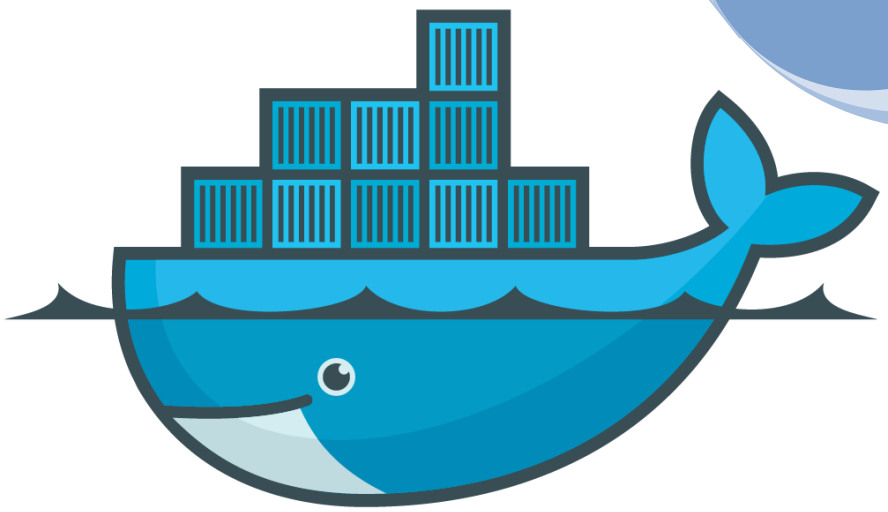
IUT Villetaneuse
Université Paris 13



docker

Virtualisation de réseaux Ethernet avec la technologie Docker

Projet tuteuré

BERNARD Jonathan
JAYARAJAH Thanushan
SAKPONOU David
VARELA Jeremy

Tuteur : LODDO Jean-Vincent

2016 – 2017

Remerciements

Avant toute chose, il nous paraît opportun de remercier tous ceux qui ont contribué à la réussite de notre projet par leur implication et leur sens de responsabilité.

A commencer par notre tuteur M. LODDO Jean-Vincent, qui nous a accompagnés le long de cette période de projet, par son suivi, ses conseils pertinents quant à l'organisation et l'avancement de notre étude et sa disponibilité pour répondre aux problèmes que nous avons rencontrés.

Aussi nous tenons à remercier notre professeur de communication, Mme PAULIAN Claire, pour la méthodologie quant à l'élaboration d'un dossier qu'elle nous a apporté tout au long de l'année.

Nous remercions aussi nos camarades qui nous ont soutenus et qui nous ont posés des questions pertinentes afin que nous puissions nous remettre en question et adapter notre projet afin qu'il puisse eux aussi le comprendre.

Table des matières

Introduction

Notre projet tuteuré s'inscrit dans le cadre de notre DUT en Réseaux et Télécommunications à l'IUT de Villetaneuse.

Le but de ce projet était de s'approprier la technologie Docker, que nous n'avions jamais étudiée auparavant, et de mettre en pratique les connaissances que nous avons acquises tout au long de notre formation, telles que l'administration de système linux, afin de pouvoir regrouper une application et toutes ses dépendances dans un seul et même paquetage que l'on appelle « conteneur ». Cette conteneurisation aura pour but de rendre notre application la plus indépendante possible de notre système d'exploitation (OS) afin de pouvoir la transporter facilement d'un OS a un autre qui, toutefois, doit être de même type (GNU/Linux).

Nous avons réalisé ce projet en groupe de quatre. Notre chef de projet étant BERNARD Jonathan, il s'est occupé de répartir les tâches et de nous assigner à chacun une mission. Cette organisation a eu pour but de nous apprendre à travailler en groupe dans un cadre professionnel et de structurer notre façon de travailler.

Notre cahier des charges était le suivant :

- Comprendre la technologie Docker via la documentation officielle et les différents tutoriels présents en ligne afin créer un conteneur pour l'application de simulation de réseaux TCP/IP Marionnet
- Étudier la possibilité de construire un réseau virtuel ou les machines seraient des conteneurs Docker, et l'infrastructure réseau fourni par la technologie VDE.

Notre problématique fut donc de savoir comment conteneuriser une application telle que Marionnet afin que n'importe qui puisse l'utiliser sans avoir à l'installer au préalable sur son ordinateur ?

Nous avons axé notre travail sur trois grandes étapes.

- Tout d'abord nous avons appris à créer et à utiliser un conteneur via la documentation.
- Ensuite, nous avons ensuite décidé de conteneuriser une petite application (Gedit) pour nous faire la main et comprendre le procédé.
- Enfin, nous avons conteneurisé Marionnet et avons étudié la possibilité de construire des réseaux virtuels où les machines seraient des conteneurs Docker.

Toutes les commandes effectuées ci-dessous ont été réalisées sur Ubuntu 16.04, une distribution Linux.

I) Fonctionnement de Docker

1) L'installation de Docker

L'application Docker est en elle-même assez simpliste car elle ne contient rien d'utilisable à son installation. Dockeriser une application c'est empaqueter une application et ses dépendances dans un conteneur isolé qui pourra être exécuté sur n'importe quel distribution Linux. À moins d'être un « empaqueteur », l'utilisateur de cet outil doit utiliser des conteneurs déjà existants. Mais avant tout, il faut tout de même installer Docker. Pour ce faire l'utilisateur doit utiliser ces quelques commandes en mode super-utilisateur :

- **apt-get update** : permet à l'utilisateur de mettre à jour et/ou d'installer les paquets qui sont nécessaires au bon fonctionnement de la dernière version de l'OS.
- **apt-get install docker-engine** : permet d'installer l'outil docker.

Pour le reste de ce rapport, on note que toutes les commandes qui commencent par « docker » ont été effectuées en mode super-utilisateur.

2) Le DockerHub

Une bibliothèque publique appelée DockerHub fut créée dans le but que chaque utilisateur de Docker puisse partager ses images avec les autres. Ce système d'échange permet aux utilisateurs d'interagir et de partager leurs travaux. En effet si ces images sont rendues accessibles par leurs hôtes, elles peuvent être utilisées par n'importe qui souhaitant les exploiter, ce qui a pour but de faciliter le travail des programmeurs et aussi les faire évoluer dans leurs travaux. Nous avons créé notre propre hub sur Docker appelé « projettut » afin de pouvoir nous transférer plus facilement nos images. Ces images peuvent être utilisées avec Docker en effectuant plusieurs commandes.

3) La définition des commandes

Afin de pouvoir conteneuriser une application, utiliser un conteneur ou bien même le gérer, peu de commandes sont utilisables.

• Pour pouvoir voir les images présentes sur la session de l'utilisateur, on utilise la commande **docker images**. Pour pouvoir apercevoir tous les conteneurs qui tournent sur un système, on utilise la commande **docker ps** et grâce à l'option **-a**, on peut afficher tous les conteneurs d'un système.

• La commande **rm** permet de supprimer un conteneur, tandis que la commande **rmi** permet de supprimer les images contenues dans notre système.

• La commande **pull** est une commande permettant de récupérer une image à partir du DockerHub et ainsi pouvoir l'exploiter directement sur notre machine. À l'inverse, la commande **push** permet d'envoyer une image que l'on a créée sur le DockerHub afin que d'autres utilisateurs puissent l'utiliser si on le souhaite.

• Lorsque l'on veut créer une image, il faut créer un fichier texte qui s'intitule obligatoirement Dockerfile avec des lignes de codes que nous évoquerons plus tard. Grâce à la commande **docker build -t nom_de_l'image.** , le fichier Dockerfile est transformé en une image. L'option **-t** est un tag permettant de définir le nom de l'image, et le « . » sert à indiquer que l'on crée l'image à parti du Dockerfile de notre répertoire courant.

• L'image ainsi créée, il nous suffira maintenant de l'exécuter en utilisant la commande **docker run nom_de_l'image**. Cette commande comporte de nombreuses options qui permettent de préciser ce que l'on souhaite effectuer :

- **-e** permet d'exécuter une variable d'environnement déjà présente dans notre ordinateur ;
- **-i** nous permet d'interagir à l'intérieur du terminal du conteneur (sans cette commande, tout ce que nous écrirons à l'intérieur de notre conteneur ne sera pas transmis au bash du conteneur) ;
- **-t** sert à afficher un terminal pour notre conteneur ;
- **-v** est une option qui nous permet de monter un répertoire indiqué à l'intérieur du conteneur.

4) La création d'une image : Dockerfile

Avant de construire une image, il faut créer un Dockerfile. Un Dockerfile est un fichier propre à Docker contenant toutes les commandes dont l'utilisateur a besoin pour créer une image et lancer des actions dans le conteneur. L'utilisation du Dockerfile est primordiale dans Docker : c'est une sorte de fichier contenant la recette pour que Docker puisse construire notre image. Lorsque l'on exécute notre commande docker build, Docker construit l'image à partir de ce fichier Dockerfile.

Le Dockerfile comporte différentes directives :

- **FROM** permet de définir la distribution sur laquelle nous exécutons notre conteneur, dans notre cas ce fut Ubuntu:16.04 ;
- **MAINTAINER** permet de nommer la personne qui a écrit le Docker, cette commande n'est pas indispensable ;
- **RUN** permet d'exécuter une commande lors de la construction de l'image ;
- **COPY** permet d'ajouter des fichiers locaux ou distants à l'intérieur de notre image, elle est notamment utilisée ici pour rapatrier des fichiers de configuration à l'intérieur de notre image ;
- **ENV** sert à créer une variable d'environnement ;
- **CMD** ne doit être utilisée qu'une seule fois, c'est une commande que l'on exécute lorsque l'on démarre le conteneur.

Maintenant que nous avons réuni et expliqué toutes les commandes nécessaires à l'utilisation de Docker, nous allons maintenant les appliquer.

Nous avons décidé en premier lieu de conteneuriser une petite application, en l'occurrence gedit, afin de nous exercer et être à l'aise avec la manipulation des commandes Docker. Ceci nous a permis de mieux comprendre le sujet et d'effectuer des tests avant de nous attaquer au gros du projet.

II) La conteneurisation de Gedit

Gedit, est une application de traitement de texte que nous utilisons souvent en cours pour créer des fichiers de programmation ou pour d'autres utilisations diverses Nous avons donc conteneurisé cette application car elle n'utilise qu'une interface graphique et est, de ce fait, assez simple à conteneuriser, et parce qu'elle nous est familière.

1) Méthode 1 : extraction des fichiers de configuration

Pour fonctionner, une application doit être munie de beaucoup de librairies présentes initialement sur la machine. Les librairies d'une application sont les instructions prêtes à être utilisées par les programmes. Pour que ces librairies fonctionnent, elles s'appuient sur des dépendances. Ces dépendances, elles, sont propres à chaque OS. Le Gedit que nous utilisons fonctionne sur Ubuntu, c'est pourquoi nous avons tenté d'extraire les librairies de notre fichier et d'installer ses dépendances dans notre Dockerfile. Pour extraire les librairies, nous avons utilisé la commande **ldd**, puis nous avons tenté d'importer toutes ces librairies dans notre Dockerfile.

Nous avons ensuite tenté, grâce à un script, d'exécuter toutes ces librairies à l'ouverture de notre conteneur pour qu'il puisse installer les dépendances. Nous avons eu accès aux dépendances grâce à la commande **apt-cache showpkg gedit**. Enfin nous avons installé l'exécutable de gedit pour qu'il fonctionne en appelant ses librairies et ses dépendances.

À l'exécution, le message d'erreur nous informe que le module gtk ne peut pas ouvrir le display : c'est le problème de l'interface graphique avec Docker.

```
root@928bc2704a39:/DockerTest# gedit
Failed to connect to Mir: Failed to connect to server socket: No such file or di
rectory
Unable to init server: Could not connect: Connection refused

(gedit:14): Gtk-WARNING **: cannot open display:
```

Figure 1 : Problème avec l'interface graphique

2) Méthode 2 : interface graphique avec SSH

N'ayant pas réussi à avoir une interface graphique, nous avons décidé de changer de méthode.
Nous avons donc décidé grâce aux connaissances que nous avons acquises durant notre formation de créer un lien entre notre conteneur et l'extérieur afin que le conteneur puisse utiliser l'interface graphique de notre machine.

Pour cela nous avons décidé de nous servir du protocole Secure Shell (SSH), qui est un protocole de communication sécurisé permettant de se connecter à distance à une machine afin d'utiliser son shell. Nous avons donc eu l'idée de mettre les lignes de commandes permettant de créer une liaison SSH dans notre Dockerfile comme montré dans l'illustration ci-dessous :

```
FROM ubuntu:16.04

RUN apt-get update && apt-get install -y openssh-server
RUN apt-get -y install gedit

RUN mkdir /var/run/sshd
RUN echo 'root:password' | chpasswd
RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin yes/' /etc/ssh/sshd_config
RUN sed 's@session\s*required\s*pam_loginuid.so@session optional pam_loginuid.so@g' -i /etc/pam.d/sshd
RUN echo "export VISIBLE=now" >> /etc/profile

EXPOSE 22
CMD ["/usr/sbin/sshd", "-D"]
```

Figure 2 : Dockerfile de la méthode 2 : SSH

Dans notre Dockerfile, nous avons d'abord installé gedit à l'aide de l'outil **apt-get** ainsi qu'un serveur SSH **openssh-server**. Nous avons ensuite créé un répertoire **sshd**, puis nous avons indiqué que nous allions utiliser le mot de passe de root. La commande **sed** nous permet d'écrire un programme qui nous permettra de modifier le contenu d'un fichier.

Nous avons donc utilisé ces lignes de commandes pour nous permettre d'utiliser SSH en tant que root et d'éliminer tous les problèmes liés à l'authentification et aux droits d'accès, ce qui est nécessaire si nous voulons utiliser SSH sans problème. Nous avons ensuite exporté une variable d'environnement **VISIBLE** vers le fichier **/etc/profile**, fichier important pour la connexion des utilisateurs. Nous avons ensuite défini le port SSH (22) sur lequel la liaison opérera, et nous avons lancé notre script.

Malheureusement cette solution n'a pas fonctionné, il a donc fallu que l'on trouve une autre solution afin de pouvoir utiliser l'interface graphique de notre ordinateur à partir du conteneur.

3) Méthode 3 : interface graphique avec xauth

N'ayant toujours pas réussi à utiliser une interface graphique avec SSH, nous avons encore décidé de changer de méthode. Pour cette méthode nous nous sommes inspirés d'une vidéo que nous avons trouvée sur Youtube.

Pour réaliser les opérations de cette vidéo nous avons dès le départ effectué la commande **xauth list**. Nous avons mis le résultat cette commande de côté car nous allons le réutiliser plus tard. Nous avons importé la distribution Ubuntu 16.04 avec **FROM** dans le Dockerfile, puis nous avons appliqué la commande **RUN apt-get update && apt-get -y install gedit**.

Nous avons pensé au fait que si l'on installe gedit directement avec la distribution mise en place, les dépendances et les librairies seraient automatiquement installées. Nous avons donc ensuite installé xauth grâce à la commande **RUN apt-get install -y xauth** qui va nous permettre d'utiliser l'interface graphique.

Après avoir construit notre image, nous avons utilisé la commande **docker run -i -t --net=host -e DISPLAY -v /tmp/.X11-unix** pour démarrer le conteneur. Ceci permet au conteneur d'utiliser internet et l'interface graphique. Une fois dans le conteneur, nous devons taper la commande **xauth add + le contenu que nous avions mis sur le côté** pour que l'application puisse sortir du conteneur et utiliser notre vraie distribution de Linux comme interface graphique.

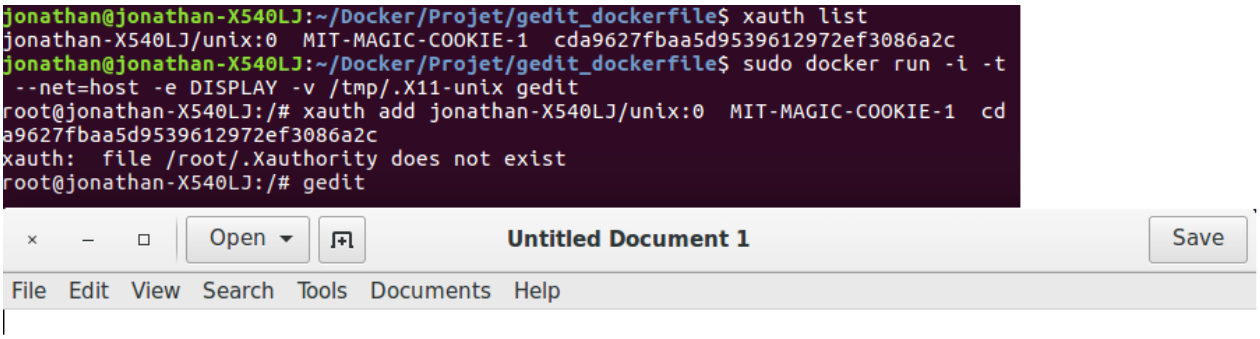


Figure 3: Exécution de gedit avec la méthode 3 : xauth

4) Méthode 4 : interface graphique avec xhost

À ce stade, nous aurions pu estimer que le conteneur fonctionne, mais nous avons décidé que nous devions automatiser entièrement le conteneur pour que l'utilisateur n'ait pas besoin de copier puis coller le contenu de xauth list qui utilise ses données personnelles.

Finalement, dans notre Dockerfile nous avons, comme auparavant, importé la distribution Ubuntu 16.04, mis à jour avec update, et installé gedit. Ensuite, nous avons écrit la commande **CMD ["/usr/bin/gedit"]** pour exécuter directement l'application gedit.

Pour faire fonctionner le conteneur, certaines machines n'autorisent pas par défaut l'utilisation de l'écran pour Docker. Pour ces machines la commande à effectuer est **xhost +local:docker**.
Enfin, on peut run l'image pour démarrer le conteneur : **docker run -e DISPLAY=$DISPLAY -v /tmp/.X11-unix/:/tmp/.X11-unix nom_image**

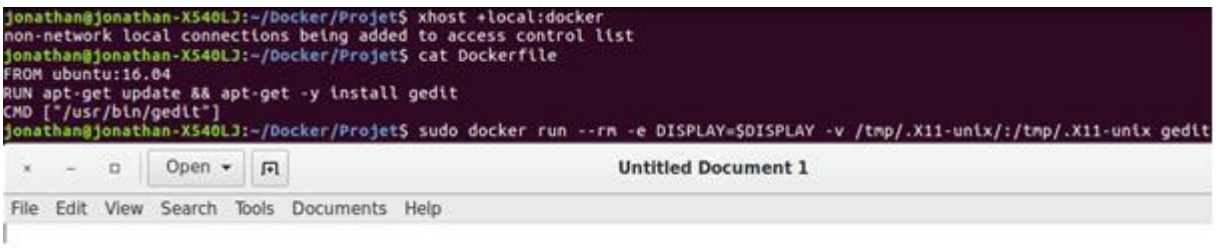


Figure 4 : Exécution de gedit avec la méthode 4 : xhost

III) La conteneurisation de Marionnet

Maintenant que nous savons conteneuriser une application simple, nous allons réaliser ce que nous demande le cahier des charges, c'est-à-dire de conteneuriser l'application Marionnet.

Marionnet est une application de simulation de réseaux virtuels TCP/IP afin de lancer des réseaux complexes de machines sans utiliser de supports physiques.
On peut l'installer en téléchargeant tout d'abord le fichier d'installation sur le site de Marionnet à l'adresse suivante :

http://www.marionnet.org/downloads/marionnet_from_scratch/marionnet_from_scratch

1) Commandes à effectuer avant d'utiliser Docker

Nous prenons d'abord le daemon de l'application et le plaçons dans le répertoire du Dockerfile en le renommant **marionnet-daemon** comme l'indique le fichier d'installation marionnet_from_scratch : **cp /etc/init.d/marrionet marionnet-daemon**

Ensuite, comme pour gedit auparavant, pour pouvoir utiliser l'interface graphique de Marionnet il faut autoriser l'application Docker à utiliser le contrôle d'accès au serveur X. Pour cela, sur notre PC, nous devons exécuter la commande **xhost local:docker**.

2) Le Dockerfile

• Nous indiquons tout d'abord quelle sera la distribution de Linux dans le conteneur : **FROM ubuntu:16.04**

• Marionnet possède beaucoup de dépendances, il faut donc toutes les installer avec la commande **RUN apt-get update && apt-get install -y**. Parmi ces librairies on trouve xterm, indiquant quel est le terminal utilisé, ou encore x11-xserver-utils qui permet d'utiliser l'interface graphique dans le conteneur.

• Nous installons ensuite les packages dont on aura besoin après la construction du Dockerfile :
wget : permettra de télécharger le fichier d'installation ;
sudo : permettra d'utiliser des commandes en tant que super-utilisateur ;
libcanberra-gtk-module : permettra d'utiliser gtk pour l'interface graphique.

• Nous téléchargeons ensuite le fichier d'installation et lui mettons le droit d'exécution :
RUN wget
http://www.marionnet.org/downloads/marionnet_from_scratch/marionnet_from_scratch
RUN chmod +x marionnet_from_scratch

• On rajoute xterm en variable d'environnement du conteneur avec la commande **ENV TERM=xterm** afin de pouvoir utiliser l'émulateur de terminal habituel du shell pour les machines virtuelles de Marionnet. Si l'on ne met pas cette commande, la construction du Dockerfile s'arrêtera et le shell renverra le code retour 2 défini dans le fichier marionnet_from_scratch indiquant que l'installation s'est arrêtée à cause d'un signal.

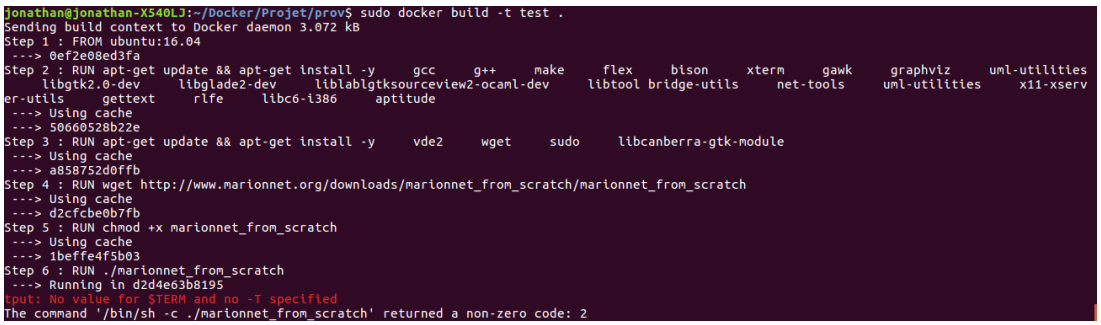


Figure 5 : Erreur due à l'absence de la variable TERM

• On supprime le fichier /etc/issue avec **RUN rm /etc/issue** pour éviter que le programme d'installation lise ce fichier et nous demande d'ajouter une ligne dans ce fichier. Sinon, le shell renverra le code retour 3 indiquant que l'installation s'est arrêtée à cause d'une erreur inattendue. Cette erreur est due au fait que l'on ne donne pas de valeur à la question du programme.

```
  Shall I install the marionnet daemon in your runlevels ([y]/n)? Exiting because of an unexpected error in line 1537
The command '/bin/sh -c ./marionnet_from_scratch' returned a non-zero code: 3
```

```
function are_we_in_ubuntu_11_or_greater {
 if [[ -f /etc/issue ]]; then
   local A=$(head -n 1 /etc/issue)
   local a b c
   read a b c <<<"$A"
   if [[ "$a" = "Ubuntu" ]]; then
     IFS="." read a c <<<"$b"
     [[ "$a" -ge 11 ]]
   else
     return 1
   fi
 else
  return 1
 fi
}
```

Figure 6 : Erreur 3 due à l'existence du fichier /etc/issue

• Ensuite, nous copions le daemon de Marionnet (qui doit être dans le répertoire courant pour utiliser COPY) dans le répertoire /etc/init.d : **COPY marionnet-daemon /etc/init.d**

• Nous exécutons le fichier d'installation avec **-m** pour utiliser la version en cours de développement (trunk). Cette exécution va télécharger et installer les noyaux et les systèmes de fichiers des différentes distributions de GNU/Linux : **RUN ./marionnet_from_scratch -m trunk**

• On se place dans le répertoire /dev contenant les systèmes de fichiers avec la commande **WORKDIR /dev** pour éviter un message d'erreur de la part de Marionnet.

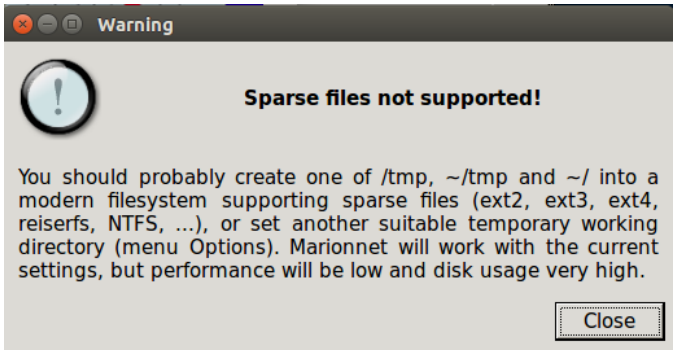


Figure 7 : Message d'erreur car l'on n'est pas dans un système de fichier (/dev par exemple)

• Avec la commande CMD, on démarre le daemon de Marionnet placé dans /etc/init.d auparavant, et on démarre Marionnet : **CMD /etc/init.d/marionnet-daemon start && marionnet**

Si on utilise une commande RUN à la place de CMD pour démarrer le daemon, ce dernier démarrera dans la construction de l'image mais sera arrêté dans le conteneur, et l'avertissement suivant sera affiché lorsque l'on démarrera Marionnet :

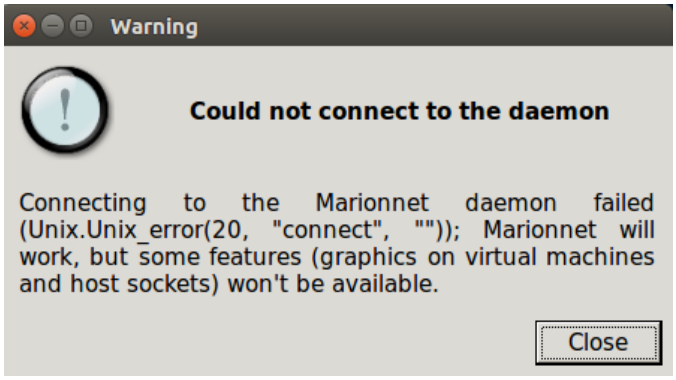


Figure 8 : Message d'erreur car le daemon n'est pas démarré

Nous avons fait des recherches pour démarrer directement le daemon lors du démarrage du conteneur grâce à systemd et pas en utilisant la commande CMD, mais nous n'avons trouvé aucun moyen de le faire.
Ceci est un petit problème étant donné que Marionnet nous renvoie un avertissement si on exécute l'application en tant que root alors qu'il faut utiliser root pour démarrer le daemon, et on ne peut utiliser qu'une commande CMD dans un Dockerfile, ce qui fait que l'on ne peut pas changer d'utilisateur.
Voici ce que nous avons essayé : le propriétaire du fichier est « test », on exécute le daemon en tant qu'utilisateur « test » (comme on le voit avec whoami) et le daemon est démarré. Or, si on regarde le statut, le daemon de Marionnet est arrêté.

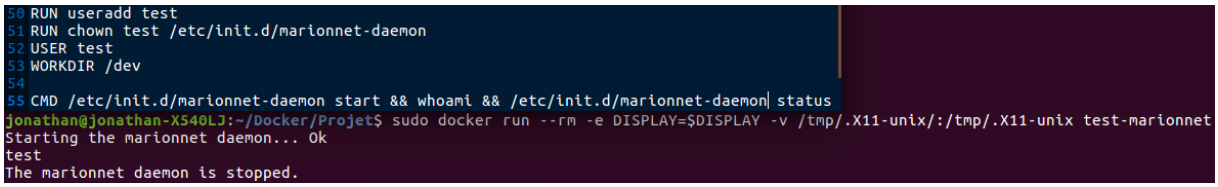


Figure 9 : Problème daemon / changement d'utilisateur

3) Les commandes Docker

Nous pouvons maintenant construire notre image de Marionnet. Nous faisons cela avec la commande **docker build -t marionnet .**

Une fois les 7 GB de l'application construits, nous pouvons démarrer le conteneur avec la commande **sudo docker run -e DISPLAY=$DISPLAY -v /tmp/.X11-unix/:/tmp/.X11-unix test-marionnet**
Le **-e** prend la valeur de la variable display de l'ordinateur exécutant la commande run et la place en variable d'environnement dans le conteneur avec la même valeur.

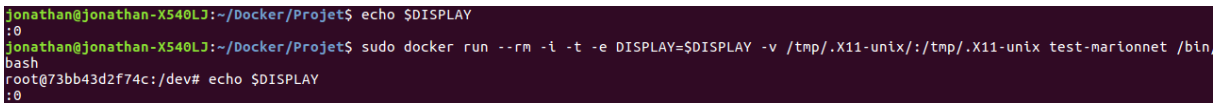


Figure 10 : DISPLAY à l'intérieur et à l'extérieur du conteneur

Le **-v** crée un volume en prenant les fichiers de /tmp/.X11-unix de notre ordinateur pour les placer au même endroit dans les conteneurs.

Maintenant que l'on a run notre conteneur, le service du daemon démarre et Marionnet s'exécute : l'application a été conteneurisée.

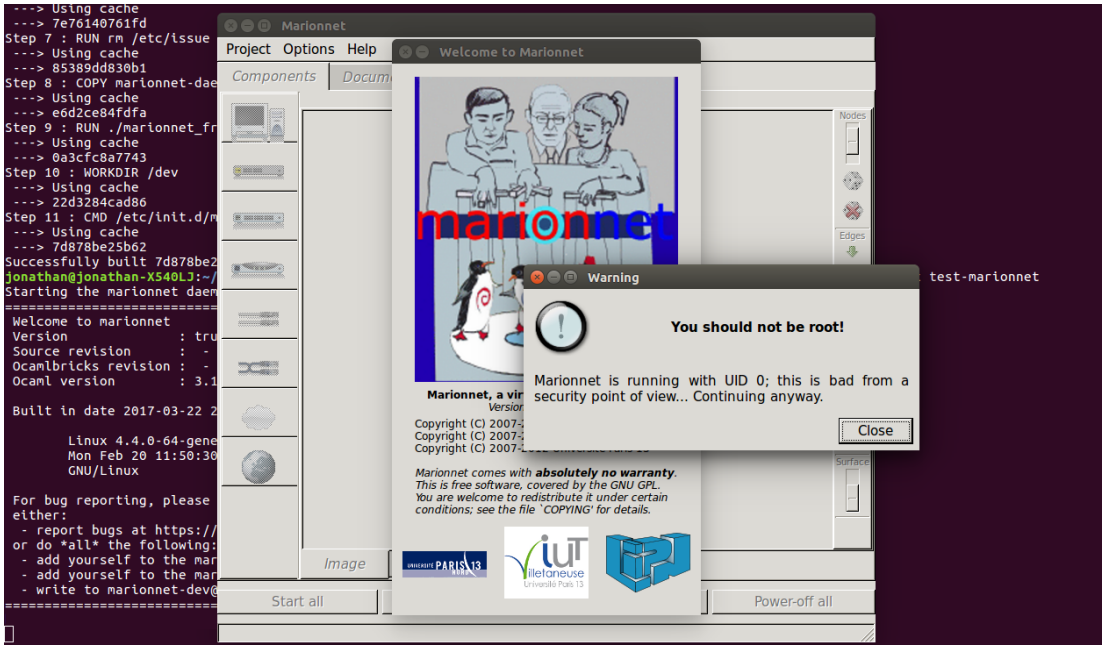


Figure 11 : Marionnet conteneurisée

IV) Conteneurisation des machines virtuelles de Marionnet

1) Le cahier des charges provisoire

Nous cherchons un moyen de conteneuriser les machines virtuelles de l'application Marionnet.
Quand on clique sur **start** dans Marionnet, une machine et son conteneur sont créés : **docker start**.
Quand on clique sur **suspend** dans Marionnet, une machine et son conteneur sont interrompus : **docker stop**.
Quand on clique sur **stop** dans Marionnet, une machine et son conteneur sont définitivement arrêtés : **docker kill**.
De plus, il faudrait lier les conteneurs avec **docker attach** lorsque l'on ajoute un câble pour que les machines puissent communiquer.

2) Le projet

La finalité de notre projet est d'étudier la possibilité de créer des réseaux virtuels où les machines de Marionnet seraient des conteneurs, et où l'infrastructure réseau permettant la communication entre ces machines serait fournis par la technologie VDE, c'est-à-dire l'ajout des équipements tels que les hubs, les routeurs, ou encore les switchs.

Durant la première partie de notre projet nous avons réussi à conteneuriser l'application Marionnet entièrement. Le but est maintenant de conteneuriser les machines virtuelles instanciées par Marionnet. Ainsi nous pourrons simuler nos réseaux sans avoir à recréer nos machines.

De ce fait, les actions interagissant avec les machines à l'intérieur de Marionnet doivent pouvoir être exécutées à partir d'un Dockerfile étant donné que nos machines seront des conteneurs.
Lorsque l'on clique sur « start » à l'intérieur de Marionnet la machine virtuelle est lancée et le terminal est affiché. Nous pouvons ainsi y exécuter nos commandes et configuré notre machine.

Cela doit pouvoir être possible même lorsque la machine virtuelle est un conteneur. Pour cela il faudrait peut-être modifier le fichier de configuration de Marionnet afin de pouvoir rendre les machines virtuelles indépendantes de l'application et ainsi pouvoir les conteneuriser.

Restera ensuite le problème des commandes à exécuter qui sont la suspension de la machine via le bouton « suspend » sur Marionnet qui permet d'éteindre une machine et la commande stop de Marionnet qui permet d'arrêter définitivement une machine.

Une solution que nous avons envisagée pour lier les conteneurs contenant les machines virtuelles avec la commande « docker attach » qui permettra, lorsqu'on ajoutera des câbles, de relier les machines entres elles et qu'elles puissent communiquer.

Nous n'avons malheureusement pas eu le temps d'approfondir cette partie du projet et de rassembler toutes les solutions possibles en vue de l'élaboration de ce nouveau réseau particulier.

Conclusion

Nous avons aimé travailler sur ce projet car, souhaitant tous les quatre poursuivre nos études, il nous a permis de découvrir et de nous approprier une nouvelle technologie qui pourrait nous être très utile. En effet la technologie Docker a de nombreux avantages, comme le fait qu'on puisse partager nos conteneurs via le Docker hub et ainsi interagir avec les projets d'autres utilisateurs comme nous : c'est en quelque sorte une technologie de partage de connaissances. La technologie Docker permet aussi de faciliter la vie de ses utilisateurs, ceux-ci n'ayant plus besoin de faire un dur travail d'installation et de téléchargement étant donné que tout cela se fait directement en lançant le conteneur.

Ayant beaucoup appris de cette technologie, nous pourrons la réutiliser pour des projets futurs ou dans le cadre de notre vie professionnelle. Ceci pourrait donc nous permettre d'approfondir nos connaissances et d'avoir de nouveaux outils de travails.

Ce projet fut aussi le premier projet de groupe que nous avons appréhendé de manière professionnelle. Dans le cadre de notre projet tuteuré nous avons décidé de fonctionner comme une équipe d'administrateurs réseaux avec un chef de projet et différentes tâches assignées à chacun, cela nous a permis de nous immerger dans le monde de l'entreprise et de comprendre comment travailler efficacement et uniformément en groupe. Cette méthodologie a eu pour but principal de nous préparer à notre stage de fin d'études.

Nos travaux ont suscité l'intérêt de certains techniciens réseaux qui nous ont reçus en entretien de stage et qui nous ont demandé comment fonctionnait cette technologie.

Pour finir, grâce à ce projet nous avons développé notre capacité d'écoute, d'autonomie et de communication.

Annexe

Dockerfile pour la conteneurisation de Marionnet

```
FROM ubuntu:16.04

MAINTAINER David, Jérémy, Jonathan, Thanushan

# Installation des applications qu'utilise Marionnet
RUN apt-get update && apt-get install -y \
    gcc \
    g++ \
    make \
    flex \
    bison \
    xterm \
    gawk \
    graphviz \
    uml-utilities \
    libgtk2.0-dev \
    libglade2-dev \
    liblablgtksourceview2-ocaml-dev \
    libtool bridge-utils \
    net-tools \
    uml-utilities \
    x11-xserver-utils \
    gettext \
    rlfe \
    libc6-i386 \
    aptitude \
    vde2

# Installation des applications utiles pour le cahier des charges et le Dockerfile
RUN apt-get install -y \
    wget \
    sudo \
    libcanberra-gtk-module

# Téléchargement du fichier d'installation de Marionnet
RUN                                                                                    wget
http://www.marionnet.org/downloads/marionnet_from_scratch/marionnet_from_scratch
RUN chmod +x marionnet_from_scratch

# Commandes pour faire fonctionner le fichier d'installation
ENV TERM=xterm
RUN rm /etc/issue

# Copie du démon marionnet
COPY marionnet-daemon /etc/init.d

# Téléchargement de Marionnet
RUN ./marionnet_from_scratch -m trunk
```

```
# Mettre /dev en tant que pwd lors de l'entrée dans le conteneur
WORKDIR /dev

# Démarrage du démon et de l'application
CMD /etc/init.d/marionnet-daemon start && marionnet
```

Bibliographie

I) Fonctionnement de Docker
https://docs.docker.com/engine/getstarted/step_one
https://remileblond.fr/2016/01/docker-intro
https://docs.docker.com/v1.10/engine/reference/commandline/run
http://linuxfr.org/news/docker-tutoriel-pour-manipuler-les-conteneurs
https://docs.docker.com/engine/reference/builder
https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices
https://www.digitalocean.com/community/tutorials/docker-explained-using-dockerfiles-to-automate-building-of-images
https://hub.docker.com/r/projettut

II) Conteneurisation de gedit
https://forums.docker.com/t/start-a-gui-application-as-root-in-a-ubuntu-container/17069
https://dzone.com/articles/docker-how-to-ssh-to-a-running-container
https://forums.docker.com/t/ssh-from-a-container-to-the-host-os/7958
https://fr.wikipedia.org/wiki/X_Window_System
https://www.youtube.com/watch?v=RDg6TRwiPtg
Pages de manuel de apt-get, apt-search, ssh, xauth et xhost

III) Conteneurisation de Marionnet
http://www.marionnet.org/site/index.php/fr/documentation/installation/a-partir-des-sources
https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=712093
https://answers.launchpad.net/marionnet/+question/281179
http://stackoverflow.com/questions/24549746/switching-users-inside-docker-image-to-a-non-root-user
http://www.linux-france.org/~mdecore/linux/doc/memo2/node46.html
https://fr.wikipedia.org/wiki/Xterm
https://www.debian.org/doc/debian-policy/ch-opersys.html

IV)
http://blog.cquad.eu/2014/09/21/docker-les-images-et-comment-faire-communiquer-deux-conteneurs/

Glossaire

Application : Programme exécutable à partir d'un système d'exploitation

Conteneur : Machine virtuelle qui n'inclut pas de système d'exploitation

Daemon : Processus qui s'exécute en arrière-plan

Empaqueter : Ranger dans des boites

Interface Graphique : La vision du logiciel qu'a l'utilisateur via son écran

Image (Docker) : Une image est un conteneur statique. On pourrait comparer une image à une capture d'un container à un moment donné

Librairies : Fonctions utilitaires regroupées et mises à disposition pour être utilisée par les programmes

Logiciel : Ensemble de programme relatif au fonctionnement d'un ensemble de traitement de données

Logiciel d'émulation : Substitution d'un élément matériel par un logiciel

Machine virtuelle : Appareil informatique créé par un logiciel d'émulation

Processus : Programme en cours d'exécution

Programme : Ensemble d'opération destinée à être exécutée par un ordinateur

Protocoles : Ensemble de règle à respecter

Script : Programme en langage interprété qui permet de manipuler les fonctionnalités d'un système informatique

Shell : Interpréteur de commande qui permet d'accéder aux fonctionnalités internes du système

Système d'exploitation : Ensemble de programme qui créé une interactivité entre le matériel et le logiciel

Terminal : Point d'accès de communication entre l'utilisateur et la machine

Table des Illustrations