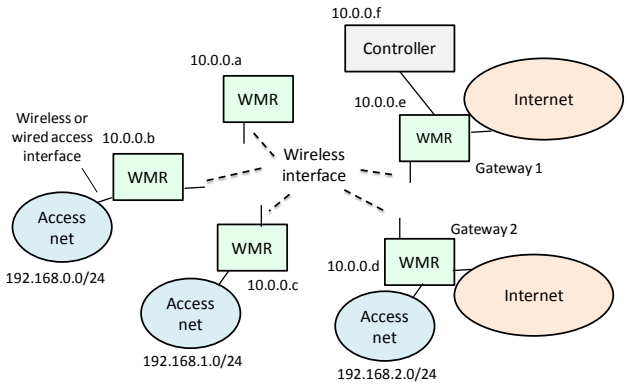

Figure 2. Wireless mesh network

Overall, in wired layer 2 networks both in-band and out-of-band control solutions usually demand of standard layer 2 switching mechanisms to route OpenFlow control messages.

III. WIRELESS MESH SOFTWARE DEFINED NETWORK

The deployment of an OpenFlow based SDN in a WMN environment presents some novel issues, like the setup of a robust control framework that allows the switches both to communicate with the controller, and to face the emergency condition in which the controller is unavailable, e.g. due to a network partition or a controller failure. In fact, VLANs cannot be used to support out-of-band control strategy and the WMN routing mechanism is usually not based on layer 2 mechanisms (Spanning Tree algorithm / auto learning), but on layer 3 routing protocols like OLSR.

A solution to create an OpenFlow wireless mesh with out-of-band control was proposed in [6], using different SSIDs for the control and data network. This work, which to our knowledge is the only that makes practical use of OpenFlow in a wireless mesh scenario, relies on the capability of the wireless driver to support multiple SSIDs. Differently, in this paper we propose an architecture which uses a single SSID, in-band control strategy and also supports controller failures. We use the OpenFlow centralized controller to engineer the routing of data traffic and use OLSR to locally set up the control-rules used by OpenFlow control traffic. Moreover, OLSR is also used to push *emergency-rules* in the switch. Such rules route data traffic in emergency conditions, during which the OpenFlow controller fails or is unreachable.

The reference network scenario is shown in Figure 2. A WMN is composed of Wireless Mesh Routers (WMRs) which provide connectivity to a set of Access networks (either offering a wired or wireless interface to user terminals). A subset of the WMRs operate as Gateways and provide connectivity towards the Internet. This configuration is typical of the current Wireless Community Networks. In our SDN based approach, we add the OpenFlow controller, connected to a WMR through a wireless/wired connection.

Control traffic and data traffic use different IP subnets. For instance, the subnet 10.0.0.0/16 is used for control traffic, while other subnets are used for data traffic.

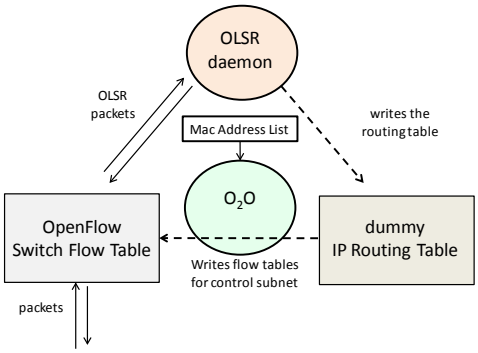

Figure 3. OpenFlow and OLSR interaction

The controller and the WMR wireless interfaces get an address of the control subnet, while other interfaces of the network get an IP address belonging to different subnets, e.g. 192.168.x.0/24, each announced in OLSR as an HNA network.

In-band control network

As we want to deploy an in-band control, we need to locally set-up the control-rules to forward OpenFlow control packets, which are packets with destination IP address belonging to the control-subnet. To this aim we use the OLSR routing protocol to learn the topology of the control-subnet and then exploit this knowledge to setup the control-rules. Accordingly, an OLSR routing instance runs on each WMR node and the IP address of the controller is also advertised by OLSR using a Host and Network Association (HNA) messages with /32 mask.

Figure 3 reports the main entities of a WMR involved in the interplay between OLSR and OpenFlow. The control-rules used by OpenFlow message are configured by the OLSR-to-OpenFlow (O_2O) entity, by inspecting an IP routing table handled by the OLSR daemon. This IP routing table configured is a "dummy" one, i.e. not actually used by the operating system when forwarding IP packets. In the Linux case this is a user defined routing table, different from the kernel main one, and never referenced in the Routing Policy Data Base.

An entry of the dummy routing table has the form <control-subnet IP address/32, next-hop, output interface>; the O_2O module converts it in a rule of the OpenFlow table whose match is "IP destination == control-subnet IP address" and whose action is "change source MAC address with the MAC address of outgoing interface and the destination MAC address with the MAC address of the next-hop". Therefore the O_2O module needs to know the MAC addresses of the WMRs; this IP-to-MAC translation can be provided offline (as in our current preliminary implementation) or can be distributed by a novel OLSR plug-in, so that each WMR can learn the MAC addresses of all other WMRs[1]. To follow topology change, the O_2O sets a timeout (e.g. 60s) to the inserted control-rules and at the timeout expiration the dummy IP table is dumped again on the OpenFlow Table.

[1] Note that while an ARP or ARP-like mechanism could be viable to learn the MAC addresses of one-hop nodes, the central controller however needs to know all the associations between IP addresses and MAC addresses of all network nodes.

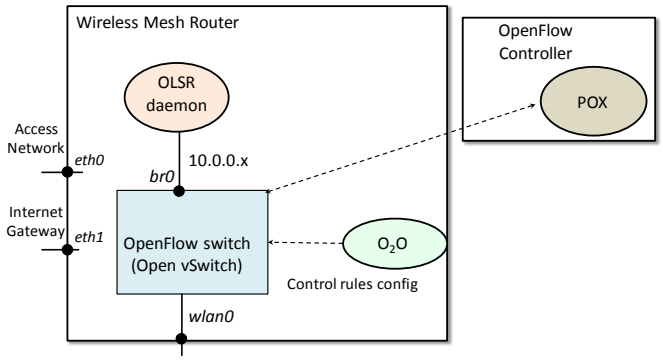

Figure 4. WMR architecture

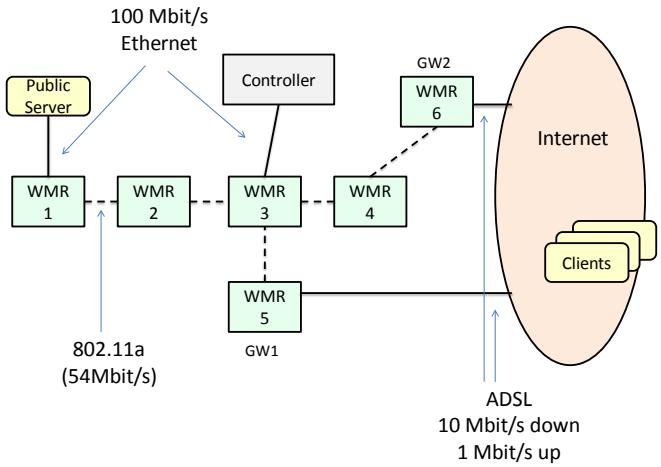

Figure 5. Use-case scenario

In addition to the control-rules used to route OpenFlow traffic, the flow tables are also filled with other control-rules needed to support the OLSR operations. These rules are used to forward the incoming OLSR packets to the OLSR daemon in the WMR node and to let the outgoing OLSR packets exit from the proper interfaces.

Data network

Let us now consider how to handle the traffic for IP destinations outside the control-subnet, i.e. either to the access networks or to the Internet. Assume that a packet is generated in a host of the access network and destined to an Internet address outside the wireless mesh network (but the same will also apply to packets destined to a host of the access network as this occurs when packets come back from the Internet or for mesh internal communications).

The packet will be received by the WMR on its access network interface. Then a match is searched in the flow table. In case a match is found, the related action is carried out. Otherwise, the IP packet is embedded in a OpenFlow packet-in, which is transferred to the controller using the in-band control network. When the controller receives the packet-in, it applies the programmed routing logic, e.g. the one we describe in section IV.

To support controller operations, the IP subnets of the Access Networks are advertised by WMRs and gateway WMRs by using OLSR Host and Network Association (HNA) messages. Moreover, gateway WMRs also advertise the default route 0.0.0.0/0. In doing so, each WMR node knows the full network topology and the controller can inquiry the connected WMR to learn this information, which is fundamental to implement traffic engineering logic for data traffic.

Emergency conditions

Using standard OpenFlow means, the O_2O module periodically controls the liveliness of the controller. In case of controller failure (e.g. due to hardware or communication issue) the O_2O enters in an emergency status during which it removes all the rules inserted by the controller from the flow table and dumps all the OLSR routing table, i.e. including the routes outside the control-subnet and the default route advertised by the gateways.

In doing so the routing of the mesh becomes substantially controlled by OLSR, while the forwarding is always carried out through OpenFlow mechanisms. When the controller becomes reachable, the O_2O leaves the emergency status and removes from the flow table the rules associated to routes outside the control-subnet, thus forcing ongoing data flows to send packet-in data units to the controller, which will decide how to re-route them.

WMR architecture

The architecture of a WMR node in our scenario is shown in Figure 4. It includes: one wireless interface belonging to the Wireless Mesh Network (wlan0); an optional wired interface towards client Access Networks (eth0); an optional wired interface used as a gateway to the Internet (eth1); a virtual interface br0, which is a software bridge using OpenFlow switching logic, e.g. Open vSwitch [15]. A generic "real" WMR node may have additional wireless or wired interfaces towards client Access Network and additional wireless interfaces can be bridged to br0 if a multi-channel WMR is used.

The br0 interface has an IP address belonging to the control-subnet, wlan0 does not have an IP address, eth0 has an address of the Access Networks subnet and eth1 of the subnet connected to the Internet. OLSR is connected to br0, and br0 is used as destination for any packets generated by the node and directed towards the WMR. To this aim we use the trick of inserting in the main routing table of Linux a fake IP address (e.g. 10.0.254.254) as gateway of all the routes whose outgoing interface is br0 (i.e. of the routes directed toward the WMN). To avoid ARP generation, we also statically insert in the ARP table a fake MAC address for the fake IP address.

It is noteworthy that in this architecture we have two different levels of controllers setting up OpenFlow rules: a local distributed controller taking care of control-rules that is the couple O_2O and OLSR, and a remote centralized controller taking care of rules for data traffic.

IV. USE-CASE: GATEWAY BALANCING

In this section we propose an application of the wmSDN architecture in which OpenFlow switching is used to balance traffic among the gateways of a Wireless Mesh Network. The gateway balancing logic is implemented in the controller and we show that external clients can fetch data from public servers inside the wmSDN at a higher throughput, with respect to the one they achieve using a WMN with only plain OLSR routing and IP forwarding.

We consider the case of a WMN that is an Autonomous System and uses its public address space to address internal public servers (e.g. mail, web, video) The WMN also uses a private address space to address private hosts, such as user laptops or desktops. The WMN is connected to the Internet through a set of WMRs acting as gateways, which also have a BGP peering relation with the related access ISPs; i.e. the WMN has a multi-homing configuration. This configuration is rather common in Wireless Community Networks, like Ninux.org in Italy or Guifi.net in Catalonia.

Figure 5 reports an example of such a configuration, in which the WMN is formed by 6 WMRs, two of which are BGP gateways and provide Internet access through an ADSL connection with an uplink bandwidth of 1 Mbit/s. A public server is connected to WMR 1 over Ethernet and clients located on the Internet fetch data from this server.

In such scenarios, using plain OLSR IP routing in the WMN implies that all the traffic sent from a server towards Internet clients flows through the BGP gateway closest to the server. Indeed, each gateway announces through OLSR the default route (0.0.0.0/0) and OLSR inserts this default route in the IP routing tables of WMRs by using a shortest path strategy (or using Expected Transmission Count [13]). For instance, in case of Figure 5 all the traffic between the public server and the Internet clients flows through GW1, which is the gateway closest to the server.

Using a wmSDN in these scenarios makes it possible to carry out forwarding operations on a flow-basis and to route different flows on different gateways in order to better exploit the uplink capacity provided by all the mesh gateways. As a proof of concept, we implemented a simple round robin Gateway Selection Algorithm (GSA) for the OpenFlow controller. The GSA pushes rules in the flow table of WMRs, aimed at routing single data flows towards the selected GW. A rule identifies a flow through a match criterion based on the couple IP source and IP destination. The rule action is twofold: i) to change the source MAC address with the MAC address on the WMR node and the destination MAC address with the one of the next-hop WMR in the path toward the selected GW; ii) to forward the packet on the wireless interface toward the next WMR of the path. The next-hop WMR computation is carried out using the network topology, periodically learned by the controller by contacting the OLSR JSONinfo plugin [13] of the connected WMR (the WMR n.3 in case of Figure 5).

At the start of a new data flow the controller receives OpenFlow *packet-in* messages from the WMRs, since they do not know how to route a new flow. The controller assigns the least recently assigned gateway to this flow, in a round robin fashion. Then the controller pushes a rule in the flow table of the requesting WMR.

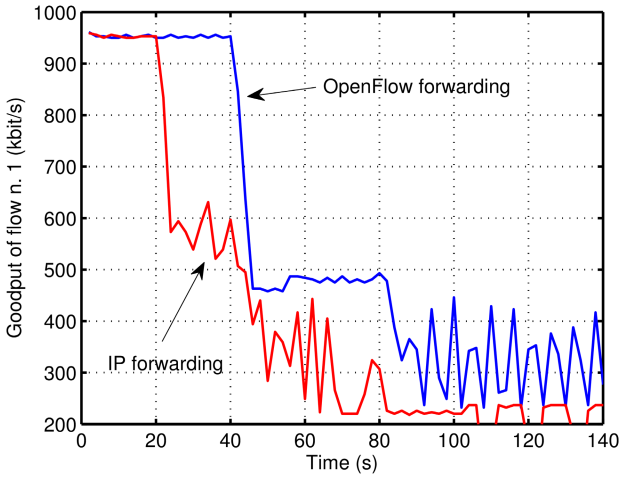

Figure 6. Goodput of a TCP stream from the public server versus time, while changing the total number of stream generated by the server: one new stream at 20,40,60,80,100 s

For instance in case of Figure 5 when the public server with IP address IP_S sends out the first packet P towards an Internet client with IP address IP_A, WMR 1 does not know how to forward this packet and sends P within an OpenFlow *packet-in* message to the controller. The GSA algorithm selects GW1, and pushes an entry in the flow table of WMR 1, whose match condition is IP source address == IP_S and IP destination address == IP_A, and whose action is to switch the source MAC address with the one of WMR 1 and the destination MAC address with the one of WMR 2 (which is the next-hop WMR toward GW1) and then to send out the packets through the wireless interface. The packet P is sent back from the controller to WMR n.1, which uses the new rule to properly forward it. The same procedure is repeated in all WMRs during the travel of this packet towards GW1.

After the initial *flow setup* phase, all the remaining packets of the data flow do not require any other interaction with the controller. When the data flow ends, after a brief timeout an OpenFlow switch automatically removes the related rule to free flow tables from unused entries.

When a second flow towards a client with address IP_B starts, the same flow setup procedure occurs, but the controller assigns the gateway GW2 to the flow, maximizing the use of the overall Internet uplink bandwidth of the mesh. Obviously, in case of gateways with different uplink bandwidth the round robin approach should be properly weighted.

To follow topology changes e.g. due to gateway or WMR failures, every time that a topology change is detected by GSA (through the JSONinfo plugin of OLSR), the controller forces the removal of all the rules it has pushed in the WMRs flow tables. In doing so, all the ongoing data flows will restart the flow setup phase, which will now take into account the changed topology. For example in the scenario of Figure 5, assuming two active flows, one outgoing from GW1 and one outgoing from GW2, in case of GW2 failure, the flow setup

procedures are restarted and both flows will be routed out through GW1.

It is noteworthy that the described procedures are meant neither to be optimal, nor to outperform other similar approaches, like IP source routing or multipath transport protocols. Indeed, the goal of this section is limited to show a simple application of the wmSDN that improves performance compared to a "plain" OLSR+IP WMN. Nonetheless, we argue that even the more complex load balancing algorithm proposed in the literature (e.g. [19][20]) can be implemented in a wmSDN by a proper programming of the controller logic and we leave this for further study.

Moreover, the rough approach of shutting down and setting up again all the rules when a topology change occurs may generate an excessive packet-in traffic or temporary lags in packet delivery. Also this optimization aspect is left for further study.

V. Performance Evaluation

We implemented the deployment framework and the gateway balancing logic through Bash and Python scripts, supported by Open vSwitch [15] and the POX controller [14]. We tested the related performance using an *emulated* environment built with NS3, Linux containers and Core tools [16][17], representing the network of Figure 5. The details of the emulation environment are described in [18]. With respect to Figure 5, in our emulated testbed we simply consider for the Internet side one node with the two "ADSL" links toward GW1 and GW2 and direct Ethernet links to a number (up to 6) of Internet Clients. We do not run BGP and assume that all packets generated by Internet clients access the mesh through GW1. These packets are routed within the mesh on the shortest path towards the server by rules pushed by the OpenFlow controller during the flow setup phase. Packets generated by the mesh public server are routed by the OpenFlow controller on the shortest path towards the gateway selected by the balancing logic.

Gateway Balancing Logic Performance

The first experiment we carried out is aimed to show the performance of the gateway balancing logic, which is also a functional proof of the proposed deployment framework. We consider six Internet clients that receive data through (long-lived) TCP connections, whose source is the public server shown in Figure 5. The first client starts to fetch data at time 0, while the other clients start at 20, 40, 60, 80, 100s respectively.

Figure 6 reports the goodput (useful data rate without TCP/IP header) achieved by the first client if the WMN uses OpenFlow and if the WMN uses plain IP forwarding and OLSR routing. In the 0-20 s time interval the first client is alone. In case of OpenFlow, the controller assigns the client TCP flow to GW1, and thus the achieved goodput is close to the GW1 uplink rate (1Mbit/s) minus the TCP/IP overhead. In case of IP forwarding, OLSR routes this flow, as well as all the other next ones, towards the gateway closest to the server (shortest path strategy), which happens to be GW1 as well. Therefore, in this single-flow case using OpenFlow or IP forwarding does not change the achieved goodput.

During the 20-40s interval there are two flows. In case of OpenFlow, the first flow is kept on GW1, while the second flow is assigned to GW2 by the controller[2]. Consequently, the presence of this second flow does not affect the goodput of the first flow, being the network bottlenecks the gateway uplinks rather than the wireless links. In case of plain IP forwarding both flows are routed to GW1, thus the first flow halves its goodput, since the uplink of GW1 is now shared by two flows.

During the 40-60 s interval there are three flows. In case of OpenFlow, the first and the second flow are kept to GW1 and GW2 respectively, and the third flow is assigned to GW1 by the controller. Consequently, the first and the third flow now share the uplink of GW1 and goodput of GW1 is close to the half of the GW1 uplink. In case of IP forwarding, all flows are routed on GW1, thus goodput of first flow is reduced to about a third of GW1 uplink capacity.

Generalizing the analysis we see that in case of OpenFlow we are able to exploit all the gateways, thus the goodput of the first flow decreases when two new flows are added, being the two gateways assigned to flows in a round robin fashion. Conversely, the plain OLSR+IP solution uses a single gateway and so the goodput of the first flow decreases at the start of each new flow. Moreover, in cases of more than one flow, the overall traffic sent out by the server using SDN is about 2 Mbit/s, while using OLSR+IP it is only 1 Mbps.

Gateway Fault Handling

In the second reported experiment we verify the effectiveness of the gateway balancing logic to handle gateway faults. We consider two TCP flows directed to two Internet clients and whose source is the public server in the WMN. The flows start at time t=0 s, and during the interval 180-360s the gateway GW2 is faulty (WiFi interface off).

Figure 7 reports the goodput of the two flows during the experiment. When flows start, the controller assigns flow n.1 to GW1 and flow n.2 to GW2. Therefore the goodput of the two flows is close to the ADSL uplink rate, i.e. 1 Mbit/s. At 180s, GW2 fails and the flow n.2 has a brief interruption up to the time when the controller detects the topology change using OLSR; in our OLSR configuration this latency is about 10 seconds. After the detection of the topology change, the controller removes all rules injected in the flow tables of the WMRs, forcing a new flow setup phase that ends up in re-routing all flows to GW1. From now on, both flows share the same gateway GW1 and hence the goodput halves. At time t = 360s the GW2 failure ends, after about 10 seconds the controller detects a topology change and finally re-routes flows on two separate gateways. The goodput of the flows become again close to 1 Mbit/s.

Controller Failure Handling

The third experiment we carried out aims to show the effectiveness of O_2O module in handling controller failures. We consider two TCP flows directed to two Internet clients and whose source is the public server. The flows start at time 0,

[2] TCP ACK sent by clients however enters the mesh from GW1. In fact this gateway is chosen by BGP.

and during the period 40-80s the controller has an outage that we simulate by shutting down the related POX process.

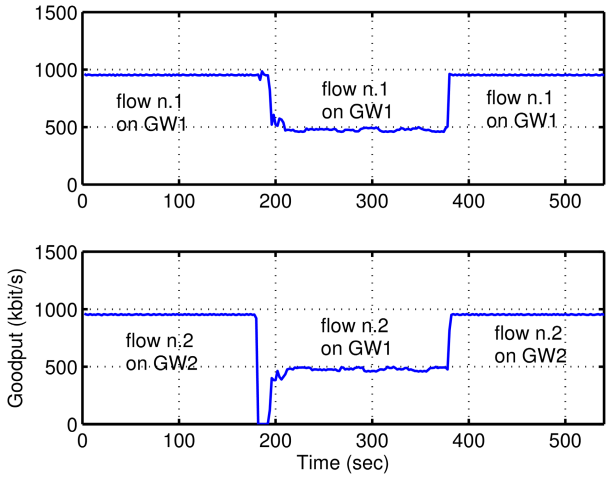

Figure 7. Goodput of two TCP streams vs. time. During the interval 180-360s the gateway GW2 has an outage.

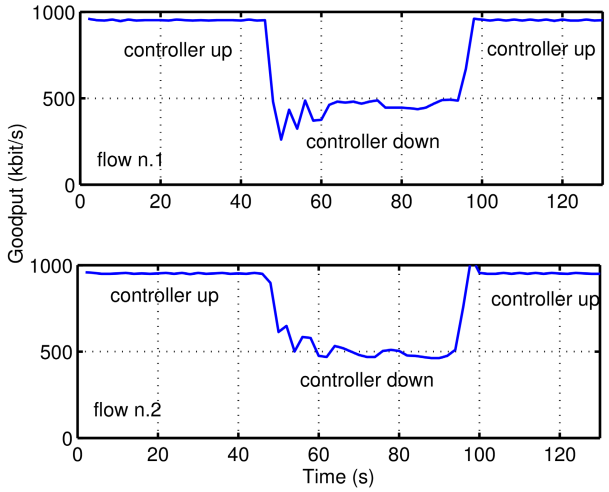

Figure 8. Goodput of two TCP streams generated by a public server of the WMN vs. time. In the interval 40-80s the controller has an outage.

Figure 8 reports the goodput of the two flows. In the interval 0-40 the controller is up, one flow is assigned to GW1 and the other flow is assigned to GW2. During the controller outage period, the O_2O module removes the rules set by controller and dumps the whole set of OLSR routes (including the default 0.0.0.0/0) in the flow table. Thus both flows are sent out through GW1 and the related goodputs halve. It is noteworthy that the delay between the controller failure instant and the actual reaction of O_2O module is about 10 s and this is due to the timeout we configure to check the controller, i.e. 15 s. At t = 80s, the controller outage period ends, after about 10s the O_2O module detects the presence of the controller and then removes from the flow tables the previously inserted OLSR routes that do not concern the control subnet. Consequently a flow setup phase occurs, in which the GSA algorithm in the controller assigns a flow to a gateway and another flow to the another gateway, and performances become again equal to the ones achieved at the start of the test.

VI. CONCLUSIONS

Wireless Mesh Networks may benefit from the flexibility and the simple management provided by the Software Defined Networking paradigm, implemented by OpenFlow. The use of wireless resources can be optimized by a central server, which can reason and perform processing actions on multiple levels of the protocol stack.

We proposed a solution to integrate SDN functionality in a Wireless Mesh, trying to face the reliability concerns related to this environment. The proposed wmSDN approach integrates "ready-to-market" technologies. Indeed it exploits OLSR, Linux based OpenFlow tools and our scripts that could be easily deployed in Linux-based wireless IP routers, typically operating in actual Wireless Mesh (Community) Networks.

REFERENCES

[1] "Freifunk: non commercial open initiative to support free radio networks in the German region," http://www.freifunk.net/.

[2] "Funkfeuer free net," http://www.funkfeuer.at/.

[3] "Ninux.org wireless community network," http://ninux.org/.

[4] "Guifi.net," http://guifi.net/en.

[5] McKeown, N. et al.; OpenFlow: enabling innovation in campus networks, ACM SIGCOMM Comp. Comm. Review, Vol.38, No.2, 2008

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

[7] M. Mendonca, K. Obraczka, and T. Turletti, "The case for software-defined networking in heterogeneous networked environments", ACM conference on CoNEXT student workshop, 2012.

[8] S. Hasan, Y.B. David, R.C. Scott, E. Brewer, S. Shenker, "Enabling Rural Connectivity with SDN", Technical Report No. UCB/EECS-2012-201

[9] J. Ahrenholz, C. Danilov, T. R. Henderson, J. H. Kim, "CORE: A real-time network emulator", IEEE Military Communications Conference, MILCOM 2008.

[10] H. Tokito, M. Sasabe, G. Hasegawa, H. Nakano, "Achieving Load Balancing in Wireless Mesh Networks Through Multiple Gateways", Eight International Conference on Networks, 2009

[11] The OpenFlow Consortium, "OpenFlow website." [Online]. Available: http://www.openflowswitch.org/wp/downloads/

[12] T. Clausen , P. Jacquet , "Optimized Link State Routing Protocol", IETF RFC3626

[13] A. Tonnesen and T. Lopatic and H. Gredler and B. Petrovitsch and A. Kaplan and S.O. Tuecke, "olsrd Website," URL: http://www.olsr.org/

[14] POX controller website: http://www.noxrepo.org/

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

[16] "HOWTO Use Linux Containers to set up virtual networks", http://www.nsnam.org/wiki/index.php/HOWTO_Use_Linux_Containers_to_set_up_virtual_networks

[17] Common Open Research Emulator (CORE) home page, http://cs.itd.nrl.navy.mil/work/core/

[18] wmSDN Web site: http:// netgroup.uniroma2.it/wmSDN

[19] Devu Manikantan Shila, Tricha Anjali, "Load aware traffic engineering for mesh networks", Computer Communications, Volume 31, Issue 7, 9 May 2008, Pages 1460-146

[20] S. Waharte, B. Ishibashi, R. Boutaba, D. Meddour," Interference-aware routing metric for improved load balancing in wireless mesh networks", IEEE ICC 2008