Suite du Projet S5 102: Découverte de route optimal dans un overlay de routage

Prototype en python se basant sur RIPE Atlas pour mesurer un overlay de routage et trouver des routes optimales entre chaque point.

Installation

Pour installer le prototype, il faut cloner notre répertoire sur redmine:

```
git clone https://redmine.telecom-bretagne.eu/git/mesuresoverlayripeatlas
```

Pour faire fonctionner le prototype, il faut installer les dépendances du projet:

- Pour utiliser le module `Measure` :

```
pip install ripe.atlas.cousteau
pip install ripe.atlas.sagan
apt-get install python-graph-tool
```

La méthode d'installer python-graph-tool sur ubuntu est écrite ci-dessous dans la partie de `Visu`

- Pour utiliser le module `Visu` :

Pour Ubuntu, ajouter les lignes souvantes dans `sources.list`:

```
sudo gedit /etc/apt/sources.list
```

les lignes à ajouter:

```
deb http://downloads.skewed.de/apt/DISTRIBUTION DISTRIBUTIO
N universe
deb-src http://downloads.skewed.de/apt/DISTRIBUTION DISTRIB
UTION universe
```

où DISTRIBUTION peut être un des trusty, topic, vivid, wily qui depend de la version de Ubuntu:

```
14.04   Trusty Tahr
14.10   Utopic Unicorn
15.04   Vivid Vervet
15.10   Wily Werewolf
16.04   LTS   Xenial Xerus
```

Après:

```
sudo apt-get update
sudo apt-get install python-graph-tool
```

S’il y a une erreur de la clé

```
Release: The following signatures couldn't be verified beca
use the public key is not available: NO_PUBKEY votre_n°_de_
clé
```

il faut

```
sudo apt-key adv --recv-keys --keyserver keyserver.ubuntu.c
om votre_n°_de_clé
sudo apt-get update
```

Si vous avez encore des problemes de la version g++ et gcc Installer gcc 4.9 & g++ 4.9 sur Ubuntu 12.04 OU Ubuntu 14.04

```
sudo apt-get install python-software-properties
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt-get update
sudo apt-get install gcc-4.9
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bi
n/gcc-4.9 50
sudo apt-get install g++-4.9
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bi
n/g++-4.9 50
```

Documentation

Les parties suivantes décrivent les différents composants du prototype: la partie mesure, algorithme, visualisation et contrôleur.

Partie Mesure

La partie Mesure comprend la classe `Measure`, le fichier `links.csv` et le fichier `measure.csv`

Vue d'ensemble

La classe `Measure` se base sur une fonction: `makeMeasure(self, target_prob, probe_list)` Cette fonction s'occupe de faire la mesure de toutes les arêtes d'une liste de sondes à une sonde cible. Le fichier `measure.csv` rassemble toutes les mesures effectuées. Elle se présente sous la forme suivante:

Sondes	**1**	**2**	**3**	**4**	**...**
1	0	47	20	30	
2	47	0	7	18	
3	20	7	0	23	
4	30	18	23	0	

Les valeurs dans chaque case représente le rtt.
Le fichier `links.csv` fait le lien entre l'id utilisé par le prototype (qui

va de 1 à 50) avec l'id qu'on utilise sur une implémentation de measure (l'id RIPE Atlas par exemple). Le fichier se présente sous la forme suivante:

id_probe	id_probeAtlas	...
1	18299	
2	24833	
3	2972	
4	17342	

`AtlasMeasure` est une implémentation de `Measure` en utilisant les sondes **RIPEAtlas**.

AtlasMeasure

RipeAtlas est un réseau de probe mondiale qui mesure la connectivité et la disponibilité d'Internet.Dans le prototype, les sondes seront donc les noeuds du graphe, et les arêtes seront modélisés par une mesure du ping entre deux sondes.

Choix des sondes

Les sondes ont été choisies en regardant la localisation des data centers connus de google. Les cartes dénottant ces emplacements se trouvent sur le lien suivant: http://www.haute-disponibilite.net/2008/04/14/carte-et-localisation-des-datacenter-de-google/

Sur Ripe, on utilise la carte suivante pour placer les sondes: https://atlas.ripe.net/results/maps/network-coverage/?filter=

Le choix des sondes est un peu détaillé dans le fichier `choix_sonde.txt` Quelques sondes ont dûes être changé car elles refusaient d'envoyer des mesures. La disponibilité des sondes changeant avec le temps, il faut quotidiennement vérifier si elles sont encore disponibles avant de faire d'intépréter les mesures.

Code

Le code de la partie mesure est réparti dans deux fichiers, un pour les mesures et l'autre pour les threads.

AtlasMeasure

La mesure se base sur la fonction: `makeMeasure(self, target_prob, probe_list)`

```
# Make a measure of rtt between a list of probe to one prob
e
    def makeMeasure(self, target_prob, probe_list):
        # we take the id of the probe in the measure file
        id_target = self.getProbeId(target_prob)
        id_probe_list = []
        for probe in probe_list:
            # we change the id to string for the join funct
ion
            # which fail with int
            id_probe_list.append(str(self.getProbeId(probe)
))

        address_probe_list = []
```

```
        # we get the address from atlas api
        target_address = self.getProbeAddress(id_target)

        # we create the ping object (Cousteau librairy)
        ping = Ping(
            af=4, target=target_address, description="measu
re to probe"
            +" %d from %s" %(target_prob, str(probe_list).s
trip('[]')))
        nb_probe = len(probe_list)
        values = ",".join(id_probe_list)

        # We create the source object (Cousteau librairy)
        # definition of the probes used in the msm
        source = AtlasSource(type="probes", value=values, r
equested=nb_probe)
        epoch = datetime(1970,1,1)
        # we create the measure with our apikey and the obj
ects created
        atlas_request = AtlasCreateRequest(
            # We delay the starting time to be able to retr
eive all data
            # with passiv methods
            start_time=datetime.utcfromtimestamp(
                                (datetime.utcnow()-epoch).total
_seconds()+5),
            key=self.ATLAS_API_KEY_CREATE,
            measurements=[ping],
```

```
            sources=[source],
            is_oneoff=True
        )
        (is_success, response) = atlas_request.create()
        response['measurements']
        # we then get the msm_id and return it
        id_result = response['measurements'][0]
        print id_result
        return id_result
```

Cette fonction se base sur plusieurs petites fonctions permettant de rendre le code plus lisible. (Le nom est en général assez explicite). On fait en sorte que la mesure parte avec un court délai (ici 5 secondes, pour avoir le temps de lancer le thread qui contient une socket qui écoute passivement si la réponse arrive.

Il y a aussi une fonction `makeGraph` qui mesure le graphe complet à partir d'un nombre de sondes. Cette fonction utilise la fonction précédente pour effectuer toutes les mesures.

```
def measureGraph(self,nb_sonde):
        # We initialize the measureFile
        self.initialize()
        # We create a list of nb_sonde probes with Atlas_id
        sondes = [int(i) for i in range(1,nb_sonde+1)]
        threads = []
        # We create a queue to sock the results of each thr
```

```
eads
        resultQueue = Queue.Queue()
        # We create nb_sonde-1 measure (and threads) to hav
e a complete graph
        for i in range(nb_sonde-1):
            thread = Thread_makeMeasure(
                "Thread-%s"%i,
                self.makeMeasure(sondes[i], sondes[i+1:]),
                nb_sonde-i-1,
                resultQueue
            )
            thread.start()
            threads.append(thread)
        # We create a thread to read from the Queue
        thread_write = Thread_writeMeasure(
                    "Thread-write",
                    resultQueue
                    )
        thread_write.start()
        # We wait for the measure to finish
        for thread in threads:
            thread.join()
        print "All Thread finish"
        # We notify the writing thread that the job is over
        # It will only stop if the queue is empty
        thread_write.not_finished = False
        print "Notifying the last thread to finish"
        # We wait for the write thread to finish
```

```
        thread_write.join()
        print "Done"
        return
```

La partie récupération des données et écriture de ces données est assurée par des threads. Il y a deux types de threads, `Thread_makeMeasure` qui s'occupe d'utiliser la fonction `makeMeasure` pour un certain nombre de sondes, et qui récupère le résultat, et `Thread_writeMeasure` qui s'occupe de prendre le résultat du thread précédent de la queue et de l'écrire dans le fichier rassemblant toutes les mesures en utilisant `setRTT` de la classe `AtlasMeasure`.

Thread_Atlas

Ce fichier possède donc deux classes: `Thread_makeMeasure` et `Thread_writeMeasure`.

```
class Thread_makeMeasure(threading.Thread):

    def __init__(self, nom, msm_id, nb_msm, queue):
        threading.Thread.__init__(self)
        self.name = nom
        self.msm_id = msm_id # measurement id
        self.nb_msm = nb_msm # number of measure expected
        self.queue = queue

    def run(self):
        t1 = time.time()
```

```
        not_stop = True
        while not_stop:
            print "Starting %s"%self.nom
            # We use Cousteau to create the Socket
            atlas_stream = AtlasStream()
            atlas_stream.connect()
            # Measurement results
            stream_type = "result"
            # Bind function we want to run with every resul
t received
            atlas_stream.bind_stream(stream_type, self.on_r
esult_response)
            # Subscribe to the stream
            stream_parameters = {"msm": self.msm_id}
            atlas_stream.start_stream(stream_type=stream_ty
pe,
                                      **stream_parameters
                                      )
            # we stop the thread if time > 5mn
            # we test the condition every 5sec
            while self.nb_msm>0 and (time.time()- t1) < 5*6
0:
                atlas_stream.timeout(5)
            print "Ending %s"%self.nom
            not_stop = False

    # The callback function
    def on_result_response(self, *args):
```

```
        # We decrease the nb_msm on every message received
        self.nb_msm -= 1

        # We need to use some function from AtlasMeasure
        from AtlasMeasure import AtlasMeasure
        measure = AtlasMeasure()
        # args is list of json for each probe in the measur
ement
        # with the streaming api, we get them one by one so
 len(args)=1
        result = args[0]

        # We get the atlas probe_id
        target = measure.getProbeIdFromAddress(result['dst_
addr'])
        source = result['prb_id']

        # Then we get our probe id
        target = measure.getRealId(target)
        source = measure.getRealId(source)

        # there are 3 packets so we take the median
        rtt = result['avg']

        # We write a list with target,source and rtt to the
 queue
        self.queue.put([target,source,rtt])
```

Le Thread crée une socket qui va écouter un port de Ripe Atlas contenant la mesure en cours. Dès qu'une mesure sera présente, la fonction de callback `on_result_response` est appelée qui permet d'analyser le json reçu est de stocker les arguments qui nous intéresse dans la queue.
L'ordre de source et de target n'est pas important puisqu'on suppose que la mesure est symétrique.
Le Thread se stoppe soit quand on a reçu le nombre de mesure attendu (qui est égale au nombre de sondes dans la source) ou quand on a attendu 5 mn (si la mesure n'arrive pas en 5 mn c'est qu'elle est soit perdue soit échouée).

```
class Thread_writeMeasure(threading.Thread):

    def __init__(self, nom, queue):
        threading.Thread.__init__(self)
        self.name = nom
        self.queue = queue
        self.not_finished = True

    def run(self):
        # We need to use some function from AtlasMeasure
        from AtlasMeasure import AtlasMeasure
        measure = AtlasMeasure()

        # The queue can be empty but the job unfinished
        # and the opposite too (for a brief amount of time)
```

```
        while self.not_finished or not self.queue.empty():
            if not self.queue.empty():
                args = self.queue.get()
                target = args[0]
                source = args[1]
                rtt = args[2]
                print "Processing: %s to %s"%(target,source
)
                measure.setRtt(source, target, rtt)
```

Le Thread prend les mesures dans la liste et les passe à la fonction `setRtt`. Ce thread est nécessaire pour ne pas avoir des lectures et des écritures en concurrence sur le fichier de mesure. Il attend que tous les threads se finissent

Représentation des données

La classe `Node` se représente les nœuds dans le graphe :

```
class Node:
    def __init__(self):
        self.ip_address = '0.0.0.0'
        self.connections = {}
        self.routingPath = {}
```

La propriété `connections` est un dictionnaire qui contient tous les distances (un chiffre `float` qui représente RTT) entre ce noeud et les

autres noeud du graphe :

```
{
    numéro_noeud1 : rtt_float,
    numéro_noeud2 : rtt_float,
    ...
}
```

La propriété `routingPath` est un dictionnaire qui contient les plus courts chemins entre ce nœud et les autres nœuds du graphe:

```
{
    numéro_noeud1 : path1,
    numéro_noeud2 : path2,
    ...
}
```

La classe `path` contien une value de `float` qui représente le distance plus court et une liste des numéros de nœud pour le chemin:

```
class Path:
    def __init__(self, aPath, aValue):
        utils.Utils.isList(aPath)
        utils.Utils.isFloat(aValue)
        self.path = aPath
        self.value = aValue
        # exapmle : aValue = 1.34 , aPath = [0,3,4,1]
```

Algo

La classe `Algo` est une interface qui définit la méthode `findRoutingPath(self, graph)`. Cette méthode s'occupe de trouver les plus courts chemins entre tous les nodes dans le graphe. Il y a deux implémentations dans le répertoire `algo/` : `DijkstraAlgo` et `Floyd_WarshallAlgo`:

DijkstraAlgo

L'algorithme `Dijkstra` est pour trouver les plus courts chemins depuis un point en utilisant un `priorityDictionary`.

```
def __DijkstraFromSource(graph, sourceNode):
    utils.Utils.isStr(sourceNode)
    if sourceNode not in graph:
        raise ValueError('the source node is not in the gra
ph')

    D = {}  # dictionary of final distances
    P = {}  # dictionary of predecessors
    Q = priorityDictionary()    # estimated distances of no
n-final vertices
    Q[sourceNode] = 0

    for anode in Q:
        D[anode] = Q[anode]
```

```
        graph[sourceNode].routingPath[anode]=node.Path([],0
.0)

        for neighbor in graph[anode].connections:
            sourceToNeighborLenght = D[anode] + graph[anode
].connections[neighbor]
            if neighbor in D:
                if sourceToNeighborLenght < D[neighbor]:
                    raise ValueError, "Dijkstra: found bett
er path to already-final vertex"
            elif neighbor not in Q or sourceToNeighborLengh
t < Q[neighbor]:
                Q[neighbor] = sourceToNeighborLenght
                P[neighbor] = anode
        graph[sourceNode].routingPath[anode].value = Q[anod
e]
        end = anode;
        while 1:
            graph[sourceNode].routingPath[anode].path.appen
d(end)
            if end == sourceNode: break
            end = P[end]
        graph[sourceNode].routingPath[anode].path.reverse()
```

Le programme applique l’algorithme dans un boucle pour chaque nœud :

```
def findRoutingPath(self,graph):
    utils.Utils.isDict(graph)
    for key in graph:
        DijkstraAlgo.__DijkstraFromSource(graph,key)
```

Floyd_WarshallAlgo

L'algorithme `Floyd Warshall` est pour trouver les plus courts chemins depuis touts les points du graphe.

```
def findRoutingPath(self,graph):
    utils.Utils.isDict(graph)

    dist = {}
    pred = {}
    for u in graph:
        dist[u] = {}
        pred[u] = {}
        for v in graph:
            dist[u][v] = 1000
            pred[u][v] = -1
        dist[u][u] = 0
        for neighbor in graph[u].connections:
            dist[u][neighbor] = graph[u].connections[neighb
or]
            pred[u][neighbor] = u

    for t in graph:
```

```
        for u in graph:
            for v in graph:
                newdist = dist[u][t] + dist[t][v]
                if newdist < dist[u][v]:
                    dist[u][v] = newdist
                    pred[u][v] = pred[t][v]
    for s in graph:
        for d in graph:
            graph[s].routingPath[d]=node.Path([],0.0)
            end = d;
            graph[s].routingPath[d].value = dist[s][d]
            while 1:
                graph[s].routingPath[d].path.append(end)
                if end == s: break
                end = pred[s][end]
            graph[s].routingPath[d].path.reverse()
```

Partie Visualization

Le module `visu` se charge de généré un graphe depuis le résultat de `Mesure` et `Algo`. Visu présente le graphe interactive dans une fenêtre d'utilisateur. La fenêtre contiens également une liste des nœuds. Ce liste permet aux utilisateurs de choisir un paire des nœuds pour générer le graphe pour le routage entre ces deux nœuds choisis.

La fenêtre des développée avec `PyGtk`. http://pygtk.org/

Le graphe est développé avec `GraphTool`. https://graph-tool.skewed.de/

Partie Controller

Le module `Controller` intègre les trois autres modules (`Measure`, `Algo` et `Visu`). Il va lancer tout d'abord une fenêtre de login et l'utilisateur peut choisir le nombre de sondes qu'il souhaite pour lancer la simulation. Un fois il lance la simulation, il va :

- Utiliser le module `Measure` pour lancer mesure les trajets entre les sondes choisis et stocker les résultats dans les fichiers.
- Utiliser le module `Algo` pour calculer les plus courts chemins.
- Prendre les résultats de `Algo` et utiliser le module `Visu` pour générer le grqphe dqns une fenêtre d'utilisateur.