Semester Project - SDN assisted DMM

Monia Chouaibi- Lucas Croixmarie

June 11, 2015

Part I
Project tools

*** TODO matching criterias *** TODO action instructions , priority *** TODO pushing flows vs ponctual orders *** TODO flow gathered in table, table relachionship (general)

Part II
Project Implementation

1 Enhance a simple switch in a real router

Introduction The implementation of the SDN controler, has been written from the code of simple_switch.py provided in the Virtual Machine dirstibuted by SDNhub.com. The initial code is quite limited and allows a switch to handle message (only icmp echo reply and request) forwarding between hosts directely linked to it. Then to improve the code to get a controller able to achieve the previously described DMM solution the first step is to enable the controller with router capabilities which involves making it aware of the underlaying topology, making it handle the icmpv6 control messages received by the switches and then making it order switches to forward packets across the network. Those steps are respectively described below.

1.1 Discovering network topology

1.1.1 Retrieving network backbone's topology

Ryu controller to access the underlying network topology including nodes and the links between them, then it has to be launched with the "–observe-link" option. In order to build data structures where topology information are stored, the controller uses the LLDP messages exchanged between switches when the network is just created. That is why this option allows the controller to be aware of all the switches of the network and all the links between then but it can't retrieve any information about the hosts.

An important point is as we didn't find any way to find out when the discovery procedure was done, (ie detecting the instant when the topology data structures are fully completed by Ryu), our controller waits for the report of the reception of the first IPv6 message by one of the switches of the network to start reading into those data structures and building objects. Indeed we assume that IPv6 messages are exchanged long time after the whole network discovery has been done.

Our Ryu controller embed a function called collectRoutingInfo() that has been created in the purpose of grabbing topology details and information given by mininet obtained during the discovery phase. It is then called once, when the first IPv6 message is submitted by a switch to the controller and uses the topology module of Ryu. Mininet information are collected this way:

```
 #All the topology information are obtained from the app_manager
appManager = app_manager.RyuApp()
#Collecting switches and links information
self.switchList = ryu.topology.api.get_all_switch(appManager)
self.linkList = ryu.topology.api.get_all_link(appManager)
```

Two lists : self.switchList and self.linkList are filled up, with respectively switch and link objects. Those objects embed many attributes that turns out to be useful for the controller in the following parts that is why they are stored this way and not in only keeping their identifier or reference.

1.1.2 Setting up a virtual adressing plan

Mininet may assign MAC and IP address to every node of the constructed network but since we wan't all the configuration decisions to be made by the controller, it will re-define virtually all the IP and MAC addresses of the network. "Virtually" meaning that new given addresses are not written back on switchs interfaces to update the old ones but when the controller asks a node to send a packet it will also specify source and destination addresses the switch has to set on the packet and thoses addresses will be the ones it has defined itself.

Then, once every connection between every switch is registered, the collectRoutingInfo() function defines new IPv6 addresses and uses a dictionnay called bindingList to store what is the new assigned IPv6 address to each interface of each switch (the key is the tuple formed by the switch identifier and the interface identifier among the switch and the value is the assigned IPv6 address). New addresses depend on the identifiers of the switch itself, on the interface number and also on the identifier of the switch on the other side of the link.

Here is the code filling up the bindingList structure from the linkList and the switchList previously built.

```
        for link in self.linkList:
            if (link.src.dpid,link.src.port_no) not in self.bindingList and (lin
                self.bindingList[link.src.dpid,link.src.port_no] = '2000:'+str(l
                self.bindingList[link.dst.dpid,link.dst.port_no] = '2000:'+str(l
```

Mac Addresses are also redifined the same way but as they all are generated the same way, they are not stored anywhere but computed on the fly every time they are needed. Here is the function that construct them:

```
    #return the MAC address associated to DATAPATH_id and port_id
    def generateMAC(self, dpid, portid):
        addMAC = 'a6:0'+str(dpid)+':00:00:00:0'+str(portid)
        return addMAC
```

The way address are forged depend on the interfaces to which they are assigned, indeed interfaces domain can be divided in two partitions, the backbone interfaces and the local network interfaces. The first one corresponds to interfaces in which a link between two switches is pluged, and the second one corresponds to interfaces in which a link between a switch and a host is pluged. Backbone interfaces all share the same two bytes prefix : '2000:' and backbone interfaces connected by a link share the same four bytes prefix : '2000:AB' where A and B are the switch to which interfaces belong (order or A and B depends on the link object from ryu.topology module). Then the last two bytes of the address is defined by the interface number among the switch. For example if we considere the third interface of a switch number 2 through which the switch linked to switch number 5, interface's address is 2000:25::3.

Then this addressing convention introduces a limit of the number of switch that can handle the controller, as the identifier of two switches must fit in two bytes for backbone addresses creation, and since indentifiers are kept in decimal system (not hexadecimal) an identifier can't exceed the value of 99, therefore it is not possible to have more than 99 switches on the network.

Since local network interfaces are not discovered yet by the controler as they are not registered on ryu.topology module's data structure, the controler can't assign them addresses right now.

Just after address assignement another data struture is built, it's called networkGraph, it's a dictionnary binding each switch to its switch neighbor list. For this structure routing algorithm are launched to resolve the one hop path to reach one switch from another one.

Here is a example of addressing plan following the addressing convetions described above:

TODO insert picture of network addressed map

1.2 Handling ICMPv6 configuration messages

Introduction This initialization work described in the previous part is done when the controller is sollicitated for the first time by a switch whih has received an IPv6 packet. Once completed the received packet has to be handled as well as the next incomming ones. Then when the controller is reported of the reception of a IPv6 packet by a switch, it first figures out the type of the packet and after run the apropriate instructions.
Our controller only works with ICMPv6 messages, other kinds of messages are filtered out.

1.2.1 Router Solicitation message

The first type of message of a switch can receive is ICMPv6 Router Solicitation messages, those one are sent by hosts when they get there interface turned on or when they access to a new network.

What the controler does first in this case is checking if the ingress interface is not already registered as a backbone interface, if it is the controller does nothing. Otherwise handling keep going and as now controller is sure that the source is a host, it register its MAC address (obtained from the source address field of the frame containing the Router Solicitation Message) in a data strucure called coveredHosts. It stores hosts that have registered inside the subnetwork of each switch, in other words it stores for each switch the hosts that are suposed to be linked to it. This structure is a dictionnary of dictionnaries : the first level key is the switch identifier and is bound to a dictionnary where keys are IPv6 addresses that the hosts has forged while joining the sub-network and values are the couple host's MAC address and the number of the switch's interface that is linked to the host (to make things clear hear is an example:
dpid1 : host1IP:(host1MAC,intfLocal1),host2IP:(host2MAC,intfLocal2) , dpid2 : host3IP:(host3MAC,intfLocal1) .

An important point is since the host doesn't have any IPv6 address yet, the one it will generate from IPv6-autoconfiguration process is guessed from its MAC address and from the switch sub-domain in which the it is. It is important to have in mind that if the host uses a different way to forge its Global IPv6 address, the controller won't recognize it.

The bindingList is also extended, indeed if the Router Solicitation message is received on an interface nerver used before, as the controler just discovers it, it stores the interface in the coveredList: now its knowledge of the network topology gets extended to local network interfaces and hosts to which they are linked. Then as before an IPv6 address is assigned to this new discovered interface and the controler has also a convention for local network interfaces. The 2 bytes prefix of the address depends on the switch, indeed switches define sub-domain among the network, but the first half-byte of the prefix is always set to 2. Then the last two bytes of the address is like before, defined by the interface number among the switch. For example is the fourth interface of switch number 7 through which the switch is linked to an host, interface's address is 2007::4. If the Router Advertisement reaches an already registerd interface, nothing described happens on the bindingList.

This is just after this step that the mobility management is done, the controler finds out if the host that has sent the Router Solicitation message cames from another sub-network and trigger or not mobility management procedure. For the moment we will skip this part, considering first a controler that make the network behaves normally, without any extra mecanism.

Last, the controller forges the ICMPv6 Router Advertisement to be sent by the solicited switch to the host that just contacted it, it first create the core of the message this way:

```
icmp_v6 = icmpv6.icmpv6(type_=icmpv6.ND_ROUTER_ADVERT,
data=icmpv6.nd_router_advert(ch_l=64, rou_l=4,
options=[icmpv6.nd_option_pi(length=4, pl=64, res1=7, val_l=86400,
pre_l=14400, prefix=prefix)]))
\end{listing}

with the variable prefix set to the switch's␣local␣network␣interface
IPv6␣address␣to␣which␣the␣host␣is␣bound.␣This␣packet␣is␣then
encapsulated␣in␣a␣IPv6␣packet␣(with␣source␣address␣set␣to␣the␣local
scope␣address␣of␣the␣interface,␣generated␣on␣the␣fly␣like␣MAC
addresses)␣and␣in␣a␣ethernet␣frame␣and␣is␣forwarded␣to␣the␣switch.\\
\newline
As␣we␣want␣every␣Router␣Solicitation␣messages␣to␣be␣reported␣by␣the
switches␣to␣the␣controler␣in␣order␣to␣keep␣track␣of␣hosts␣moves␣across
the␣network,␣no␣flow␣handling␣Router␣Solicitations␣messages␣are␣pushed
down␣to␣the␣switch␣but␣only␣a␣ponctual␣order␣asking␣to␣forward␣the
provided␣Router␣Advertisement␣message␣on␣the␣specified␣interface.\\
\newline
Here␣is␣the␣associated␣code␣of␣a␣ponctual␣order␣embedding␣a␣Router
Advertisement␣message␣(under␣pck_generated␣name)␣sent␣by␣the␣controler
to␣the␣switch␣(called␣datapath␣here):

\begin{lstlisting}[frame=single,language=Python]
actions␣=␣[parser.OFPActionOutput(out_port)]
out_ra␣=␣parser.OFPPacketOut(datapath=datapath,
buffer_id=ofproto.OFP_NO_BUFFER,␣in_port=0,␣actions=actions,
data=pkt_generated.data)
datapath.send_msg(out_ra)
```

```
\end{listing}

The switch will execute the given order in forwarding to the host the
Router Advertisement message and will keep reporting any Router
Solicitation messages comming next to the controler.\\

\subsubsection{Neighbor Solicitation message}

A second kind of ICMPv6 message that can be reported by switches to
the controller are ICMPv6 Neighbor Solicitation messages, there are
two reason for an host to send such a message to its local switch. The
first one is in order to resolve the MAC address associated to a given
IPv6 address : the target address. In this case the option field of
the Router Solicitation message is not empty, and the controller
checks if the target address is one of the virtually assigned
addresses the solicited switch's interfaces. If yes the controler
forges the corresponding Neighbor Advertisement message that contains
the IPv6 address of the spotted interface and transmits it back to the
switch along a forwarding order for being relayed to the host, exactly
as for Router Advertisement messages.\\
\newline
As several hosts can be connected to the same switch and then get
configured with the same prefix whereas they are linked through
different interfaces, the controler also resolves inside domain
requests : when a Neighbor Solicitation messages received by a switch
has a target address corresponding to one of another host on the local
network. Here, as every packet between hosts in the sub-network goes
through the switch, the packet containing frame built by the sender
will have its destination MAC address set to the MAC address of the
switch's interface it is linked to.\\
\newline
If the option field of the Neighbor Solicitation message is null that
means that it has been sent by the host for address conflict
resolution purposes, in this case, as address conflicts are not
considered, the controller doesn't do anything : all the host
registration process inside controller data structure is done at
Router Solicitation message reception.\\
\newline
As address conflicts are not handled by the controller, if an host
comes up with a new reconfigured IPv6 address it won't be recognized
by the switch since this address is not obtained from the usual IPv6
autoconfiguration process.\\
\newline
Router Solicitation and Neighbor Solicitation messages are the only
two kinds of ICMPv6 control messages handled by the controller, as the
controler redefines itself the whole backbone addressing plan and as
```

```
address␣conflict␣is␣not␣managed␣there␣is␣no␣need␣to␣care␣about␣ICMPv6
Router␣Advertisement␣and␣Neighbor␣Advertisement␣Messages.

\subsection{ICMPv6␣Echo␣request␣\&␣reply}

The␣last␣kind␣of␣message␣we␣want␣to␣be␣handled␣by␣the␣controller␣are
ICMPv6␣Echo␣messages,␣they␣are␣representing␣data␣packets␣in␣the
simulations.␣\\
When␣the␣reception␣of␣on␣ICMPv6␣Echo␣packet␣is␣reported
by␣a␣switch␣to␣the␣controller,␣the␣controller␣first␣looks␣at␣packet's
destination address and behaves according to it.

\subsubsection{Answering to Echo messages}
Once the controller gets packet's␣destination␣address␣it␣checks␣if
this␣address␣belongs␣to␣one␣interface␣of␣the␣solicited␣switch␣using
the␣bindingList,␣from␣which␣it␣retrieves␣switch's interfaces this way:

\begin{lstlisting}[frame=single,language=Python]
localAddressesList = [ self.bindingList[localPort] for localPort in self.binding
```

If the destination address is indeed one belonging to the switch, there is two possible scenarios : if the message is an Echo Reply, nothing has to be sent back the the network and the controller doesn't do anything. If the message is an Echo Request, that means that someone is pinging the switch and it has to reply.

Exactely as when the controller was ordering the switch to send a Router Advertisement or a Neighbor Advertisement message, it first constructs the ICMPv6 Echo Reply message and the encapsulating packets, and pushes it to the switch along with a punctual forwarding order toward the interface the Echo Request was coming from.

Here we choosed not to push flow to the switch even if it can bother the controller because as ICMPv6 Echo Reply message is constructed from the associated Echo Request message it would have been necessary to push flows specific to each Echo Request's destination address that have their specific actions. Therefore flow table could have been over populated with Echo message related flows which are not the interesting ones.

1.2.2 Forwarding Echo messages

When the destination address of the Echo packet is not of of the solicited switch's address, that means the switch is one intermediate node on the Echo message's path and have to forward it toward its destination, regardless if its an Echo request or an Echo reply.

Then the controller figures out what is the local network to which the destination address belongs to. To do so it extracts from the address the number contained on its first two bytes, from it it get the identifier of the switch the destination should be linked to unless if it is a backbone interface that is aimed and the result is null, in this case it extracts the last byte of the address where is written switch's identifier.

Then two cases are possible, either the destination node is an host direcly linked to the solicited switch (in this case the extracted identifier is the one of the switch itself) and here the controller checks if there is an host registered in the coveredHosts list of the switch that own Echo message's destination address. If no host is found is the list the packet is dropped, but if one is found the controller feches host's details from the coveredHosts list that are : host's MAC address and the switch's interface to which it's linked to. With all of this the controller has everything he needs to relay the echo message toward its destination.

The other case is that the extracted network identifier is not the one of the switch itself meaning that another switch or an host located in a remote local network is aimed by the message. Here the controller finds out next hop switch toward the destination : for this purpose is uses the structure called networkGraph constructed during the initialisation phase and runs on it the breadth-first algorithm to get the shortest path beteween the solicited switch and the destination switch. From the path, the controller learns which switch is the next one to reach the final one. The last step consist in, given a switch and one of its direct neighbour, finding the interface of the switch that is linked to the neighbor, the function routing() has been written for this purpose, the idea is a to scans all the links of the network untill finding the one linking the two specified switches and look on which interface the link is plugged on the switch, here is the code:

```
def routing(self, source, dest):
    for l in self.linkList:
        if l.src.dpid==source and l.dst.dpid==dest:
            return l.src.port_no
```

Since this function uses the linkList structure it only works with interfaces linking one switch to another, that is why interfaces linked to hosts are stored in the coveredHost structure.

When we reach this point in every cases we have resolved the switch's interface on which the Echo message has to be forwarded. And as the message has to be encapsulated in a new MAC frame, new source MAC address and destination MAC address must be set. The source MAC address depends on the switch's interface and is computed on the fly as previously said, and the destination MAC address is fetched from the coveredHost structure if the next hop is an host and if the next hop is a switch the routing function is called on this switch in a symetrical way to resolve identifiers of the other side interface on the link, and they its MAC address can be generated.

When we reach this point, in every cases all the element needed for forwarding has been resolved (MAC addresses and Output interfaces) and here for the first time a flow is pushed to the switch from the controler instead of punctual order. This flow consists in matching every next Echo message reaching the switch with the same destination address and to forward them toward the interface that has just been resolved in changing MAC address with the ones provided, here is the associated code:

```
action =
[parser.OFPActionDecNwTtl(),parser.OFPActionSetField(eth_src=new_mac_src),
parser.OFPActionSetField(eth_dst=new_mac_dst),parser.OFPActionOutput(outputIntf)
]
match = parser.OFPMatch( eth_type=0x86dd, ip_proto=58, ipv6_dst=(ping_dst,'ffff:
self.add_flow(datapath, 1, match, action,tblId=1)
```

Now the switch knows how to handle alone this specific kind of message and won't forward them anymore to the controller. This is how switches get progressively autonomous, by getting instructed at the receiption each new Echo message to forward.

Now the controller is able to handle switches over a normal network that is not requiring any extra services, but our purpose is to handle host mobility over the network, then we can imagine that other flows will be pushed down to switches so it is necessary to organise flows properly into switches order to avoid any conflict between flows of different purposes.

2 Handle host mobility across the network

2.1 Introduction

2.1.1 Flow organisation inside switches

Once pushed to a switch, flows are grouped into ordered tables that can be bound together, our controller defines 3 tables insides switches : the first one (table number 0) and the last one (table number 2) are dedicated to flows related to mobility handling and their purpose will be explained later, for the moment the only thing to know about them is that the default entry policy of table number 0 is to forward message to the table number 1.

The table number 1 (the second one) is dedicated to flows related to classic message forwarding, like the flows pushed for relaying ICMPv6 Echo messages. At this point tables 0 and 2 are empty and the second one get progressivelly populated witch flow for Echo messages forwarding. For each switch, when a packet is received, it checks if it matches one of the entries of the first table, if not it checks if it matches one of the entries of the second table. If no match is found after having scanned the second table, the switch doesn't know how to handle it and asks the controller what to do with it (it's asking for an order or a new flow), that is what happens when a Echo message with a new destination reaches the switch.

2.1.2 Basic idea of how mobility management is done

Host mobility is ensured first in keeping track of them all around the network, by storing and updtating the list of sub-networks each node has visited. So that when a host gets to a new network, all the old ones registered on the list are retrieved and involved in the mobility management procedure.

2.2 Retrieving Mobile Host history and enable local forwarding

2.2.1 Retrieving Mobile Host history

When a Router Solicitation is submited to the controller, after it has updated coveredHosts and bindinList data structures, it refers to the module called mobilityPackage to know if the host has visited sub-network before connecting to the solicited switch. This module is very simple consists in one dictionnary called trackingDict which store the network history for each host and provides a unique function called getTraceAndUpdate(), that returns the list of previously visited networks by a specific host identified by its MAC address and appends to this list the identifier of the switch which notifies the Router Solicitation message to the controller.

The controller next build a tunnel between each previously visited returned by the module and the solicited switch which is now the one to which the mobile host is linked, and therefore the one to which all the mobile host's pending communications has to be re-routed.

2.2.2 Enabling local forwarding of remote addresses

When a host gets connected to a switch, the controller computes the Global IPv6 address the host will generate and stores it into the coveredHost data structure so that when a ICMPv6 Echo message is aiming this particular address the controller can resolve the interface to witch the host is linked and the Echo message can be forwarded.

When now the host has visited other networks before connecting to a new switch, by setting up tunnels the controller makes all the active communications in which the host is involved, going through the new switch. Then, in this case the new switch receives packets coming from everywhere in the network and has to forward them on the interface the mobile host is now linked to. Since Those communications have been started by the host other sub-netoworks, they are made of packets for which the host address is a previously generated one and the new switch doesn't have any ideas of them. Therefore to be able to achieve correctly the local forwarding, the switch has find out which are those previous addresses used by the host so that output interface can be resolved.

A simple solution to this issue consists in first resolving the previously IPv6 addresses used by the host and then pushing new flows from the controller to the new switch's table number 1 (the table for routing purposes) that would match packets whose destination address is the ones just resolved and forward them on the interface where the host registered. The probleme now is that some packets that doesn't come from any tunnel may be forwarded to one of the new switch local interface instead of being routed normaly tho the switch associated to their destination address. Therefore those new flows would mess up the normal routing procedure and we wan't to keep this very particualer local forwarding of remote addresses only for packets comming from tunnels and keep other packets out of it.

That is why a third flow table is set up inside switches, it contains all the flows related to local forwarding of remote address and act as described just before. The important point is that flow table can only be accessed by messages comming from a tunnel. As host's previous network are know as well as host MAC address, the controller computes the global IPv6 addresses the host has generated when it was in those networks, and then pushes flows that match packets having those addresses as destination address and forward them to the new switch's interface on which the Router Soliciation message has been received.

2.3 Setting up tunnels

2.3.1 General scheme

Tunnels are established between the switch just joined by the Mobile Host and the ones it was connected to before. In this way all the messages aiming an address that the host has forged in a old sub-network reach the covering router that forwards them through the tunnel just set up to the new switch that extract them out of the tunnel and relays them to the host.

In the reverse direction, when the host sends a message with a old IP address as source address, this message is tunneled to the switch covering the sub-newtork where this old address has been built (no route optimization mecanisms are set up). This old switch then extract packets out of the tunnel and forwards them toward their final destination.

2.3.2 Tunnel properties

Tunnels are materialized with Vlan tags, as it only deals with the layer 2 of switches' stacks, the handling is lighter and faster for them.

Moreover for a given direction, only one tunnel exists between two switches and it is shared between hosts, this makes the number of flows to push for mibility purposes lower, indeed, the first host that goes from a network A to a netork B will trigger the establishment of a tunnel between the associated switches and every next host that do the same crossing from A to B will have its message going conveyed through this same tunnel.

Tunnels are unidirectionals on the way hosts move, in the sense that they convey messages (in both directions) to ensure host mobility from a network A to another newtork B but if the host goes back to A from B another tunnel will be used.

2.3.3 Tunnel related flows

A tunnel between a previously visited switch A and the currently visited switch B is set up by the controller first in pushing two flows to both switches A and B. This time flows are related to host mobility, they are then store in the first table (table 0) of each switch.

Two flows are pushed to the first table of switch A:

The first one matches packets coming from the network whose destination address is the one that the Mobile Host forged when it was in the sub-network of switch A (let's call this address "host's old address"), and its action consists in pushing a VLAN tag on those packets, and forwarding them toward router B, in changing MAC addresses. Here the VLAN tag is the concatenation of switch A and switch B's identifiers (let call this value "V"). To avoid any undesirable tunnel binding effect the flow matches only packets without VLAN tag.

The second flow matches packets whose source address is the host's old address and which are including a VLAN tag set to V. The associated action consists in first getting rid of the VLAN tag and then in relaying the new packet to the flow table number 1 of the switch so that it will be passed through the table like a normal packet from the local network and be forwarded as usual to the external network.

Two other flows are pushed to the first table of switch B:

The first one matches all the received packet whose source address is the host's old address. The associated action is to push a VLAN tag with the value V and then to forward packets toward router A, in changing MAC addresses. As IPv6 addresses are unique there is no risk that this flow matches packets received on a backbone interface of switch B and forwards them into the tunnel because the mobile Host is the only entity of the network that sends packets with this precise source address. Here again, to avoid any undesirable tunnel binding effect the flow match only packet without VLAN tag.

The second flow matches packets from router A that include a VLAN tag set to V, then the associated action consists in first stripping the VLAN tag out of packets. Second, as those packets have their destination addresse set to the host's old address they are then relayed to the flow table number 2 of switch B so that the local output interface will be resolved based on the destination address of the packets.

Now that flows are pushed to the entry point and to the exit point of the tunnel, the controller has now to tell switches of the network this tunnel is crossing to relay packets going through the tunnel. In applying the breadth-first algorithm over the networkGraph data structure, the controller gets the list of the intermediate switches to instruct, and push to each of them two flows :

The first one matches packets that include a VLAN tag set to V and whose source adress is host's old address, the associated action is forwarding them in keeping them unchanged (except their MAC addresses) to the next hop on the path to reach B.

The second one matches packets going in the reverse direction, those ones include a VLAN tag set to V and their destination adress is host's old address, the associated action is forwarding them in keeping them unchanged (except their MAC addresses) to the next hop on the path to reach A.

With this set of flow host mobility is ensured all over the network but there is currently one configuration that makes the program not working : when the tunnel entry point has to forward encapsulated packet on the same interface on which it received them just before, it doesn't forward anything out on the interface and nothing is send inside the tunnel. This problem is not fixed today!

2.4 Advanced mobility

It's important to keep in mind that the mobile host may not only go from one network to another but may roam across many different ones and also go back to previously visited networks. Therefore the tunnel establishement algorithm described before is a trade between having a simple sequence of operations to be done by the controller and try not to make switches flow table soaring after host have roamed for a while, that is why shared tunnel solution has be selected.

2.4.1 Subsequent Handover

When the mobile host after having left its home network A to go to network B, changes again of network and goes to network C. There are now two address for which mobility have to be ensured : the one acquired in network A and the one acquired in B, that means that two tunnels have to be set up : one between switch A and switch C and another between switch B and switch C, moreover the previously tunnel from A to B must not be used anymore.

Once installed into a switch a flow can be updated when a new flow with the same matching criterias is pushed to this switch, this is what happens when the host gets to network C. Before the host joins switch C, two tunnel flows are installed into switch A : one ensures that every packet aiming the mobile host's old address is forwarded in a vlan tunnel toward B, let's call this flow FA1. The other one ensures that every packets going from the vlan tunnel is piped to the routing table, let's call it FA2.

When the mobile node reaches network C, a new vlan tunnel is set up between switches A and C, FA1 is then updated because a new flow matching every packets aiming mobile host's old address forged in network A is pushed, and this new flow makes switch A forwards them into the new vlan-tunnel toward C, from now switch A doesn't forward anything more packets related to the considered mobile host in the tunnel it shares with swith B (but it still can forward into it packets related to other mobile hosts). The second new pushed flow matches packets based on a new vlan tag, then it doesn't update FA2 as tunnels between A and B and between A and C use different tags.

Then switch A has now 3 flows in its flow table number 0 : two of them handle host mobility toward network C and the last one is now useless for the considered host but still important to handle mobility of other mobile nodes that have moved from network A to network B.

The two new flows pushed to switch B when the mobile node gets in network C are exactely analog to the ones pushed to switch A when the host moved from network A to network B, but they are associated with the new vlan tunnel between switch B and switch C. One of the two already existing flows related to the vlan tunnel established with switch A, was in charge of forwarding packets caming from the tunnel to switch B's flow table number 2, let's call it FB1. The other one was matching packets with the mobile host's old address as source address and was sending them into the tunnel, let's call it FB2. As the mobile node is not anymore in network B, FB2 becomes completely useless as the mobile host is no more emitting directly to switch B, but FB1 is still used for other mobile nodes that have moved from network A to network B.

Two pairs of flow are then pushed to switch C they are analog to the pair pushed to switch B when the mobile node reached network B from network A, but one pair is related to the tunnel between switch A and switch C and the other to the tunnel between switch B and switch C.

2.4.2 Subsequent Handover Complexity

In this scenario of subsequent handover, when the node gets to network C, 8 flows are pushed by the controller, and every time a mobile node moves to a new network, n time 4 flows will be pushed with n the number of visited networks. Indeed the fact of having simple flow pushing algorithm makes the number of OpenFlow messages quite important. However, our method doesn't present a great space complexity regading to switches flow tables, and especially for the first flow table. Indeed as tunnel are shared, among the four flows pushed during the first handover between network A and network B, one (FA1) is updated, two are still usefull for other mobile hosts (FA2 and FB2), and only one becomes unused (FB1) untill the mobile node goes back to network B.

2.4.3 Back to a visited network

If the mobile host, after having visited network C, keeps roaming and goes back to network B, the mobility of the the address acquired in network A and of the one acquired in network C have to be ensured, moreover packets going to the address that the mobile node has forged in network B doesn't have to be transfered in a tunnel anymore.

First, two flows are pushed to switch A and two others are pushed to switch B and as they are exactly the same as the one pushed when the host moved first from network A to network B (the vlan tag is still the same), there won't have new flows in switch A and switch B's flow table but the flow FA1 will be once again updated and incomming packets aiming the host's old address from network A will be forwarded to the tunnel toward B.

Two other pairs of flow are pushed to switch C and switch B again, but as we said tunnel are unidirectionnal in the sense that one tunnel ensure mobility between two switch for a given direction, then two more entries are written in both switch B and switch C's flow table.

Packets going to the address that the mobile node forged into newtork B when it got there for the first time were matched by a flow entry that sent them into the tunnel between switch B and switch C. Now this flow entry is updated by the controller that pushes a new flow to B with the same matching criterias and that forwards packets on the second flow table of switch B, so that packets going to this specific address are forwarded normally by B toward the local interface the mobile Host just connects to.

When the mobile nodes goes back to a previously visited network, old flow entries are used again, and then flow table size doesn't become very high. As each mobile node is associated to the list of the networks that he visited, if it goes back to previous networks, several networks can occur multiple time on the list, then in order to avoid to push muliple times flows related to the same tunnel during the same handover procedure, the controller keeps in memory wich tunnel it has already updated in order not to send new flows an already updated tunnel.

3 Observations and results

Introduction This part is following the steps of what is supposed to be presented during the final presentation, its role is to illustrate and make clearer the concepts presented in the previous section.

3.1 Network topology and simple ping

3.1.1 Topology

Let's considere a network composed by five switches and six hosts, organised according to the following scheme : **** TODO INSERT NETWORK PLAN AND PROVIDE CODE IN APPENDIX

A mininet script has been written to reproduce this topology, to make things clearer the script assign to each switch address that will be virtally generated by the controller. It also make the default route configuration on hosts easier.

Once both mininet and the controller are launched, after few seconds hosts get configured with global IPv6 addresses, here is a view of h10's interface configuration: **** TODO insert picture h10$_{a}utoconfiguration.png$

3.1.2 Simple Ping

To enable hosts to send messages, thay have to be given a default route, here the local router is the default route.

From now hosts are able to ping each other, the first ping messages won't be conveyed to their destination as flows are getting pushed to switches but once they received all the information from the controller, messages are well relayed. Here is an example with h10 pinging h31's IPv6 address :

**** TODO insert picture h10$_{p}ing_{h}31.png$

The first message of this series of ping has triggered flow pushing to the second flow table of switches on the path from s10 and to s31, at the begining those tables were empty and now they get populated with the occurence of new ping messages, here is the content of the flow tables of s10. **** TODO insert s10$_{d}umpflows.png$

At this moment the two other tables are still empty.

3.1.3 Simulating one hop mobility

As making hosts move from one router to another with mininet looks possible to implement in a python script, but not with command line instruction. The idea to overcome this issue is to use IP and MAC spoofing inside the network. Indeed let's configure h50 with the same addresses as h31 while h31 is turned off, as h50 presents h31 identifiers the controller will treat it as if it was h31. Here are the spoofing instructions:

**** TODO insert h50$_spoofs_h31.png$

Now, if h2 pings again h1's address, ping messages are still well exchanged but now the ttl of the ping response is equal to 59 whearas it was equal to 61 before, that means that there is two more hops now on the path from h10 to h31's address. With a packet sniffer it is possible to see ping messages going from s1 to s3 and then being relayed in VLAN tagged packet to s5, h31's address mobility is then provided. **** TODO insert picture h10$_ping_h31spoofed.png$

Ping messages are now received and treated by h50 that now plays the role of h31 as we can see from a packet capture on h50's interface:

**** TODO insert picture h50$_tcpdump.png$

Flow tables have been updated, the first flow table of s3 is now containing two flows that transfer packets going to h31's address in the tunnel toward s5. The first and third flow table of s5 have also been populated as we can see:

**** TODO insert s5$_dumpflows.png$

3.1.4 Simulating advanced mobility

Let's now turn h50 off and make h40 impersonate h31 exactly as the same way we did before with h50, the controller will then believe that h31 has now moved from s5 coverage to s4 coverage. Then ping messages go now through a new tunnel between s3 and s4, and second tunnel is set up between s5 and s4, we can retrieve them with the dump of s4 flow table:

**** TODO insert s4$_dumpflows.png$

When the mobile node moves back under s3 coverage after having visited s4 network, flow tables are updates and ping messages are now routed again to s3 and s3 now forwards packets going to h1'address not anymore on a tunnel but on its local interface where is now plugged h31.

Part III

Futur Enhancements and Conclusion

4 Enhancements

4.1 Controller algorithms

4.1.1 Having less flow to push

We already said that each time a node moves to a new network after having visited n networks, 4 time n flows has to be pushed down by the controller. Then after a while it can turns out to be lot of flow to send for the controller. In order to limit this number a new way to handle mobility would be only to set up a tunnel between the switch of the just network left by the host and the one of the newtork just reached, then mobility would be ensured with this series of tunnel bound one after the other one among which switches would forward packets going to the address the mobile node has forged under their coverage but also packets going to the address the mobile has forged in the network visited before : comming from the serie of tunnel.

4.1.2 Handling the first packets of flow

As routing flows are pushed reactively the first packets of a serie that triggers a flow pushing are lost. This can be avoided in implementing a buffering mechanism inside the controller or in making it tell switches to forward those packets to their destination while flows are being set up.

4.1.3 Handling other types than icmpv6

Flows pushed to both the first or the second flow table of each switch match ipv6 ping messages packet, this has to be changed in the future to allow other types of message to be treated. The question then is gather all the network traffic type in one general matching flows or assign specific flows for each supported protocol.

4.1.4 Handle address confict within the same sub network

TODO : with other host, we suppose that the node compute it global ip@ the same way as the controller

4.1.5 Introducing access control to mobility service

As mobility management is presented as a service it would be nice to control which user can use it. Then the implementation of a policy decision an enforcement entity could be done which would be consulted when a new user shows up in the network. The authentication can be first based on the mac address, and then on more advanced criterias.

4.2 Interaction with mininet

4.2.1 Make hosts move for real

Yet a way to make host moves from one switch to antother within the mininet virtual network hasn't been found, that is why our way was to trick the SDN controller with addresses spoofing. But as hosts doesn't properly move in our simulation we do not really know how the system really reacts and may be the messages exchanged between the mobile node and the switch are not exactelly the same. It appears that allocating several local network interfaces on switches may help but this involve changes of the controller behaviour as described before.

4.2.2 From comand line to a batch program

Our demonstration has been done in typing one by one all the mininet instructions that turns out to be quite the same, it would make the interaction with mininet easier and faster especially during the test phases to load once an instruction file instead of writing them all every single time.

4.3 fixing the bug

5 Conclusion

5.1 Status and scope of the program

what is it doing? limitation? why is it limited?

5.2 Context, how can it be used in real life

5.3 Personal impressions