CR Gestion de projet - M1 EEEA

Léah LAURENT Noë MERCIER-BROSSE

January 3, 2025

Contents

1 Création du serveur CLI **2**

2 Développement python **3**

2.1 Reseau.py 3

2.2 StrategieReseau.py 3

2.3 Écriture des tests 4

3 Utilisation de git **4**

1 Création du serveur CLI

Pour le serveur Jenkins, on a décidé d'utiliser une Machine Virtuelle tournant sur *VirtualBox* et utilisant une image *Ubuntu Server*. Nous avons des machines personnelles tournant sous Linux, donc nous n'avions pas accès à WSL, ceci étant donné, cette solution offrait un bon compromis entre taille de la VM et rapidité de configuration.

Comme nous utilisions une machine virtuelle locale et que nous allons avoir besoin d'accéder à des services internet sur celle-ci (Serveur Web, SSH), il a fallu jouer avec les paramètres réseau de VirtualBox. Heureusement, VirtualBox propose de gérer le NAT du routeur virtuel sur lequel est interfacé notre VM. Ainsi, on a pu exposer le port 9090 (HTTP utilisé par Jenkins) de la VM sur le port 9090 du réseau local, et le port 22 (SSH) sur le port 2222 (fig. 1). Ceci étant fait après avoir ouvert le pare-feu de la VM (avec *sudo ufw enable ssh*), nous allons pouvoir nous connecter via SSH à la VM et profiter du confort de notre interface locale.

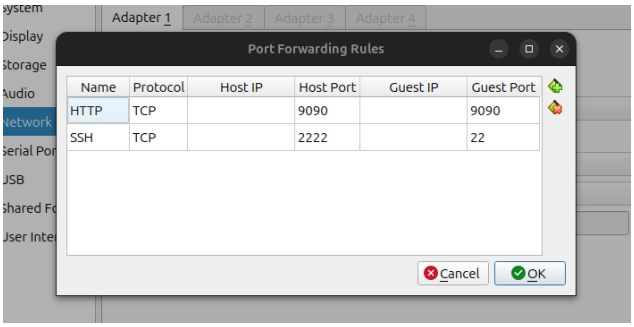


Figure 1: Configuration du port forwarding de notre VM

Ensuite, en suivant les instructions données pour l'installation de Jenkins (installation et démarrage du service, déploiement des clés SSH). Nous pouvons accéder au tableau de bord du service. Etant donnée notre configuration utilisant un NAT, on accède pas à ce service via l'adresse IP de l'interface *eth0* de la VM mais via l'adresse *localhost (127.0.0.1)*. Une fois connectés, on a du relancer plusieurs fois l'installation de tous les plugins, mais tout a fini par se finir correctement.

Ensuite, on a pu mettre en place la pipeline en permettant à Jenkins d'utiliser le JenkinsFile de notre repo Github sans problème. On a pu enfin la lancer pour la première fois et vérifier que tout fonctionnait proprement (fig. 2)

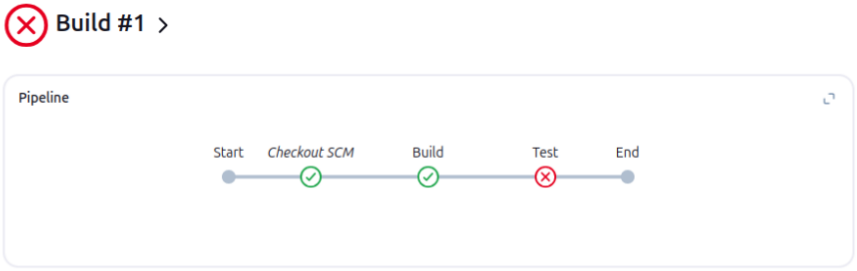


Figure 2: Task Failed successfully!!

2 Développement python

Une fois la configuration de Jenkins terminée, nous pouvons entamer l'implémentation du code du projet PowerGrid. L'objectif principal de ce projet étant de nous familiariser avec les outils de gestion de projets, il est logique, dans ce contexte, de répartir les tâches de développement logiciel entre les membres de l'équipe. Ainsi, chacun de nous peut se concentrer sur différentes fonctions à implémenter, en travaillant sur nos branches respectives *dev-noe* et *dev-leah*. Cette approche permet de paralléliser le travail, et une fois nos tâches terminées, nous pourrons fusionner nos branches pour obtenir une version unique et consolidée, intégrant nos implémentations respectives.

2.1 Reseau.py

La fonction *valider_ reseau* utilise une approche de recherche en profondeur DFS à partir du noeud d'entrée pour vérifier la connectivité de l'ensemble du réseau. Elle parcourt tous les arcs du réseau et, en utilisant une liste de noeuds à visiter, elle s'assure que chaque noeud est accessible depuis le noeud d'entrée. Si tous les noeuds du réseau sont visités au terme de cette recherche, la fonction retourne True, confirmant ainsi que le réseau est bien connecté. Sinon, elle retourne False, indiquant qu'il existe des parties du réseau non accessibles.

Quant à la fonction *valider_ distribution*, elle permet de s'assurer que tous les clients présents dans le terrain sont correctement distribués sur les noeuds du réseau. Elle vérifie, pour chaque client, si celui-ci se trouve bien sur un noeud existant. Si un client est détecté en dehors des noeuds du réseau, la fonction retourne False, indiquant une mauvaise distribution des clients. Dans le cas contraire, si tous les clients sont correctement placés, elle retourne True.

2.2 StrategieReseau.py

Une approche orientée objet a été adoptée pour la gestion des stratégies de configuration, permettant ainsi de personnaliser le comportement en fonction du type de stratégie choisie. La classe StrategieReseau définit une interface générale avec une méthode configurer, qui est censée retourner une configuration spécifique pour le réseau.
Deux sous-classes ont été créées pour spécifier deux comportements distincts pour cette méthode : *StrategieReseauManuelle* et *StrategieReseauAuto*. Ces sous-classes permettent de définir des stratégies de configuration différentes, tout en maintenant une interface commune. Cela permet d'appeler la même méthode *configurer()* tout en ayant des implémentations différentes selon le type de stratégie.

La stratégie manuelle repose sur l'ajout de noeuds et d'arcs sur le terrain, choisi par l'utilisateur. Tous les cas particuliers générant des erreurs ou du non-sens sont gérés.
La stratégie automatique propose 3 "types" de configuration du réseau pour recouvrir le terrain :

- Un réseau qui couvre tout le terrain
- Un réseau qui couvre toutes les lignes contenant des clients, sans prendre en compte les obstacles
- Un réseau recouvert par l'algorithme BFS (Parcours en largeur), offrant des solutions optimales

Cette dernière stratégie, explore de manière systématique tous les noeuds voisins, garantissant que la couverture du terrain est optimale, indépendamment de la configuration d'entrée. En effet, quelle

que soit la disposition du terrain, y compris la présence d'obstacles ou la distribution des clients, l'algorithme parvient toujours à trouver la solution la plus adaptée pour relier les différents points du réseau.

Exemple de résolution :

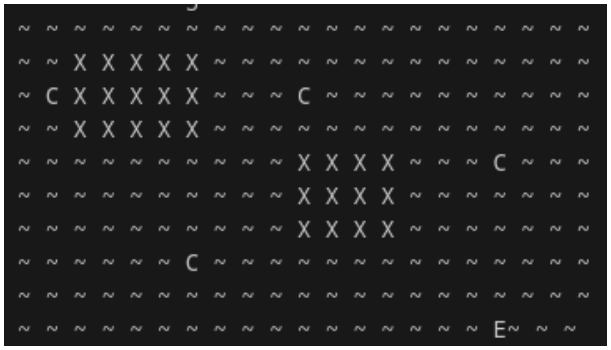


Figure 3: Terrain sans réseau

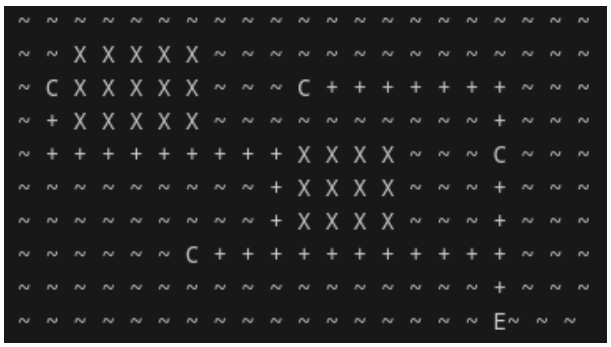
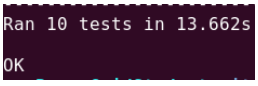


Figure 4: Réseau après BFS

2.3 Écriture des tests

Après avoir terminé l'implémentation des fonctions, nous avons développé des tests unitaires pour en valider le comportement. Ces tests ont permis de s'assurer que chaque fonction répond correctement aux différents cas d'utilisation, y compris les situations limites.
L'écriture de ces tests a également joué un rôle clé dans la détection précoce d'éventuelles erreurs, tout en garantissant une base solide pour les étapes suivantes du projet. Cette démarche a renforcé la fiabilité globale du code et assuré une meilleure maintenance.



3 Utilisation de git

Tout au long du projet, nous avons utilisé *Git* pour collaborer efficacement en groupe et organiser le travail de manière structurée.
Chacun de nous a créé des branches dédiées au développement (*dev-noe* et *dev-leah*) ainsi que des branches spécifiques pour la rédaction du rapport LaTeX (*rapport-noe* et *rapport-leah*). Cette approche nous a permis de travailler simultanément sur différentes parties du projet, sans interférer avec les contributions des autres membres de l'équipe.
L'utilisation des branches nous a offert une grande flexibilité : nous pouvions tester, modifier, et affiner nos contributions avant de *push* et de *merge* vers la branche principale (*master*) une fois notre travail jugé abouti.
Bien que nous maîtrisions déjà *Git* et le travail collaboratif, ce projet nous a encore renforcé dans notre compréhension de l'importance et de l'efficacité qu'apportent les branches dans la gestion de versions et la collaboration.

Lien GitHub vers notre projet

Historique des versions et visualisation des branches avec la commande

git log –graph –oneline –branches