

# SDN & NFV - Master REOC

Compte rendu de TP - Aude Jean-Baptiste et Romain Moulin

## TP1 : SDN

Accès au sujet : <http://tiny.cc/TP-SDN>

### Partie 1 : Prise en main de Floodlight et de Mininet / Containernet

*Pinger* Hôte 2 à partir du Hôte 1. Le ping aboutit-il ?

Après l'ajout du flow bloquant le trafic provenant de Host1 non (dans les deux sens ça passe pas).

### Partie 2 : Redirection transparente de paquets IP

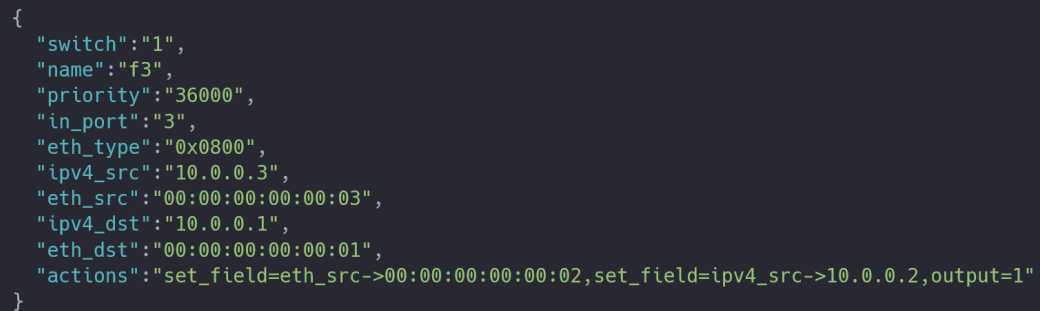
Si vous observez du trafic à ce niveau, il s'agit essentiellement des paquets LLDP envoyés par le contrôleur pour découvrir la topologie. À partir de la console **xterm** de Host1, *pinger* Server1 et ensuite Server2. Qu'observez-vous ?

Le ping se passe normalement, on observe le trafic ICMP sur le serveur destinataire du ping et rien sur l'autre.

À partir de Host1 *pinger* Server1. Qui de Server1 ou de Server2 reçoit le paquet ? La redirection transparente a-t-elle abouti ? Justifiez votre réponse. Installez la dernière règle de flux pour rendre effective la redirection transparente.

Lorsque l'on pingue Server1, c'est Server2 qui reçoit le paquet, on a bien une redirection de trafic. En revanche, la redirection n'est pas transparente, car les paquets ICMP reply portent les adresses IP et MAC de Server2, alors que le client attend une réponse de Server1.

La dernière règle de flux appliquée pour obtenir une redirection transparente peut être trouvée ci-dessous.



```
{
  "switch": "1",
  "name": "f3",
  "priority": "36000",
  "in_port": "3",
  "eth_type": "0x0800",
  "ipv4_src": "10.0.0.3",
  "eth_src": "00:00:00:00:00:03",
  "ipv4_dst": "10.0.0.1",
  "eth_dst": "00:00:00:00:00:01",
  "actions": "set_field=eth_src->00:00:00:00:00:02,set_field=ipv4_src->10.0.0.2,output=1"
}
```

*Figure 1: Dernière redirection pour le trafic retour*

## Partie 3 : Intégration de modules à Floodlight

Pour cette partie, nous avons décidé de faire une application de load balancing à la demande. Pour cela nous avons utilisé la même topologie en étoile que dans la partie 2 en ajoutant un troisième serveur. Par défaut, le client contacte le Server1. Nous avons par la suite développé un programme python qui vient piloter le controller SDN.

Pour l'implémentation, nous avons fait une cli à l'aide du module Cmd de python. En tapant la commande `redirect_server X`, nous redirigeons tout le trafic vers le serveur X. Pour cela, nous effectuons simplement des requêtes POST au controller afin d'implémenter des flows aller et retour pour rediriger le trafic comme dans la question 2. Nous supprimons également à chaque fois les anciens flows afin d'éviter tout conflit au niveau du switch.

```

from cmd import Cmd
import requests
import json

class MyPrompt(Cmd):
    # redirect traffic to server 2
    server2JsonAllez = {
        "switch": "1",
        "name": "f21",
        "priority": "36000",
        "in_port": "1",
        "eth_type": "0x0800",
        "eth_src": "00:00:00:00:00:01",
        "ipv4_src": "10.0.0.1",
        "eth_dst": "00:00:00:00:00:02",
        "ipv4_dst": "10.0.0.2",
        "actions": "set_field=eth_dst->00:00:00:00:00:03,set_field=ipv4_dst->10.0.0.3,output=3"
    }
    # redirect traffic to server 3
    server3JsonAllez = {
        "switch": "1",
        "name": "f31",
        "priority": "36000",
        "in_port": "1",
        "eth_type": "0x0800",
        "eth_src": "00:00:00:00:00:01",
        "ipv4_src": "10.0.0.1",
        "eth_dst": "00:00:00:00:00:02",
        "ipv4_dst": "10.0.0.2",
        "actions": "set_field=eth_dst->00:00:00:00:00:04,set_field=ipv4_dst->10.0.0.4,output=4"
    }
    # modify source addresses of traffic from server 3
    server2JsonRetour = {
        "switch": "1",
        "name": "f22",
        "priority": "36000",
        "in_port": "3",
        "eth_type": "0x0800",
        "eth_src": "00:00:00:00:00:03",
        "ipv4_src": "10.0.0.3",
        "eth_dst": "00:00:00:00:00:01",
        "ipv4_dst": "10.0.0.1",
        "actions": "set_field=eth_src->00:00:00:00:00:02,set_field=ipv4_src->10.0.0.2,output=1"
    }
    # modify source addresses of traffic from server 3
    server3JsonRetour = {
        "switch": "1",
        "name": "f32",
        "priority": "36000",
        "in_port": "4",
        "eth_type": "0x0800",
        "eth_src": "00:00:00:00:00:04",
        "ipv4_src": "10.0.0.4",
        "eth_dst": "00:00:00:00:00:01",
        "ipv4_dst": "10.0.0.1",
        "actions": "set_field=eth_src->00:00:00:00:00:02,set_field=ipv4_src->10.0.0.2,output=1"
    }

    def do_exit(self, inp): # exit the cli
        return True

    def delete_all_flows(self): # delete all flows with delete requests
        requests.delete("http://10.0.2.15:8080/wm/staticflowpusher/json",
            data=json.dumps({"switch": "1", "name": "f32"}))
        requests.delete("http://10.0.2.15:8080/wm/staticflowpusher/json",
            data=json.dumps({"switch": "1", "name": "f22"}))
        requests.delete("http://10.0.2.15:8080/wm/staticflowpusher/json",
            data=json.dumps({"switch": "1", "name": "f21"}))
        requests.delete("http://10.0.2.15:8080/wm/staticflowpusher/json",
            data=json.dumps({"switch": "1", "name": "f31"}))

    def redirect_flow(self, id): # create flows to redirect to server <id>
        self.delete_all_flows() # delete all previous flows
        if(id==2): # redirect to server 2
            requests.post("http://10.0.2.15:8080/wm/staticflowpusher/json",
                data=json.dumps(self.server2JsonAllez))
            requests.post("http://10.0.2.15:8080/wm/staticflowpusher/json",
                data=json.dumps(self.server2JsonRetour))
        elif(id==3): # redirect to server 3
            requests.post("http://10.0.2.15:8080/wm/staticflowpusher/json",
                data=json.dumps(self.server3JsonAllez))
            requests.post("http://10.0.2.15:8080/wm/staticflowpusher/json",
                data=json.dumps(self.server3JsonRetour))
        elif(id!=1): # server 1 by default
            print("Error: Please enter a valid ID (1, 2 or 3)")
            return
        print("Traffic redirected to server " + str(id))

    def do_redirect_server(self, inp): # call redirect_server <id> to execute
        id=int(inp)
        self.redirect_flow(id)

    MyPrompt().cmdloop() # launch the prompt

```

Figure 2: Le code de notre Load Balancer

A des fins de tests, nous avons lancé un ping depuis le Host1 et changé à la volée (pendant l'exécution du ping) le serveur vers lequel le trafic est redirigé. Nous avons pu observer que du point de vue de l'Host1, tout est transparent. En effet, le ping est envoyé et revient avec l'adresse IP du serveur 1. Cependant à l'aide de TCPdump lancé sur les différents serveurs, nous pouvons voir que le trafic est bien redirigé vers les différents serveurs en fonction de ce qui est demandé sur la CLI.

## TP2 & TP3 : Projet SDCI

Accès au sujet : [tiny.cc/Projet-SDCI-Infos](http://tiny.cc/Projet-SDCI-Infos)

### Objectifs du projet

L'objectif de ce projet est de partir d'une architecture classique en IoT, comprenant des capteurs / actionneurs (devices) communiquant avec un serveur par l'intermédiaire de gateway intermédiaire et finales, comme présenté Fig. 3.

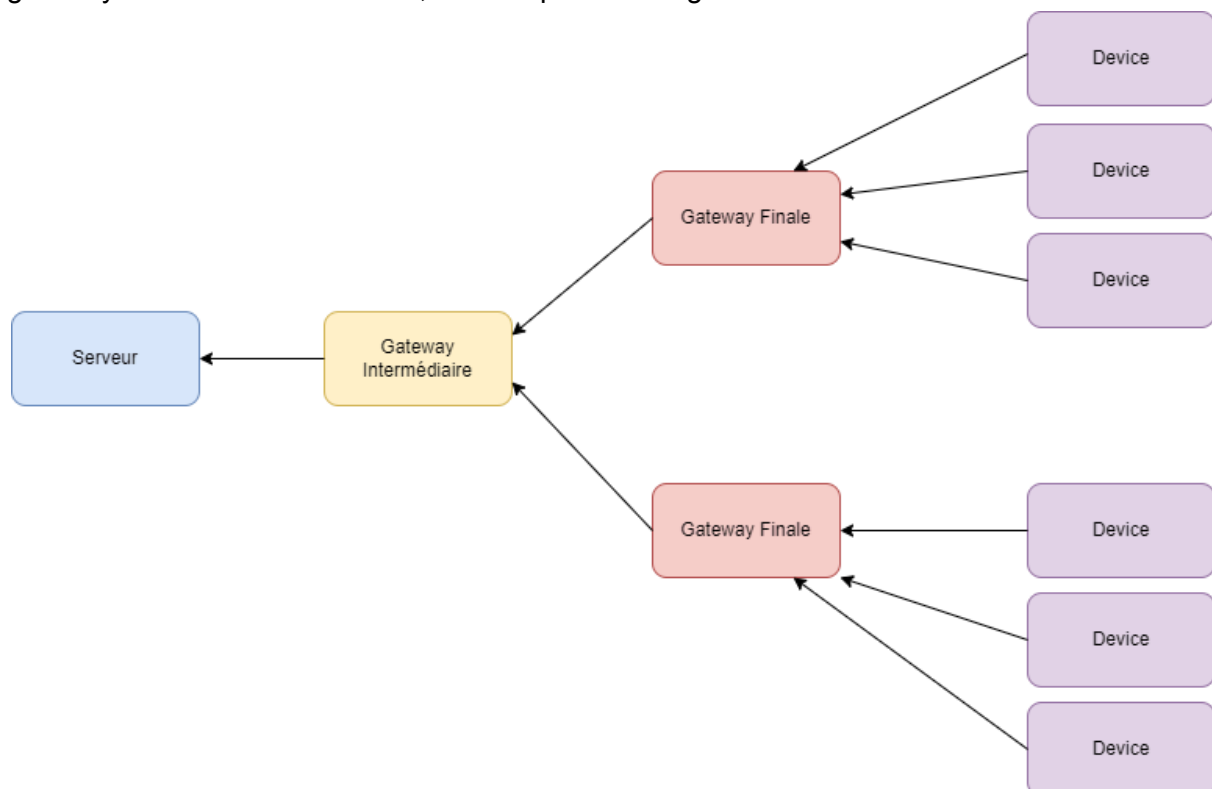


Figure 3: La topologie logique initiale

À partir de cette architecture, on souhaite implémenter une solution de redirection du trafic à la volée lors d'augmentation de l'activité d'un ou plusieurs devices, par l'ajout de gateway intermédiaires lorsque cela s'avère nécessaire. Cette approche NFV (network function virtualization) va de pair avec de la SDN (software defined network) afin d'assurer une redirection transparente du trafic.

Pour implémenter cette solution, nous reposerons sur un réseau containernet (un fork de mininet supportant la mise en place de container docker en guise d'hosts). Notre réseau tournera sur une VM équipée d'un OS Ubuntu.

## Utilisation d'un metadata server

Lors de la première séance, nous avons choisi l'approche suivante :

- création d'une image docker contenant toutes les librairies et fichiers javascript nécessaires pour lancer le serveur, la gateway intermédiaire, etc.
- création d'un metadata serveur où chaque élément lancé (serveur, gateway intermédiaire, etc) vient chercher sa configuration.

Afin de simplifier l'instanciation du réseau avec containernet, nous avons visé une topologie plate comme détaillé Fig. 4.

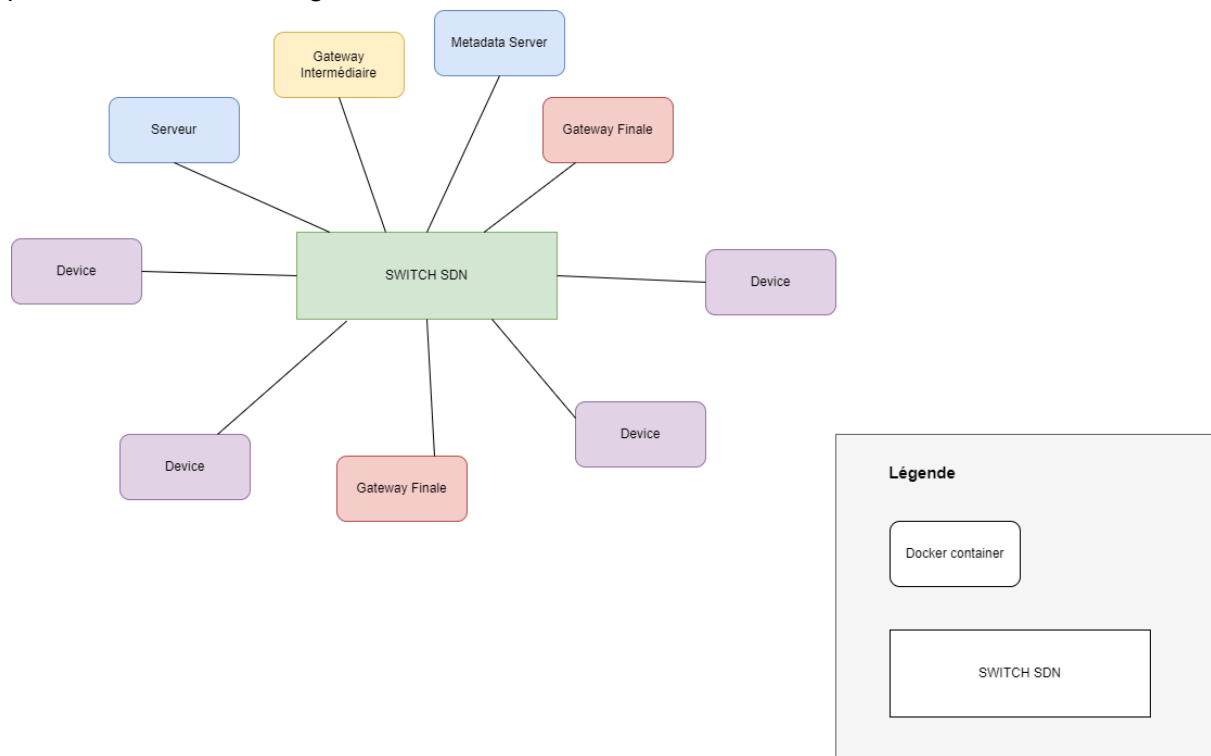


Figure 4: Topologie visée lors de la première séance

Notre travail nous a amené au code suivant pour le metadata serveur, présenté Fig. 5.

```

let app = require('express');
let request = require('request');
const port = 8080;

app.use(express.json());

let nodes = {"server":{}, "gi":{}, "gf":[{}]}

app.post('/register', (req,res)=> {
  const { type, ip, port, name} = req.body;
  if(type == "server"){
    nodes["server"] = {ip, port, name};
  }
  if (type == "gi"){
    nodes["gi"] = {ip,port,name};
  }
  if (type == "gf"){
    nodes["gf"].push({ip,port,name});
  }
  console.log("register " + JSON.stringify({ip,port,name}));
  res.status(200);
})

app.get('/server', (req, res)=> {
  res.status(200).send(nodes["server"]);
})

app.get('/gi', (req, res)=> {
  res.status(200).send(nodes["gi"]);
})

// return la liste des gf
// dans notre implementation les devices utilisent la GF sur le localhost
app.get('/gf', (req, res)=> {

  console.log("request reçue sur /gf");
  console.log("on envoie " + JSON.stringify(nodes.gf));
  res.status(200).send(nodes["gf"]);
})

app.listen(port, ()=>{
  console.log(`Server listening on port ${port}`)
})

```

Figure 5: Le code de notre metadata server

Comme vous pouvez le constater Fig. 5, nous avons choisi d'implémenter une API REST avec les endpoints suivants :

- /register (POST) : permet à un élément de s'enregistrer
- /server (GET) : permet à la/les gateway intermédiaires de récupérer les paramètres (ip, port, nom) du server afin d'établir avec lui une communication
- /gi (GET) : permet aux gateway finales de récupérer les paramètres de la gateway intermédiaire afin d'établir une communication avec elle
- /gf (GET) : permettrait, dans une future implémentation, aux devices de récupérer la liste des gateway finales afin de choisir avec laquelle communiquer. En pratique, dans cette implémentation, le choix a été fait que les devices soient lancés sur le même host que leur gateway finale, afin de faciliter la mise en place de la topologie.

Nous avons ensuite modifié les différents fichiers fournis pour qu'ils s'enregistrent auprès du serveur à l'aide du code présenté Fig. 6.

```
// register to metadata server
let ip_meta = "10.10.10.10"
let iface = os.networkInterfaces()["eth0"];
let ip_serv = iface[0].address
let req_res = request('POST', `http://${ip_meta}/register`, {json:{type: "server", ip:ip_serv, port:cfg.local_port,
name:cfg.local_name}});
if (req_res.statusCode==200){
  console.log("successfully registered");
} else {
  console.log("Problem with registration");
}
```

Figure 6: Code pour s'enregistrer sur le metadata server

Les différents éléments devant récupérer une configuration sur le metadata serveur se sont vus ajouter le code présenté Fig. 7.

```
let req_server = request('GET', `http://${ip_meta}/server`);
if (req_server.statusCode == 200){
  console.log("Information about server received");
}else{
  console.log("Problem when getting server information")
}
```

Figure 7: Code permettant de récupérer une configuration auprès du metadata serveur

Cette solution fonctionnait bien en localhost sur la VM. Malheureusement, nous n'avons pas pu mener cette approche au bout car la réalisation d'une image docker générique contenant tous les éléments nécessaires pour être "instancié" en temps que serveur, gateway finale, etc, s'est avérée plus complexe que prévu.

Nous avons été confrontés à de nombreux problèmes de version qui ont rendu impossible l'installation de toutes les librairies npm nécessaires pour le code en raison d'une version d'Ubuntu trop vieille. Nous avons donc choisi une autre approche pour la deuxième séance.

## Changement d'approche

Entre les deux séances, nous avons tenté de réinstaller sur une de nos machines personnelles une VM Ubuntu avec une version récente de l'OS, docker, npm, etc. La version plus récente de l'OS nous a effectivement permis de résoudre les problèmes avec npm. Malheureusement, les choses se sont compliquées au moment d'installer le contrôleur Ryu. En effet, nous avons installé la version la plus récente de python, qui s'est avérée

incompatible avec ryu. Après de vaines tentatives de downgrade la version de python installée sur la VM, nous avons choisi de recommencer nos tentatives sur la nouvelle VM fournie lors de la deuxième séance.

Cette nouvelle VM ne posait pas de problèmes d'installation, bien que le fait qu'elle ne présentait pas d'interface graphique nous a demandé un petit temps d'adaptation.

Hélas, le serveur du LAAS où sont hébergés les fichiers pour lancer le serveur, les gateway, etc, a subi une panne générale durant la séance. Cela nous a empêché de continuer notre projet d'image docker contenant tous les fichiers.

Las de toutes ces péripéties, nous avons décidé de réaliser le script nécessaire à la redirection de trafic sans possibilité de le tester. Nous sommes confiants quant au fait que l'expérience acquise au cours de ces deux séances de debug intensif et la fine compréhension des concepts de NFV et SDN nécessaires au TP permettent à ce script d'être fonctionnel même en l'absence de test.

Vous trouverez ce superbe script, fruit de notre travail, dans la partie suivante.

## Script final

Le script final permettant la redirection de trafic vers une nouvelle gateway intermédiaire en cas d'augmentation de trafic est présenté Fig. 8.



```

import requests
import json

gf1_to_gi2 = { # Redirect the traffic from final gateway 1 originally destined to intermediate gateway to newly created
intermediate gateway 2
    "dpid":1,
    "priority":11111
    "match": {
        "ipv4_src":"10.0.0.3", ## 10.0.0.3 is the IP source of final gateway 1
        "eth_src":"00:00:00:00:00:03", ## MAC address of the final gateway 1
        "ipv4_dst":"10.0.0.2", ## 10.0.0.2 is the IP source of intermediate gateway 1
        "eth_dst":"00:00:00:00:00:02" ## MAC address of the intermediate gateway 1
    },
    "actions":[{
        "type":"OUTPUT", ## Change the output of the switch to the newly created intermediate gateway 2
        "port":12
    },
    {
        "type":"SET_FIELD",
        "eth_dst":"00:00:00:00:00:12", ## Change the destination IP to match the IP and MAC of the newly created intermediate
gateway 2
        "ipv4_dst":"10.0.0.12"
    }]
}

gi2_to_gf2 = { ##For the packets that comes back to final gateway 1 from newly created intermediate gateway 2
    "dpid":2,
    "priority":11111
    "match": {
        "ipv4_src":"10.0.0.12", #IP of intermediate gw 2
        "eth_src":"00:00:00:00:00:12", #MAC of intermediate gw 2
        "ipv4_dst":"10.0.0.3", #IP of final gw 1
        "eth_dst":"00:00:00:00:00:03" #MAC of final gw 1
    },
    "actions":[{
        "type":"SET_FIELD", #Change the source IP and MAC of the packet to intermedite gw 1
        "eth_src":"00:00:00:00:00:02",
        "ipv4_src":"10.0.0.02"
    },
    {
    }]
}

def redirection():
    host1 = net.addDocker("h1", ip="10.0.0.12", mac="00:00:00:00:00:02", dimage="sdcl:1.0.0") #Add a new docker to the containernet
architecture with static IP and MAC address
    net.addLink(host1,s1) # Create a new link between the new host and the switch S1
    requests.post("http://localhost:8080/stats/flowentry/add", data=json.dumps(gf1_to_gi2)) # Redirect the traffic from the final
gateway 1 to the new intermediate gateway
    requests.post("http://localhost:8080/stats/flowentry/add", data=json.dumps(gi2_to_gf2)) # Change the source address for the
answer to make it transparent for the final gateway

if __name__ == '__main__':
    redirection()

```

Figure 8: Script final

## Conclusion

Ces TP nous ont permis de mieux comprendre et appréhender les concepts de NFV et SDN ainsi que leur utilité dans un contexte type IoT.

Nous avons également développé nos compétences en virtualisation (VM, containers, mininet).