Intent Based Application

Group Project Report for Advanced Network Architectures And Wireless Systems

Francesco Iemma
Yuri Mazzuoli
Giovanni Menghini

M.Sc. in Computer Engineering

Academic Year 2021/22

Contents

Introduction

This project is aimed to design and develop an intent based application. The scenario is the following: *"Consider a set of clients that can communicate through a redundant network (e.g., based on a spine-leaf topology.) An external application can request to install paths between host pairs, by just specifying the endpoint identifiers, we refer to this as an host-2-host intent. The network has to allow communications only among the specified host pairs. Moreover, the network has to automatically reconfigure in case of link failures. Design and realize a system that allows users to request and withdraw host-to-host intents, and configures the underlying network accordingly."*
The objectives of the projects are:

1. *"Implement a Floodlight module that exposes a RESTful interface allowing clients to create/delete host-to-host instents. The module will then dynamically install and update flow rules in the network to allow the communication among specified pairs. Possible path switches must occur transparently to clients."*

2. *"Test the overall system using Mininet and Floodlight, devising proper scenarios to demonstrate the above functionalities."*

Chapter 1

Implementation

The objective of the intent based application is mainly to expose a REST interface to allow the request of an **intent**. An **intent** is a request, done by an host, to have a link with another host, this link must tolerate link failures. Hence the controller must implement a module that is in charge of:

- Computing the best path between hostA and hostB
- Installing in the switches of the network the proper rules to establish this path
- Being responsive in case of a link failures and establishing a new path to maintain the link alive

In the implementation of the application we have re-used some Floodlight modules extending some classes where needed. In the following sections we will analyze the choices done and the classes extended in order to implement the features shown previously.

1.1 Floodlight Forwarding

The forwarding in Floodlight is implemented providing the abstract class ForwardingBase[1] which is in charge of providing a base class for implementing a forwarding module. The forwarding module "is responsible for programming flows to a switch in response to a policy decision". The implementation of the abstract class must implement the following abstract method:

```
    public abstract Command processPacketInMessage(IOFSwitch sw, OFPacketIn pi,
    IRoutingDecision decision, FloodlightContext cntx);
```

The Floodlight standard implementation of this abstract class is the class Forwarding[2].

1.2 Floodlight Extended Forwarding Module

The main module of our package is IntentForwarding. This module is an extension of the floodlight forwarding module[3] and we will call it also extended forwarding module. In short the added functionalities are:

- An array list that remembers the list of active intents (intents database).
- The extension of the method *ProcessPacketInMessage* in order to check if the packet that triggered the packetIn is from two host that are allowed to communicate (an intent between the two exists) or not.

[1] net.floodlightcontroller.routing.ForwardingBase
[2] net.floodlightcontroller.forwarding.Forwarding
[3] net.floodlightcontroller.forwarding.Forwarding

1.2.1 Intent Life Cycle

An intent is created, is active and then, when the time indicated in the intent establishment request finishes, expires.
When an intent is created the following operations are done:

- It is checked if an intent with the same IPs already exists. If yes no other operations are performed.
- A new timeout task is created and scheduled in order to be executed in the amount of time indicated by the intent timeout
- The intent is added to the array list

Notice that the creation of the timer is needed in order to implement the intent expiration and to perform some operations when this event takes place.
The operations that must be done when the intent expires are:

- Install rules in certain switches in order to deny the communications between the hosts of the expired intent
- Delete the intent from the intent database

Thus the main problem is to decide which are the switches where we have to install the rules.
Let assume that we have two host: *host1* and *host2* and an intent between them exists and it is going to expire. In order to prevent the two hosts to communicate after the expiry and in order to minimize the remaining packets in the network from hosts of an expired intent, the rules are installed in the first switch encountered by the *host1* when sends the message to *host2* (we will call this switch *switch1*) and in the first switch encountered when *host2* sends a message to *host1* (we will call this switch *switch2*). In practice we install the rules in the access switch that allow an host to communicate with the rest of the network.
The rules installed on each of the two switches are:

- Deny IPv4 from *host1* to *host2*
- Deny IPv4 from *host2* to *host1*

Both rules are valid for 5 seconds by default. See figure 1.1 for a graphical representation.

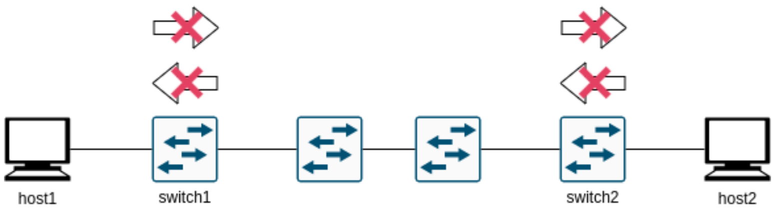

Figure 1.1: Graphical representation of where the rules for denying the communications between two hosts are installed

Thanks to this rules no other packets are allowed to pass from *switch1* and *switch2*. It is possible that some packets are in the network because they have passed the access switches before the intent expiry, in that case there are no problems because they are discarded in any case when they arrive in one of the two access switches. Those packets are a few (possibly 0) and so they doesn't affect the performance heavily, the alternative would be installing rules on each switch of the network in order to discard immediately these packets but this would be more costly.
When the rules are no longer valid for each packet arriving to *switch1* or *switch2* a packetIn is generated and thus the *processPacketInMessage* method will check if that packet is allowed (an intent has been re-established) or not (the old intent has expired and no other intent have been established).
Notice that due to the fact that the rules installed in the switches are valid for a certain time, no intent can be established until that rules are no longer valid. In the default case the rules are valid for 5 seconds and so a new intent regarding one of the two hosts can be established 5

seconds after the expiry of the old intent. That time can be changed by the network manager through REST.
The last problems that need a solution is how we can know which is the identifier of *switch1* and *switch2*. When the timeout task (the task that will be performed when the intent expires) is created we pass to the constructor:

- A reference to the object IntentForwarding, this is necessary in order to call the function who is in charge to install rules in the switches
- A reference to the object representing the intent associated to that timeout task. That object is **HostPair** and it is composed by:
  - IPv4Address host1IP
  - IOFSwitch sw1
  - IPv4Address host2IP
  - IOFSwitch sw2
  - long timeout

  That reference is necessary in order to retrieve the identifier of *switch1* and *switch2*. That identifiers are set when the first packetIn coming from the first switch encountered by the packet sent by an host is handled by the extended forwarding module i.e. IntentForwarding (for more on this see 1.2.2).

Here we can see the code executed when we add an intent.

```
public boolean addNewIntent(HostPair newPair) {
  System.out.print("AddNewIntent Called\n");
  if(intentsDB.contains(newPair)) {
    log.info(" Intent already present in Intents List");
    return false;
  }

  long timeout = newPair.getTimeout();
  Timer timer = new Timer();
  TimerTask task = new TimeoutTask(newPair, this);
  timer.schedule(task, timeout);

  intentsDB.add(newPair);
  return true;
}
```

1.2.2 IPv4 Management

When a packetIn regarding an IPv4 packet arrives the operations done are:

- The intents database is searched:
  - If exists an intent between the sender and the receiver (or viceversa) then
    * If the sender is *host1* and *switch1* (in the object HostPair representing the intent) is null then *switch1* is set equal to the identifier of the switch that has sent the packetIn that has triggered this execution of *processPacketInMessage*. Same thing is done for *host2* and *switch2*.
    * Then the method of the super class (Forwarding.java) is invoked
  - Otherwise two rules are installed on the switch that sent the packetIn in order to deny the communication between the two hosts for a certain amount of time (as we have seen the default timeout is 5 seconds but that value can be customized through REST API).

  Here a section of the method *processPacketInMessage*. Notice that when setting the switch id, is the method *setSw(IOFSwitch switch)* that checks if the switch is already set or not and set the switch only if it is not already set.

```
    HostPair hp = null;
    int hpIndex = intentsDB.indexOf(new HostPair(sourceIP, destinIP));
    if (hpIndex != -1)
    hp = intentsDB.get(hpIndex);

    if(hp != null && intentsDB.contains(hp)) {
      System.out.printf("allowing: %s - %s on switch %s \n",
      sourceIP.toString(), destinIP.toString(), sw.getId());
      if(hp.getHost1IP().equals(sourceIP))
      hp.setSw1(sw);
      if(hp.getHost2IP().equals(sourceIP))
      hp.setSw2(sw);
      return super.processPacketInMessage(sw, pi, decision, cntx);
    }
    denyRoute(sw, sourceIP, destinIP);
    denyRoute(sw, destinIP,sourceIP);
    return Command.CONTINUE;
```

1.2.3 ARP Management

The ARP protocol is used to retrieve the layer 2 address of an host given its IP address. When an ARP packet arrives to a switch a packetIn is sent to the controller and so the processPacketInMessagge is invoked, the method understands that the packet that has triggered the packetIn is an ARP packet and so it must be handled differently.
The first operation done is checking if an intent between the sender and the receiver exists using their IPs. If an intent exists then the method of the super class is invoked otherwise the ARP packet is blocked.
The rules to avoid communication between the MACs addresses carried inside the ARP packet are installed only if that packet is an ARP response, in that case the target MAC address is not 00:00:00:00:00:00. In fact installing a rule that deny to forward an ARP messagge from a certain mac X to 00:00:00:00:00:00 means deny X to runs the ARP protocol because for each ARP request the target mac is 00:00:00:00:00:00 i.e. unspecified mac. Viceversa deny an ARP from 00:00:00:00:00:00 to any mac has no sense.

1.3 REST API For Establishing An Intent

When the hostA wants to send an intent to make a connection with hostB it has to use the REST API. This API is provided by the classes: IntentWebRoutable, AddNewIntent, DelIntent and GetIntent; these classes use methods from IntentForwarding.
When the REST API receives the intent, it establishes it inserting an entry in a dedicated data structure where it is indicated:

- The Source
- The Destination
- Intent Timeout

The possible API's that we can use are:

- "/addNewIntent/json", that is used to insert new intent. This API gets from a JSON file two IPs (representing source and destination) and a timeout. This use the addNewIntent function of the IntentForwarding class that, first check if new intent is already presents, then in case it isn't put it in intentDb that is an ArrayList of HostPair. Moreover, this function also set the timeout time of the pair.
- "/delIntent/json", that takes from a JSON file two IPs and a timeout and removes the intent corresponding to them. To do this, it use the delIntent function of IntentForwarding, that iterates the intentDB list and removes the intent with IPs passed in through JSON.
- "/getIntents/json", that is used to get all intents present in specific moment. It use the getIntents function of IntentForwarding that simply return intentDb.

Then the intent will be pushed in the network when the source host tries to send a packet to the destination host. When this happens a packetIn sending from the first switch that receives the packet will be sent to the controller and will be handled by the forwarding module seen in 1.2

1.4 Responsiveness To Link Failures & Topology Changes

To be responsiveness to topology changes we can exploit the ITopologyListener inteface. In fact in the Floodligh documentation for the TopologyService it is written: *"All the information about the current topology is stored in an immutable data structure called the topology instance. If there is any change in the topology, a new instance is created and the topology changed notification message is called. If other modules want to listen for changes in topology they can implement the ITopologyListener interface."*[4].
Thus we can implement a class that implements ITopologyListener and that each time a topology change arrives it recompute the paths of intent involved in the topology change (i.e. the ones that has a link which doesn't work anymore). To do so we can use the TopologyInstance (updated before sending the notification message that the topology has changed) to understand which intent are involved in the topology change.

[4]https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343623/TopologyService+Dev

Chapter 2

Testing

Languages and Frameworks

In order to test our system, we need to use a tool to create a *virtual network* inside our machine; we used `mininet`, exploiting the `python2` APIs. To simulate input from an external application we also used a python2 library, called `requests`, which is able to act like an HTTP client.

2.1 Scenario

We want to simumlate a tipical data center scenario with a single LAN, implemented in a *Spine-Leaf* topology. This configuration is widely used thanks to it's easy scalability and sufficent redundancy.

Figure 2.1: an example of *spine-leaf* topology