



## Software Defined Communication Infrastructure

**Systèmes distribués et big data - INSA Toulouse - 2023** [Théo Fontana](#) , [Jose Organista](#)

**Ce document est un export pdf du readme de notre projet git. Il est disponible avec une mise en page plus agréable et les ressources accessible au lien suivant <https://github.com/TheoFontana/SDCI>**

## Presentation du Projet

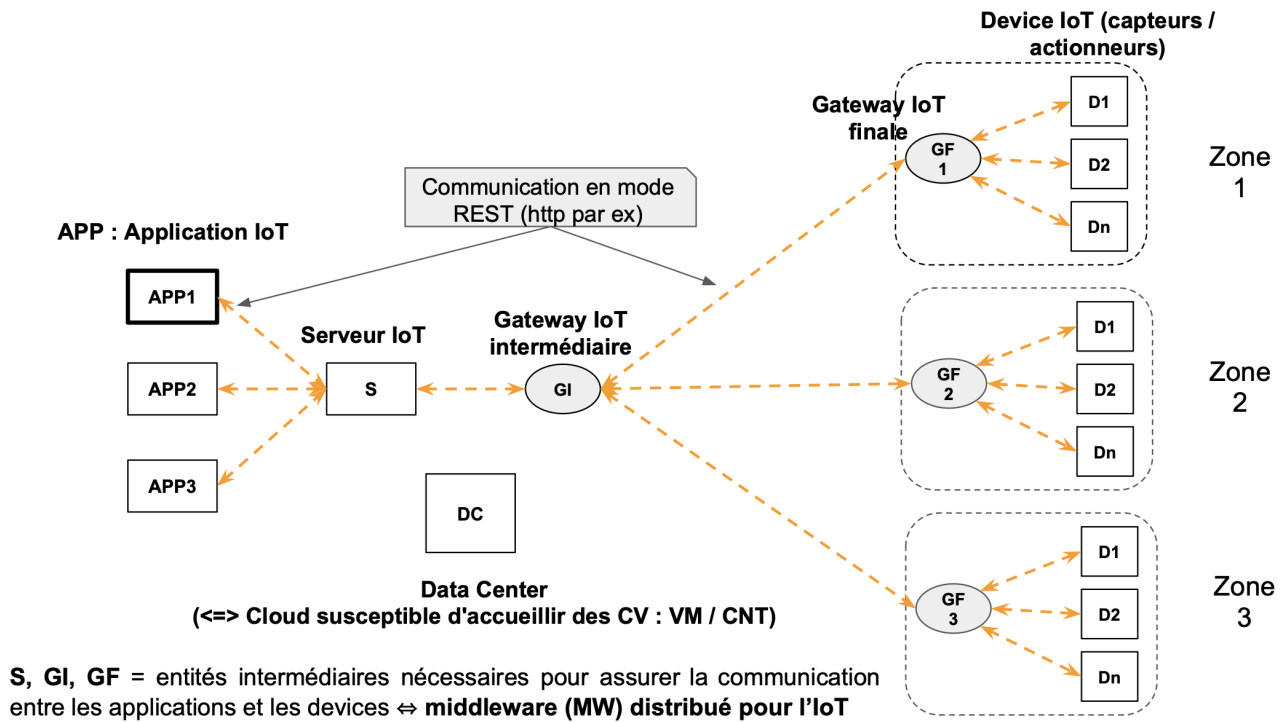
Ce projet est réalisé dans le cadre de la mineur SDCI en dernière année de l'école d'ingénieur INSA en spécialité systèmes distribués et big data.

### Objectifs

- Déployer dynamiquement et de façon transparente des fonctions de réseau virtuelles (VNF)
  - permettant de répondre aux besoins fonctionnels et/ou non fonctionnels d'applications distribuées relevant d'une activité de l'Internet des objets (IoT)
  - en appliquant les concepts et techniques relevant de la virtualisation de fonctions de réseau (NFV) et des réseaux pilotables par le logiciel (SDN)
- Développer une approche de gestion autonome de la mise en œuvre des VNF ciblées via le concept de l'Autonomic Computing (AC)

### Activité IoT ciblée

Nous ciblons une activité de supervision/intervention à distance sur différentes zones dotées de capteurs / actionneurs, par le biais d'applications



En cas d'incident dans une zone du trafic supplémentaire est généré par ses capteurs / actionneurs. Ceci peut entraîner la saturation de la gateway intermédiaire (GI) générant ainsi une baisse de performances incompatible avec les besoins en QoS des applications.

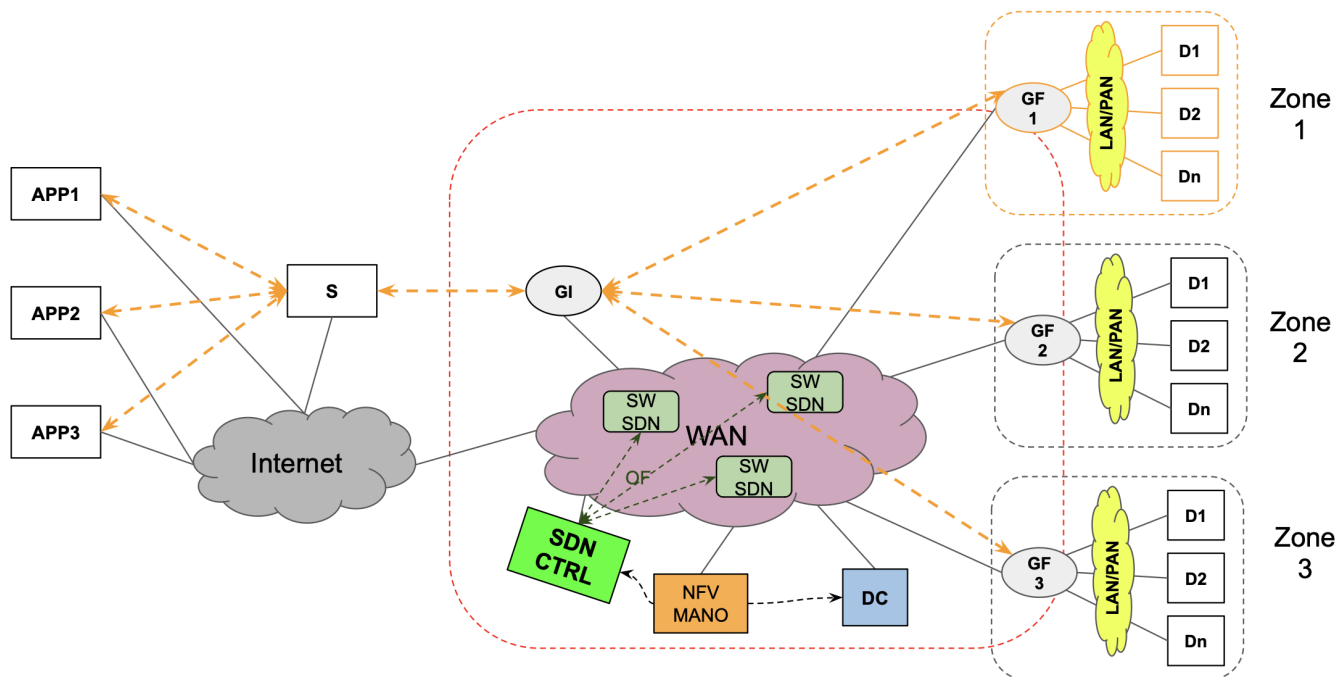
Une phase d'adaptation est alors nécessaire pour rétablir les performances. Plusieurs stratégies peuvent être adoptées :

- Déployer une seconde gateway sous forme de VNF et rediriger le trafic provenant de la zone 1 (ou des zones 2 et 3) vers cette gateway.
- Déployer une VNF d'ordonnancement différencié priorisant le trafic issu de GF1.
- Supprimer les flux de données en provenance de la zone 2 et 3.
- Déployer d'un loadbalancer sous forme de VNF.

## Vision IT de l'activité ciblée

### Hypothèse sur l'infrastructure IT

- GI, GF et DC sont connectés via un réseau grande distance (WAN) géré par un opérateur dont la portée d'action inclut : les nœuds internes du réseau (switch), les nœuds middleware (GI et GF) et le DC
- Un orchestrateur de VNF (VNF-ORCH) est connecté au WAN : il permet de déployer des VNF sur le DC et de gérer leur cycle de vie.
- Le WAN est doté de capacités SDN :
  - Ses nœuds internes sont des switch SDN programmables via Open Flow
  - Il inclut un contrôleur SDN interagissant avec les switch SDN via Open Flow



## Plateforme et outils mis à disposition

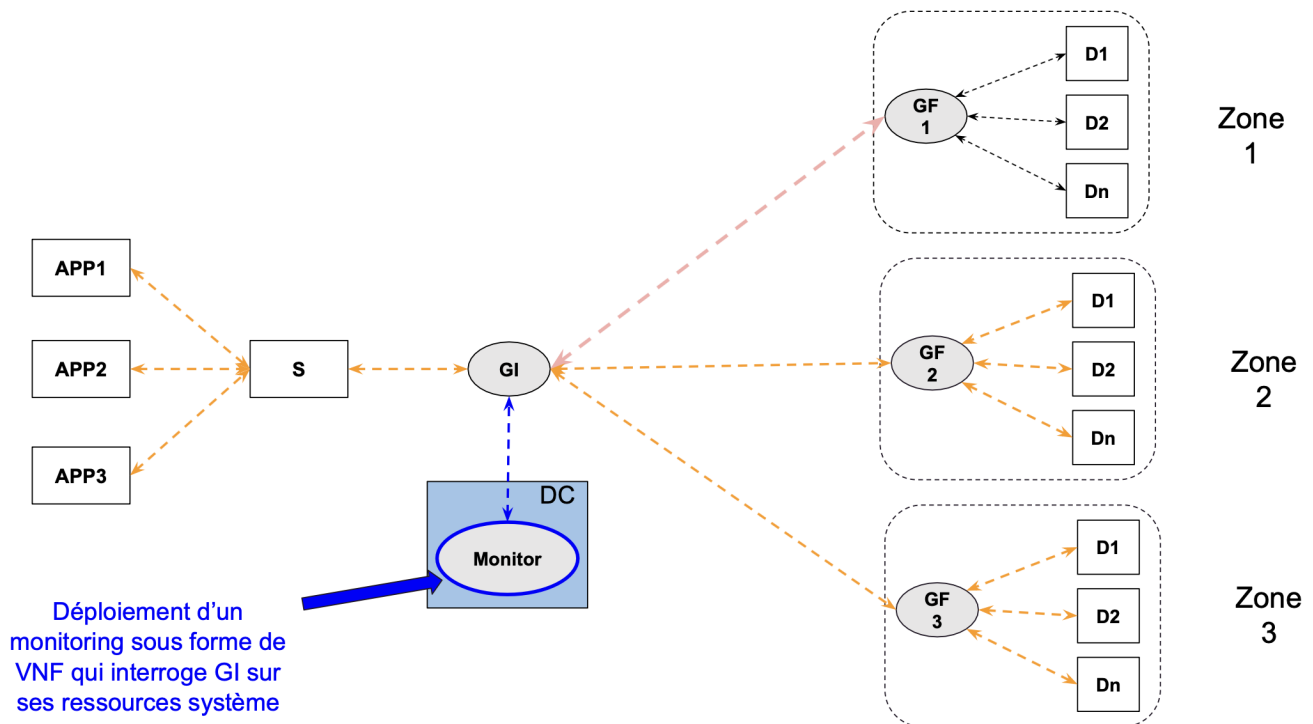
- Plateforme d'émulation de réseau : [ContainerNet](#)
- Contrôleur SDN : [RYU](#)
  - [documentation API](#)
- MANO standardisé ETSI NFV : [OSM](#)
  - [documentation API vim-emu](#)
- Middleware IoT/M2M en NodeJS (see [Middleware](#))

## Travail demandé

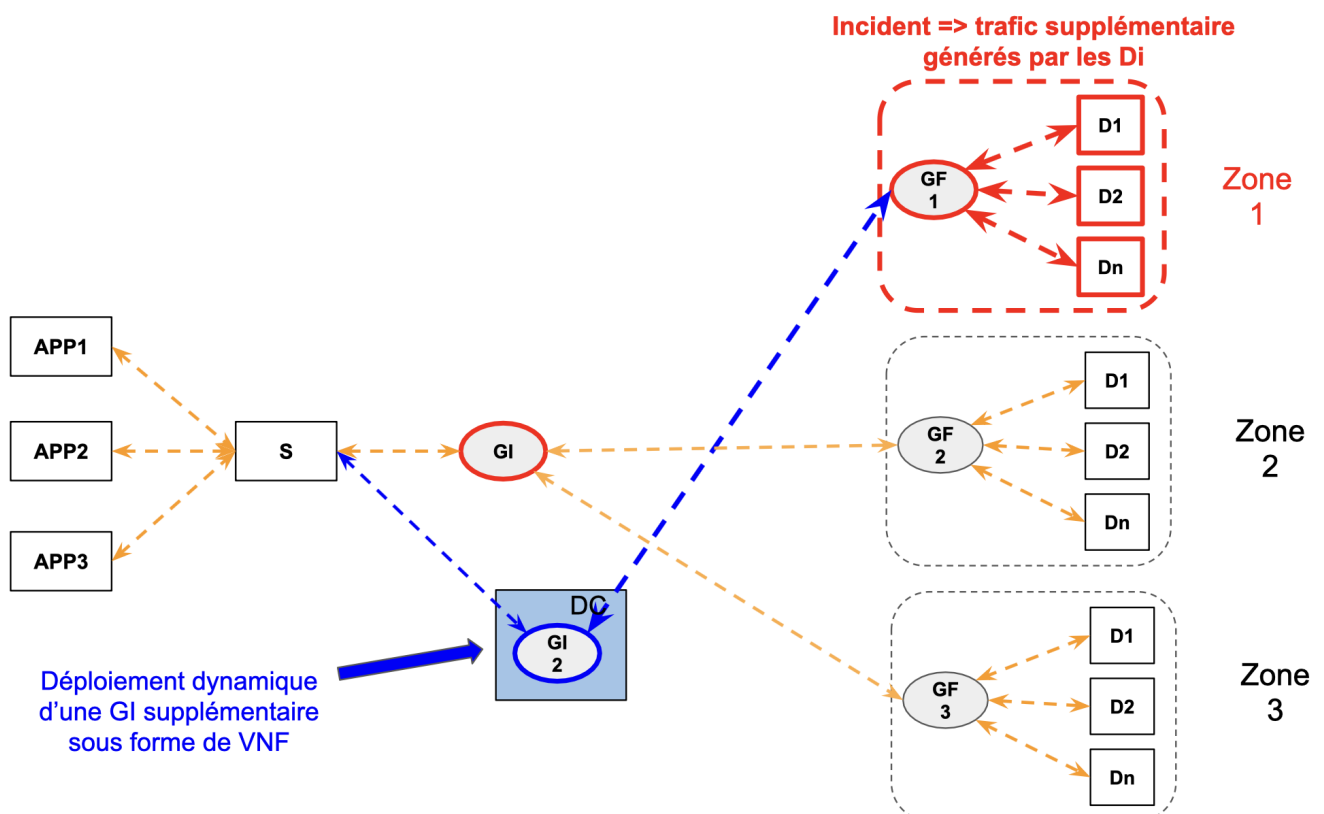
Mettre en place l'adaptation requise lorsque la gateway intermédiaire est saturée, suivant le cadre de l'Autonomic Computing

## Use cases étudiée

Notre groupe avait pour mission de monitorer la gateway intermédiaire pour surveiller sa charge à partir de métrique système tel que la charge du CPU.



Nous devons ensuite, en cas de dégradation des performances déployer une nouvelle gateway et rediriger le trafic en provenance de la zone 1 vers cette dernière. Le trafic de la zone 2 et 3 continuant d'utiliser la gateway initiale.



## Conception des solutions

### Composants en jeu

Nous disposons d'un Mano qui nous expose un service permettant de déployer et d'arrêter des VNF dans un datacenter via des requêtes sur son API REST.

Nous avons egalement un controller SDN qui nous permet de mettre à jour les tables SDN des differents switch de notre reéseau via son API REST.

Les interactions entre notre general controller, le MANO et le SND controller sont resumés dans le diagramme de structure composite suivant

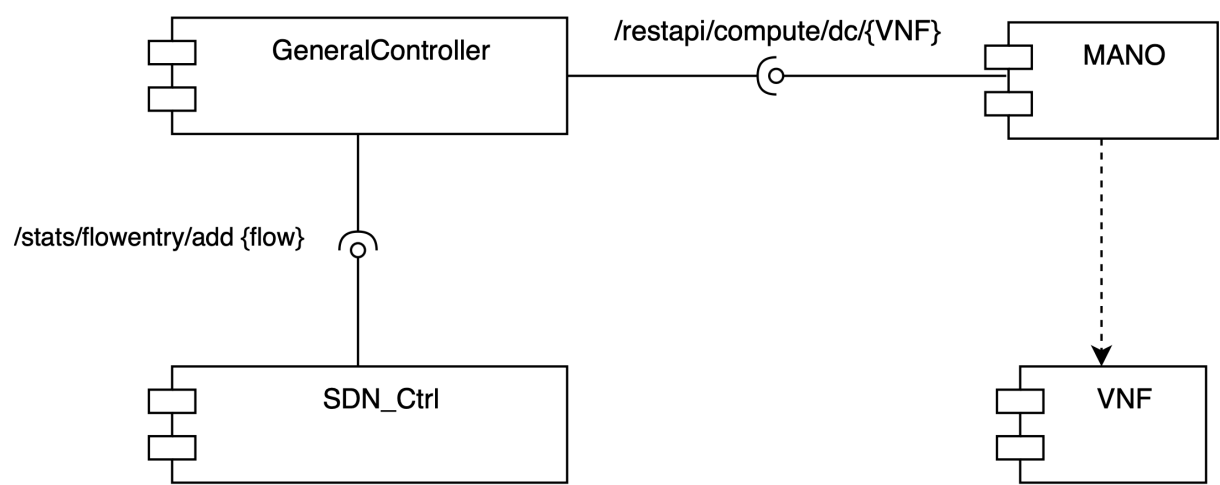


Diagramme de structure composite

Monitoring

Pour le monitoring nous proposons de déployer la VNF de monitoring au démarrage du general controller. Une fois que celui ci à eu la confirmation que la VNF est correctement déployé, il l'interroge periodiquement pour récupérer les informations système de la gateway intermediaire. Il verifie à chaque iteration que le système n'est pas en surcharge.

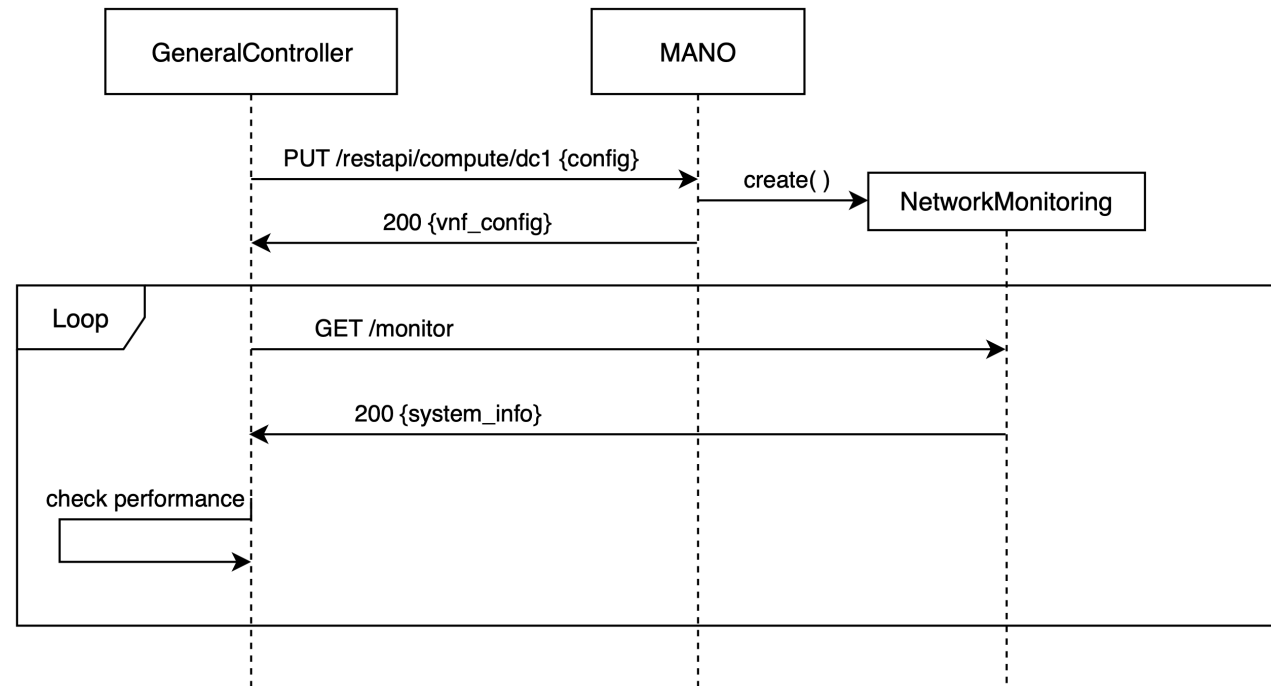


Diagramme de séquence du monitoring

Adaptation

Pour l'adaptation, notre general controller devra demander le déploiement d'une nouvelle gateway dans le datacenter via l'API du MANO. Si ce deploiment s'est bien déroulé, il demande la redirection du trafic de la zone 1 en direction de cette VNF grace à l'API du controlleur SDN.

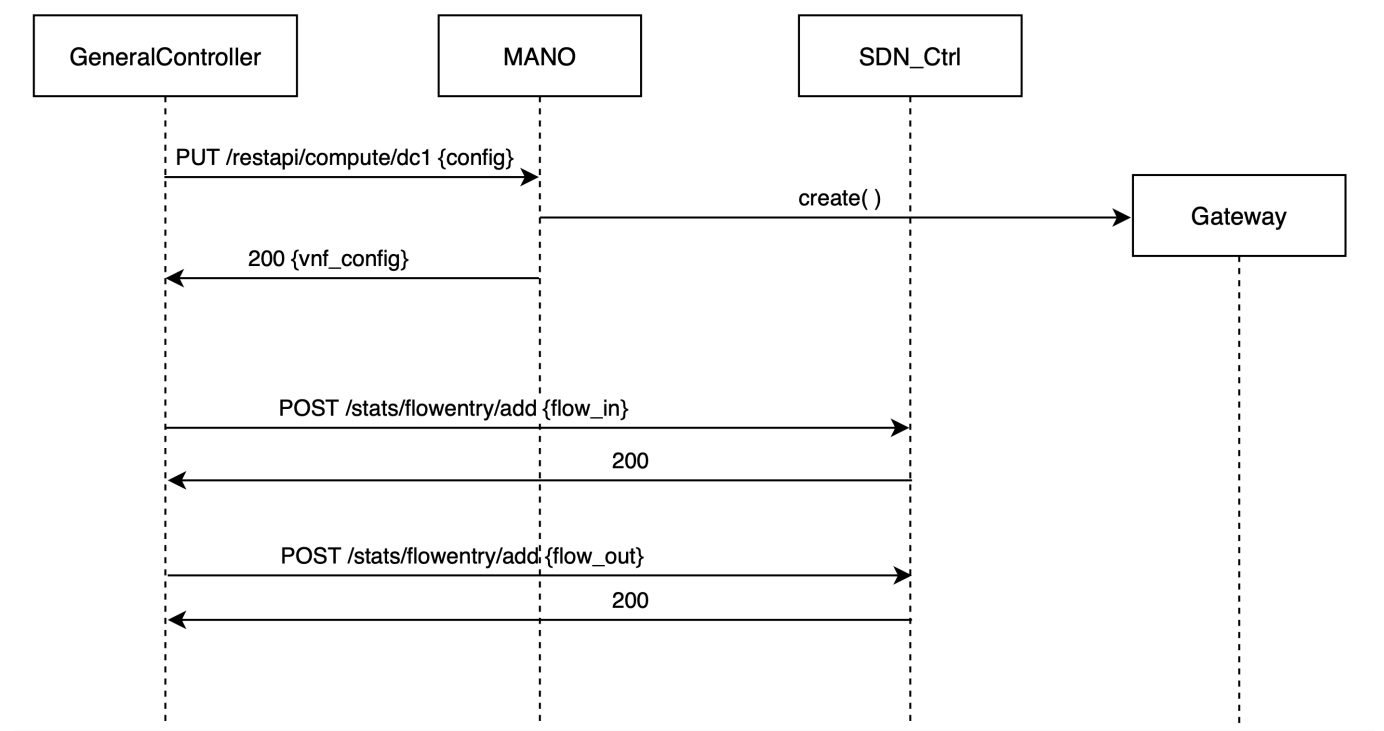
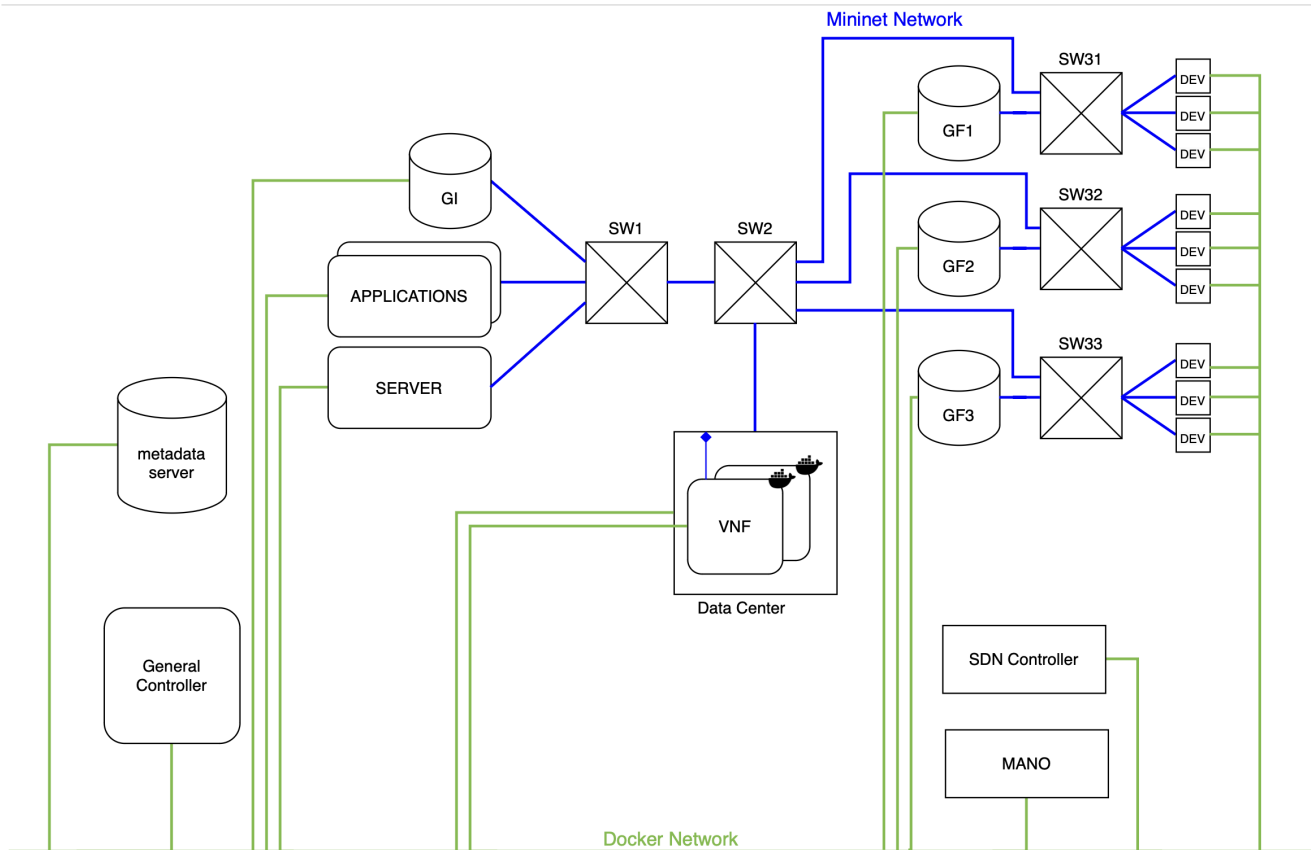


Diagramme de séquence de l'adpatation

### Choix d'implementation

Topologie déployé

Nous avons choisi de déployer le reseau suivant



Le reseau bleu est le réseau émulé mininet. Nous avons choisi de représenter les différentes zones avec un switch simulant un LAN.

Le reseau vert représente le réseaux VLAN Docker reliant tous nos containers. Il est utilisé pour assurer la communication entre :

- les noeuds middleware et le metadata serveur
- le GC, le Mano, le controlleur SDN et les VNFs

Pour le déploiement des differents noeuds middleware, nous avons créé un unique Dockerfile permettant de créer l'image associée au noeud. Celui ci recupère d'identifiant de l'instance à déployer en variable d'environnement et lance un script de démarrage spécifique en fonction du type d'instance lorsque le container est lancé.

```
FROM ubuntu:trusty

ARG SCRIPT
ARG NODE_VERSION=14
ENV INSTANCE_ID=''
...
ADD $SCRIPT .
...
ENTRYPOINT sh /componnent/$SCRIPT && /bin/sh
```

Extrait du [Dockerfile](#)

Le script de démarrage est chargé de récupérer la configuration de l'instance sur le metadata serveur pour pouvoir lancer le service avec les bons paramètres.

```
curl -o conf.json metadata_server/$INSTANCE_ID

LOCAL_NAME=`cat conf.json | jq '.local_name'`
LOCAL_PORT=`cat conf.json | jq -r '.local_port'`
LOCAL_IP=`cat conf.json | jq '.local_ip'`
FILE_URL=`cat conf.json | jq -r '.file_URL'`

curl -LO $FILE_URL
node server.js --local_ip $LOCAL_IP --local_port $LOCAL_PORT --local_name $LOCAL_NAME
```

Exemple de script de démarrage pour le serveur [start\\_server.sh](#)

## Metadata serveur

Nous avons réalisé le metadata serveur en Node.js. Il renvoie la configuration de l'instance à deployer suite à une requête **GET** sur l'identifiant de l'instance souhaité.

```
app.get('/:id', function(req, res) {
  var id = req.params.id;
  var conf_instance = config[id];
  if (conf_instance)
    res.status(E_OK).send(JSON.stringify(conf_instance));
  else
    res.sendStatus(E_NOT_FOUND);
});
```

Extrait de [metadata\\_server.js](#)

L'ensemble des configurations est stocké dans un fichier général de configuration json.

```
{
  ...
  "gw1": {
    "local_ip": "10.1.0.11",
    "local_port": 8282,
    "local_name": "gw1",
    "remote_ip": "10.1.0.10",
    "remote_port": 8181,
    "remote_name": "gwi",
    "file_URL": "https://homepages.laas.fr/smedjiah/tmp/mw/gateway.js"
  },
  ...
}
```



Extrait de [config.json](#)

## General controlleur

Nous avons choisi de ne pas utiliser le squelette de general controller fourni mais de développer nous même un prototype plus simple en Python afin de nous faciliter le développement et les tests.

## Monitoring

Notre strategie de monitoring est pour l'instant assez simple. Lorsque notre VNF recoit une requette **GET** de la part du general controller elle interroge la gateway sur son endpoint **/health** et retourne la reponse reçu au GC. Cette strategie nous permet de déplacer le traitement de la réponse au niveau du GC celui si peut donc choisir à quel rythme monitorer ce qui peut potentiellement reduire la charge sur la gateway.

```
app.get('/monitor', function(req, res) {
  request({method: 'GET', uri: `http://10.1.0.10:8181/health`}, (error,
response, body) => {
    if (!error && response.statusCode == 200){
      res.send(body);
    } else {
      res.send(error);
    }
  });
});
```

Extrait de [monitor.js](#)

Pour deployer la VNF, nous utilsons l'API REST de vim-emu

```
def start_monitoring():
  # URL to add new vnf
  url = 'http://127.0.0.1:5001/restapi/compute/dc1/vnf_monitor'
  headers = {'Content-type': 'application/json'}
  d = {"image":"vnf_monitor:0.2", "network":""
(id=vnf_monitor,ip=10.1.0.100/24)}
  r = requests.put(url, headers = headers, data = json.dumps(d))
  return r.status_code, r.json()
```

Extrait de [controller.py](#)

Nous avons chois de basé notre monitoring sur la metrique **currentLoadSystem** car c'est celle qui variait le plus rapidement lorsque nous simulons une charge sur la gateway durant nos tests. Lorsque celle ci depasse le seuil fixé, nous devons deployer notre VNF d'adpatation.

## Adaptation

La première étape de l'adaptation est de déployer une nouvelle gateway intermediaire dans le datacenter en utilisant l'API REST de vim-emu. l'image de la gateway intermediaire precedement construite à du être

légèrement modifié pour qu'elle concovienne aux requirements de vim-emu :

- Le serveur node.js doit tourner de background
- Les sripts de démarrage et d'arrêt de la VNF doivent être passé en variable d'environnement dans le Dockerfile.

Nous devons ensuite rediriger le trafic de la gateway final de la zone 1 à direction de la gateway intermédiaire vers notre VNF.

Nous avons pris la décision d'identider ces flux avec uniquement les adresses IP source et destination. En effet le seul trafic circulant sur notre réseaux entre ces instance est le trafic applicatif que nous souhaions rediriger. Si ce n'était pas le cas nous aurions également du utiliser les numéros de port pour identifier ces flux.

Nous devons donc :

- modifier l'adresse IP destination des paquet provennat de **GF1** en direction de **GI** (*aller*)
- modifier d'adresse IP srouce des paquets provenant de la **GI\_VNF** en direction en direction de **GF1** (*retour*)

Cela est réalisé en ajoutant des *flow* dans la table SDN du switch 2 à l'aide de l'API de controller SDN de la façon suivante (pour l'aller)

```
curl -X POST -d '{
  "dpid": 2,
  "table_id":0,
  "priority":11111,
  "match":{
    "nw_src": "10.1.0.11",
    "nw_dst": "10.1.0.10",
    "dl_type": "2048",

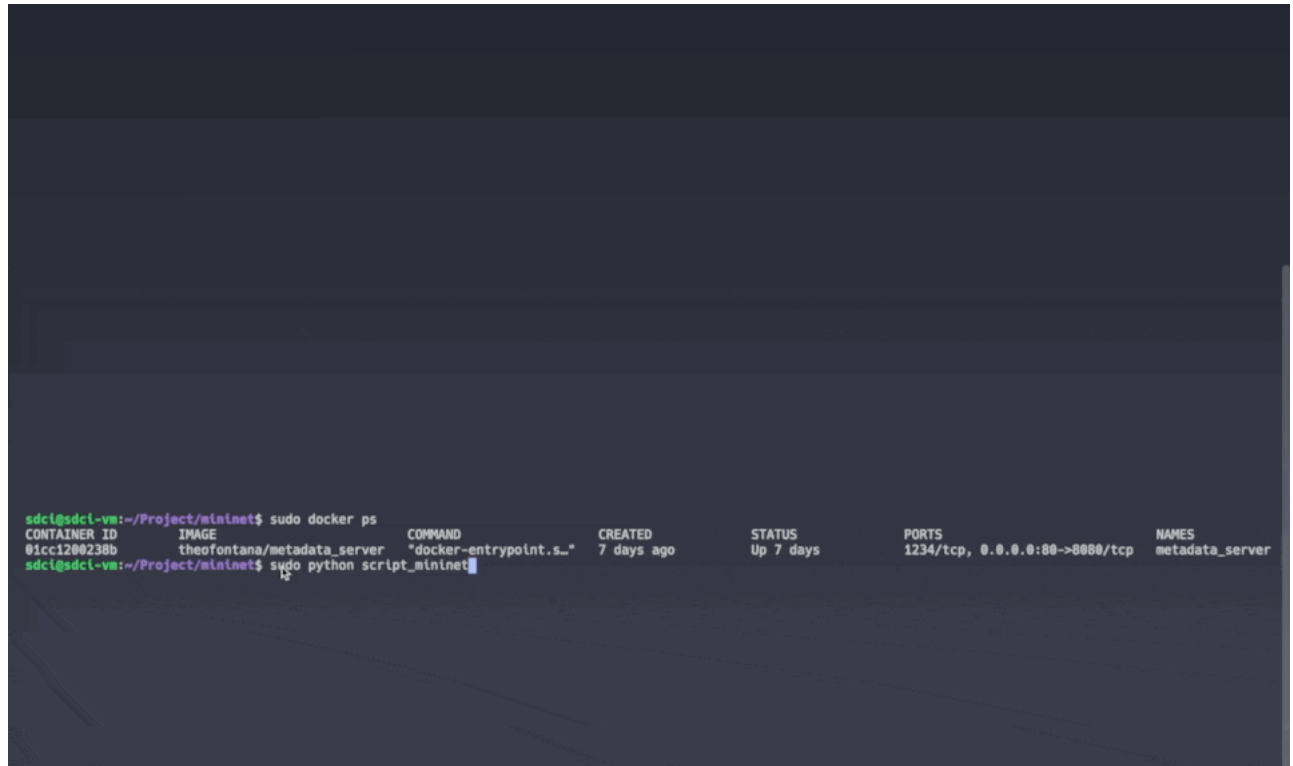
  },
  "actions":[
    {
      "type": "SET_FIELD",
      "field": "ipv4_dst",
      "value": "10.1.0.60"
    },
    {
      "type": "OUTPUT",
      "port": "NORMAL"
    }
  ]
}' http://localhost:8080/stats/flowentry/add
```

Extarit de [redirect\\_gwi\\_to\\_vnf.sh](#)

## Sénario de démonstation

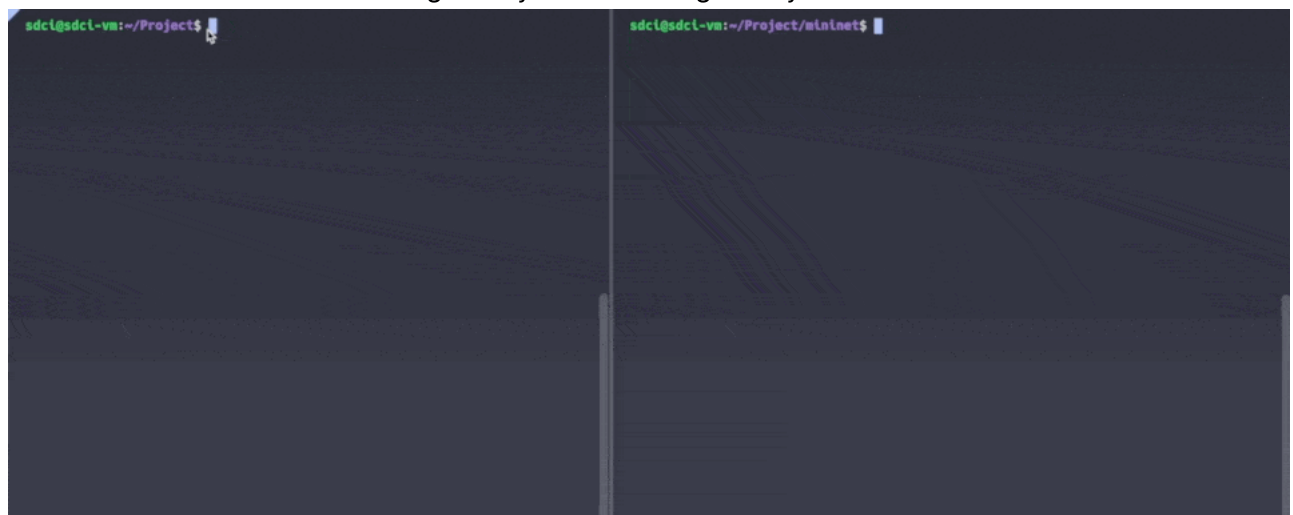
Pour notre démonstration nous souhaitons

### 1. Lancer notre topologie avec mininet



```
sdci@sdci-vm:~/Project/mininet$ sudo docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED        STATUS        PORTS                    NAMES
01cc1200238b   theofontana/metadata_server   "docker-entrypoint.s..."   7 days ago    Up 7 days    1234/tcp, 0.0.0.0:80->8080/tcp   metadata_server
sdci@sdci-vm:~/Project/mininet$ sudo python script_mininet
```

### 2. Tester la communication entre la gateway finale 1 et la gateway intermediaire



```
sdci@sdci-vm:~/Project$
sdci@sdci-vm:~/Project/mininet$
```

3. Démarrer notre general controller et voir que le monitoring se lance.

The left terminal window shows the output of a script that starts a gateway. It displays statistics for a data center (d) and a server (s). The gateway is successfully started and is listening on port 8181.

```
sdcl@sdcl-vm:~/Project/GC$
```

	0	189	100	189	0	0	43498	0	--:--:--	--:--:--	--:--:--	0.3800
ent	% Total	% Received	% Xferd	Average Speed	Time	Time	Time	Time	Time	Time	Time	Time
d	0	3516	100	3516	0	0	77770	0	--:--:--	--:--:--	--:--:--	10
s	0	3516	100	3516	0	0	77770	0	--:--:--	--:--:--	--:--:--	78133

```
*** GATEWAY SUCCESSFULLY STARTED ***
-----
LOCAL_NAME      "gw1"
LOCAL_IP        "10.1.0.10"
LOCAL_PORT      8181
REMOTE_NAME     "srv"
REMOTE_IP       "10.1.0.1"
REMOTE_PORT     8080
-----
undefined
"gw1" listening on : 8181
{}
{}
{}

```

The right terminal window shows the execution of a Docker container named 'mn.gwf\_1' which runs a bash shell. It performs several curl and ping commands to test connectivity.

```
sdcl@sdcl-vm:~/Project/mininet$ sudo docker exec -it mn.gwf_1 bash
root@gwf_1:/component# curl 10.1.0.10:8181/ping
{"pong":1675423328522}root@gwf_1:/component# curl 10.1.0.10:8181/ping
{"pong":1675423331846}root@gwf_1:/component# curl 10.1.0.10:8181/ping
{"pong":167542333746}root@gwf_1:/component#
```

4. Générer un fort trafic depuis la gateway final 1 vers la gateway intermediaire
5. Observer que le General controller detecte une dégradation des performance sur la gateway et lance une nouvelle gateway intermediaire dans le data center.
6. Observer que le trafic générer est redirigé et n'arrive plus à la gateway intermediaire mais à la VNF.
7. Vérifier que ces opérations ont été transparentes au niveau applicatif.

The left terminal window shows the output of a Python script that monitors the system. It displays various load and resource metrics for the gateway and the server.

```
sdcl@sdcl-vm:~/Project/GC$ python controller.py
({'docker_network', 'u', '172.17.0.17'})
({'avgLoad', 0.07})
({'currentLoadSystem', 0.43442279002794837})
```

```
{
  loadIrq: 0,
  rawLoad: 10916330,
  rawLoadUser: 7953910,
  rawLoadSystem: 2919260,
  rawLoadNice: 43160,
  rawLoadIdle: 662707320,
  rawLoadIrq: 0
},
{
  load: 1.6256072757193971,
  loadUser: 1.184386933223733,
  loadSystem: 0.43547839632814567,
  loadNice: 0.005741946167518742,
  loadIdle: 98.3743927242806,
  loadIrq: 0,
  rawLoad: 10956390,
  rawLoadUser: 7982620,
  rawLoadSystem: 2935070,
  rawLoadNice: 38700,
  rawLoadIdle: 663031120,
  rawLoadIrq: 0
}
}
```

The right terminal window shows the output of a script that configures the system. It displays various debug messages and the configuration of the gateway and the server.

```
DEBUG:dcemulator.net:addLink: n1=s31 intf1=s31-eth6 -- n2=gwf_2 intf2=gwf_2-eth0
DEBUG:dcemulator.net:addLink: n1=s32 intf1=s32-eth2 -- n2=dev_4 intf2=dev_4-eth0
DEBUG:dcemulator.net:addLink: n1=s32 intf1=s32-eth3 -- n2=dev_5 intf2=dev_5-eth0
DEBUG:dcemulator.net:addLink: n1=s32 intf1=s32-eth4 -- n2=dev_6 intf2=dev_6-eth0
DEBUG:dcemulator.net:addLink: n1=s31 intf1=s31-eth7 -- n2=gwf_3 intf2=gwf_3-eth0
DEBUG:dcemulator.net:addLink: n1=s32 intf1=s32-eth5 -- n2=dev_7 intf2=dev_7-eth0
DEBUG:dcemulator.net:addLink: n1=s32 intf1=s32-eth6 -- n2=dev_8 intf2=dev_8-eth0
DEBUG:dcemulator.net:addLink: n1=s32 intf1=s32-eth7 -- n2=dev_9 intf2=dev_9-eth0
*** Configuring hosts
root srv gw1 gwf_1 gwf_2 gwf_3 dev_1 dev_2 dev_3 dev_4 dev_5 dev_6 dev_7 dev_8 dev_9
*** Starting controller
c0
*** Starting 7 switches
dcl.s1 fs1 s1 s2 s31 s32 s33 ...
*** Starting CLI:
containernet> DEBUG:dcemulator.node:Starting compute instance u'vnf_monitor' in d
ata center 'dcl'
vnf_monitor: kwargs {'environment': {'VNF_NAME': 'u'vnf_monitor'}, 'datacenter': d
cl, 'flavor_name': 'tiny', 'ip': '10.0.0.29/8'}
vnf_monitor: update resources {'cpu_quota': -1}
DEBUG:dcemulator.net:addLink: n1=vnf_monitor intf1=vnf-monitor -- n2=dcl.s1 intf2
=dcl.s1-eth2
*** vnf_monitor : { './start_vnf.sh', }
```

```
root@gwf_1:/component# curl 10.1.0.10:8181/ping
{"pong":1675424761582}root@gwf_1:/component# curl 10.1.0.10:8181/ping
{"pong":1675424763030}root@gwf_1:/component# curl 10.1.0.10:8181/ping
{"pong":1675424763937}root@gwf_1:/component# curl 10.1.0.10:8181/ping
{"pong":1675424764967}root@gwf_1:/component#
```

demo pour les points 4, 5, 6 et 7

## Axes d'ameliorations

- Actuellement lors de la redirection du trafic le trafic de *retour* entre la VNF gateway intermediaire et la gateway finale est adressé à la gateway intermediaire au niveau MAC. Nous n'avons pas pu debugger ce problème qui fait que la GWI reste saturé même après la redirection de trafic effectué.
- Il nous faudrait ensuite ajouter une strategie pour revenir au cas nominal en supprimant la VNF deployé une fois que le trafic redevient normal.
- Il pourrait également être interessant de monitorer la VNF deployer pour s'assurer que celle ci ne soit pas non plus en surcharge et possiblemnt deployer une nouvelle gateway intermediaire avec un load balancer en cas de problème.

## Conclusion

Ce projet a été l'occasion de nous familiariser avec les concepts de l'autonomus computing dans un contexte IoT où les applications ont des besoin en QoS et génèrent un trafic varibale. Nous avons pu développer et déployer dynamiquemnt des VNFs en charge de surveiller l'état d'instances sur le reseau et d'assurer des performances sufisante aux applications. Grace à SDN nous avons pu dynamiquement modifier le routage au sein de notre réseaux de manière transparente pour les applications.

Nous aurions cependant aimer pouvoir aprofondir la partie SDN pour pouvoir definir plus finement les flux de communications et aller plus loin dans les strategies mise en oeuvre pour s'adapter a la dégradation de l'état du middleware.