AmarOk

Geste de savoir

p2p internals #3

10 octobre 2019

Table des matières

Dans les précédentes parties, nous avons un peu parlé de comment réaliser une connexion pair à pair, mais aussi des problèmes de connectivité possibles. Ces problèmes sont principalement dus au NAT qui nécessite d'utiliser des méthodes pour le contourner (comme les serveurs TURN) ou des protocoles pour tenter diverses méthodes de contournement (comme ICE) :

Il est maintenant temps de parler un peu plus concrètement d'applications et de structures distribuées.

Pour de nombreuses applications distribuées, l'ensemble des noeuds composants le système ne peut pas être stable. En effet, des noeuds sont continuellement entrain de rejoindre ou de quitter le réseau pour diverses raisons. Il est donc nécessaire d'utiliser une structure permettant à la fois de connecter cet ensemble de noeuds, de manière à maximiser la couverture de cet ensemble (on ne veut pas qu'un noeud se trouve séparé du réseau), tout en ayant le meilleur temps de réponse possible.

Dans ce billet, je vais introduire une structure distribuée qui répond à ce besoin et qui est beaucoup utilisée dans les applications fonctionnant en pair à pair : la DHT (*Distributed Hash Table*).

1. Une DHT c'est quoi ?

Une *DHT* comme son nom l'indique est une table de hachage, mais stockée sur un ensemble de noeuds (contenant chacun une partie de la DHT) pouvant être répartis sur différents appareils. BitTorrent est un exemple de logiciel ayant participé à la démocratisation de cette technologie.

Dans une DHT, la valeur de la fonction de hachage détermine quel pair possède la donnée. Les deux principaux algorithmes sont Kademlia et Chord. Dans la suite de ce billet, je parlerais surtout de Kademlia.

La DHT possède aussi deux autres caractéristiques :

1. Elle n'a pas d'autorité contrairement à la blockchain (qui aura surement son article). Donc aucun noeud n'est considéré de confiance.
2. Le stockage ne garantit pas de base la persistance des données. Généralement un temps de vie est donné aux valeurs.

2. Et comment ça marche ?

Imaginons un ensemble de noeuds, répartis un peu partout dans le monde. Sans arrangement logique, il est difficile de demander à chaque noeud de se connecter à quelques autres, tout en étant certain que chaque pair peut échanger avec n'importe quel autre pair du réseau. Il est donc nécessaire d'ordonner un peu le réseau différement. Imaginons que chaque noeud possède un identifiant random (représenté par un hash). Il est alors possible de les ordonner (*0000,0001,...,1111*). En général, on choisit alors de les disposer en cercle comme dans l'image suivante :

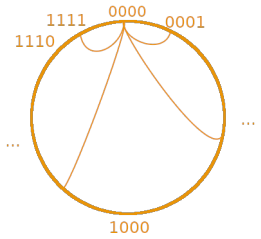

FIGURE 2.1. – DHT

Imaginons maintenant N noeuds, disposés en cercle. Les deux méthodes les plus simples pour connecter ces noeuds sont soit de les connecter chacun à son voisin pour faire passer les messages ou alors que tout le monde se connecte à tous les autres pairs. Dans la première, on minimise donc le nombre de connexions (N pour tout le réseau), mais le temps de transmission est alors de N bonds. Dans le second cas, on a énormément de connexions ($N * N$) mais chaque message est transmis le plus rapidement possible. Cependant, des méthodes utilisant moins de connexions tout en étant rapide peuvent être mises en place, tout en étant simple à calculer.

Ainsi, dans une DHT de type Kademlia, chaque noeud possède un identificateur tiré au hasard de 160 bits. Les clés des hashes font la même taille. Pour définir la distance entre deux noeuds, la méthode **XOR** est utilisée. Ainsi, les noeuds ...0101010 et ...0100010 auront pour distance ...0001000.

2.0.1. Stockage

Imaginons que Alice veut stocker une valeur sur la DHT : $hash(foo) = bar$. $hash(foo)$ est alors une clé de 160 bits. Il est alors possible de trouver dans un temps logarithmique $O(log(N) + \omega)$ le noeud le plus proche qui servira pour stocker la donnée *bar*. Cependant, comme celui-ci est potentiellement capable de se déconnecter durant les prochaines secondes, il est nécessaire de stocker cette valeur de manière redondante. Ainsi, la valeur sera stockée sur les k noeuds les plus proches pour garantir une certaine durée de vie de la valeur. **IPFS** définit ce k à 20 (à confirmer), **BitTorrent** à 8 (ce qui garantit ~10 minutes de temps de vie).

2.0.2. Routage

Chaque noeud de la DHT divise l'espace en espaces appellés *Bucket* jusqu'à obtenir un bucket où k noeuds (dont le courrant) sont présents. Ainsi, la table de routage du noeud ($ID = 10010...$) sera alors :

Bucket id	**noeuds**
X	0aaaa 0bbbb
1	11aaa 11bbb
10	101aa 101bb
100	1000a 1000b
1001	10011 10011
...	...

Le dernier bucket contiendra F.

Un protocole est alors définit pour interagir. Tout d'abord, l'insertion de noeud dans la table de routage est définie par :

Soit un noeud X, appartenant à un bucket B. Si le bucket B a encore de la place, alors X est ajouté. Sinon, si le bucket contient des noeuds expirés, alors on ajoute B au bucket ou si c'est le bucket de F, alors on le scinde. Dans les autres cas, on drop l'insertion.

Les ordres suivants sont aussi définis :

1. **ping**
2. **find(n)** pour obtenir la liste des k noeuds les plus proches de n.
3. **put(k,v)**
4. **get(k)**

2.0.2.1. Rejoindre un réseau

Maintenant que le protocole est déterminé, il est nécessaire de parler d'une étape importante, celle de rejoindre un réseau. En fait, chaque noeud du réseau est un potentiel point d'entrée. Cependant, en général, des noeuds connus et persistant ont un rôle de *bootstrap*, c'est-à-dire qu'ils vont servir de point d'entrée aux nouveaux noeuds, lorsqu'il souhaite rejoindre pour la première fois un réseau (eg *router.bittorrent.com*, *bootstrap.jami.net*, etc).

2.0.2.2. Maintien de la table de routage

Comme la table de routage ne peut pas être fixe (les noeuds ayant la possibilité de joindre/quitter le réseau), il est nécessaire de maintenir une sorte de synchronisation avec le reste de la DHT. Périodiquement, un noeud tentera les opérations suivantes **find(F)** pour maintenir la correspondance avec ses voisins et pour chaque bucket **find(i)** où i est un identifiant random de B pour remplir le bucket (si B est vide, l'identifiant est pris dans un bucket voisin) et d'éliminer les noeuds expirés.

2.0.3. Recherche d'une valeur

Au final, si Bob veut trouver où les valeurs sont stockées pour $hash(foo)$, il n'aura qu'à envoyer des $find(hash(foo))$ sur la DHT, afin de trouver les k noeuds les plus proches de $hash(foo)$. Lorsque la liste de noeuds obtenus est stationnaire, la recherche est terminée.

Ainsi, il est possible pour tous les pairs d'envoyer ou de récupérer une valeur sur à la liste récupérées via les méthodes **get(k)** et **put(k,v)**.

3. Exemples de projets utilisant une DHT

Comme nous l'avons vu, la DHT est une structure permettant de stocker relativement simplement des données sur un réseau distribué. Il est à noter que ce type de réseau peut facilement avoir des dizaines de millions de noeuds avec un énorme pourcentage de noeuds inaccessibles (NAT, te voilà) avec des noeuds apparaissant seulement pour quelques secondes.

Même si une DHT est forcément plus lente et bien plus complexe à analyser, de nombreux projets autres que Bittorrent utilisent une telle technologie (je reviendrais sur la plupart dans la suite de cette série) :

— Jami ou Tox pour de la communication pair à pair.
— Yacy
— I2P dans un premier temps
— IPFS ou GNUnet pour naviguer
— etc.

4. Exemple de code

Wabi est une ville intelligente où des capteurs de pollution sont présents un peu partout. Chaque capteur agit comme un noeud DHT où les capteurs rafraichissent les données récupérées toutes les minutes sur le réseau à l'addresse $hash(pollution_level)$ via la valeur suivante :

```
SIGN(sensor_key,{
    "lvl_percent":78,
    "lat":39.010941,
    "long":125.723739
})
```

Pour ce faire, chaque habitant a en plus la possibilité de se connecter au réseau de la ville pour suivre la pollution ou d'ajouter son capteur. Ainsi, voici un petit script python à destination des capteurs ou utilisateurs de la ville. La bibliothèque suivante a été utilisée :

— Opendht possédant un wrapper Python ainsi que des primitives de plus comme l'opération **listen(k)** permettant de suivre un stream pour une clé donnée. De plus, la bibliothèque supporte des opérations de crypto pour envoyer des valeurs signées.

```python
#!/usr/bin/env python3
# Example
# In one terminal
# python3 sensor.py
# On another
# python3 sensor.py
    "{\"lvl_percent\":39,\"lat\":39.093214,\"long\":125.688883}"
# python3 sensor.py
    "{\"lvl_percent\":39,\"lat\":39.017443,\"long\":124.7365321}"
import opendht
import asyncio
import base64, json
import sys

def listen_cb(node, v):
    try:
        json_object = json.loads(v.data.decode())
        latitude = json_object["lat"]
        longitude = json_object["long"]
        lvl = json_object["lvl_percent"]
        if latitude and longitude and lvl:
            key = str(latitude) + "x" + str(longitude)
            node.sensors_map[key] = lvl

                print(f"Sensor at lat: {latitude}, long: {longitude} - pollutio
    except:
        print("Illegally formatted value received")
    return True

class DhtNode:
    def __init__(self, is_bootstrap):
        self.sensors_map = {}
        self.n = opendht.DhtRunner()
        self.n.run(ipv4="", ipv6="", port=4242 if is_bootstrap else
            2424)
        if not is_bootstrap:
            self.n.bootstrap("localhost", "4242")
        self.key = opendht.InfoHash.get("pollution_level")

    def follow_stream(self):
        self.n.listen(self.key, lambda v:
            loop.call_soon_threadsafe(listen_cb, self, v))
        loop = asyncio.get_event_loop()
        loop.run_forever()

    def put(self, data):
        v = opendht.Value(arg.encode('utf-8'))
        self.n.put(self.key, v)

```

```python
if __name__ == "__main__":
    node = DhtNode(len(sys.argv) == 1)
    if len(sys.argv) > 1:
        for arg in sys.argv[1:]:
            node.put(arg)
    else:
        node.follow_stream()
```

Si vous souhaitez améliorer le code pour vous amuser (ajouter la signature des valeurs, la vérification de la validité de la signature, montrer une carte, etc), n'hésitez pas.

---

5. Aller plus loin

— La papier décrivant Kademlia
— Les vidéos d'Anne marie Kermarrec :

Élément externe (Video) —
Consultez cet élément à l'adresse https://www.youtube.com/embed/WeXRfWJ2snA?feature=oembed.