UNIVERSITÉ D'ÉVRY – VAL D'ESSONNE

Réalisation en python d'un outils de gestion de services distribués

à l'aide de l'API python clustershell

Guillaume Dubroeucq
Théo Poccard
Nicolas Chapron

le 18 octobre 2016

Professeur
Patrice LUCAS
adresse mail
patrice.lucas@cea.fr

Année universitaire 2016-2017

Table des matières

1 Introduction

1.1 Objectif

Développée et utilisée au CEA, Clusterhell est une bibliothèque en Python qui permet d'exécuter en parallèle des commandes distantes sur des nœuds au sein d'un cluster. Elle fournit 3 outils en ligne de commande qui nous permettent de bénéficier des fonctionnalités de la bibliothèque : clush,nodeset et clubak. ClusterShell fournit également une API Python permettant d'intégrer les fonctionnalités de ClusterShell au sein d'un script python.
Ce projet nous demande de réaliser un outil de gestion distribuée permettant d'administrer des services de système sur plusieurs nœuds, et cela en utilisant l'API Python ClusterShell.

1.2 Projet

Ce projet est composé de 3 parties distinctes :
— Une version simple en ligne de commande
— Une version accueillant un fichier de configuration
— Une version avec une interface graphique

Les deux scripts ainsi que l'interface graphique seront entièrement programmé en python version 2.7.

Nous allons dans un premier temps implémenter une version basique de gestion de services avec des fonctionnalités simple comme : start, stop, restart , etc.. sur un ensemble de nœuds distants. Puis une fois cette base réalisée, nous allons mettre en place une configuration statique de la répartition des services grâce à des fichiers de configuration. Pour finir nous développerons une IHM à partir des éléments déjà crées afin de parfaire l'outil de gestion des services distribuées.

2 ClusterShell

2.1 Présentation

ClusterShell est un outil d'administration distribuée. Il permet d'exécuter des commandes à distance sur un ensemble n de nœuds.

Pour que ClusterShell fonctionne, il faut installer le paquet "clustershell" dû coté master (celui qui va administrer les nœuds à distance). Cette outil est agent-less coté client, c'est à dire qu'il n'y a pas de service à installé sur les nœuds à administrer, ClusterShell nécessite seulement au préalable d'avoir une connexion SSH valide (accès avec échanges de clés et sans mot de passe) sur chacun des nœuds qu'il va administrer.

2.2 Commandes CLI

Commençons tout d'abord par définir les 3 fonctionnalités de la bibliothèque ClusterShell définit plus haut : crush,nodeset et clubak.

- Nodeset : Permet la création et la manipulation de liste de nœuds . En effet on peut créer des listes machines ainsi que des étendus de nœuds, on peut effectuer plusieurs opérations sur ces listes (union, exclusion, intersection , etc...). Cela facilite fortement la manipulation des noeuds.
- Clush : Permet l'exécution des commandes en parallèle sur des noeuds distants, prend également en charge les groupes de noeuds.
- Clubak : Regroupement de sorties standards qui permet de présenter de manière plus synthétique un résultat d'exécution un peu trop verbeux et répétitif.

2.3 API Python

ClusterShell délivre une API python permettant de manipuler cette outil dans nos propres scripts python.

Pour utiliser cette API, il suffit de télécharger le paquet "clustershell".

Par la suite, il suffit d'intégrer l'api python ClusterShell à notre script avec cette ligne :

```
from ClusterShell.Task import task_self, NodeSet
```

FIGURE 1 – Import

3 Script gestion des services

3.1 Présentation

Cette première partie consiste à créer un script en python permettant d'utiliser les fonctionnalités de base des services, en étendant cette fonctionnalité de façon distribuée sur un ensemble choisi de nœuds. Pour ce faire, on utilise l'api python Clustershell qui va nous permettre de créer un script pouvant effectuer ces actions.

3.2 Fonctionnement

Le script fonctionne très simplement, il prend 3 éléments en paramètre : les nœuds que l'on veut utiliser, le service concerné et la commande que l'on veut exécuter. Comme par exemple :

`./script.py node2 cron restart` ; cela correspond à relancer le service cron sur le node 2.

```
7   import sys
8   from ClusterShell.Task import task_self, NodeSet
9
10  def main():
11      if len(sys.argv) <= 3:
12          print "Veuillez entrer des paramètres (./script.py node[1-3] cron stop)"
13      else:
14          if((len(sys.argv)%2)==0):
15              try:                # vérifie les erreurs de syntaxe pour les nodes
16                  nodeset=NodeSet(sys.argv[1])
17              except:
18                  print("Erreur: Problème avec la syntaxe \"%s\"" % sys.argv[1])
19                  return
20              for i in range(2,len(sys.argv)):
21                  if i%2 == 0:    # pour chaque chiffre pair
22                      if(sys.argv[i+1] == 'start' or 'stop' or 'status' or 'restart' or 'reload'):
23                          task = task_self()
24                          cli="service %s %s"% (sys.argv[i], sys.argv[i+1])
25                          task.shell(cli, nodes=sys.argv[1])
26                          task.run()
27                          for output, nodelist in task.iter_buffers():
28                              print ''
29                              print '%s: %s' % (NodeSet.fromlist(nodelist), output)
30                      else:
31                          print("Erreur: Veuillez vérifier que la commande soit compris dans [start,stop,status,restart,reload]")
32          else:
33              print("Erreur: Veuillez vérifier le nombre de paramètres")
34
35  if __name__ == '__main__':
36      main()
37
```

FIGURE 2 – Script Simple

Le script en lui même vérifie dans un premier temps si des éléments sont mis en paramètre puis si il n'y a pas d'erreur de synthaxe sur les nœuds.
Une fois ces deux conditions remplies, on vérifie que la commande demandée corresponde à une des commandes suivantes : start,stop,status,restart,reload. On concatène les informations qui ont été passés en paramètre. Task-shell permet d'ajouter les commandes (cli) à exécuter à distance sur les nœuds. Et enfin on exécute le tout.
La dernière boucle for permet d'afficher les erreurs obtenues.

4 Script avec fichier de configuration

4.1 Présentation

Cette deuxième partie consiste à améliorer le script précédent en ajoutant un fichier de configuration. Le script aura besoin seulement d'un fichier de configuration en argument contrairement à précédemment où il avait besoin d'une description complète des noeuds, des services et des actions à effectuer. Le fichier de configuration regroupera en lui les actions à accomplir. Nous avons choisi d'utiliser le langage YAML pour le fichier de configuration.

4.2 YAML

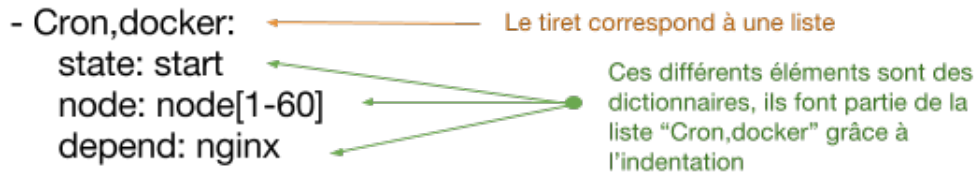
4.2.1 Présentation

YAML (Yaml Ain't Markup Language) est un format de représentation des données par sérialisation, c'est à dire que YAML fait passer d'un format de description ("human friendly") à un objet représenté par des combinaisons de listes, dictionnaires et données scalaires.
YAML est conçu pour être facilement manipulable par des humains, il permet par la suite de facilement manipuler les données décrient par l'utilisateur dans de nombreux langages (C/C++,Python,Java,PHP,...).

4.2.2 Fichier de configuration .YAML

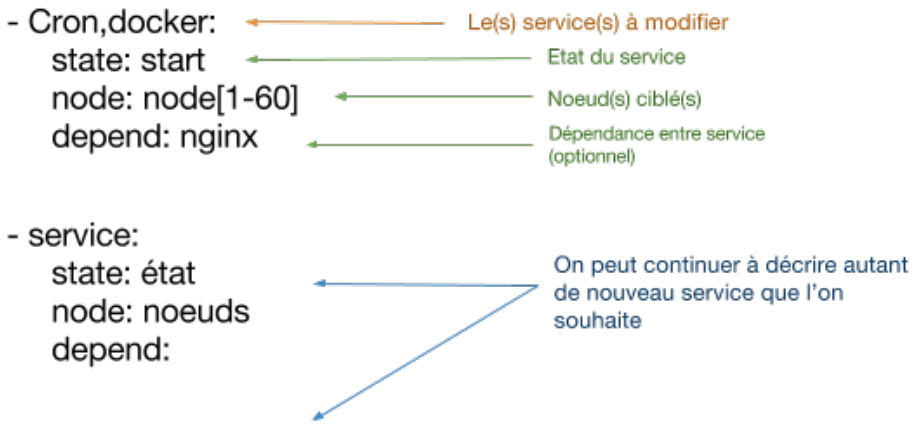
Nous allons voir comment se compose un fichier YAML, il y a énormément de manière de synthétiser un fichier YAML, nous ne pourrons pas voir toutes les possibilités offertes par ce langage.

Voici un exemple de fichier .yaml :



Dans notre cas, nous utilisons des collections (crées par l'indentation, voir l'exemple si dessus), il existe aussi les scalaires, c'est à dire l'ensemble des types qui ne sont pas des collections (donc aucune indentation), nous n'en utiliserons pas dans notre cas.

Voici ci-dessous la syntaxe du fichier YAML utilisé dans notre script :



4.3 Installation

Pour pouvoir utiliser le langage YAML, il faut au préalable installer le paquet "pyyaml" disponible dans le gestionnaire de paquets pip.

```
pip install pyyaml
```

Par la suite, pour inclure YAML dans notre script python, il faut importer sa librairie :

```
import yaml
```

À présent, nous possédons tous les outils pour pouvoir utiliser YAML dans notre code python.

4.4 Script

Il suffit maintenant d'ouvrir le fichier YAML en question qui sera passer en argument du script de la façon suivante :

```
fichier = sys.argv[1]
with open(fichier,'r') as stream:
    try:
        doc=yaml.safe_load(stream)
```

Si il n'y a pas d'erreur dans l'ouverture et la sérialisation, on récupère un objet python (dans ce cas la variable "doc") parfaitement fonctionnelle.

Il faut ainsi vérifier la syntaxe de cette objet pour s'assurer que l'utilisateur a bien respecter la syntaxe imposée par notre script :

```
 doc=yaml.safe_load(stream)
#debug(doc) # information (optionnel)
 if(check_service(doc)): # Contrôle les services
    service=add_key(doc) # Met les services en état
    passe=0
    for r in sys.argv:
        if(r=="-force"):
            passe=1
    if(passe==0):
        while(rep !='y' and rep !='n'):
            rep = raw_input("Confirmer (y/n) : ")
        if(rep=='y'):
```

La fonction `check_service(doc)` permet une première vérification de la syntaxe du fichier YAML, notamment si le fichier comporte des listes (requis dans notre cas).

La fonction `add_key(doc)` permet de récupérer tout les services du fichier et de les afficher à l'utilisateur, ce dernier devra valider lui même si les services affichés sont cohérent et dans ce cas là, valider. Il est possible d'outre passer cette vérification en ajoutant "-force" en argument, c'est utile si on a besoin d'automatiser le script. Il est toutefois fortement conseiller de vérifier le fichier YAML avant de l'automatiser.

Dès que l'utilisateur confirme le contenu du fichier ou qu'il ait forcé le passage avec "`-force`", le script continu :

```
if(rep=='y'):
    print ""
    if(check_attribut(doc,service)): # Contrôle les attributs des services
        recapitulatif=clustershell(doc,service)
        recap(service,recapitulatif)
```

La fonction `check_attribut(doc,service)` permet de contrôler les attributs de chaque service pour vérifier si il n'en manque pas et qu'ils ne sont pas vide ou erroné.

Après cette vérification, la fonction `clustershell(doc,service)` est appellé, le script commence à vérifier notamment si il y a des dépendances grâce à une fonction interne à `clustershell(doc,service)` et dans ce cas, le script vérifie si les services dépendants sont installés et activés. Si ce il manque une dépendance, toutes les actions liées à cette dépendance sont annulées Si tout est bon, le script effectue sur les noeuds les commandes distribuées dans `clustershell(doc,service)`.

4.5 Affichage

Notre setup de test est le suivant :
— Un master
— 3 noeuds appelés respectivement node1,node2 et node3
La particularité est que seulement node1 possède nginx d'installé contrairement à node2 et node3.

Tout les noeuds sont accessibles par le master par SSH sans mot de passe. On commence par un cas simple :

```
---
- cron:
    state: start
    nodes: node[1-3]
```

FIGURE 3 – example1.yaml

On exécute le script avec le fichier example1.yaml en argument :

```
./yamlscript.py example1.yaml
```

FIGURE 4 – exécution du script

Le script nous demande de confirmer le contenu du fichier `example1.yaml`, dans notre cas le script affiche bien le service cron. On rappelle que l'on peut outrepasser cette confirmation en ajoutant "`-force`" en argument du script.

```
D'après le fichier "example1.yaml", les services concernés sont:
   0: cron

Confirmer (y/n) :
```

FIGURE 5 – Confirmation

Pour finir, le script fait son travail avec ClusterShell sur les noeuds indiqués et indique le résultat des commandes, si il y a "OK", cela veut dire que la commande sur l'intégralité des noeuds a été un succès. Il y a une partie "RECAP" qui permet de voir en fin de script sur quel service il y a eu des erreurs ou non.

```
TASK: [cron] ********************************************************************************
cron           :  state=start     nodes=node[1-3]
OK

RECAP ***************************************************************************************
[cron]cron                     : ok=1         failed=0
```

FIGURE 6 – Output de l'exécution example1.yaml

Maintenant nous allons faire un fichier yaml plus complet en provoquant volontairement des erreurs, voici le fichier `example2.yaml` :

```yaml
- cron,nginx:
    state: start
    nodes: node[1-3]
    depend: ssh
- ssh:
    state: start
    nodes: node123
- cron:
    state: start
    nodes: node[1-3]
    depend: nginx
- ssh:
    state: start
    nodes: node[1-3]
```

FIGURE 7 – example2.yaml

Ce fichier example2.yaml comporte des erreurs volontaires comme l'activation de nginx sur tout les noeuds (nginx est installé seulement sur node1), l'activation de ssh sur node123 (qui n'existe pas), ainsi que la dépendance nginx sur cron sur tout les noeuds.

```
TASK: [cron,nginx] **************************************************************************
cron           :  state=start     nodes=node[1-3]
OK

nginx          :  state=start     nodes=node[1-3]
Error: node[2-3]: Failed to start nginx.service: Unit nginx.service failed to load: No such file or directory.

TASK: [ssh] *********************************************************************************
ssh            :  state=start     nodes=node123
Error: node123: ssh: Could not resolve hostname node123: Name or service not known

TASK: [cron] ********************************************************************************
cron           :  state=start    nodes=node[1-3]
Error depend node[2-3]: le(s) service(s) ['nginx'] n'est(sont) pas activé(s) ou installé(s)

TASK: [ssh] *********************************************************************************
ssh            :  state=start     nodes=node[1-3]
OK

RECAP ***************************************************************************************
[cron,nginx]cron               : ok=1         failed=0
[cron,nginx]nginx              : ok=0         failed=1

[ssh]ssh                       : ok=0         failed=1

[cron]cron                     : ok=0         failed=1

[ssh]ssh                       : ok=1         failed=0
```

FIGURE 8 – Output de l'exécution example2.yaml

On constate les erreurs évidentes. Dans une commande distribuée, sur 10 noeuds, il suffit d'une seule erreur pour que l'ensemble de l'action soit marqué d'un "FAIL".

5 Création d'une IHM

Pour la création d'une interface graphique, nous nous sommes tournés vers l'environnement de développement Qt. Qt est basé sur le langage C++ pour créer ses IHM. Cependant, il existe le module PyQt permettant de programmer facilement une interface graphique en python.

5.1 Qt Creator et PyQt

5.1.1 La conversion ui > py

Au départ, on utilise Qt Creator pour pouvoir créer les fenêtres avec tous les composants graphiques nécessaires. Lorsque l'on crée une fenêtre, Qt nous génère un fichier **.ui**. A l'aide de l'utilitaire **pyuic**, on peut convertir ce fichier en python.

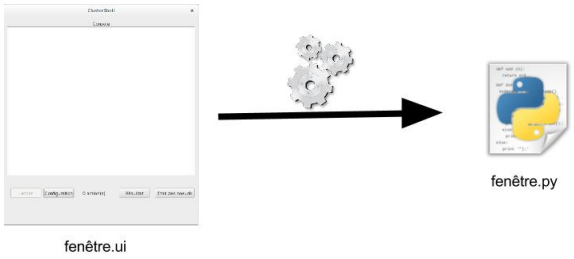


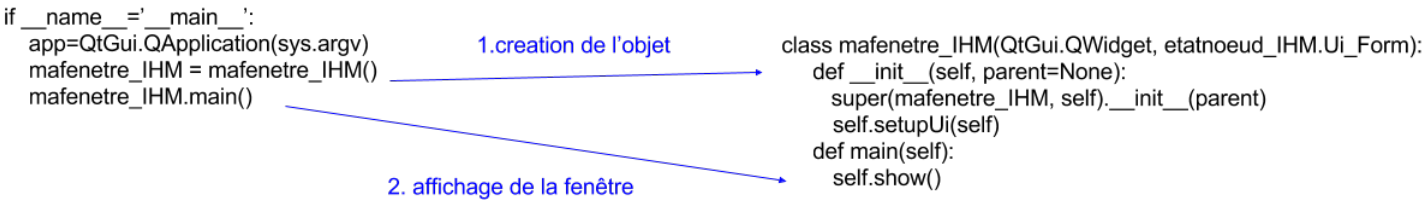
FIGURE 9 – Conversion ui - py

Pour convertir on utilise la commande suivante : **pyuic4 fenêtre.ui > fenêtre.py**

5.1.2 Les classes d'interfaces graphique

Une fois généré, le fichier python (qui représente la fenêtre) est composé d'une classe qui porte le nom **Ui_Form** pour une fenêtre ou **Ui_MainWindow** si c'est la fenêtre principale. Dans cette classe se trouve une fonction qui s'appelle **setupUI** qui a pour rôle d'initialiser et configurer tous les composants graphiques qui se trouve dans la fenêtre (exemple : un bouton avec sa taille de base, sa taille maximum etc...). C'est la première fonction à être appelé lors du lancement de la fenêtre.

Pour afficher la fenêtre, on doit importer le fichier et instancier la classe générée ci-dessus dans un nouveau fichier python (IHM.py dans notre cas) qui sera le fichier principal (fichier expliqué un peu plus loin) :



5.1.3 Les signaux et les slots

Pour de la programmation événementielle, on utilise deux moyens qui sont propres à Qt : les **signaux** et les **slots**. Chaque composant graphique (comme un bouton) possède des signaux et des slots qui vont permettre d'intéragir avec d'autres composants et fonctions(exemple : ouvrir une fenêtre via un bouton).

Un signal : Un signal est un message envoyé par une classe lors du déclenchement d'un événement comme le clic sur un bouton
Les slots : Les slots sont tout simplement des fonctions qui seront déclenchés par les signaux. Les fonctions peuvent être créées par nous même ou cela peut être des fonctions propres à une classe de Qt (exemple : la fonction **quit** de **QApplication** qui quitte le programme.

Voici un petit exemple pour mieux comprendre :

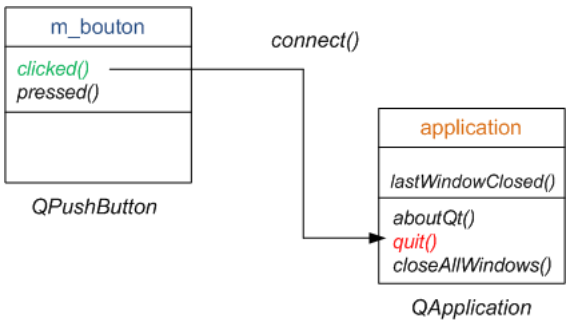
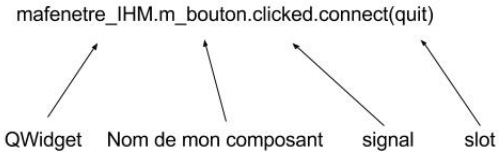


FIGURE 10 – Les signaux et slots

Pour pouvoir assigner un slot à un signal on doit utiliser la fonction **connect** que l'on définit dans la classe de notre fenêtre :



La définition d'un slot se fait exactement de la même manière que celle d'une fonction de base. Il faut également ajouter "**@pyqtSlot()**" devant la fonction. C'est grâce à pyqtSlot que Qt va différencier les slots des fonctions.
Voici un exemple de slot :

```
@pyqtSlot()
def monslot():
    print "Hello World"
```

5.1.4 Le programme principal

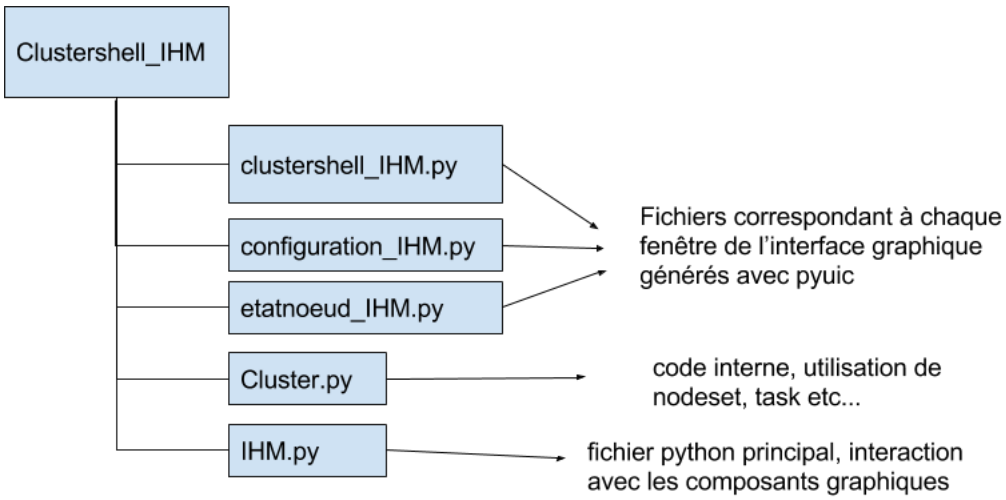
Comme dans tout fichier python, une fonction main existe et doit être appelée pour lancer le programme principal. Le fichier python principal (**IHM.py**) va contenir :

- Les signaux et les slots
- Les classes d'interface graphique
- Le code lié aux interactions avec les composants graphiques (ajout d'item dans une **QListWidget**, variation de la **QProgressbar**, récupération des informations d'une **QLineedit** (barre de saisie)).

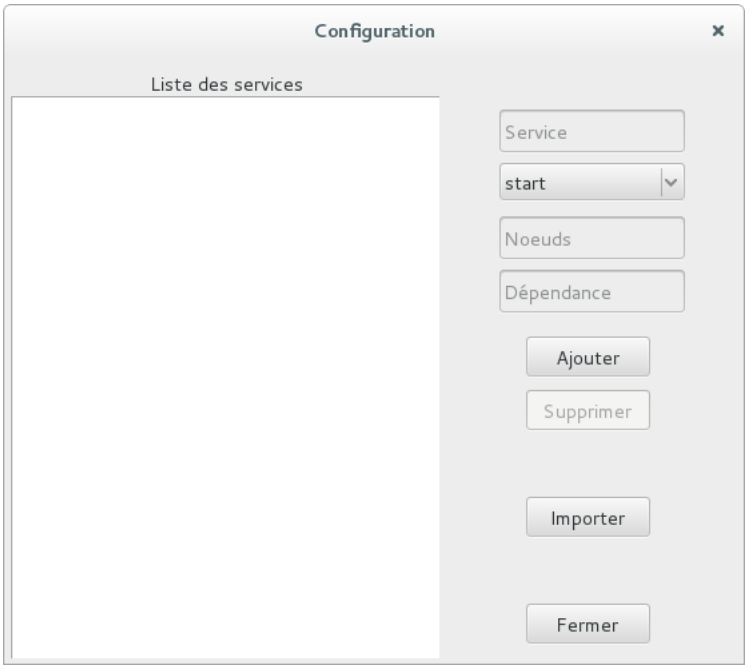
Nous avons décider de séparer par soucis de clarté le code Qt (IHM.py) du code ClusterShell qui se tiendra dans un fichier appelé `cluster.py`.

5.1.5 Arborescence des fichiers

Afin de mieux comprendre comment notre IHM à été créé, voici un récapitulatif de l'ensemble de nos fichiers python qui sont regroupés dans le dossier clustershell_IHM.



5.2 Configuration des services



Cette fenêtre va servir de fenêtre intermédiaire avant l'affichage des résultats sur la fenêtre principale. On dispose de deux choix :

- Importer un fichier au format **YAML** qui recense tous les noeuds et services que l'on veut administrer.
- Créer une liste de services où l'on peut manuellement rentrer le nom des services voulu avec les noeuds.

Lorsque nos noeuds et services sont configurés, on utilise une liste (**QListWidget**) afin d'avoir un bref récapitulatif du lancement des séquences de tâches sur les noeuds à effectuer :

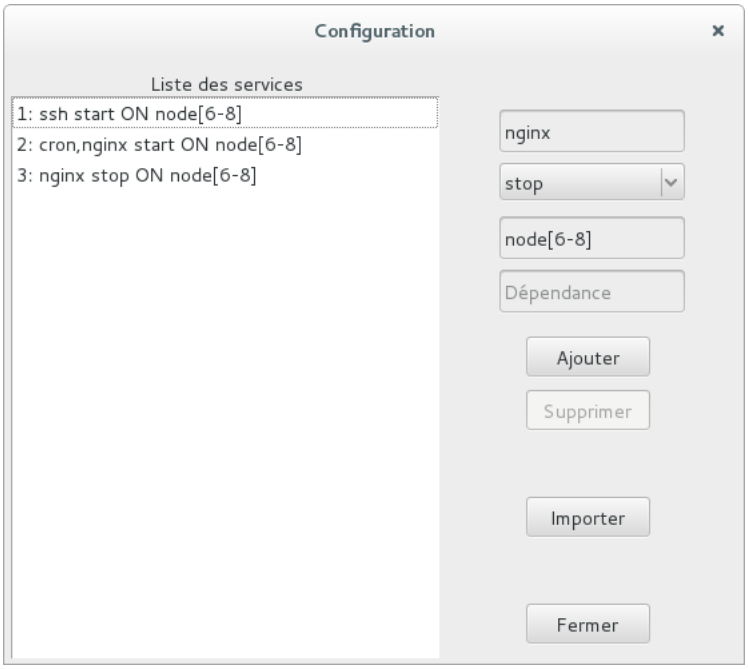
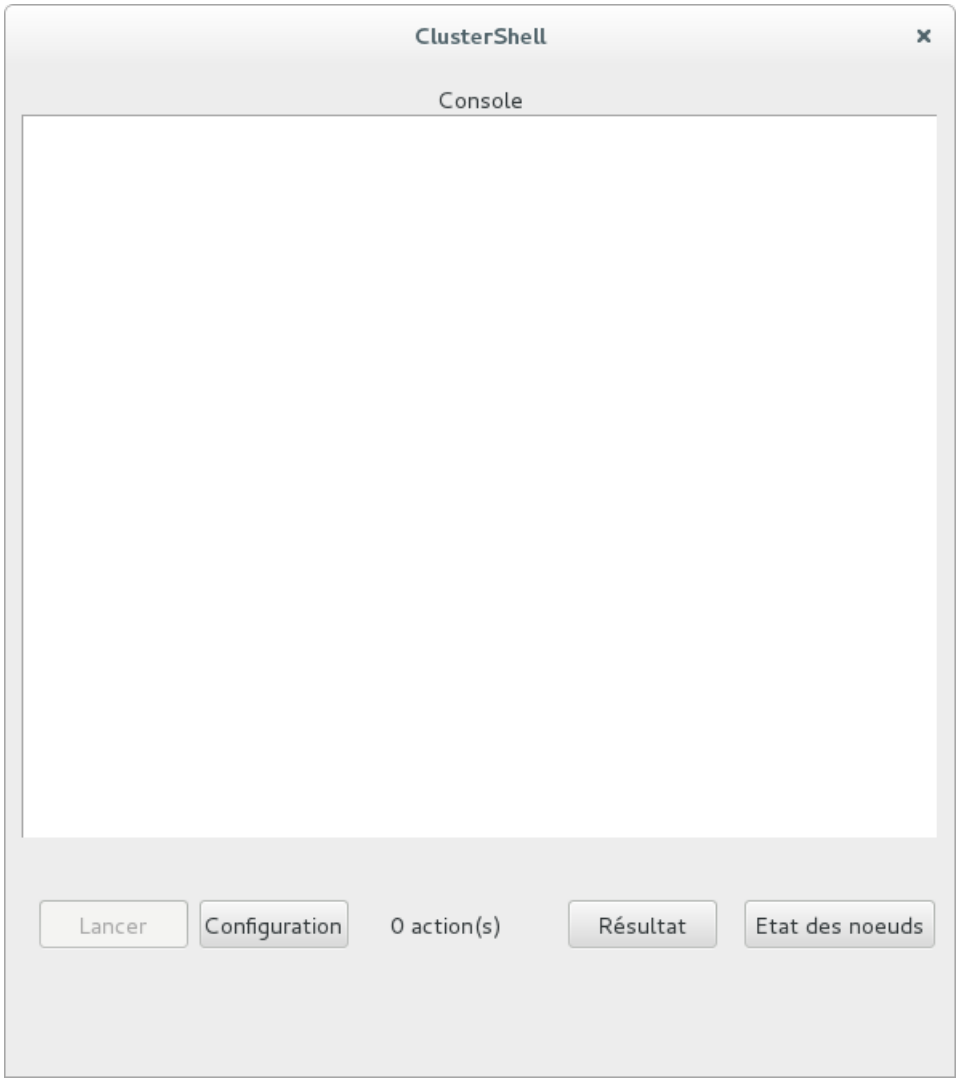


FIGURE 11 – Séquences de tâches à exécuter

Il est possible d'ajouter plusieurs services à la fois,(chaque service doit être séparé par une virgule) pour plusieurs ensemble de noeuds.
Dans **IHM.py** une fonction s'occupera d'ajouter les services dans la **QListWidget** pour les afficher ensuite. De plus, lors de l'ajout / supression d'un service, toute les informations seront envoyés dans **cluster.py**.
Il faut également gérer le cas des dépendances, car un service peut avoir besoin qu'un autre service soit installé / activé. Pour cela, on a créé une barre de saisie supplémentaire (dépendance) permettant de vérifier au préalable si le service en question est bien disponible.

5.3 Visualisation des résultats



La fenêtre principale est le point d'entrée de notre IHM, elle va permettre de visualiser la sortie de chaque noeud de notre cluster et d'analyser rapidement les résultats.
Une fois que la configuration des noeuds / services à été effectué dans la fenêtre de configuration, il est possible de lancer l'exécution des commandes via le bouton "**Lancer**" (cliquable uniquement si une liste de services à été créé via la fenêtre de configuration).

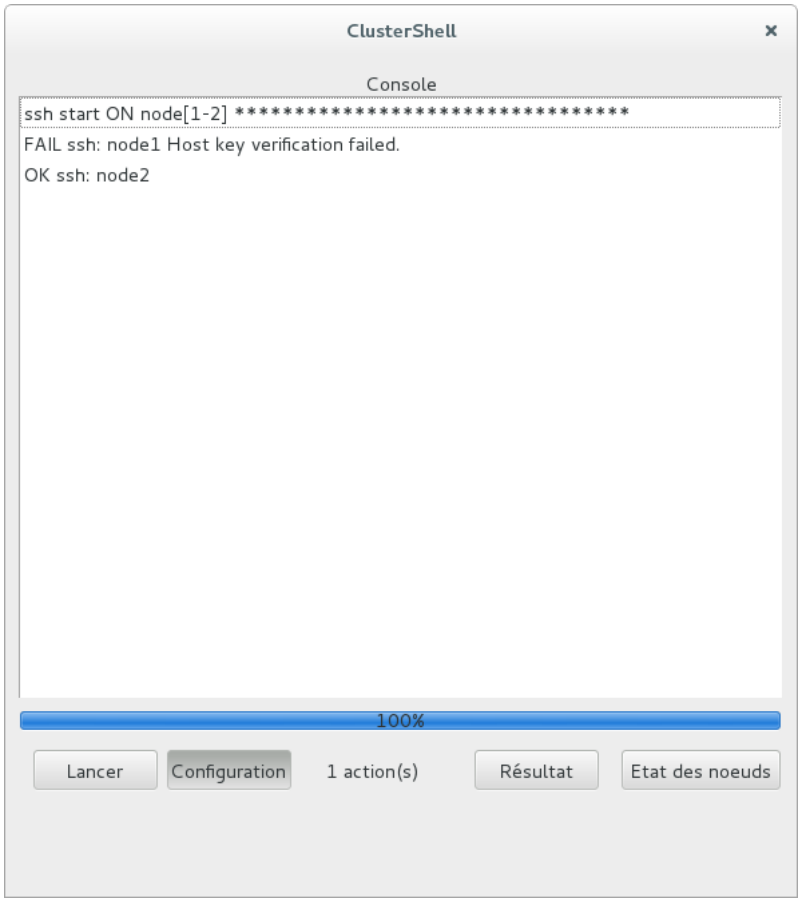


FIGURE 12 – Exemple de résultat d'exécution

5.3.1 Affichage des résultats

L'affichage se fait simplement à l'aide d'une **QListWidget** qui va récupérer en entrée le résultat de l'exécution.
Ligne 1 : On récapitule la tâche qui est effectuée
Ligne 2,3 : Le résultat est affiché de manière claire avec une réponse de type FAIL ou OK.

5.3.2 Détail de l'exécution

Nous avons également fait un compteur qui récapitule le nombre d'actions a réalisé grâce à un composant de type **QLabel**. On a aussi implémenté une barre de progression qui permet de suivre l'avancement de l'exécution des tâches sur l'ensemble des noeuds.

Voici le code qui s'exécute lors du clic sur le bouton Lancer :

```
def lancer():
    del(clustershell_IHM.list_recap[:])
    clustershell_IHM.listWidget.clear()
    clustershell_IHM.progressBar.reset()
    clustershell_IHM.progressBar.update()
    clustershell_IHM.progressBar.show()
    for i in range(0,len(clustershell_IHM.list_service)):
        nom=clustershell_IHM.list_service[i].nom
        action=clustershell_IHM.list_service[i].action
        noeuds=clustershell_IHM.list_service[i].noeuds
        dependance=clustershell_IHM.list_service[i].dependance
        if(clustershell_IHM.list_service[i].dependance==""):
            #clustershell_IHM.listWidget.insertItem(clustershell_IHM.listWidget.count()+1,"%s %s ON %s" % (nom,action,noeuds))
            clustershell_IHM.listWidget.addItem("%s %s ON %s **********" % (nom,action,noeuds))
            print("%s %s ON %s **********" % (nom,action,noeuds))
        else:
            clustershell_IHM.listWidget.addItem("%s %s ON %s (depend: %s) **********" % (nom,action,noeuds,dependance))
            print("%s %s ON %s (depend: %s) **********" % (nom,action,noeuds,dependance))


        clustershell(clustershell_IHM,clustershell_IHM.list_service,i)
        clustershell_IHM.progressBar.setValue((100/(len(clustershell_IHM.list_service))*(i+1)))
        clustershell_IHM.progressBar.update()
    clustershell_IHM.progressBar.setValue(100)
    clustershell_IHM.progressBar.update()
```

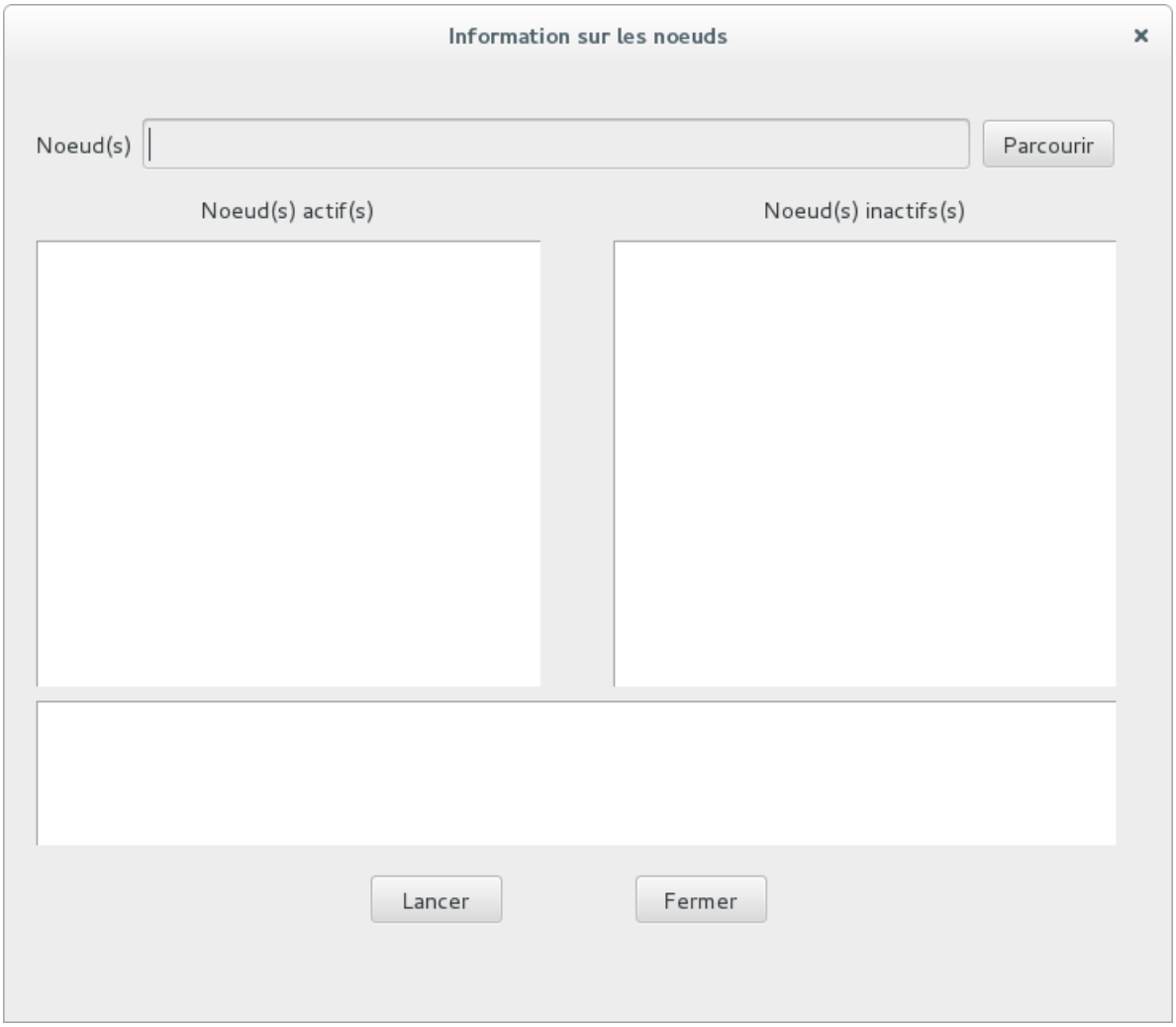
Détail du code

1. On nettoie notre console et on reset la barre de progression.
2. On récupère toutes les informations (noeuds, services, action, dépendance).
3. On affichage le récapitulatif de la tâche selon s'il y a une dépendance ou non.
4. On appelle la fonction **clustershell** et on lui passe toutes les informations. De plus comme cette fonction se trouve dans un autre fichier que la fonction lancer, on lui passe la fenêtre en paramètre. Cette fonction va effectuer la vérification des dépendances, lancer les tâches puis afficher le résultat dans la console.
5. On utilise la barre de progression de type **QProgressbar** et on la met à jour en fonction du nombre de services.

5.3.3 L'accès au différentes fonctionnalités

— **La vérification de l'état des noeuds :** Accessible depuis le bouton "Etat des noeuds"
— **La configuration des services / noeuds :** Accessible depuis le bouton "Configuration"
— **La visualisation des résultats :** Les résultats sont affichés sur la console de la fenêtre principale.
— **La journalisation des résultats :** La création de logs se fait via le bouton "Résultat"
— **L'exécution des séquences de tâches :** Une fois la configuration remplie, l'exécution se fait via le bouton "Lancer" de notre fenêtre principale.

5.4 Vérification de l'état des noeuds



Il est souvent utile de savoir si un noeud est bien présent ou non pour effectuer une quelconque tâche. On a alors créé une fenêtre permettant de lister les noeuds actifs, des noeuds qui ne l'étaient pas.
On a utilisé en tout 3 **QListWidgets** ; une pour les noeuds actifs, une pour les noeuds qui ne répondaient pas et une dernière pour l'affichage des erreurs.
Si l'utilisateur veut tester ses noeuds, il dispose de deux choix :

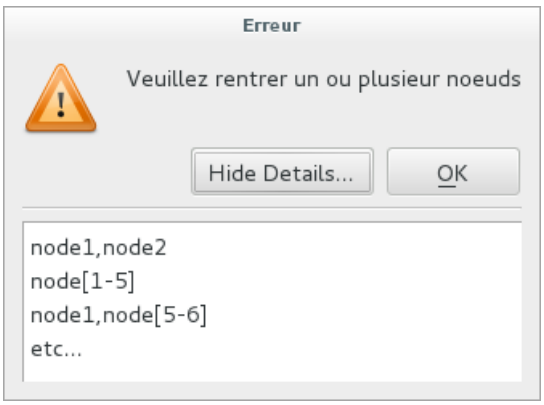
- Chercher le(s) noeud(s) qu'il veut vérifier en les renseignant dans la barre de saisie (**QLineEdit**)
- Si jamais l'utilisateur avait déjà écris son fichier au format YAML pour pouvoir administrer ses services, il peut par avance tester les noeuds présents dans le fichier. Le fichier est récupéré via le bouton (**QPushButton**) "Parcourir".

5.4.1 Test des noeuds via la barre de saisie

Lorsque l'utilisateur tape une liste de noeuds et lance la vérification des noeuds via le bouton "Lancer", on utilise un signal qui va appeler une fonction de vérification des noeuds :

```
etatnoeud_IHM.pushButton.clicked.connect(check_etat_noeud)
```

Lors de l'appel à **check_etat_noeud** on vérifie bien-sur si quelque chose a bien été entré dans la barre de saisie. L'objet **msg** de type **QMessageBox** va afficher une boite de dialogue :



Pour tester si les noeuds sont actifs on a finalement lancé la commande "**echo Hello**". Si le noeud répond simplement par "**Hello**", on considère que le noeud est présent, sinon on affiche l'erreur.

5.4.2 Test des noeuds via un fichier

Lors de l'ouverture du fichier au format YAML, il était nécessaire de tester avant tout si la syntaxe du fichier était correcte. Cependant cette vérification à déjà été faite auparavant lors de l'importation d'un fichier au niveau de la configuration des services.
Comme les fonctions utilisés n'étaient pas entièrement compatible avec le contexte il a fallu écrire de nouvelles fonctions basé sur les anciennes en les modifiant.

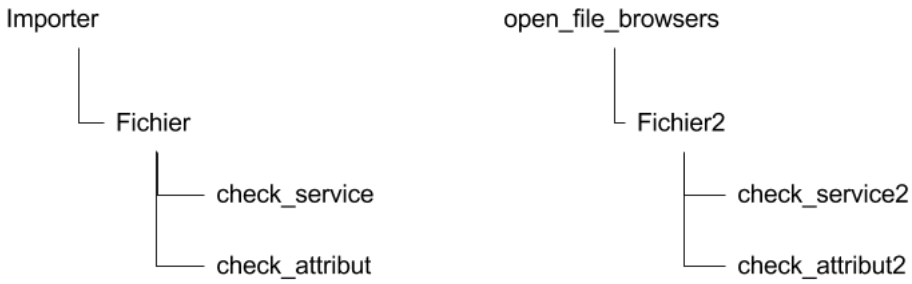


FIGURE 13 – Fonctions de configuration vs Fonctions de vérification de noeuds

5.5 Mise en place des résultats d'exécution dans des logs

Lorsque la séquence de tâches à été effectué, il est possible d'appuyer sur le bouton "Résultat" pour générer un fichier log. Ce fichier permettra de sauvegarder, garder un historique des résultats qui ont été réalisé. Le fichier se trouvera directement dans le même répertoire que le projet sera nommer sous la forme "log_Year-monthday_Hour :Minute :Second".
De plus il est également possible de voir le résultat dans le terminal.

5.6 Génération d'un exécutable via Pyinstaller

Il est possible de générer un exécutable afin d'éviter de lancer le programme par ligne de commande. L'avantage de cette exécutable est qu'il peut être exporter sur d'autre système quel qu'il soit (Windows, Linux, OSX, FreeBSD,...)
Pour cela on peut utiliser **pyinstaller**.
Il suffit de créer l'exécutable via la commande : **pyinstaller -F IHM.py**

L'option -F permet d'intégrer toutes les dépendances dans l'exécutable (taille de l'exécutable = 30 Mo).

Le nom de l'exécutable sera le même nom que le fichier python.

6 Sources

6.1 Références

Yaml : http://pyyaml.org/wiki/PyYAMLDocumentation
Nodeset : http://clustershell.readthedocs.io/en/latest/api/NodeSet.html
Task : http://clustershell.readthedocs.io/en/latest/api/Task.html
Qt GUI : http://doc.qt.io/qt-4.8/qtgui-module.html
https://pythonspot.com/en/pyqt4/
http://pyqt.sourceforge.net/Docs/PyQt4/qtgui.html

6.2 Annexes

Lien du projet (avec le code source) :

https://github.com/gdubroeucq/Clustershell_GG