**Cloud Computing Project**

**Date :** 27 avril 2025  
 **Étudiant :** Kenny GAYAKPA

**Sujet : Déploiement de Docker et Kollaps – Résumé d’installation et de configuration**

Use [http://kollaps.dev](http://kollaps.dev/) to emulate cloud systems, playing with different network topologies, using containerized systems.

**Déploiement de Docker et Kollaps – Résumé d'Installation**

### **1. Installation de Docker et Kollaps**

J’ai procédé à l’installation de tous les paquets requis pour faire fonctionner Docker et Kollaps, en suivant les instructions officielles disponibles sur leurs sites respectifs.  
 Quelques erreurs sont survenues pendant l’installation (liées principalement à des dépendances manquantes ou des problèmes de permissions), mais elles ont été résolues avec succès.

### **2. Vérification de l’installation**

Afin de m’assurer du bon fonctionnement de l’environnement, j’ai réalisé plusieurs vérifications :

* **Version de Docker :** docker --version  
   → Confirme que Docker est bien installé.
* **État du service Docker :** docker info  
   → Donne des informations détaillées sur l’état du service.
* **Version de Docker Compose :** docker compose version  
   → Vérifie la présence de Docker Compose, essentiel pour exécuter des configurations à l’aide d’un fichier docker-compose.yml.
* **Conteneurs actifs :** docker ps  
   → Permet de voir les conteneurs en cours d’exécution, notamment ceux liés à Kollaps.
* **Réseaux Docker :** docker network ls  
   → Affiche les réseaux définis dans Docker.
* **Test de l’état de Kollaps :** curl http://localhost:8080  
   → Permet de vérifier si Kollaps répond correctement. mais échoue

### **3. Lancement des services**

Avant de démarrer l’expérience, les services nécessaires ont été lancés :

* Démarrage de Docker :  
   sudo systemctl start docker
* Démarrage de Kollaps via Docker Compose :  
   docker compose up

### **4. Génération du fichier topology.yaml**

Un fichier topology.yaml est requis pour le déploiement via Docker Swarm. Il est généré à partir du fichier topology.xml en utilisant l’outil **KollapsDeploymentGenerator** (fourni avec Kollaps).  
 Cet outil convertit une topologie décrite en XML en un fichier YAML

### **5. Déploiement via Docker Swarm**

La commande suivante permet de déployer la stack définie dans topology.yaml :

docker stack deploy -c topology.yaml kollaps\_example

Cela crée une stack Docker nommée kollaps\_example, c’est-à-dire un ensemble coordonné de services déployés sur un réseau interne.

**ici la descripètion du fichier xml << topology.xml>> dans kollapse**

Cette topologie définit une simulation réseau avec Kollaps, où trois clients (client1, client2, client3) envoient du trafic vers un serveur central via deux commutateurs (s1 et s2). Les clients sont connectés à s1 avec des latences et débits différents, puis s1 est relié à s2, qui connecte finalement au serveur. Le client3 a un débit plus faible, simulant une connexion limitée. Un service de supervision (dashboard) est également présent. Aucune dynamique réseau (pannes ou changements) n’est définie pour l’instant.

**Date :** 04 mai 2025  
**Étudiant :** **Kenny GAYAKPA**

### **Contexte**

Cette semaine, nous avons tenté de lancer l’interface **dashboard** de **Kollaps**, qui permet de visualiser les topologies réseau et leurs configurations. Cette étape est cruciale pour valider l’installation et les scénarios de simulation.

### **Problème rencontré**

Après l’exécution de :

docker stack deploy -c topology.yaml kollaps\_example

l’interface web du **dashboard** (accessible théoriquement via http://localhost:8088/) reste inaccessible.

### **Avancement**

#### **Installation complète**

Toutes les étapes préalables ont été réalisées sans message d’erreur.

#### **Génération du fichier YAML**

Le fichier topology.yaml a été généré avec succès à partir du fichier XML de topologie.

#### **Vérifications en cours**

* **État des services Docker Swarm :**
  + docker service ls
  + docker stack ps kollaps\_example  
     → Ces commandes permettent de vérifier si kollaps\_example\_dashboard est en état **Running**.
* **Analyse des logs :**
  + docker service logs kollaps\_example\_dashboard
  + docker logs <ID\_du\_container\_dashboard>  
     → Pour diagnostiquer d’éventuels problèmes de lancement ou de liaison réseau.
* **Réseaux Docker :** Vérification que :
  + le réseau kollaps\_network est bien de type overlay et attachable,
  + aucun pare-feu ou conflit de port ne bloque l’accès à l’interface.

### **Prochaines étapes**

1. **Re-générer** le fichier topology.yaml avec KollapsDeploymentGenerator

**Re-déployer** le stack après suppression :  
docker stack rm kollaps\_example

docker stack deploy -c topology.yaml kollaps\_example

1. **Analyser en détail les logs** du service dashboard
2. **Tester à nouveau l’accès à l’interface** http://localhost:8088/  
   **Réinitialisation complète de l’environnement Docker** si nécessaire
3. **Migration vers Kubernetes :**
   * Installer Kubernetes
   * Reprendre les étapes de génération et déploiement
   * Comparer la stabilité et les performances du dashboard sous Kubernetes

### **TODO – Semaine prochaine**

* Vérifier le fonctionnement du dashboard avec Kubernetes
* Répartir avec mon binôme les topologies à tester
* Lancer plusieurs simulations
* Analyser les résultats des expériences
* Préparer un **résumé des observations** pour la prochaine réunion

**Date :** 11 mai 2025  
**Étudiant :** **Kenny GAYAKPA**

### **Reprise de l’installation de Docker sur une machine distante (Ubuntu)**

### **Contexte**

Après une tentative d’exécution avec Kubernetes non concluante, j’ai décidé de revenir à Docker. Avant cela, **toutes les installations précédentes ont été supprimées via une purge complète** du système, afin de repartir sur une base propre.

### **Objectif**

Installer Docker sur une machine distante fonctionnant sous Ubuntu et vérifier son bon fonctionnement à l’aide d’un test simple.

### **Étapes d'installation**

#### **1. Mise à jour et prérequis**

sudo apt-get update

sudo apt-get install ca-certificates curl

* ca-certificates : pour la vérification des certificats SSL
* curl : pour le téléchargement de fichiers depuis des URL

#### **2. Préparation des clés GPG Docker**

sudo install -m 0755 -d /etc/apt/keyrings

sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc

sudo chmod a+r /etc/apt/keyrings/docker.asc

#### **3. Ajout du dépôt Docker**

echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] \

https://download.docker.com/linux/ubuntu $(. /etc/os-release && echo "${UBUNTU\_CODENAME:-$VERSION\_CODENAME}") stable" | \

sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update

→ Cette étape permet d'ajouter le dépôt officiel Docker à APT pour pouvoir installer les versions les plus récentes.

#### **4. Installation de Docker**

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin

### **Vérification de l'installation**

#### **Test de bon fonctionnement :**

sudo docker run hello-world

Cette commande télécharge une image de test et exécute un conteneur qui affiche un message de confirmation.

### **Conclusion**

* L’installation de Docker sur la machine distante Ubuntu a été réalisée avec succès, **sans erreur**.
* Le test avec l’image hello-world a confirmé que Docker fonctionne correctement dans cet environnement.
* La machine est désormais prête pour accueillir des déploiements Docker (comme Kollaps).

### 

### **Construction de l’environnement Kollaps avec Docker**

### **Objectif**

Après avoir confirmé le bon fonctionnement de Docker sur la machine distante Ubuntu, l’objectif est désormais de :

* Cloner le dépôt Git de **Kollaps**
* Construire les images Docker nécessaires :
  + kollaps:2.0
  + kollaps-deployment-generator:2.0

### **Étapes techniques**

#### **1. Démarrage du service Docker**

sudo service docker start

#### **2. Vérification du bon fonctionnement**

sudo docker run hello-world

Résultat : le test de conteneur s’est déroulé avec succès, confirmant que Docker fonctionne toujours correctement.

#### **3. Clonage du dépôt Kollaps (avec sous-modules)**

git clone --branch master --depth 1 --recurse-submodules https://github.com/miguelammatos/Kollaps.git

* Récupère la dernière version du projet Kollaps depuis GitHub
* Inclut également les sous-modules nécessaires au bon fonctionnement du projet

#### **4. Activation de Docker BuildKit**

export DOCKER\_BUILDKIT=1 Permet d’activer **BuildKit**, le moteur de build moderne de Docker, qui :

#### **5. Construction de l’image kollaps:2.0**

Depuis le dossier examples :

docker build --rm -f dockerfiles/Kollaps -t kollaps:2.0 .

* Utilise le fichier dockerfiles/Kollaps pour créer l’image principale du simulateur
* L’image obtenue contient l’environnement de simulation Kollaps

#### **6. Construction de l’image kollaps-deployment-generator:2.0**

docker build -f dockerfiles/DeploymentGenerator -t kollaps-deployment-generator:2.0 .

* Cette image est responsable de la **génération de fichiers de déploiement** (YAML) pour Docker Swarm ou Kubernetes
* Elle inclut :
  + Le code source de Kollaps
  + Les dépendances Python nécessaires à la génération des scénarios de simulation

**Conclusion**

Docker est pleinement opérationnel et testé  
 Le dépôt **Kollaps** a été cloné avec succès, y compris ses sous-modules  
 Les deux images Docker essentielles ont été construites sans erreur :

* kollaps:2.0
* kollaps-deployment-generator:2.0

### **État du système**

Le système est désormais **prêt à exécuter des scénarios de simulation réseau ou de performance sur des architectures cloud virtualisées** via Kollaps.

**Rapport – Déploiement de l’environnement de simulation Kollaps avec Docker Swarm**

### **Objectif**

Finaliser la mise en place de l’environnement de test Kollaps en :

* Construisant l’image Docker pour l’interface **Dashboard**
* Configurant un **cluster Docker Swarm**
* Créant les services nécessaires dans ce cluster
* Générant les images de simulation pour l’application **iperf3**

### **Étapes et commandes exécutées**

#### **1. Construction de l’image Docker du Dashboard**

docker build -f examples/utils/dashboard/Dockerfile -t kollaps/dashboard:1.0 examples/utils/dashboard/

Cette commande construit l’image Docker du dashboard, utilisé pour visualiser les topologies réseau simulées.

#### **2. Initialisation du cluster Docker Swarm**

docker swarm init

#### **3. Création du réseau Docker pour les services Swarm**

docker network create --driver=overlay --subnet=11.1.0.0/20 kollaps\_network.

#### **4. Déploiement du Dashboard en tant que service Swarm**

docker service create \

--name kollaps\_dashboard \

--replicas 1 \

--publish published=8088,target=8088 \

--network kollaps\_network \

kollaps/dashboard:1.0

Déploie le **dashboard** accessible via le port 8088.

#### **5. Vérification du bon fonctionnement du service**

docker service ls

docker service ps kollaps\_dashboard

Permet de s’assurer que le service kollaps\_dashboard est bien lancé et en état **Running**.

**Problème constaté :** malgré le déploiement réussi, **le dashboard n’est pas accessible via le navigateur à l’adresse http://localhost:8088**.

### **Génération des images applicatives avec KollapsAppBuilder**

./KollapsAppBuilder iperf3

Génère les images suivantes :

* kollaps/iperf3-client:1.0
* kollaps/iperf3-server:1.0
* Une nouvelle image kollaps/dashboard:1.0

Le script utilise des Dockerfiles basés sur Alpine Linux incluant bash, iperf3, bind-tools, etc., pour simuler le trafic et les communications réseau.

### **Génération et déploiement d’un scénario réseau**

#### **Objectif**

Générer un fichier YAML à partir d’un fichier XML de topologie, puis le déployer via **Docker Stack**.

#### **Commandes exécutées**

##### **1. Génération du fichier YAML à l’aide de KollapsDeploymentGenerator**

./KollapsDeploymentGenerator iperf3/topology.xml -s ../iperf3-experiment.yaml

Convertit le fichier XML en un fichier YAML prêt à être utilisé avec Docker Swarm pour le déploiement d’une stack de simulation.

### **Conclusion**

* **Cluster Docker Swarm** initialisé avec succès
* **Dashboard**, **images iperf3**, et **réseau Docker** créés sans erreur
* **Dashboard** non accessible sur http://localhost:8088 malgré le déploiement
* Génération de scénario réseau réussie via KollapsDeploymentGenerator

### **Prochaines étapes**

* Diagnostiquer l’inaccessibilité du dashboard :
  + Vérifier les **ports ouverts**docker service logs kollaps\_dashboard
  + Tester avec curl localement pour voir si le port 8088 est réellement exposé
  + Exécuter les simulations et observer le comportement réseau dans l’interface graphique (si celle-ci est fonctionnelle)

**Date :** 12 mai 2025  
 **Étudiant :** **Kenny GAYAKPA** **Rapport – Remise à Plat et Redéploiement de l’Environnement Kollaps**

## **1. Objectif**

Remettre à zéro l’environnement Docker et reconfigurer l’ensemble de la pile Kollaps, incluant :

* Nettoyage complet de l’environnement Docker
* Reconstruction des images nécessaires
* Génération de l'application simulée iperf3network
* Création et déploiement d’une stack Swarm propre
* Tentatives d’accès au dashboard via différents moyens

## **2. Nettoyage de l’Environnement Docker**

Afin de repartir sur une base propre, les commandes suivantes ont été exécutées :

docker stack rm iperf3network

docker container prune -f

docker image prune -af

docker network prune -f

## **3. Positionnement dans le Répertoire du Projet**

cd ~/Kollaps

Positionnement dans le répertoire racine du projet.

**4. Reconstruction des Images Docker**

### **a. Activation de Docker BuildKit**r

export DOCKER\_BUILDKIT=1

### **b. Construction des images :**

#### **Image principale Kollaps :**

docker build -f dockerfiles/Kollaps -t kollaps:2.0 .

#### **Image du Dashboard :**

Avant la compilation, la ligne suivante a été **commentée** dans le Dockerfile :

# ADD ./baremetal/.ssh /root/.ssh/

Puis :

docker build -f dockerfiles/Dashboard -t kollaps/dashboard:1.0 .

#### **Image du Deployment Generator :**

docker build -f dockerfiles/DeploymentGenerator -t kollaps-deployment-generator:2.0 .

## **5. Génération de l’Application Simulée : iperf3network**

cd examples

chmod +x KollapsAppBuilder

./KollapsAppBuilder iperf3network

L’image kollaps/iperf3-network:1.0 a bien été générée. Vérification :

docker images | grep kollaps

## **6. Génération du Fichier de Stack**

Conversion du fichier XML en YAML via :

./KollapsDeploymentGenerator iperf3network/topology.xml -s topology.yaml

Produit le fichier topology.yaml pour le déploiement Swarm.

## **7. Création des Réseaux Docker**

### **Réseau iperf3network\_outside**

À **ne pas créer manuellement**, car il est généré automatiquement lors du déploiement.

### **Réseau kollaps\_network**

docker network create \

--driver overlay \

--subnet 11.1.0.0/20 \

kollaps\_network

## **8. Déploiement de la Stack**

docker stack deploy -c topology.yaml iperf3network

## **9. Étapes de Vérification**

## **Services actifs**

docker service ls

**Conteneurs actifs**

docker ps

## **10. Accès au Dashboard**

### **Méthodes testées :**

**Via curl :**  
curl http://localhost:8088

**Via http (outil CLI HTTPie) :**  
http http://localhost:8088

❌ **Aucun retour exploitable dans les deux cas.**

**Accès graphique via SSH avec X11 forwarding :**  
ssh -X [ubuntu@ccbd-2025-10.maas](mailto:ubuntu@ccbd-2025-10.maas)

## **11. Relance Complète en Environnement Propre**

### **Arrêt des services**

docker stack rm iperf3network

docker service rm kollaps\_dashboard || true

**Suppression des réseaux**

docker network rm iperf3network\_outside || true

docker network rm kollaps\_network || true

### **Recréation du réseau**

docker network create \

--driver overlay \

--subnet 11.1.0.0/20 \

kollaps\_network

**Redéploiement**

docker stack deploy -c topology.yaml iperf3network

**Vérification**

docker service ls

docker ps

**Accès graphique**

**firefox http://localhost:8088**

## **Conclusion**

* L’environnement Kollaps a été **remis à plat** avec succès.
* Toutes les **images Docker** ont été reconstruites proprement.
* La **stack iperf3network** a été déployée dans un cluster Docker Swarm.  
  Le **dashboard reste injoignable via HTTP direct**, mais peut être consulté via un navigateur lancé dans la VM (via X11 ou en local).

**Rapport – Contournement de l’accès graphique au Dashboard Kollaps**

**1. Problème rencontré**

L’accès à l’interface graphique du **Dashboard Kollaps**, déployé sur la machine distante via Docker Swarm, s’est révélé **inopérant via un navigateur standard**.

### **2. Tentative d’accès via Firefox (X11 forwarding)**

Connexion SSH avec redirection X11 établie :  
ssh -X ubuntu@ccbd-2025-10.maas

**Lancement de Firefox :**  
firefox <http://localhost:8088>

**Résultat :** le navigateur **ne se lance pas**, probablement en raison d’un manque de support graphique ou de dépendances X11 sur la machine distante.

### **3. Installation d’un navigateur alternatif : Epiphany**

En réponse à l’échec avec Firefox, un navigateur plus léger a été installé :

sudo apt install epiphany-browser -y

epiphany-browser http://localhost:8088

### **4. Problèmes rencontrés avec Epiphany**

Malgré l’installation réussie, plusieurs erreurs sont survenues au lancement :

**Messages d’erreur liés aux drivers EGL** :  
libEGL warning: pci id ... driver (null)

**Crashs récurrents du processus de rendu** :  
Web process crashed

Résultat : l’accès graphique via Epiphany **n’est pas stable ni exploitable**.

### **5. Vérification manuelle du Dashboard (en ligne de commande)**

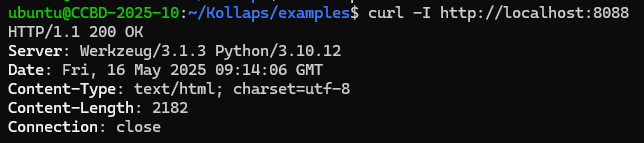
Face à l’échec des navigateurs graphiques, une vérification simple a été effectuée pour tester la **disponibilité HTTP** du service :

curl -I <http://localhost:8088>

Cette commande renvoie uniquement les **en-têtes HTTP**, ce qui permet de vérifier si le service est bien **exposé et joignable** sans tenter de charger l’interface graphique.

### **Conclusion**

* La redirection X11 fonctionne, mais les navigateurs ne sont pas fiables dans l’environnement graphique de la VM distante.
* Ni **Firefox** ni **Epiphany** n’ont permis d’accéder correctement au dashboard.
* L’accès **en ligne de commande** avec curl permet de vérifier que le **service HTTP du dashboard est bien actif**, même si l’affichage graphique reste impossible dans la VM.



### **6. Vérification de l'écoute réseau sur le port 8088**

Un scan de ports a été effectué pour confirmer que le service du **dashboard Kollaps** écoute bien sur le port 8088 :

**ss -tuln | grep 8088**

**Résultat : le port 8088 est bien ouvert et écouté en local.**

Cependant, **le navigateur graphique distant ne parvient toujours pas à afficher l’interface**, tandis que ce même navigateur peut accéder à des sites externes (ex. Google),

### **7. Tentative d’accès via navigateur console**

Installation d’un navigateur en mode texte :

sudo apt install w3m

w3m http://localhost:8088

**Résultat : malgré l’écoute active sur le port, l’interface web ne s’affiche pas**

### **8. Forçage du déploiement du Dashboard sur le bon nœud Swarm**

Afin de s’assurer que le service **dashboard** s’exécute **localement sur la machine utilisée pour le tunnel SSH** (ccbd-2025-10), un placement forcé a été appliqué.

#### **a. Modification du fichier topology.yaml**

Dans la section correspondant au service dashboard-4b826bd8-d92a-45a4-8951-376e6b9cf3ea, une directive deploy a été ajoutée comme suit :

deploy:

placement:

constraints:

- node.hostname == ccbd-2025-10

replicas: 1

Cela garantit que le conteneur du dashboard sera lancé **uniquement sur le nœud manager**, c’est-à-dire la machine où le tunnel SSH est établi.

#### **b. Redéploiement de la stack**

docker stack rm iperf3network

docker stack deploy -c topology.yaml iperf3network

#### **c. Vérification de l’emplacement du conteneur Dashboard**

docker service ps iperf3network\_dashboard-4b826bd8-d92a-45a4-8951-376e6b9cf3ea

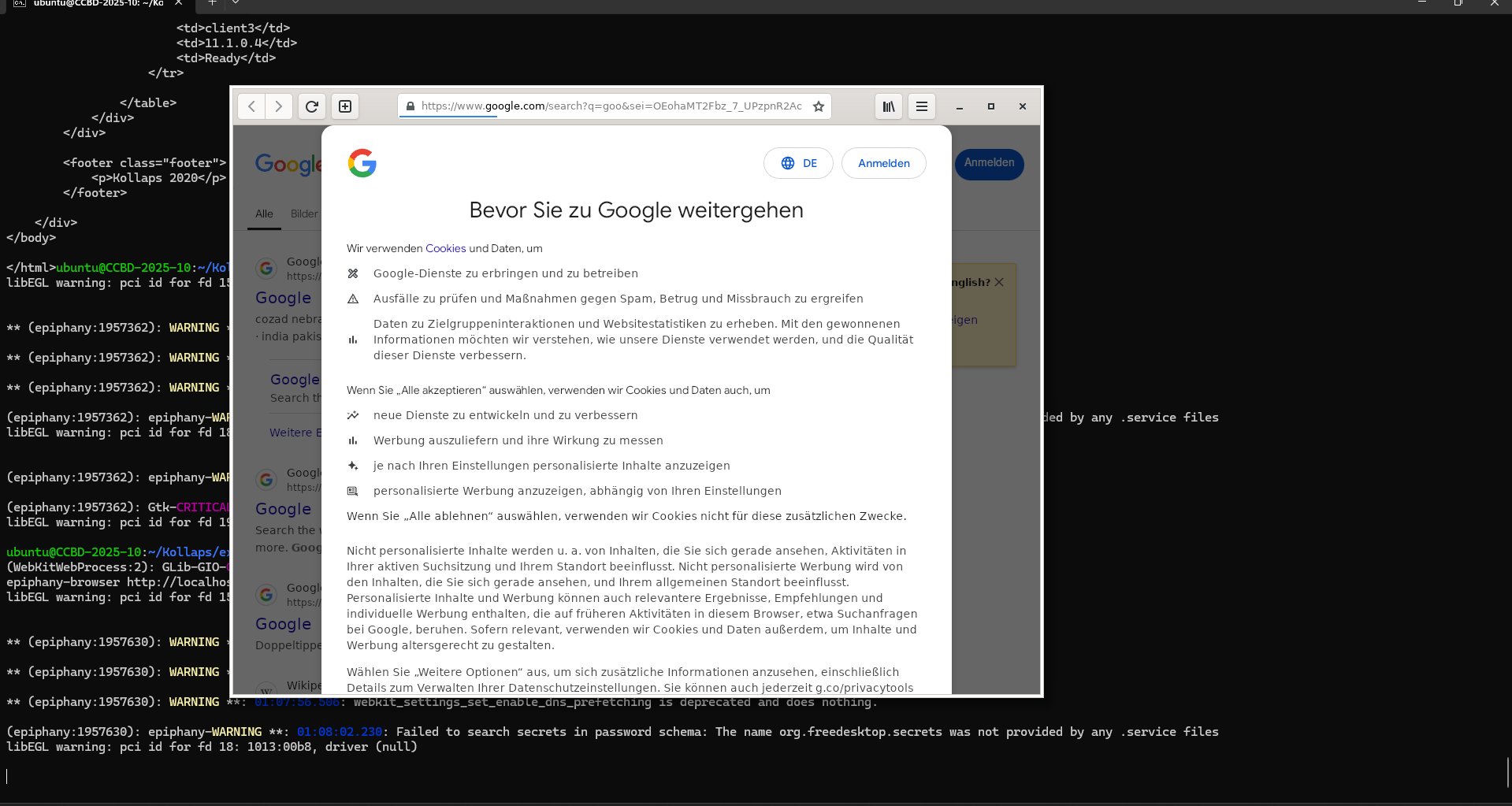
Cette commande permet de confirmer que le service **est bien lancé sur le nœud ccbd-2025-10**.

### **9. Relance du tunnel SSH (redirection de port)**

ssh -L 8088:localhost:8088 ubuntu@ccbd-2025-10.maas

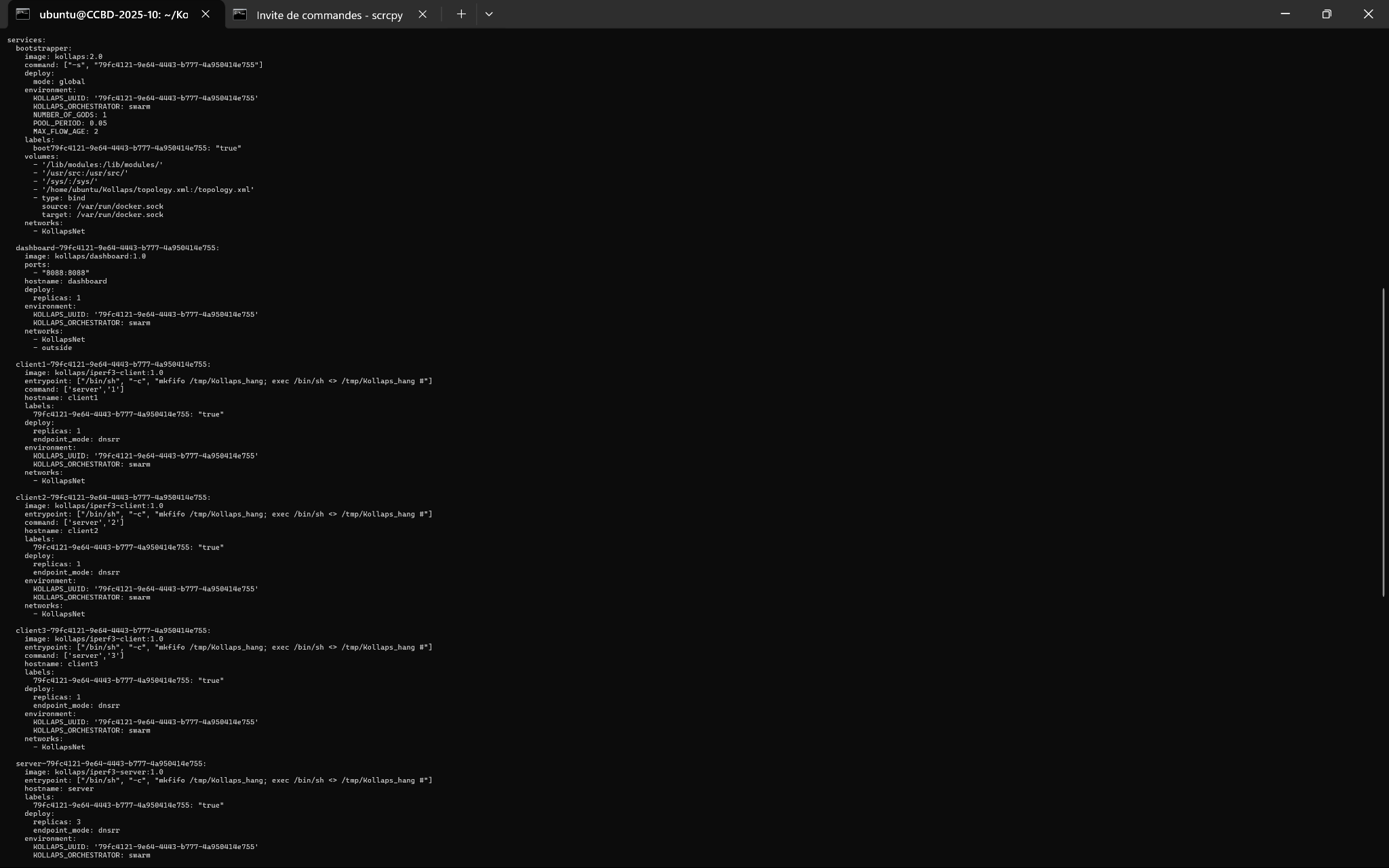
Puis ouvrir dans un navigateur **local** :

<http://localhost:8088>.  
**Conclusion : toujkours impossible d’ouvrir le dashboard de Kollaps**

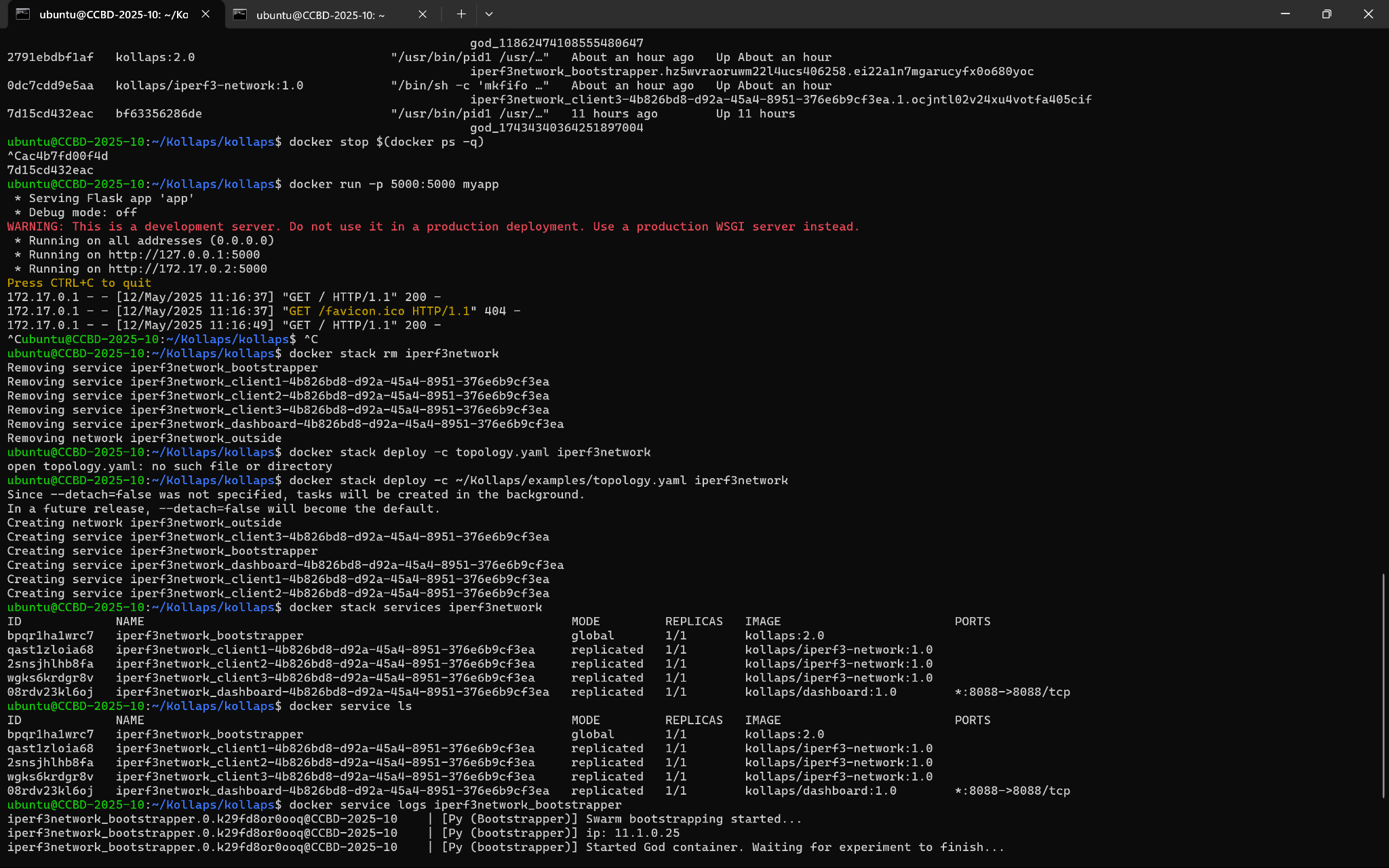


**D**ate: 12 Mai 2025  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Etudiant: Kenny GAYAKPA\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Experiments/Deployment**



Semaine pro : port forwarding VM, find what doesn’t work on WSL



Khaireddine Gatti - 12.05.2025

**Date :** 16 mai 2025  
**Étudiant :** **Khaireddine Gatti**

**Rédaction d’un GitHub Issue pour signaler un bug sur le Dashboard de Kollaps**

### **Contexte**

Dans le cadre des tests de l’environnement de simulation **Kollaps** avec **Docker Swarm**, un dysfonctionnement a été observé : **le dashboard de Kollaps reste inaccessible via le port 8088**, malgré un déploiement réussi du service.

### **Action entreprise**

Ouverture d’un **GitHub Issue** afin de **signaler le problème** à l’équipe de développement du projet Kollaps.

### **Lien vers le rapport de bug / issue**

🔗 [**Kollaps dashboard inaccessible via Docker Swarm · Issue #56**](https://github.com/miguelammatos/Kollaps/issues/56)

### **Objectif**

* Décrire les étapes de reproduction du bug
* Mentionner le contexte d’exécution (Docker Swarm, machine distante, tunneling SSH, etc.)
* Indiquer les tests déjà effectués (ports, redéploiement, contraintes de placement, etc.)

**Ensemble de tests à effectuer (Drive : *à tester* – répertoire OnDrive)**

En raison de l’inaccessibilité actuelle du **dashboard web** de Kollaps via Docker Swarm, il est nécessaire de **valider les fonctionnalités réseau et système de manière manuelle**,

| **Test** | **Outil** | **Objectif / Ce que ça mesure** |
| --- | --- | --- |
| **Connectivité** | ping | Vérifie la latence de base et la connectivité entre nœuds/services |
| **Bande passante** | iperf3 | Mesure le débit réseau en TCP/UDP |
| **Transactions applicatives** | netperf | Analyse le taux de réponse et le débit applicatif |
| **Accessibilité des services** | curl, wget | Teste la disponibilité des endpoints HTTP |
| **Résilience à la charge** | stress, top, htop | Simule de la charge CPU/Mémoire, surveille la stabilité |
| **Routage réseau** | traceroute | Affiche le chemin parcouru par les paquets |
| **Monitoring réseau** | docker stats, iftop, nload | Surveille en temps réel l’usage CPU/RAM/réseau des conteneurs |

fichiers XML utilisés pour les tests  
**LCS** – topologie avec variation de jitter et crash à 300 s

**failover** – scénario de basculement avec server1 qui crash puis revient

**CBBP** – chute brutale de bande passante puis rétablissement

**PRECS** – partition réseau temporaire entre serveurs et réseau

**RUBSC** – rolling upgrade de trois serveurs avec crash et recovery

**Date :** 17 mai 2025  
 **Étudiant :** **Kenny GAYAKPA** **Visualisation de la Topologie Kollaps sans Dashboard – Génération Graphique via Script Python**

### **Contexte**

En raison de l’indisponibilité persistante du **dashboard graphique de Kollaps**, il est nécessaire de **générer une représentation visuelle de la topologie réseau autrement**.

**Solution**  : Utilisation d’un script Python yaml2graph.py pour convertir un fichier topology.yaml en image graphique au format **SVG**, **PNG**.

### 

### **Fonctionnement du script yaml2graph.py**

Ce script effectue les opérations suivantes :

1. **Analyse (parsing)** du fichier topology.yaml décrivant la topologie réseau utilisée par Kollaps
2. **Génération automatique d’un fichier .dot** au format **Graphviz**, représentant les nœuds (services) et les liens (réseaux)
3. **Création d’une image** de la topologie à partir du .dot, via les moteurs de rendu Graphviz (neato, circo, etc.)

### **Organisation des fichiers**

* Un dossier graphvizgenerator est créé pour contenir :  
  + Le fichier topology.yaml
  + Le script yaml2graph.py

### **Installation des dépendances**

Depuis le dossier graphvizgenerator, exécuter les commandes suivantes :

sudo apt update && sudo apt install graphviz

pip install pyyaml

Ces outils permettent respectivement :

* **Graphviz** : transformation des fichiers .dot en images
* **PyYAML** : lecture/écriture des fichiers YAML en Python

### **Exemples de génération d'images**

#### **✅ Image orientée en SVG (moteur Neato)**

python3 yaml2graph.py -i topology.yaml -d topology.dot -f svg -e neato --directed

#### **✅ Image au format PNG (moteur Circo)**

python3 yaml2graph.py -i topology.yaml -d topology.dot -f png -e circo

**script python dans le fichier** yaml2graph.py

**topologie LCS** :

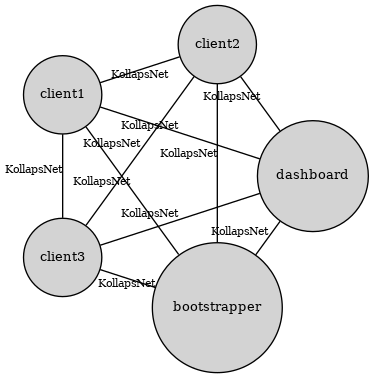
Dans cette expérience, on observe le comportement de trois clients (*client1*, *client2*, *client3*) connectés à un réseau partagé. Il n’y a pas de serveurs, seulement des clients qui communiquent entre eux ou avec un système de test. Un tableau de bord (*dashboard*) est utilisé pour surveiller ce qu’il se passe.

Les trois clients sont connectés à un point central (le pont *s1*) avec une bonne connexion (vitesse de 666 Mbps et faible délai au départ).

L’objectif de ce scénario est de tester **la qualité du réseau en jouant sur le "jitter"**, c’est-à-dire les variations imprévisibles dans le temps de transmission des données. Voici ce qui se passe au fil du temps :

* **À 0 seconde**, tous les clients se connectent et commencent à communiquer normalement.
* **À 30 secondes**, on introduit un jitter modéré de **10 millisecondes** pour tous les clients, ce qui simule un réseau un peu instable.
* **À 60 secondes**, le jitter passe à **20 millisecondes**, représentant une situation plus chaotique, comme lors d’une congestion réseau.
* **À 90 secondes**, le jitter est réduit à **2 millisecondes**, ce qui correspond à une amélioration soudaine de la qualité du réseau.
* **À 300 secondes**, tous les clients plantent en même temps, simulant une panne générale ou une fin de charge.

Ce scénario permet de tester comment les clients réagissent face à des perturbations dans la qualité du réseau (instabilité de la connexion), sans changer la bande passante ni provoquer de coupures.



**topologie failover** :

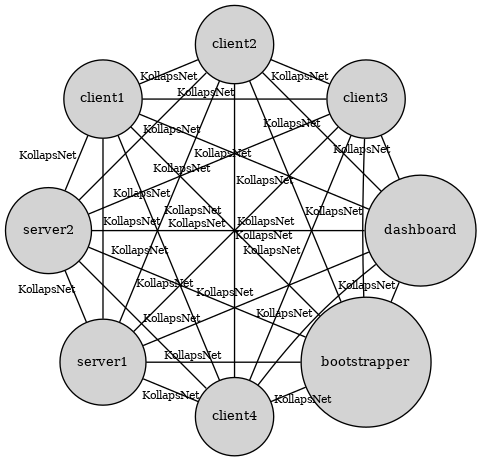
Dans cette expérience, il y a deux serveurs (*server1* et *server2*) qui travaillent en parallèle pour répondre aux demandes, et quatre clients (*client1* à *client4*) qui envoient des données à ces serveurs. Un tableau de bord (*dashboard*) permet de suivre ce qui se passe en temps réel.

Tous les éléments sont connectés à un même point central (*br0*), comme s’ils étaient branchés sur un même réseau, avec de bonnes connexions (rapides et stables).

Voici le déroulement :

* **À 0 seconde**, tout le monde se connecte : les deux serveurs et les quatre clients commencent à fonctionner normalement. *client1* et *client2* utilisent *server1*, tandis que *client3* et *client4* utilisent *server2*.
* **À 60 secondes**, *server1* tombe en panne : il s’arrête brutalement. Les clients qui l’utilisaient doivent alors basculer automatiquement vers *server2* pour continuer à fonctionner.
* **À 180 secondes**, *server1* redémarre et revient en service. On vérifie s’il est bien réintégré et si les clients peuvent à nouveau l’utiliser.
* **À 300 secondes**, l’expérience se termine.

Ce scénario sert à tester la **capacité du système à basculer automatiquement vers un autre serveur en cas de panne**, puis à **reprendre un fonctionnement normal quand le serveur est de nouveau disponible**.



**topologie CBBP** :

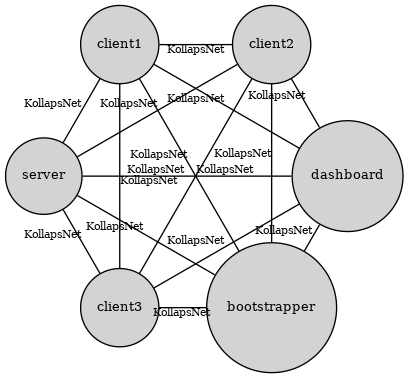
Dans cette expérience, on teste un système avec un seul serveur et trois clients (*client1*, *client2*, *client3*) qui envoient des données au serveur. Un tableau de bord (*dashboard*) permet de surveiller ce qu’il se passe pendant l’expérience.

Tous les éléments sont connectés à un point central (*br0*), qui joue le rôle de carrefour du réseau. Au début, les connexions sont très rapides (1 Gbps) et stables.

Voici le déroulement :

* **À 0 seconde**, tous les clients et le serveur démarrent et commencent à communiquer normalement avec un réseau rapide.
* **À 30 secondes**, une panne est simulée : la bande passante chute brutalement pour tout le monde, passant de 1 Gbps à **10 Mbps**. Cela ralentit énormément le trafic, comme si la connexion devenait soudainement très lente.
* **À 90 secondes**, tout revient à la normale : les vitesses de transmission sont rétablies à 1 Gbps pour tous.
* **À 120 secondes**, l’expérience se termine.

Ce scénario sert à évaluer comment le système réagit à une **forte dégradation temporaire de la bande passante**.



**topologie PRECS** :

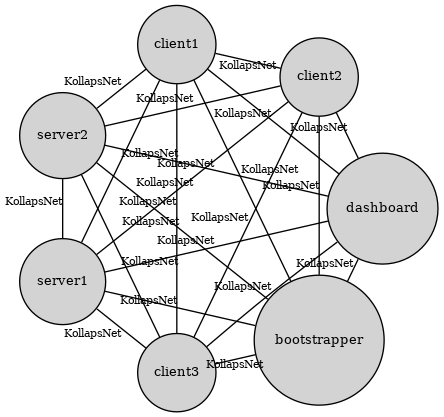
Dans cette expérience, on teste un système composé de deux serveurs (*server1* et *server2*) qui fonctionnent ensemble, et de trois clients (*client1*, *client2* et *client3*) qui envoient des données. Un tableau de bord (*dashboard*) permet de suivre l’activité du réseau.

Tous ces éléments sont connectés à un pont central (*br0*), qui joue le rôle de point de communication unique, avec des connexions rapides et stables.

Voici ce qui se passe au cours du temps :

* **À 0 seconde**, tous les serveurs et les clients se connectent et commencent à échanger des données. *client1* et *client2* utilisent *server1*, tandis que *client3* utilise *server2*.
* **À 60 secondes**, une panne réseau est simulée : **les deux serveurs sont coupés du réseau**. C’est comme si leurs connexions étaient débranchées. Les clients ne peuvent donc plus échanger avec eux.
* **À 120 secondes**, la connexion est rétablie : les serveurs peuvent à nouveau recevoir et répondre aux demandes.
* **À 180 secondes**, l’expérience se termine.

Ce scénario sert à tester **la tolérance du système à une coupure complète des serveurs**, et à vérifier s’il retrouve un fonctionnement normal une fois la connexion rétablie.



**topologie RUBSC** ::

Dans cette expérience, on simule une situation où un système évolue progressivement avec des mises à jour de serveurs, tout en continuant à fonctionner. Il y a :

* Trois serveurs version 1 (*server1-v1*, *server2-v1*, *server3-v1*) qui démarrent au début.
* Trois serveurs version 2 (*server1-v2*, *server2-v2*, *server3-v2*) qui seront activés plus tard pour remplacer les précédents.
* Quatre clients (*client1* à *client4*) qui envoient de la charge au système.
* Un tableau de bord (*dashboard*) pour observer le déroulement en temps réel.

Tous ces éléments sont connectés à un point central (*br-backend*), qui permet à tous de communiquer rapidement et efficacement.

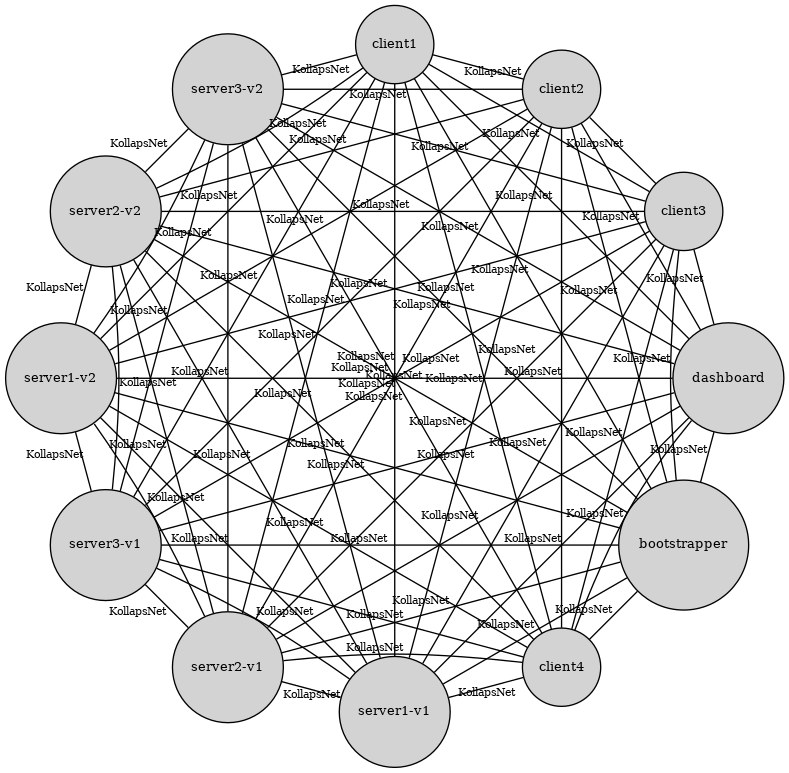
Voici le déroulement de l’expérience :

* **À 0 seconde**, les trois serveurs version 1 et les quatre clients démarrent normalement.
* **À 60 secondes**, on commence une **mise à jour progressive** (*rolling upgrade*) : *server1-v1* est remplacé par *server1-v2*.
* **À 120 secondes**, c’est au tour de *server2* d’être mis à jour.
* **À 180 secondes**, *server3* est remplacé à son tour par sa version mise à jour.  
  **À 240 secondes**, une panne est simulée : *server2-v2* tombe en panne.
* **À 270 secondes**, ce serveur redémarre et revient dans le système.
* **À 300 secondes**, l’expérience se termine.

Ce scénario permet de tester :

* La **continuité de service pendant une mise à jour**.
* La **résilience en cas de panne d’un serveur fraîchement mis à jour**.
* La **capacité du système à récupérer automatiquement** après un problème.

C’est une situation réaliste utilisée pour vérifier si les mises à jour logicielles peuvent se faire **sans interruption pour les utilisateurs**.



**Rapport de résolution – Inaccessibilité du Dashboard Kollaps via Docker Swarm**

### **Contexte**

Suite à l’ouverture de l’[issue GitHub #56](https://github.com/miguelammatos/Kollaps/issues/56) signalant l'inaccessibilité du **Dashboard Kollaps** sur le port 8088 via Docker Swarm, plusieurs pistes ont été explorées pour diagnostiquer et corriger le problème. Ce rapport documente la démarche qui a permis de **résoudre définitivement** le problème.

## **1. Correction du Dockerfile du Dashboard**

**Problème identifié :** L'image Docker du dashboard exécutait par défaut app.py au lieu de Dashboard.py.

**Solution :** mise à jour du Dockerfile pour forcer l’exécution de Dashboard.py :

FROM python:3.9

WORKDIR /app

COPY . /app

RUN pip install flask

EXPOSE 8088

CMD ["python", "Dashboard.py"]

## **2. Script de nettoyage complet de l’environnement Docker**

Un script shell a été rédigé pour supprimer toutes les stacks, services, conteneurs, images, volumes et réseaux Swarm afin de **repartir d’un environnement propre** :

docker stack rm iperf3network || true

docker service rm $(docker service ls -q) 2>/dev/null || true

docker container rm -f $(docker ps -aq) 2>/dev/null || true

docker image rm -f $(docker images -q) 2>/dev/null || true

docker volume rm $(docker volume ls -q) 2>/dev/null || true

docker network rm $(docker network ls --filter "driver=overlay" -q) 2>/dev/null || true

docker network rm kollaps\_network 2>/dev/null || true

## **3. Vérification de l'état global**

Après le nettoyage, plusieurs commandes ont été exécutées pour vérifier que l’environnement était vide :

docker stack ls

docker ps -a

docker images

docker network ls

docker volume ls

docker service ls

## **4. Reconstruction des images Docker**

Après activation de BuildKit, les trois images nécessaires ont été reconstruites :

cd ~/Kollaps

export DOCKER\_BUILDKIT=1

docker build -f dockerfiles/Kollaps -t kollaps:2.0 .

docker build -f dockerfiles/DeploymentGenerator -t kollaps-deployment-generator:2.0 .

docker build -f dockerfiles/Dashboard -t kollaps/dashboard:1.0 .

## **5. Génération de la topologie iperf3network**

cd ~/Kollaps/examples

chmod +x KollapsAppBuilder

./KollapsAppBuilder iperf3network

./KollapsDeploymentGenerator iperf3network/topology.xml -s topology.yaml

docker network create --driver overlay --subnet 11.1.0.0/20 --attachable kollaps\_network

## **🚀 6. Déploiement de la stack Swarm**

docker stack deploy -c topology.yaml iperf3network

### **⚠️ En cas d'erreur réseau :**

docker network rm ingress || true

docker swarm leave --force

docker swarm init

docker network create --driver overlay --subnet 11.1.0.0/20 --attachable kollaps\_network

docker stack deploy -c topology.yaml iperf3network

## **7. Publication du port 8088 (Dashboard)**

Pour exposer le port du dashboard depuis Swarm, la configuration suivante a été ajoutée **dans le fichier YAML du service** :

ports:

- target: 8088

published: 8088

protocol: tcp

mode: host

Après modification, le stack a été redéployé.

## **8. Test final via navigateur console**

Pour valider localement que le service répond :

links http://localhost:8088

Le dashboard est maintenant accessible depuis un navigateur local via<http://localhost:8088>

**Après correction du Dockerfile, nettoyage de l’environnement, reconfiguration des ports, et redéploiement de la stack, le Dashboard fonctionne comme prévu.**

### **Développement d’un script de collecte automatisée des métriques réseau dans un environnement Kollaps**

#### **🎯 Objectif**

Dans le cadre du projet **Cloud Computing avec Kollaps**, un script Bash nommé collect\_metrics.sh a été développé pour automatiser la **collecte de métriques réseau** issues des simulations orchestrées via **Docker Swarm**.

Ce script permet de :

* Identifier automatiquement les conteneurs **clients** et **serveurs** au sein de la stack,
* Lancer des tests de **débit et de performance** réseau avec iperf3,
* Mesurer la **latence, perte de paquets et jitter** via ping,
* Gérer automatiquement l’infrastructure nécessaire aux tests,
* Générer un fichier CSV standardisé contenant toutes les mesures.

### **Nom du script : collect\_metrics.sh**

#### **Utilisation**

chmod +x collect\_metrics.sh

./collect\_metrics.sh <stack\_name>

**Paramètre requis :**

* <stack\_name> : nom de la stack Docker Swarm déployée (ex : iperf3network)

**Exemple :**

./collect\_metrics.sh iperf3network

Ce script produit un fichier de sortie :

metrics\_<stack\_name>\_<timestamp>.csv

**Fonctionnalités détaillées**

#### **1. Détection automatique des conteneurs**

* Identification des **clients** (client\*) et **serveurs** (server\*) à partir des noms des conteneurs Docker,
* Filtrage du dashboard (non pertinent),
* Nettoyage des noms pour les rapports.

#### **2. Initialisation des serveurs iperf3**

* Démarrage automatique du serveur iperf3 (iperf3 -s) sur chaque conteneur serveur, en tâche de fond.

#### **3. Tests de performance réseau**

Pour chaque couple client → serveur :

* Test iperf3 (5 secondes),
* Extraction du débit montant, descendant, total,
* Calcul de volume de données échangées,
* Capture des retransmissions TCP,
* Gestion des cas d’échec avec indicateurs clairs.

#### **4. Tests de latence avec ping**

* Exécution d’un ping de 10 paquets du client vers le serveur,
* Extraction des latences (min, moyenne, max),
* Calcul approximatif du jitter (max - min),
* Détection du taux de perte de paquets.

#### **5. Format de sortie standardisé**

Un fichier .csv est généré à chaque exécution avec les colonnes suivantes :

timestamp,client,server,upload\_throughput\_mbps,download\_throughput\_mbps,total\_throughput\_mbps,data\_transferred\_mb,min\_latency\_ms,avg\_latency\_ms,max\_latency\_ms,jitter\_ms,packet\_loss\_percent,failed\_requests\_count,error\_rate\_percent,tcp\_retransmissions

Les valeurs sont horodatées et prêtes à être utilisées pour des traitements automatisés (ex. fusion, visualisation, analyse comparative).

### **Date : 21 mai 2025**

**Étudiant :** Khaireddine Gatti  
**Titre :** Automatisation robuste de la collecte et synthèse des métriques réseau

### **Contexte**

Dans le cadre de simulations réseau avancées avec Kollaps, chaque scénario (défini via une topologie YAML) introduit des comportements dynamiques : mises à jour progressives, crashes simulés, coupures réseau, baisses de bande passante, variations de latence, etc.

Pour étudier ces phénomènes de manière rigoureuse, il est essentiel de **collecter des métriques réseau** (débit, latence, pertes) à des instants stratégiques, synchronisés avec les événements du scénario.

### **Problèmes avec l’ancienne méthode**

Le script collect\_metrics.sh original ne réalisait qu’une **capture statique unique**, manuelle, ce qui posait plusieurs limites :

* Manque d’automatisation pour suivre l’évolution temporelle,
* Risques de désynchronisation entre événements et captures,
* Difficulté à comparer plusieurs étapes ou itérations.

### **Solution mise en œuvre**

#### **1. Collecte automatique répétée avec schedule\_metrics.sh**

Un nouveau script schedule\_metrics.sh a été développé. Il joue le rôle **d’orchestrateur temporel** :

* Il exécute collect\_metrics.sh de manière **automatique à intervalles réguliers**, définis par l'utilisateur.
* Il regroupe toutes les mesures dans un dossier dédié.
* Il fusionne ensuite les résultats dans un unique fichier CSV complet.

**Syntaxe :**

./schedule\_metrics.sh <stack> <intervalle\_s> <durée\_totale\_s> <topologie>

**Exemple :**

./schedule\_metrics.sh iperf3network 30 300 RUBSC

Cela déclenche une collecte toutes les 30 secondes pendant 5 minutes.

**Fonctionnement :**

* Création d’un dossier collect\_<stack>\_<topo>\_<timestamp>/
* À chaque itération :
  + Appel de collect\_metrics.sh <stack>
  + Déplacement du fichier généré dans le dossier
* À la fin :
  + Fusion de tous les CSV en un fichier metrics\_<stack>\_<topo>\_merged\_<timestamp>.csv

#### **2. Collecte précise des performances avec collect\_metrics.sh**

Ce script :

* Identifie dynamiquement les **clients** et **serveurs** dans la stack Docker Swarm.
* Lance des tests ping et iperf3 entre tous les couples client ↔ serveur.
* Génére un fichier .csv contenant pour chaque paire :Débits montant/descendant, latence (min, avg, max), jitter, pertes, retransmissions TCP, taux d’échec.

**Données capturées :**

* timestamp, client, server
* upload/download throughput (Mb/s)
* latence (ms), jitter, pertes (%)
* retransmissions TCP, erreurs iperf3

**Format du fichier :** metrics\_<stack>\_<timestamp>.csv

### **Exemples de scénarios et plan de collecte**

Voici les **recommandations temporelles** pour chaque topologie :

| **Topologie** | **Événements clés** | **Durée** | **Fréquence recommandée** | **Total mesures** |
| --- | --- | --- | --- | --- |
| **RUBSC** | upgrades à 60/120/180s, crash à 240s, recovery à 270s | 300s | toutes les 30s | 11 |
| **CBBP** | chute bande à 30s, retour à 90s | 120s | toutes les 15s | 9 |
| **PRECS** | coupure à 60s, retour à 120s | 180s | toutes les 30s | 7 |
| **Failover** | crash à 60s, retour à 180s | 300s | toutes les 30s | 11 |
| **LCS** | changements de jitter à 30/60/90s | 300s | toutes les 30s | 11 |

### **Nouveau processus complet d’exécution**

1. **Réinitialiser l’environnement :**

docker network rm ingress || true

docker swarm leave --force

docker swarm init

docker network create --driver overlay --subnet 11.1.0.0/20 --attachable kollaps\_network

1. **Déployer la topologie souhaitée :**

docker stack deploy -c <fichier\_topologie>.yaml iperf3network

1. **Lancer la collecte automatisée :**

./schedule\_metrics.sh iperf3network 30 300 RUBSC

1. **Obtenir le fichier de synthèse fusionné :**

* metrics\_iperf3network\_RUBSC\_merged\_<timestamp>.csv