INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Université de Toulouse

Master Réseaux Embarqués et Objets Connectés

Projet Software Defined Communication Infrastructure

Cédric Chanfreau, **Yohan Boujon**

Institut National des Sciences Appliquées de Toulouse

15 janvier 2025

Table des matières

Introduction

Lien du repo : https://github.com/CedricChnfr/sdci-reoc

Lien du sujet : https://tiny.cc/Projet-SDCI

Lien des informations : https://tiny.cc/Projet-SDCI-Infos

1.1 Contexte du projet

Dans le cadre du master REOC *(Réseaux Embarqués et Objets Connectés)*, nous sommes amenés à réaliser un projet visant à développer une infrastructure de communication définie par logiciel à l'aide de principes comme **VNF** [1] ou **SDN** [2] pour gérer et surveiller les réseaux de manière efficace. L'objectif principal de ce projet est de créer une topologie réseau en utilisant des technologies telles que *Docker*, *Mininet* et *OpenFlow* pour simuler et gérer les composants réseau.

1.2 Objectifs Spécifiques

Le projet se concentre sur trois cas d'utilisation principaux :

1. **Ordonnanceur** : Gérer les communications entre les zones Z_1, Z_2, Z_3 et la passerelle GI. Le but d'un ordonnanceur est de donner une priorité plus ou moins élevé pour une zone donnée. Dans notre cas, la Z_1 aura une priorité sur les autres zones, ce qui fera en sorte que les paquets venant de ces périphériques seront plus vite pris en charge par le switch. Un schéma spécifique peut être associé, il faut faire attention à rediriger les paquets vers l'ordonnanceur.

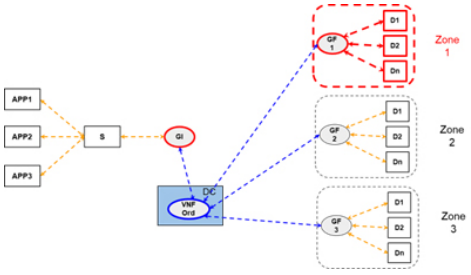

FIGURE 1 – Schématique de l'ordonnanceur

2. **Limiteur Réseau** : Implémenter des mécanismes de rejet et de limitation de débit au niveau L3 pour le trafic provenant des zones Z_2 et Z_3. Cela permettra de contrôler le flux de données et d'éviter la congestion du réseau en appliquant des politiques de gestion de la bande passante. La Z_1 encore une fois sera priorisée et aura un débit maximum "infini", alors que les autres zones seront limitées fortement, ce qui pourrait cependant provoquer la perte de paquets par congestion des buffers. Voici son schéma :

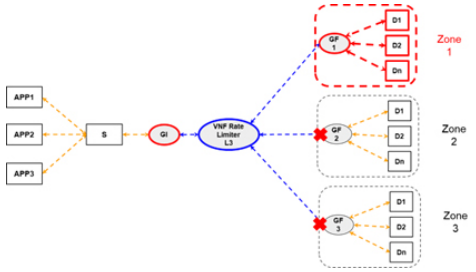

FIGURE 2 – Schématique du limiteur réseau

3. **Monitoring L2/L3 via SDN (OpenFlow)** : Mettre en place un système de monitoring pour surveiller les performances et l'état des ports réseau en utilisant SDN et OpenFlow. Ce système de surveillance fournira des informations en temps réel sur les statistiques de trafic, les paquets reçus et envoyés, les erreurs et les pertes de paquets, permettant ainsi une gestion proactive du réseau.

Pour atteindre ces objectifs, nous utiliserons plusieurs technologies et outils, notamment Docker pour la création de conteneurs qui seront utilisés dans la simulation, **Containernet** *(Une fork de Mininet)* [1] pour la simulation de la topologie réseau. Il existe aussi un logiciel nommé **Vim-Emu** [1] qui nous permettra de faire des Virtualisation des fonctions réseau. Pour la gestion SDN nous utiliseront **Ryu** [2], la version **OpenFlow** [3] liée sera la *1.6*.

1. **VNF** : Virtual Network Function, virtualisation réseau permettant de simuler, modifier de manière dynamique des équipements réseaux.
2. **SDN** : Software Define Network, à partir de règles, modifier dynamiquement les paquets envoyés, que ce soit les champs, supprimer certains paquets, créer des priorités, orchestrer les données envoyés.

Architecture Réseau

2.1 Topologie réseau

La topologie réseau est composée de plusieurs switches et hôtes connectés pour simuler un environnement réseau complexe. Voici un schéma de la topologie :

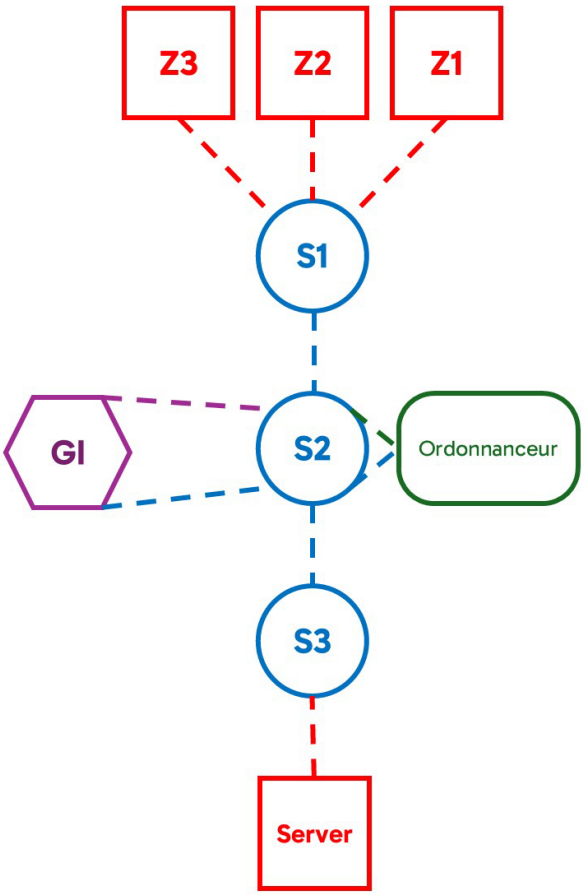

FIGURE 3 – Topologie du réseau

Comme nous pouvons le voir, chaque zone est connectée au premier switch. Dans notre simulation nous allons omettre chaque périphérique et considérer qu'une seule Zone correspond à un seul *Container*. Cela ne nous empêchera pas d'émuler une émission de plusieurs périphériques en simultanée, la seule différence est qu'au lieu d'appartenir à un sous-réseau, ce sera un unique réseau. Heureusement cela passe à l'échelle sans problème et la simulation n'est pas affectée.

Chaque Switch possède une interface la reliant à des points d'intérêts. Le premier est connecté aux différentes zones, le second à l'ordonnanceur et la gateway, le dernier au serveur de reception.

Le choix d'une telle topologie nous permettra de tester si les données peuvent être filtrés à notre convenance, dans un premier tant en utilisant des applications de la simulation tel que *Ryu* ou encore *Openflow* de manière plus général. Ensuite l'utilisation des dockers avec des applications JavaScript, Python ou GoLang nous permettra d'envoyer des données plus importantes et de vérifier si la connexion est correcte entre les deux extrémités du réseau.

2.2 Composants principaux

- **Switches** (S_1, S_2, S_3) : Utilisés pour interconnecter les différentes zones et composants du réseau.
- **Hôtes** (Z_1, Z_2, Z_3) : Représentent les zones du réseau avec des périphériques simulés, ces hôtes seront composés de conteneurs avec des clients envoyant des données.
- **Ordonnanceur** : Au cœur de notre outil de *monitoring*. Il permettra de lancer notre scénario pour ordonnancer les paquets venant de chaque zone.
- **Gateway Intermédiaire** (G_I) : Permettra de rediriger les paquets venant des différentes zones pour pouvoir modifier leur contenu.
- **Serveur** : Fonctionne comme les hôtes, possédera un conteneur mais cette fois-ci en mode serveur, attendant des données.
- **Datacenter** (D_C) : Utilise OpenStack pour s'interfacer avec le reste de la topologie.

Implémentation

3.1 Instances Docker

Chaque instance docker peut être créée à l'aide de notre script *create.sh*. Le but était simple, d'avoir un moyen efficace de générer chaque image docker à chaque fois que nous faisons une modification des scripts :

```
#!/bin/bash

SCRIPT_PATH=$(realpath "$0")
SCRIPT_DIR=$(dirname "$SCRIPT_PATH")

cd "$SCRIPT_DIR" || exit 1

for dockerfile in
↪  "$SCRIPT_DIR"/*.dockerfile; do
    image_tag="reoc:$(basename
    ↪  "$dockerfile" .dockerfile)"

    echo "Building image: $image_tag from
    ↪  $dockerfile"
    docker build -t "$image_tag" -f
    ↪  "$dockerfile" .
done
```

Au début cela nous était utile pour tester la connectivité des différentes zones, un script Python plutôt brutal a été crée pour l'occasion. Envoyant directement en UDP un flot de données constant : cela nous a permis par la suite d'identifier à quoi correspondait chaque entrée du switch S_1 dans l'API Ryu. En effet, les numéros *ID* définis dans la réponse Ryu ne correspondait à rien de connu dans notre topologie, c'est pas cette analyse que nous sommes arrivés à la conclusion que :

- *3 (port 1)* : Z_1
- *4 (port 2)* : Z_2
- *5 (port 3)* : Z_3

Cela nous sera utile pour la Sec. 4.1.

```
import socket
import signal
import sys
import time

client_socket =
↪  socket.socket(socket.AF_INET,
↪  socket.SOCK_DGRAM)
# 10.0.0.254 is the address of 'gi'
server_address = ('10.0.0.254', 8080)

def signal_handler(sig, frame):
    print("\nInterrupt received, closing
    ↪  socket...")
    client_socket.close()
    sys.exit(0)
signal.signal(signal.SIGINT,
↪  signal_handler)

try:
    message = "a"
    while True:

        ↪  client_socket.sendto(message.encode(),
        ↪  server_address)
        time.sleep(0.1)
except Exception as e:
    print(f"An error occurred: {e}")
finally:
    print("Closing socket...")
    client_socket.close()
```

Pour simuler une connexion avec chacun des périphériques, il nous a été donné des fichiers javascript envoyant certains paquets. Sur le serveur il y a aussi une API rest utilisable pour avoir différentes données, notamment *"/devices"* pour récupérer chaque périphérique connecté, *"/gateways"* similaire pour chaque gateway, *"/ping"* pour envoyer un simple ping entre le serveur et un des périphériques choisis et enfin *"/health"* pour avoir la charge réseau. Sur le principe, chacune de ces fonctions sont déjà réalisées et sont une aide pour le monitoring et d'autres fonctionnalités, malheureusement comme nous n'avons pas eu le temps de faire fonctionner ce dernier nous voulions partir sur une autre approche.

Le langage de programmation *"Go"* crée par Google était une solution que nous avions envisagé car il est proche du système et permet de faire un simple serveur REST tout comme le prototype proposé. En plus de demander moins de dépendances, mais par manque de temps il nous était impossible de faire cela. Une idée à creuser pour de possibles prochains projets.

3.2 Simulation Containernet

Dans le dossier **mininet**, il existe plusieurs fichiers nous permettant de générer la topologie à l'aide de *Containernet*, ou de supprimer chacune des liaisons. En effet, en lançant le fichier *topology_ sdn.py* il peut arriver qu'en modifiant certains paramètres, ou en jouant avec l'API Ryu, certaines liaisons ne fonctionnent plus, nous avons donc décidé de créer un script pour supprimer chacune des liaisons créés.

```
docker rm mn.serveur --force
docker rm mn.gi --force
docker rm mn.ordon --force
docker rm mn.z3 --force
docker rm mn.z2 --force
docker rm mn.z1 --force
sudo mn -c
sudo ip link delete s1-s2
sudo ip link delete s2-s1
sudo ip link delete s2-s3
sudo ip link delete s3-s2
```

Ce script nous permettait de tout remettre à zéro. Si la simulation s'arrêtait de manière intempestive, il pouvait rester des dockers connectés, ce qui nous a posé de nombreux problème lors de la configuration de certains paramètres des switch. Il est aussi parfois possible de trouver des liaisons entre les switch quand nous avions changé leurs noms en cours de route.

Un dernier point dans notre script python *topology_ sdn.py* à noter est la création d'un fichier nommé *network_ links.temp*, ce dernier est généré à la création de la topologie. Il fait un simple appel à la fonction de *Containernet* permettant d'afficher le lien entre chaque périphérique sur le réseau. Ce fichier sera par la suite utilisé dans le moniteur, dans la Sec. 4.1.

Application 'Moniteur'

4.1 Monitoring

Le monitoring correspond à une interface graphique développée avec *npyscreen* nommé *"sdci.py"*. Cette interface permet de sélectionner et d'exécuter différents scripts en fonction des tests que nous souhaitons réaliser. Dans notre cas, ce qui nous intéresse c'est le nombre de données envoyées par chacune des zones, nous allons donc nous concentre sur le premier switch S_1. Pour faire cela nous allons simplement observer chacun des débits, qu'il soit entrant, sortant, la taille des paquets, le nombre d'octets totaux, pour faire cela, un appel vers l'API de *Ryu* est necessaire :

```
curl http://127.0.0.1:8080/stats/port/1
```

Dans notre application, nous avons fait le choix qu'il soit possible d'observer cette évolution en *"temps réel"*. Lorsque l'utilisateur depuis l'application *controller* décide d'observer l'évolution des paquets un *thread* est lancé, tournant en arrière plan et récupérant chacune des données pour un port du switch donné. Cela nous permet par la suite de mettre à jour l'affichage.

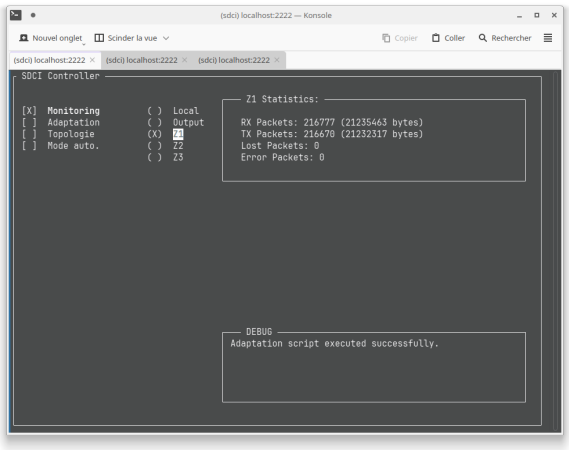

FIGURE 4 – Onglet Monitoring de l'application *sdci.py*

Quelques améliorations possibles si le temps nous l'aurait permis aurait été d'ajouter le débit courant actuel entrant et sortant, des tests auraient pu être fait pour voir comment les paquets d'erreurs ou perdus auraient interagis avec l'interface. Finalement, cette partie est entièrement fonctionnelle et nous a permis d'avancer grandement sur le mode d'ordonnancement.

4.2 Adaptation

L'adaptation permet de choisir différents scénarios, dans notre cas nous en avons deux bien délimités :

- Blocage des trames provenant de la Z_2 et Z_3, ne laissant que passer Z_1
- Réduction du débit pour les Z_2 et Z_3, laissant la Z_1 fortement prioritaire en terme autant de taille de trames que de débit brut

Pour faire ces deux tests, nous avons tout d'abord utiliser une règle bloquant chacune des zones qui ne nous intéresse pas à l'aide d'une requête POST utilisant l'API de ryu. La règle établie est la suivante :

```
{
    "dpid": 1,
    "table_id": 0,
    "priority": 1,
    "match": {
        "in_port": 2
    },
    "instructions": [
        {
            "type": "APPLY_ACTIONS",
            "actions": [
                {
                    "type": "DROP"
                }
            ]
        }
    ]
}
```

Ici nous appliquons l'action d'abandonner tous les paquets entrant venant du port d'entrée 1 (Z_2). Nous ferons de même pour Z_3, il nous suffit ensuite d'entrer dans les conteneurs que nous avons créer et de lancer une requête ping-pong entre l'hôte et le serveur, ici nous nous arrêterons à la G_I qui est suffisante pour notre exemple, voici la commande utiliser pour rentrer dans les conteneurs :

```
docker exec -it mn.<device> "bash"
```

Ainsi que le resultat des commandes : Comme nous pouvons le voir, cette fonctionnalités fonctionne très bien.

Il reste maintenant à faire une tâche bien plus compliquée dans l'adaptation : limiter le débit pour certaines zones. Un script, *qos.sh* a été créé, cette fois-ci la requête est légèrement différente et demande de créer des queues avec *Open vSwitch*[3] ainsi que de lier ces dernière avec *Ryu*.

```
"actions": [
    {
      "type": "SET_QUEUE",
      "queue_id": 1
    },
    {
      "type": "OUTPUT",
      "port": 4
    }
]
```

Cette action n'aurait pour le moment aucun effet car la queue indiquée n'existe pas, il faut donc lancer l'utilitaire *"ovs-vsctl"* [4] qui permet de créer cette dernière dans l'environnement *OpenStack*. Dans un des exemples, il est précisé qu'il est possible de paramétrer pour une certaine queue le débit maximum et minimum. C'est donc ce que nous avons fait avec une commande similaire à cela :

```
ovs-vsctl -- set port eth0 qos=@newqos --
↪  --id=@newqos create qos type=linux-htb
↪  other-config:max-rate=1000000
↪  queues:0=@newqueue -- --id=@newqueue
↪  create queue
↪  other-config:min-rate=1000000
↪  other-config:max-rate=1000000
```

Malheureusement, comme précisé précédemment, cela n'a eu aucun effet, le débit ne semble en rien modifié. Nous avons donc laisser ces scripts pour observer plus en détail le non-fonctionnement de ce dernier.

4.3 Topologie

La topologie permet de monter à l'utilisateur comment le réseau est construit, cela est utile pour savoir d'où proviennent chacune des données. Pour générer une simple topologie nous avons eu l'idée de génerer un fichier lors du lancement de *Containernet*. Ce fichier affiche la connexion entre chaque interface, il est possible de l'afficher avec le *controller* en allant simplement dans l'onglet demandé.

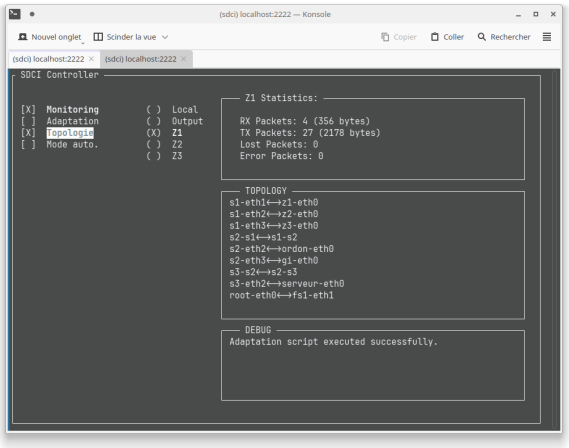

FIGURE 6 – Onglet Topology de l'application *sdci.py*

Une amélioration possible serait d'afficher l'adresse IP de chaque appareil et de rendre cela plus graphique, mais avec le temps qui nous ai donné cela n'a pas été possible. Il est aussi à noter que cette topologie n'est pas dynamique dans l'instance *Containernet* actuelle. Une amélioration possible serait de faire en sorte directement dans l'instance de génerer ce fichier toutes les secondes *(temps à titre d'exemple)*.

4.4 Mode automatique

Le mode automatique n'a pas pu être réalisé dans son entièreté dû au manque de temps, cependant il aurait pu être assez simple à implémenter dans le principe. En effet, le but de ce dernier aurait été de lancer les divers scénarios de manière randomisée. En forçant les diverses zones à envoyer des données et observer depuis le serveur les résultats.

Venant d'une formation *Automatique/Electronique*, nos connaissances dans le langage *JavaScript* étaient assez limitée et nous n'avons pas réussi à faire fonctionner les divers scripts donnés pour justement faire ces scénarios. C'est aussi l'une des raisons pour laquelle nous avions décidé de réaliser un petit programme dans le langage *Golang* pour faire cette tâche. Cependant nous avons compris le principe et les idées, et il pourrait nous être possible à l'avenir de faire tout cela.

4.5 Validation

Nous avons au final une application avec des fonctionnalités de monitoring fonctionnant dans son entièreté, avec un peu plus de temps nous aurions pu trouver le problème lier à l'adaptation du débit et réaliser un scénario d'automatisation.

3. **Open vSwitch** : stack logiciel pour les switch, réalisé spécifiquement pour les environnements réseaux virtuel

Conclusion

Dans le cadre du master *REOC*, nous avons simulé une infrastructure réseau, surveiller cette dernière et mis en place une adaptation en fonction des besoins clients. En utilisant des technologies telles que *Docker* pour créer instances de périphériques réseaux, *Containernet* qui permet d'utiliser ces derniers dans une simulation *Mininet*. Ainsi que *Open vSwitch*/*VimEmu*/*Ryu* qui sont des technologies basées sur **SDN** et **VNF** pour contrôler l'intelligence du réseau. Le but de ce projet était de réaliser les tâches suivantes :

1. **Déploiement de l'ordonnanceur** : Coordination efficace des communications entre les zones Z_1, Z_2, Z_3 et la passerelle G_I.
2. **Rejet** : Implémentation de mécanismes de rejet pour contrôler le flux de données et éviter la congestion.
3. **Monitoring** : Mise en place d'un système de surveillance en temps réel des performances et de l'état des ports réseau.

Les technologies basées sur **SDN** et **VNF** permettent de modifier le réseau en temps réel autant au niveau fonctionnel que pour simuler des comportements.

Ce projet visait donc à mettre en lumière ces technologies pour régler des problèmes de congestions qui ne peuvent pas être réglés à l'aide des couches du modèle OSI standard. la *QoS*[4] est essentielle pour la Z_1, mais cela doit être dynamique, le but derrière ce projet était donc de mettre en avant la nature adaptative de tels principes. Nous avons pu voir l'utilité de **SDN** lorsque les propriétés des paquets ont été récupérés, ou encore quand les paquets ont été supprimés. Pour **VNF**, nos Switch S_x sont des conteneurs qui peuvent modifier les paquets reçus comme ils le souhaitent. Ce qui permet un contrôle complet et avec l'application *Javascript* donné c'est exactement ce qui est mis en avant.

Pour conclure, le projet était assez dense et posait des problématiques dans plusieurs niveaux. Réaliser la topologie réseau était la première étape et nous a pris quelques temps, en effet l'API *Containernet* ne nous étant pas forcément familière, il a fallu lire de la documentation et comprendre comment la topologie devait se poser dans notre scénario. Réaliser le monitoring nous était au début assez flou, mais en observant comment l'API REST *Ryu* fonctionnait, certains principes nous sont venus assez facilement. L'idée de faire une application graphique pour chaque tâche a donc été fait. Le projet ensuite s'est déroulé de manière assez linéaire, nous cherchions à faire fonctionner tous les scénarios possible, afficher les fonctionnalités demandés. Nous sommes plutôt content de notre travail mais nous aurions voulu avoir plus de temps pour pouvoir vraiment rendre un projet complet. Cependant, nous avons beaucoup appris dans ce projet, la virtualisation réseau peut être réellement utile pour tester des protocoles en simulations ou encore voir si certaines topologies sont possibles. De plus l'utilisation d'applications aussi complètes permet de mettre en lumière la possibilité de nouveaux réseaux entièrement logiciel.

Références

[1] M. Peuster, H. Karl et S. van Rossem. "MeDICINE : Rapid prototyping of production-ready network services in multi-PoP environments". In : *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. Nov. 2016, p. 148-153. DOI : 10.1109/NFV-SDN.2016.7919490.

[2] *Ryu SDN Framework*. URL : https://ryu-sdn.org/ (visité le 15/01/2025).

[3] Nick McKeown et al. "OpenFlow : enabling innovation in campus networks". In : *SIGCOMM Comput. Commun. Rev.* 38.2 (31 mars 2008), p. 69-74. ISSN : 0146-4833. DOI : 10.1145/1355734.1355746. URL : https://doi.org/10.1145/1355734.1355746 (visité le 15/01/2025).

[4] *ovs-vsctl(8) - Linux manual page*. URL : https://www.man7.org/linux/man-pages/man8/ovs-vsctl.8.html (visité le 15/01/2025).

4. **QoS** : Qualité de Service