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, l'API Clusterhell est une bibliothèque en Python qui permet d'exécuter en parallèle des commandes local et distantes sur des nœuds d'un cluster. Elle fournit également 3 outils en ligne de commande (script utilitaires basés dessus) qui nous permettent de bénéficier des fonctionnalités de la bibliothèque : clush,nodeset et clubak.
Ce projet nous demande de réalisé et de développer un outils en ligne de commande de gestion distribué des service de systèmes permettant d'administrer ces services sur plusieurs nœuds, et cela en utilisant l'API Python ClusterShell.
Nous allons donc 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 distant. Puis une fois cette base réalisé, nous allons mettre en place une configuration statique de la répartition des services grâce à des fichiers. Et pour finir nous développerons une IHM à partir des éléments déjà crée afin de parfaire l'outil de gestion des services distribué.

1.2 Présentation des outils

Commençons tout d'abord par définir les 3 fonctionnalités de la bibliothèques de 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 ranges de nœuds, on peut effectuer plusieurs opérations sur ces listes (union, exclusion, intersection , etc...)
- Clush : Permet l'exécution des commandes en parallèle sur des machines distantes, prends également en charge les groupes.
- Clubak : Regroupement de sorties standards qui permet de présenter de manière synthétique un résultat d'exécution un peu trop verbeux.

2 Gestion des Services

3 Configuration des Services

4 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 en python une interface graphique aisément.

4.1 Qt Creator et PyQt

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

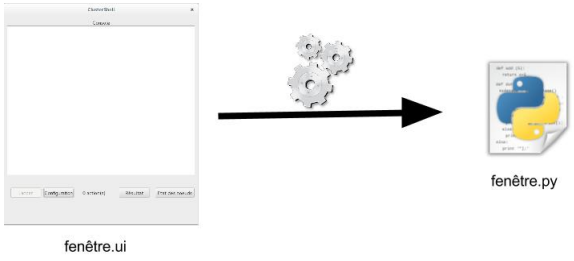


FIGURE 1 – Conversion ui - py

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

4.1.1 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 fenête 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 créés 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 :

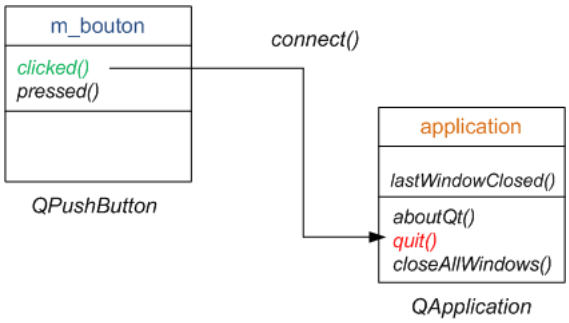
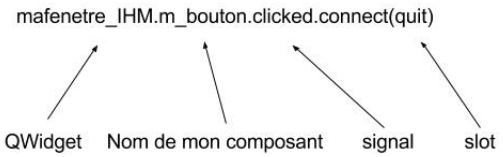
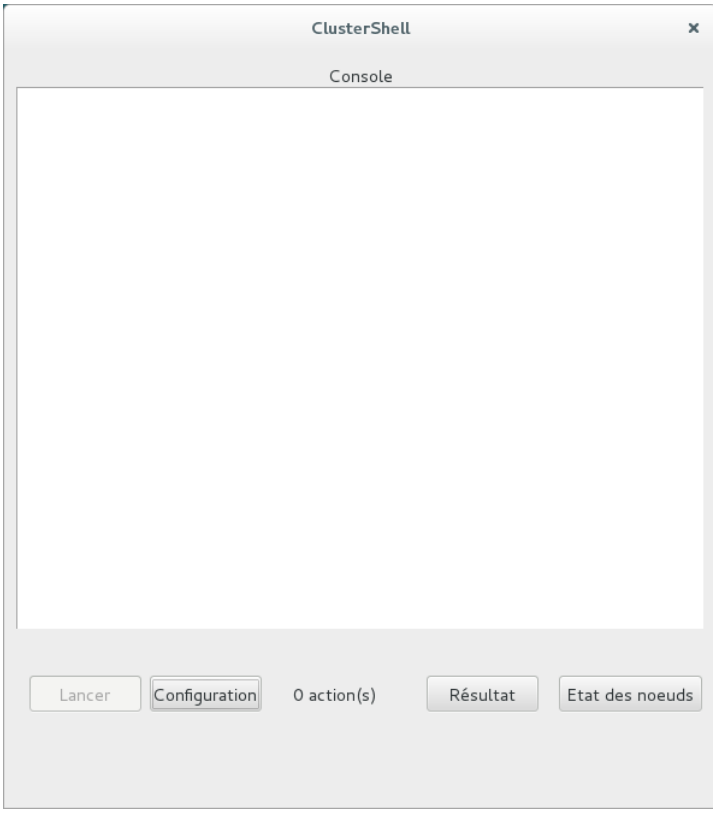


FIGURE 2 – Les signaux et slots

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



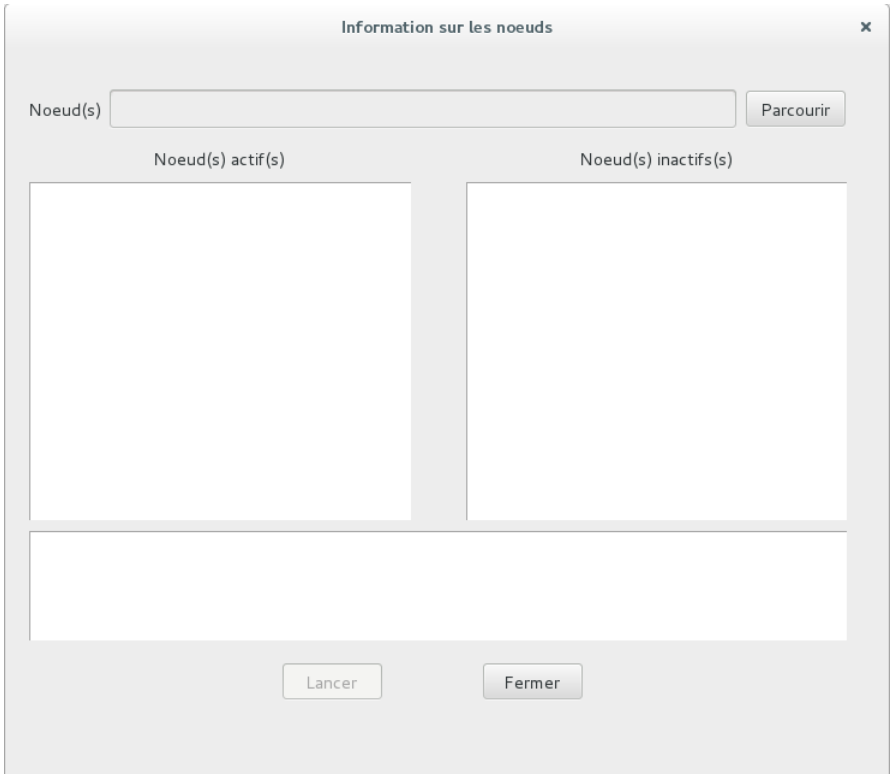
4.2 Visualisation des résultats



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.
Tout d'abord nous avons mis en place une autre fenêtre (accessible via le bouton "Etat des noeuds") permettant de vérifier l'état de noeuds. Ensuite nous pouvons configurer nos noeuds (via la fenêtre de configuration accessible via le bouton "Configuration").
De plus nous avons trouvé qu'il était essentiel de garder une trace des résultats de tous les noeuds alors vous avons gardé en mémoire ces résultats dans des fichiers qui sont crées à partir du bouton "Résultat".

4.3 Configuration des services

4.4 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 et les noeuds qui ne répondaient.
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 sont fichier au format YAML pour pouvoir administrer ses services, il peut à par avance tester les noeuds dans ce noeud. Le fichier est récupérer via le bouton (**QPushButton**) "Parcourir".

4.4.1 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 :

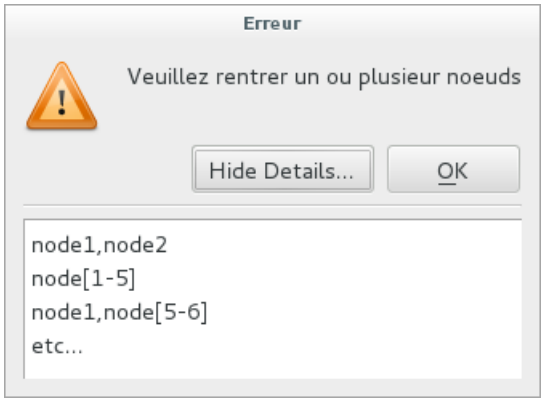
```
etatnoeud_IHM.pushButton.clicked.connect(check_etat_noeud)
```

Partie de la fonction de vérification des noeuds :
Pour tester si les noeuds sont actifs on a finalement lancé une commande sur le(s) noeud(s) en question et on à vérifié si le résultat était bien celui qu'on attendait. Ici j'ai simplement utilisé la commande "**echo Hello**".

```
noeuds = etatnoeud_IHM.lineEdit.text()
if (noeuds!=""):
    try:
        nodeset = NodeSet(str(noeuds))
        print nodeset
        for node in nodeset:
            cli = "echo Hello"
            taske = task_self()
            taske.shell(cli, nodes=node)
            taske.run()

            for output, nodelist in task_self().iter_buffers():
                if(output=="Hello"):

                    etatnoeud_IHM.listWidget.insertItem(i,"%s" % (NodeSet.fromlist(nodelist)))
                    i = i + 1

                else:
                    etatnoeud_IHM.listWidget_2.insertItem(i,"%s" % (NodeSet.fromlist(nodelist)))
                    i = i + 1
                    etatnoeud_IHM.sortie.append(output)
                    print "output: %s" % output

    except:
        msg.setText("Oups ! Probleme")
        msg.exec_()

else:
    msg.setText("Veuillez rentrer un ou plusieur noeuds")
    msg.setDetailedText("node1,node2\nnode[1-5]\nnode1,node[5-6]\netc...")
    msg.exec_()
```

Détail du code :

1. Utilisation de **NodeSet** pour vérifier si ce que l'utilisateur a tapé correspond à la syntaxe d'un noeud.
2. Utilisation de **Task** pour lancer la commande sur le(s) noeud(s)
3. La boucle for permet de récupérer le résultat d'exécution de la commande sur les noeuds.
   — Si la sortie (**output**) correspond bien à "Hello", on récupère le nonom du noeud et on l'ajoute dans la **QListWidget**.
   — Sinon on l'ajoute dans l'autre **QListWidget** qui liste les noeuds non actifs.
4. Biensur on vérifie si quelque chose à bien été entré dans la barre de saisie. L'objet **msg** de type QMessageBox va afficher une boite de dialogue :



4.4.2 Test des noeuds via un fichier

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

5 Sources

5.1 Références

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

5.2 Arborescence des fichiers