LES GUIDES DE
GNU LINUX MAGAZINE / FRANCE
HORS-SÉRIE N°90
France MÉTRO. : 12.90 € — CH : 18,00 CHF — BEL/PORT.CONT : 13,90 € — DOM TOM : 13,90 € — CAN : 18,00 $ CAD

LE GUIDE POUR CRÉER DES APPLICATIONS CLIENT/SERVEUR EN PYTHON

PROGRAMMATION RÉSEAU

COMPATIBLE RASPBERRY PI / WINDOWS / MAC OS / LINUX

DÉMARREZ...
la programmation réseau en Python avec les modules essentiels

CRÉEZ...
- un système de migration des rapports de bug de GitHub à votre GitLab
- un driver FUSE pour Google Drive
- un bot IRC
- un robot Slack
- un client XMPP

PROGRESSEZ...
en créant un script communiquant par SMS et en analysant un serveur de fichiers

Édité par Les Éditions Diamond
L 15066 - 90 H - F: 12,90 € - RD
www.ed-diamond.com

Retrouvez toutes nos publications
LES ÉDITIONS DIAMOND
sur **www.ed-diamond.com**

GNU/Linux Magazine Hors-Série
est édité par **Les Éditions Diamond**

10, Place de la Cathédrale - 68000 Colmar - France

Tél. : 03 67 10 00 20 / **Fax :** 03 67 10 00 21

E-mail : cial@ed-diamond.com
lecteurs@gnulinuxmag.com

Service commercial : abo@gnulinuxmag.com

Sites : http://www.gnulinuxmag.com
http://www.ed-diamond.com

Directeur de publication : Arnaud Metzler

Chef des rédactions : Denis Bodor

Rédacteur en chef : Tristan Colombo

Responsable Service Infographie : Kathrin Scali

Responsable publicité : Tél. : 03 67 10 00 27

Service abonnement : Tél. : 03 67 10 00 20

Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne

Distribution France :
(uniquement pour les dépositaires de presse)

MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04

Service des ventes :
Distri-médias : Tél. : 05 34 52 34 01

IMPRIMÉ en Allemagne - PRINTED in Germany

Dépôt légal : A parution

N° ISSN : 0183-0864

Commission Paritaire : K78 976

Périodicité : Bimestrielle

Prix de vente : 12,90 Euros

PRÉFACE

« Et puis c'est arrivé... une porte s'est ouverte sur un monde nouveau... j'envoie un signal, il traverse les lignes téléphoniques (...). »

« The Hacker Manifesto », The Mentor, 1986

Depuis que ces lignes ont été écrites, 31 ans se sont écoulés. Et pourtant elles n'ont jamais autant été d'actualité. Le net ne cesse de grandir, de se développer, d'accoucher de nouvelles technologies et de nouvelles façons de faire communiquer différentes entités entre elles. Le **SOAP** a laissé la place au **REST**, les réseaux sociaux se diversifient, chaque jour amène son lot de nouveaux services qui offrent chacun leur API.

Dans cet écosystème foisonnant, nous, développeuses et développeurs Python, sommes plus que chanceux. En effet, la mise en place de communications par échange d'informations passant par des connexions réseaux est au cœur du Python. Que ce soit grâce aux modules fournis par la bibliothèque standard ou les multiples bibliothèques disponibles, tout est fait pour que nous ayons tous les outils pour pouvoir travailler efficacement.

« Nous explorons... (...). Nous recherchons la connaissance... (...). ». Le magazine que vous tenez entre les mains a pour but de vous faire découvrir ou approfondir les différents moyens de mettre en places des communications réseaux avec notre langage préféré :

- Que vous vouliez en apprendre plus sur les concepts de base fondamentaux de la communication réseau bas niveau, pour mettre en place des applications ayant à la fois une interface graphique et des communications réseaux, consommer des API web REST diverses ou pour travailler directement sur les paquets réseaux, la première partie de ce hors-série est faite pour vous.
- Si vous avez plutôt envie de suivre la mouvance des chats-bots, que ce soit pour se connecter à **IRC**, **Jabber**, **Slack** ou des réseaux sociaux, là aussi, vous allez pouvoir trouver votre bonheur dans les pages qui vous attendent dès que vous aurez fini de lire cette petite préface.
- Enfin, si vous voulez mettre en place des serveurs de fichiers ou interagir avec le réel et envoyer des SMS avec l'un de vos **Raspberry Pi** préférés, alors vous allez adorer la derrière partie de ce hors-série.

Je vais profiter de la poignée de caractères qu'il me reste pour vous souhaiter une bonne lecture et de longues heures d'expérimentation avec Python et les communications réseaux. Faites tout de même attention, ne développez pas le bot de trop, la Singularité n'est pas si loin que ça...

Jean-Michel Armand

Sommaire

GNU/Linux Magazine Hors-Série N°90

PROGRAMMATION RÉSEAU

DÉMARRREZ...

LA PROGRAMMATION RÉSEAU EN PYTHON AVEC LES MODULES ESSENTIELS

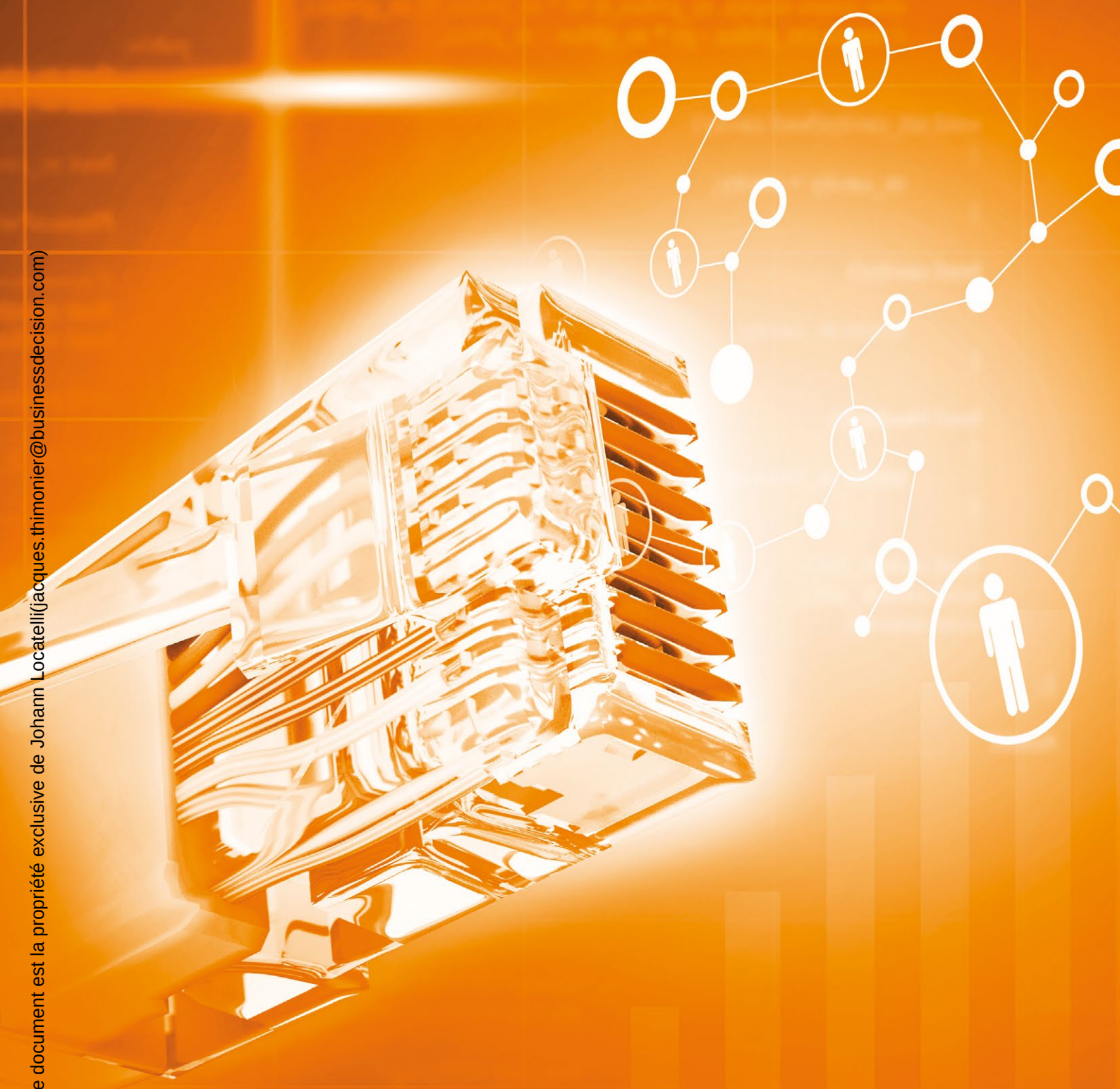
CRÉEZ...

VOS ROBOTS ET CLIENTS EN PYTHON POUR INTERAGIR AVEC DES SERVICES WEB TELS QUE GITHUB, GOOGLE DRIVE, ETC.

PROGRESSEZ...

EN CRÉANT UN SCRIPT COMMUNIQUANT PAR SMS ET EN ANALYSANT UN SERVEUR DE FICHIERS

DÉMARREZ



1 DÉMARREZ... (LES ESSENTIELS)

À découvrir dans cette partie...

Utilisez TCP et UDP en Python

Il est important de comprendre comment fonctionne le réseau avant de l'utiliser ! Cet article a pour but de vous présenter les protocoles TCP et UDP et comment les utiliser en Python avec des accès bas niveau. p. 08

Développez une application graphique utilisant le réseau

Utiliser le réseau, c'est bien, mais proposer une interface graphique, ça peut aider certains utilisateurs. Cet article est un exemple de développement d'une application réseau utilisant une interface graphique. p. 22

Utilisez des API REST en Python

Consommer des API est devenu commun. Fort heureusement pour nous, le Python est un des langages classiquement utilisés pour faire cela et donc une grande majorité des services web connus propose des bibliothèques Python pour interagir avec eux. p. 34

Scapy, le couteau suisse Python pour le réseau

Scapy est un outil écrit en Python qui permet de créer des paquets, de les envoyer sur le réseau, d'analyser les paquets reçus, d'écouter le trafic, etc. Il constitue une boîte à outils utilisable pour écrire des programmes divers et variés. p. 46

UTILISEZ TCP ET UDP EN PYTHON

Sébastien CHAZALLET

Cet article va vous présenter les concepts sur lesquels s'appuient les réseaux informatiques et la manière dont on peut les maîtriser très facilement, à l'aide de Python.

Le mot réseau définit la nature de relations entre plusieurs éléments, quelle que soit la nature de ces derniers. Si on se limite à ce qui nous intéresse vraiment, à savoir les réseaux informatiques, les éléments sont des serveurs, ordinateurs, **Raspberry Pi**, téléphones portables, ou n'importe quoi qui peut s'interfacer avec un réseau informatique, quel qu'il soit, comme des caméras de vidéosurveillance ou encore les nouveaux objets connectés dont on nous explique qu'ils vont nous changer la vie (non, je n'ai pas dit « améliorer », j'ai dit « changer »). Un vaste monde.

1. PRÉSENTATION GÉNÉRALE

1.1 Infrastructures réseau

Les réseaux ne sont pas des entités magiques, mais simplement des **infrastructures** sur lesquelles on s'appuie pour faire transiter des données. Il s'agit du câble réseau qui vous relie à votre box, de votre ligne téléphonique qui vous relie à un DSLAM (multiplexeur permettant l'accès haut débit à internet via la ligne téléphonique), de fibres qui relient sur de plus grandes distances. Ces appareils sont des outils très complexes, mais qui font un travail très simple : **transmettre un signal** d'un point à un autre. Ce signal peut-être électrique (câbles) ou lumineux (fibre, radio ou encore Wifi ou GSM, lesquels n'ont pas besoin de support pour se propager vu que la théorie de l'éther est morte depuis plus d'une centaine d'années (ou en tout cas, devrait l'être)).

Chaque machine est reliée au réseau via une **interface réseau**. Il s'agit d'un composant capable de communiquer, c'est-à-dire capable de recevoir et d'envoyer des signaux. Une machine peut avoir plusieurs interfaces réseau, comme votre ordinateur portable qui a une prise RJ45 et le Wifi.

Au niveau d'une interface réseau, un signal est simplement une succession de `0` ou de `1`, mais la machine elle-même n'a pas besoin de savoir ce que des valeurs représentent, simplement d'être capable de les envoyer, de les recevoir, c'est-à-dire de convertir un signal de quelque nature qu'il soit en représentation informatique ou inversement.

Cette capacité de transmettre ces signaux est la première couche du modèle OSI (*Open System Interconnection*), le standard de communication réseau. À ce niveau, un signal est juste une suite de bits ou d'octets.

1.2 Adressage physique

Il s'agit là de la seconde couche du modèle OSI. Elle intervient dans le cadre très précis de la communication entre machines d'un même réseau local (LAN) ou étendu (WAN). C'est le cas de votre carte réseau lorsqu'elle envoie une donnée via votre câble RJ45 ou encore votre Wifi. C'est le cas de votre concentrateur (*hub*) ou commutateur (*switch*), c'est encore le cas d'un DSLAM.

L'idée est d'identifier de manière précise chaque interface réseau (et non chaque ordinateur, ne pas confondre). Ceci est fait par l'attribution d'une adresse physique (ou adresse MAC (*Media Access Control*)) à chaque interface réseau. Cette adresse est fixée par le fabricant de l'interface réseau, mais peut être surchargée par le système d'exploitation si nécessaire. Le point important est qu'il ne doit pas y en avoir deux identiques sur le même réseau.

La couche de liaison se charge de faire transiter ces données sous la forme de trames (sans entrer dans le détail, il s'agit d'une série de bits qui suit des règles spécifiques pour pouvoir être traitée).

Maintenant se pose la question de l'adressage réseau. En effet, si Internet était un et un seul immense réseau, ce serait facile. Il suffirait d'identifier la machine à accéder et de lui faire parvenir la donnée en utilisant la couche de liaison. Mais ce n'est pas ce qu'Internet est. Internet n'est pas un réseau, mais un réseau de réseaux (*réseau-ception*). Il faut donc faire son chemin d'un réseau vers l'autre. C'est là qu'intervient la troisième couche.

1.3 Interconnexion de réseaux

On va exclure d'entrée les réseaux téléphoniques et GSM qui ont leur propre mode de fonctionnement pour nous intéresser au réseau d'entreprise et Internet. Le protocole le plus répandu est le protocole **Internet Protocol** (IP).

Pour faire simple, la couche réseau de l'OSI doit déterminer une réponse à trois problématiques : le routage, le relayage et le contrôle des flux.

Le **routage** consiste à déterminer un chemin permettant de relier deux machines. Pour cela, on s'appuie sur des machines qui ont une interface dans plusieurs réseaux physiques distincts. Ces machines sont des passerelles. Si l'adresse que vous voulez n'est pas dans le réseau local, on va transmettre le paquet à la passerelle. Et elle va alors déterminer le réseau à qui transmettre la donnée. Si elle peut identifier le réseau, alors elle le choisit. Sinon, elle aussi a une passerelle à interroger. La route est ainsi déterminée itérativement.

Le **relayage** consiste, pour chaque routeur, à transmettre un paquet de données d'un réseau à un autre. L'idée est de le rapprocher de sa destination finale, mais ceci ne veut absolument pas dire que toutes les données transitent par le même chemin. Une même donnée peut être fractionnée en plusieurs paquets qui chacun prendront leur propre route.

Le **contrôle des flux** est une fonctionnalité permettant d'éviter avant tout une congestion. Si trop de paquets se dirigent vers une même zone, ceux qui peuvent l'éviter se verront attribuer une route différente.

Pour illustrer tout cela, on peut prendre votre box internet comme exemple ; elle dispose de trois interfaces réseaux : une RJ45, une Wifi et votre ligne téléphonique. Vous êtes sur votre téléphone connecté en Wifi et vous allez vous chercher un fichier dans un répertoire partagé sur votre PC connecté en RJ45 ? vous allez adresser votre passerelle qui va comprendre que le réseau que vous souhaitez atteindre est le réseau filaire local : elle va transmettre alors la donnée. Vous voulez aller accéder à un site internet ? Votre téléphone va envoyer la donnée à nouveau à la passerelle : elle transmettra sur le réseau Internet.

1.4 Finalité

La finalité de toute cette présentation est de restituer les connaissances générales qu'il faut avoir sur le réseau pour aborder la partie programmation. En effet, ces trois premières couches sont celles sur lesquelles repose la quatrième, comme vous pouvez raisonnablement vous y attendre et c'est cette dernière qui nous intéresse, puisqu'elle consiste à gérer une communication de bout en bout.

Nous allons voir deux choses : TCP et UDP et nous allons voir dans le détail comment, en **Python**, créer de telles connexions, c'est-à-dire comment programmer un serveur et un client.

NOTE

Notons, pour terminer, qu'il existe IPv4 et IPv6, la seconde norme étant une amélioration de la première. Nous n'en parlerons pas trop ici, le sujet ayant déjà été traité, en particulier dans le hors-série sur les lignes de commandes, une partie étant entièrement dédiée aux commandes réseau.

2. TCP

2.1 Introduction

Maintenant que nous avons vu les trois premières couches du modèle OSI, nous pouvons aborder en toute confiance la quatrième et commencer à parler de TCP (*Transmission Control Protocol*).

Commençons par dire qu'il s'agit d'un protocole très ancien, extrêmement fiable et très utilisé. Un exemple d'utilisation : lorsque vous naviguez sur Internet (HTTP 1.0 et 1.1 sont construits au-dessus de TCP).

Une session TCP fonctionne en trois phases : l'établissement d'une connexion, le transfert de données et la fin de la connexion.

Un serveur TCP est essentiellement un programme qui ne fait strictement rien : il se place à l'écoute d'une éventuelle demande de connexion. Auquel cas, il répond en créant une tâche fille (*thread*) qui se charge de cette demande. Puis, il retourne écouter une éventuelle nouvelle demande de connexion.

L'établissement de la connexion se fait par une tâche fille et commence par un « *Three-way handshake* » ou poignée de main en trois temps :

- Client : Hey ! Tu m'entends ? (envoi d'un paquet **SYN**) ;
- Serveur : Je t'entends. Et toi, tu m'entends (envoi d'un paquet **SYN-ACK**) ;
- Client : Ouais, c'est bon ! (envoi d'un paquet **ACK**).

Pour être plus précis, ces échanges permettent aussi au client et au serveur de se mettre d'accord sur un identifiant pour se reconnaître lors des échanges futurs. En cas de non-réponse à l'une des requêtes, la connexion n'est pas établie.

La fermeture d'une connexion se fait par un « *Four-way Handshake* » ou poignée de main en quatre temps :

- Client : Je n'ai plus rien à te dire.
- Serveur : OK, j'ai noté. Tu ne peux plus me parler.
- Serveur : OK, moi aussi je n'ai plus rien à te dire.
- Client : OK, restons bons amis et à la prochaine !

Entre les phases 2 et 3, le client ne peut plus rien envoyer, mais le serveur le peut encore.

Pour être clair, si vous accédez à une page web contenant 8 images, 4 CSS et 2 JS, cela fait 15 connexions à établir (une pour chaque requête) et donc 45 + 60 envois de paquets juste pour gérer la connexion. C'est le principal reproche que l'on peut faire à TCP. D'un autre côté, le fait d'avoir une connexion forte présente certains avantages, en particulier lorsque l'on a vraiment besoin d'être certain que l'entièreté du message a bien été transmise. Une connexion TCP est identifiée par un 4-uplet (IP du serveur, port du serveur, IP du client, port du client). Un client peut avoir plusieurs connexions ouvertes en parallèle, elles se feront sur différents ports.

2.2 Serveur bas niveau

Commençons par préciser que tout ce que vous allez entendre ici est valable pour Python3. En effet, on fait du réseau, donc on traite des données de bas niveau. En clair, on échange des octets et non des chaînes de caractères et il s'agit d'un point très important.

Vous m'entendrez très rarement critiquer Python, mais il faut avouer que la version 2 est assez confuse à ce niveau-là. Ah, j'entends dans l'oreillette que certains d'entre vous s'attendaient à une critique plus violente ? Oui, mais on parle tout de même de la huitième merveille du monde, donc un peu de respect !

Toujours est-il, après cette digression inutile, mais nécessaire, que l'on se doit de mettre au point un protocole d'échange de données. En effet, vous n'allez pas pouvoir transmettre des objets, mais seulement des octets. Allez-vous utiliser XML ? Ou JSON ? Ce choix vous appartient. Pour notre part, nous allons rester le plus basique possible. Notre serveur va recevoir une séquence d'octets et il en renverra une autre (un « *Hello World* » tout ce qu'il y a de plus classique).

On commence simplement en important le nécessaire :

Fichier

```
import socket
```

Il faut ensuite déterminer le réseau sur lequel on écoute (ici la boucle locale) ainsi qu'un port, puis la taille maximale d'un paquet de données :

Fichier
```
params = ('127.0.0.1', 8808)
BUFFER_SIZE = 1024 # default
```

On peut maintenant initier la connexion côté serveur :

Fichier
```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(params)
s.listen(1)
```

`SOCK_STREAM` est la constante qui précise que l'on fait du TCP. Ensuite, le serveur va se mettre en attente d'un client. Tant qu'il n'y a pas de client, la méthode `accept` se comporte comme une boucle infinie. Dès qu'il y a un client, elle renvoie son adresse ainsi qu'un objet représentant la connexion :

Fichier
```
conn, addr = s.accept()
```

On peut effectivement afficher cette adresse :

Fichier
```
print('Connexion acceptée: %s' % str(addr))
```

Maintenant, on va recevoir la donnée transmise :

Fichier
```
data = conn.recv(BUFFER_SIZE)
```

Puis renvoyer notre réponse :

Fichier
```
conn.send(b'Bonjour ' + data.strip() + b'.\n')
```

On peut maintenant fermer la connexion.

Fichier
```
conn.close()
```

Puis le socket :

Fichier
```
s.close()
```

Fermer le socket pourrait vous sembler étrange. Et vous auriez raison. Il se trouve que l'on crée un serveur qui accepte de répondre à une seule condition, puis s'éteint. Dans la vraie vie, on crée un serveur qui reste en vie tant qu'on ne l'éteint pas volontairement. Ceci dit, comme précisé plus haut, une seule tâche (*thread*) ne peut pas à la fois être à l'écoute d'une éventuelle nouvelle connexion et en traiter une autre dans le même temps.

Le serveur est donc une tâche qui, lorsqu'il accepte une nouvelle connexion, crée une tâche fille et lui laisse le soin de gérer la connexion (ou toute autre stratégie similaire, comme avoir un *pool* de tâches prêtes à intervenir et les faire travailler à la demande). Bien évidemment, ceci est légèrement plus complexe et nous ne le traiterons pas dans cet article, d'autant plus que Python offre des outils de plus haut niveau qui gèrent par eux-mêmes cette complexité.

On peut aussi noter que le message peut être plus grand que la taille limite, il faudrait donc placer la méthode **recv** dans une boucle. De plus, lorsque le client termine sa connexion, il renvoie un message vide au serveur, lequel sait alors qu'il a terminé son travail et peut fermer la connexion.

D'autre part, il peut y avoir plusieurs échanges simultanés entre client et serveur et certains messages peuvent être plus longs que la taille du *buffer* choisie. Il est donc important pour gérer tout ceci de définir une convention sur le format des messages échangés, afin que le serveur sache quand un message unique est terminé et quand il reçoit un nouveau message, pour qu'il puisse traiter la donnée et renvoyer une réponse uniquement aux moments opportuns.

Enfin, il faut savoir que le moyen le plus utilisé pour couper un serveur est l'utilisation d'un signal d'interruption. Il est important de fermer le socket lors de l'arrêt du programme, quelle que soit la manière dont cet arrêt se produit. Et c'est là qu'intervient une syntaxe plus pythonique et plus adaptée :

Fichier

```
import socket

params = ('127.0.0.1', 8808)
BUFFER_SIZE = 1024 # default

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind(params)
    s.listen(1)

    conn, addr = s.accept()
    print('Connexion acceptée: %s' % str(addr))

    with conn:
        while True:
            data = conn.recv(BUFFER_SIZE)
            if not data:
                break
            conn.send(b'Bonjour ' + data.strip() + b'.\n')
```

La fermeture de connexion est implicite, gérée par le mot clé **with**, lequel est équivalent à un **try** … **finally** où ma fermeture de la connexion ou du socket seraient gérées dans le **finally**.

Si vous ne deviez retenir qu'une seule chose, ce serait la méthode d'utilisation du socket, c'est-à-dire sa création, le **bind**, le **listen** et le **accept**. Ce qu'il est important de retenir est que l'on établit une connexion permanente avec le client avec la méthode **accept**, puis que l'on travaille à partir de cet objet connexion, lequel sera éventuellement fermé implicitement ou non avec la méthode close.

2.3 Client

Voici maintenant ce à quoi peut ressembler un client :

Fichier

```
import socket

params = ("127.0.0.1", 8808)
BUFFER_SIZE = 1024 # default
```

On commence, là encore, par importer le module réseau bas niveau de Python et on définit les mêmes paramètres et la même taille d'échange qu'au niveau du serveur. Puis, on crée un socket de la même manière que pour le serveur :

Fichier
```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Enfin, on peut se connecter au serveur :

Fichier
```
s.connect(params)
```

Puis envoyer notre donnée :

Fichier
```
s.send(b"World")
```

Et attendre la réponse :

Fichier
```
data = s.recv(BUFFER_SIZE)
```

Et enfin, l'afficher :

Fichier
```
print("\tDonnée récupérée du serveur : %s" % data)
```

Puis fermer le socket dès que l'interaction avec le serveur est terminée :

Fichier
```
s.close()
```

On peut bien évidemment échanger plusieurs séries de messages avec le serveur, mais une fois que l'on ferme le socket, c'est terminé.

Pour communiquer, on utilise les méthodes **send** et **recv** en commençant par envoyer, là ou le serveur commence par recevoir. Ces méthodes sont cependant liées au socket et non pas à un objet connexion, comme c'est le cas pour le serveur.

2.4 Serveur Haut niveau

Le principe derrière un algorithme de haut niveau consiste à se concentrer sur ce qui compte réellement (c'est-à-dire ce que l'on veut vraiment faire) et à laisser le langage gérer la complexité inhérente à une utilisation correcte du protocole TCP.

Pour cela, on va utiliser une classe *Handler* dont le rôle sera de gérer tout le côté technique. Étant donné que l'on fait du TCP, on va utiliser **StreamRequestHandler** (pour rappel, lors de la création d'un socket TCP, on utilisait **SOCK_STREAM**) en créant une classe qui en hérite et qui surcharge la méthode **handle** :

Fichier
```
import socketserver

params = ('127.0.0.1', 8808)

class ExampleTCPHandler(socketserver.StreamRequestHandler):
 def handle(self):
  data = self.rfile.readline().strip()
  print('>>> Reçu: %s', data)
  self.wfile.write(b"Bonjour " + data.strip() + b".\n")
```

L'idée est que l'appel à une instance de cette classe va automatiquement appeler des méthodes qui vont préparer le terrain (gérer la connexion permanente et récupérer le

message du client), puis la méthode **handle**, puis des méthodes qui vont gérer la sortie (renvoyer un message au client et gérer l'éventuelle fin de connexion). Dans notre méthode **handle**, nous sont mis à disposition deux attributs essentiels : **rfile** qui contient le message en provenance du client (la lettre **r** insistant sur le fait que le message est lu) et **wfile** qui contient le message qui sera renvoyé au client après la fin de la méthode **handle** (la lettre **w** insistant sur le fait que le message est écrit). On doit aussi noter que la manière de lire le message est par l'utilisation de **readline** : ce qui veut dire que le message envoyé par le client doit explicitement être terminé par un **\n**. On note que le message reçu comme celui renvoyé sont de type **bytes** (octets) et que, encore une fois, le format de la donnée que l'on décide d'envoyer est un choix personnel.

Pour créer le serveur TCP, il suffit maintenant de procéder ainsi :

Fichier

```
if __name__ == '__main__':
    server = socketserver.TCPServer(params, ExampleTCPHandler)
    server.serve_forever()
```

On se retrouve donc avec un serveur capable de tourner indéfiniment, mais qui ne peut traiter qu'une seule connexion à la fois. On verra plus tard comment en traiter plusieurs simultanément.

3. UDP

3.1 Introduction

UDP (*User Datagram Protocol*) est un protocole qui, exactement comme TCP, appartient à la quatrième couche du modèle OSI, la couche de transport. Là encore, il s'agit d'un protocole assez ancien, extrêmement fiable et très utilisé. Un exemple d'utilisation : lorsque vous regardez une vidéo sur Internet. UDP est complémentaire de TCP, dans le sens où il fonctionne d'une manière totalement différente et donc permet de répondre à un besoin totalement différent. Il fonctionne sans avoir besoin d'une connexion permanente : il envoie un paquet au destinataire et si ce dernier le reçoit, c'est bien, sinon, tant pis. Pas de renvoi.

L'avantage est qu'il n'y a pas de négociation (pas de *handshake* pour créer ou terminer une négociation), ce qui fait gagner du temps. Les inconvénients sont que les paquets n'arrivent pas dans le bon ordre et potentiellement peuvent ne pas arriver : UDP ne garantit en rien la bonne livraison des paquets.

Du coup, ça fait un peu peur. L'idée, c'est que ce protocole est à utiliser si vous souhaitez transmettre rapidement de petites quantités de données et que la perte d'une partie d'entre elles n'est pas préjudiciable, ce qui est souvent le cas. Et si ce n'est pas le cas, alors utilisez TCP, il est fait pour cela. Pour être concret, lorsque vous regardez une vidéo, le serveur va vous envoyer plein de paquets, contenant images et sons. Mais votre lecteur va les garder en mémoire et les lire en respectant le temps de lecture (24 images par secondes, par exemple). Il va donc, parce qu'on n'est plus en 1950, rapidement télécharger et accumuler les données en avance, pour pouvoir vous les lire au fur et à mesure. Et s'il vous manque un morceau d'image ou de son, ce sera d'une part suffisamment rare et d'autre part suffisamment peu visible ou audible que cela ne fera pas grande différence : vous pourrez toujours apprécier votre vidéo.

Et en effet, il existe des tonnes de cas d'utilisation où l'on peut se permettre de perdre une partie de l'information sans que ce soit pénalisant et où l'on va donc décider de privilégier la vitesse à la sécurité. Sachez qu'il existe **QUIC**, un protocole expérimental qui propose de remplacer TCP par UDP pour le Web, ce qui serait un assez grand bouleversement.

3.2 Serveur bas niveau

Dans les grandes lignes, le serveur bas niveau en UDP s'écrit exactement comme celui en TCP : on utilise le même module et les mêmes paramètres :

Fichier
```
import socket
params = ('127.0.0.1', 8808)
BUFFER_SIZE = 1024 # default
```

Il faut ensuite préciser que l'on est en UDP :

Fichier
```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

SOCK_DGRAM est la constante qui précise que l'on fait de l'UDP. Il faut ensuite adapter le code aux spécificités d'UDP : pas de **listen** ou d'**accept**, parce que pas de négociation et de connexion permanente :

Fichier
```
s.bind(params)
```

Le serveur se met directement à l'écoute d'une requête :

Fichier
```
data, addr = s.recvfrom(BUFFER_SIZE)
```

Notez la sémantique. Ce n'est plus la méthode **recv**, mais la méthode **recvfrom** : on insiste sur le fait que la méthode va renvoyer à la fois l'adresse du client ainsi que la donnée, soit un 2-uplet qui est facile à gérer à l'aide de l'affectation multiple. On peut ainsi afficher le message et sa provenance :

Fichier
```
print(">>> Reçu:", data, "from:", addr)
```

Puis renvoyer le message du serveur vers le client à l'aide de la méthode **sentto** (et non la méthode **send**) :

Fichier
```
s.sendto(b"Bonjour " + data.strip() + b".\n", addr)
```

À noter que l'on précise l'adresse du client, qui nous a été donnée par **recvfrom**. Enfin, dans cet exemple à visée purement pédagogique, nous fermons la connexion après un seul échange :

Fichier
```
s.close()
```

À noter que l'on aurait pu gérer tout ceci d'une manière plus pythonique :

Fichier
```
import socket

params = ('127.0.0.1', 8808)
BUFFER_SIZE = 1024 # default

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.bind(params)

    while True:
        data, addr = conn.recvfrom(BUFFER_SIZE)
        conn.sendto(b'Bonjour ' + data.strip() + b'.\n', addr)
```

Le serveur ainsi conçu ne peut être arrêté qu'en tuant le processus.

3.3 Client

En ce qui concerne le client, rien de fondamentalement choquant :

Fichier

```
import socket

params = ('127.0.0.1', 8808)
BUFFER_SIZE = 1024 # default

s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

On utilise ici, encore une fois, `SOCK_DGRAM` au lieu de `SOCK_STREAM` pour indiquer que l'on fait de l'UDP au lieu du TCP. On n'utilise pas `connect`, parce que l'on n'ouvre pas de connexion. Mais simplement `sendto` et `recvfrom` au lieu de `send` et `recv`, comme pour le serveur :

Fichier

```
s.sendto(b"World", params)
data, _ = s.recvfrom(BUFFER_SIZE)
print("\tDonnée récupérée du serveur : %s", data)
```

On peut noter que le `recvfrom` va nous dire que l'on reçoit des données du serveur dont l'adresse est déjà connue (variable `params`), et on n'a donc techniquement pas vraiment besoin de la récupérer à nouveau.

On peut enfin terminer le socket lorsque l'on n'a plus aucune donnée à échanger :

Fichier

```
s.close()
```

Les différences entre les deux clients sont uniquement dues aux différences de fonctionnement entre les deux protocoles.

3.4 Serveur Haut niveau

C'est là que l'on va commencer à percevoir les avantages du haut niveau :

Fichier

```
import socketserver
params = ('127.0.0.1', 8808)
class ExampleUDPHandler(socketserver.DatagramRequestHandler):
 def handle(self):
  data = self.rfile.readline().strip()
  print('>>> Reçu: %s', data)
  self.wfile.write(b"Bonjour " + data.strip() + b".\n")
```

Le code présenté ici est rigoureusement identique à celui pour le serveur TCP, à l'exception notable du fait que l'on hérite de `DatagramRequestHandler` au lieu de `StreamRequestHandler`, puisque l'on fait de l'UDP.

On pourra alors lancer le serveur ainsi :

Fichier

```
if __name__ == '__main__':
 server = socketserver.UDPServer(params, ExampleUDPHandler)
 server.serve_forever()
```

Et là encore, on peut souligner la similarité avec ce que l'on a déjà vu.

Donc, pour conclure, le réseau, c'est simple une fois que la complexité est masquée !

4. THREADS

Comme nous l'avons expliqué, créer un vrai serveur nécessite d'aller plus loin au niveau système : il faut créer des tâches et les gérer proprement. Ceci nous amène forcément un peu plus loin que le cadre strict de cet article, mais il me semble important, au minimum, d'en parler. Je ne vais cependant pas détailler le fonctionnement d'une tâche (*thread*) ou regarder ce qui se passe à bas niveau, mais simplement utiliser des outils de haut niveau. Et on commencera par importer les modules dont nous allons avoir besoin :

Fichier

```
import threading
import socketserver
```

Et par définir cette constante, comme précédemment :

Fichier

```
PARAMS = ('127.0.0.1', 8808)
```

On peut alors réutiliser le *handler* que nous avions créé précédemment, mais pour mettre en évidence le fait que chaque appel à un client va générer un nouveau *thread*, on peut aussi utiliser celui-ci :

Fichier

```
class ExampleUDPHandler(socketserver.DatagramRequestHandler):
    def handle(self):
        cur_thread = threading.current_thread()
        data = self.rfile.readline().strip()
        print('>>> Reçu: %s', data, ', current thread:', cur_thread)
        self.wfile.write(b"Bonjour " + data.strip() + b".\n")
```

On peut maintenant joyeusement entrer dans la complexité inhérente à l'ajout de la fonctionnalité de *threading* qui consiste ici, tout simplement, à surcharger la classe `UDPServer` et lui adjoindre un *Mixin* :

Fichier

```
class ThreadedUDPServer(socketserver.ThreadingMixIn, socketserver.UDPServer):
    pass
```

Rien de plus ne sera nécessaire, si ce n'est qu'il faut maintenant l'utiliser d'une manière légèrement différente :

Fichier

```
if __name__ == "__main__":
    server = ThreadedTCPServer(PARAMS, ThreadedTCPRequestHandler)
    server_thread = threading.Thread(target=server.serve_forever)
    server_thread.daemon = True
    server_thread.start()
    print("Server loop running in thread:", server_thread.name)
```

Revenons dans l'ordre sur ce que l'on vient de faire : on crée l'objet serveur, qui est simplement une instance de la classe que l'on vient de créer, puis on crée un *thread* dont le rôle est de faire tourner le serveur, en utilisant sa méthode `serve_forever`, ce que l'on faisait directement dans le code principal précédemment.

Ce *thread* créera un nouveau *thread* à chaque requête qu'il recevra et restera néanmoins à l'écoute d'une nouvelle connexion. Il mourra cependant lorsque le code présent se terminera. Il faut par conséquent trouver un moyen simple de mettre en attente le processus courant :

Fichier

```
    input()
```

Ce qui signifie que pour terminer le serveur, il suffit de taper sur la touche <**Entrée**>, ce qui aura pour effet de terminer proprement le serveur, grâce à ces lignes :

Fichier

```
server.shutdown()
server.server_close()
```

Le détail des notions derrière les tâches est assez complexe à embrasser, mais sachez que ces quelques lignes suffisent à répondre à une problématique qui touche à deux des domaines de l'informatique les plus difficiles à gérer proprement et qu'elles y répondent avec un code à la fois lisible et assez facile à appréhender.

5. SERVEUR WEB

On va terminer par l'implémentation d'un serveur web, histoire de vous proposer un autre angle et de vous faire toucher du doigt ce qu'il est nécessaire d'embrasser lorsque l'on fait du réseau. Tout d'abord, précisons que le Web, sous sa forme actuelle, mais pas forcément pour très longtemps encore si l'on en croit les récents développements, est basé sur TCP. On l'a déjà effleuré, mais reprenons plus dans le détail : lorsque vous tapez l'adresse de votre site préféré, une première requête va partir pour demander à quelle IP correspond ce site. Il s'agit de DNS et c'est hors sujet en ce qui nous concerne.

Par contre, une fois que le navigateur connaît l'IP, il va établir une connexion avec cette IP et demander la page que vous souhaitez visiter. Les données de la page sont renvoyées par le serveur au sein de cette connexion. Mais pour chaque image, fichier CSS, JS ou autre média, une nouvelle connexion va être créée (ce qui inclus le processus de négociation d'ouverture et de fermeture + le transfert des données avec assurance qu'aucun paquet n'a été perdu en chemin), même si on simplifie un peu, car en réalité, les choses sont légèrement plus complexes. Toujours est-il que l'élément clé, dans la phrase précédente est « demander la page que vous souhaitez visiter » (et un œil averti aura noté qu'il s'agit en fait de la phrase d'avant la précédente, mais passons).

Comment fait-on pour demander une page ? C'est là qu'intervient ce dont j'ai pu vous parler précédemment : le client et le serveur doivent se mettre d'accord sur un format d'échange de données. C'est ce que le Web fait, à l'exception du fait que... Internet a eu une histoire chaotique.

En gros, pour les navigateurs, il existe essentiellement deux méthodes qui sont **GET** et **POST**. La seconde est utilisée pour transmettre des données du client vers le serveur, celles-ci devant être validées par le serveur, car il va les utiliser. On parle ici de l'utilisation d'un formulaire.La méthode GET est utile pour simplement afficher une page, ou encore utiliser un formulaire de recherche. Les attributs de la recherche étant visibles dans l'URL, il est facile de mettre le résultat en cache, ce qui peut avoir des applications intéressantes.

On peut aussi noter la méthode **HEADER** qui permet de récupérer les entêtes d'une page, sans son contenu. Et il est donc temps de parler des entêtes. Ces derniers permettent au client et au serveur de s'échanger des métadonnées. Par exemple, le client peut dire « je veux des pages en français de préférence. Si t'as pas ça en stock, alors en espagnol, sinon en anglais, mais c'est vraiment mon dernier choix. Aussi, je suis capable de comprendre si tu décides de compresser les pages que tu m'envoies ou encore si tu décides d'utiliser telle autre technologie. C'est toi qui vois. ».

Le serveur va quant à lui répondre l'inévitable code d'erreur qui peut être **200** s'il n'y a pas d'erreur, ce qui semble étrange à dire, mais qui ne l'est pas. On a ensuite les codes **4xx**

pour des erreurs dont on estime qu'elles viennent du client dont la célèbre erreur **404** qui signifie que la page n'existe pas et qui est une erreur côté client même si vous venez en fait de cliquer sur un lien dans le menu du site. Viennent ensuite les erreurs **5xx** qui sont des erreurs côté serveur et enfin les erreurs **3xx** qui ne sont, là encore, pas forcément des erreurs, puisque l'on trouve un avertissement comme quoi l'on n'a pas les permissions nécessaires ou encore les redirections permanentes ou temporaires.

Au-delà de ce code d'erreur, les entêtes comprennent le *content-type*, l'encodage, la date ou date de dernière modification, le cookie, etc.

Mais au-delà de cela, lorsque l'on parle de Web, on parle aussi de services web et là, les méthodes sont celles-ci :

- GET permet d'obtenir de la donnée sur une liste d'objets ou un seul objet ;
- POST permet d'ajouter un objet ;
- PUT permet de modifier un objet ;
- DELETE permet de supprimer un objet ;
- HEADER permet de récupérer les en-têtes relatives à un objet.

Et nous nous arrêterons là. Mais tout ce que nous venons de dire a une importance, puisque le *handler* que vous allez utiliser pour faire du Web se doit de correspondre à ce qu'est réellement le Web.

Commençons par importer ce dont nous allons avoir besoin :

Fichier

```
from http.server import BaseHTTPRequestHandler, HTTPServer
```

Jusque-là, pas de surprises. Le nom du *handler* et celui du serveur correspondent bien à ce que l'on veut faire. On va cependant aller un peu plus loin sur un aspect : en Web, il y a la notion d'URL et il faut être capable de comprendre une URL et nous aurons ainsi besoin de ceci :

Fichier

```
from urllib.parse import urlparse
```

Il nous faut maintenant l'immanquable séquence suivante :

Fichier

```
PARAMS = '127.0.0.1', 8016
```

Et nous pouvons y aller gaiement :

Fichier

```
class HelloHandler(BaseHTTPRequestHandler):
    def do_GET(self):
      # Récupération des informations sur le client.
    infos = [
    infos.append('client_adress: %s' % str(self.client_address))
    infos.append('address_string: %s' % self.address_string())
    infos.append('command: %s' % self.command)
    infos.append('unparsed path: %s' % self.path)
    parsed = urlparse(self.path)
    infos.append('parsed path %s' % parsed.path)
    infos.append('query: %s' % parsed.query)
    infos.append('request_version: %s' % self.request_version)
    infos.append('server_version: %s' % self.server_version)
    infos.append('sys_version: %s' % self.sys_version)
    infos.append('protocol_version: %s' % self.protocol_version)
    for k, v in self.headers.items():
```

```
            infos.append('HEADER %s: %s' % (k, v.strip()))
                # Début de la création de la réponse
            self.send_response(200)
            self.send_header(b'Content-type', b'text/html')
            self.end_headers()
                # Envoi du contenu de la page au navigateur
            infos = b'<ul><li>' + b'</li><li>'.join([bytes(i, 'utf-8') for i in
        infos]) + b'</li></ul>'
            self.wfile.write(b"""<html><head><title>Hello World</title></
        head><body><p>Hello World</p>""" + infos + b"""</body></html>""")
```

Cela semble un peu complexe au premier coup d'œil, mais revenons-y calmement. On n'a pas de méthode *handle*, puisqu'il y a en fait plusieurs manières de gérer la chose, cela dépendant de la méthode. On a donc une méthode de classe pour chaque méthode web, ici **do_GET** (la méthode web est toujours en majuscule).

Ensuite, on crée simplement une liste dans laquelle on va mettre toutes les informations que l'on peut trouver et en faire une liste au sens HTML du terme. Et on va écrire le code HTML sous la forme d'octets (car n'oubliez pas, on fait du réseau, donc on reçoit et envoie des octets), cette partie étant réalisée par les deux dernières lignes.

Dans la première liste d'information, on reconnaît l'adresse du client, sous ses deux formes, la méthode web (**command**, ici un GET) ainsi que le chemin. Le chemin correspond à l'URL, et il est *parsé* à la première ligne de la méthode. On peut donc facilement extraire l'hôte, le port, le *path* et la *query*. Ainsi, dans **www.exemple.com:1234/chemin/vers/ici?a=1&b=2**, le *path* est **/chemin/vers/ici** et la *query* est **a=1&b=2**. Enfin, une fois le *handler* écrit, il faut créer le serveur, sans que ce soit d'une quelconque difficulté par rapport à ce que l'on a déjà vu :

Fichier

```
server = HTTPServer(params, HelloHandler)
server.serve_forever()
```

CONCLUSION

Cet article a pu, l'espace d'un instant, vous plonger dans les méandres du fonctionnement de vos réseaux, tout en se limitant au TCP et à l'UDP, ce qui est loin d'être exhaustif bien que ce soit les deux protocoles les plus universellement utilisés.

Vous avez ainsi pu comprendre comment ils fonctionnent, comment ils s'implémentent à bas niveau, mais surtout vu les outils que Python met à votre disposition pour vous permettre de faire du réseau sans avoir besoin de gérer ce niveau de détail. En effet, c'est toujours mieux de savoir comment fonctionnent les choses quand on doit les utiliser, mais c'est vraiment plus agréable de laisser le langage gérer cette complexité et de ne se concentrer que sur ce qui compte vraiment : son besoin.

Sachez également que la documentation de Python sur les modules **socket** ou **socketserver** est relativement complète, assortie de quelques exemples complémentaires de ceux que vous avez pu lire ici, et que vous trouverez énormément de modules pour écrire des clients ou serveurs utilisant à peu près n'importe quelle technologie : FTP, Webdav, SSH (plus délicat, mais ça existe) ou bien d'autres et que la plupart de ces modules font partie de la bibliothèque standard de Python.

Quand je vous dis que ce langage est merveilleux ! ▮

DÉVELOPPEZ UNE APPLICATION GRAPHIQUE UTILISANT LE RÉSEAU

Sébastien CHAZALLET

Maintenant que vous savez comment fonctionne un réseau et comment faire transiter des données, il est temps de l'appliquer à un cas pratique. Nous vous proposons de construire un serveur applicatif pour une application graphique et nous allons insister sur ce qui peut ou ne peut pas transiter par le réseau.

L'architecture d'une application est un élément central. De nos jours, parmi les développements modernes, on distingue trois types d'applications :

- les applications autonomes (*stand-alone*) ;
- les applications web ;
- les applications mobiles.

Quels sont les éléments qui font que celles-ci sont plus populaires que les autres ?

Pour les premières, cela s'explique très simplement. Lorsque vous avez une application du type logiciel de bureautique, existe-t-il une meilleure façon de la créer, qu'en passant par une couche graphique simple ? Y a-t-il une raison qu'elle ne soit pas autonome ? Pour les deux questions, la réponse est non.

Pour les suivantes, c'est la souplesse de la technologie web, sa rapidité de développement et sa normalisation (oui, ça peut paraître ironique, mais cela reste vrai..., par comparaison avec les autres domaines !) et son industrialisation qui ont eu raison de la concurrence.

Enfin, le développement mobile suit une croissance importante due à l'explosion du nombre de téléphones portables sur le marché, même si son objectif principal semble être d'emprisonner des utilisateurs qui ne semblent demander que cela, récolter par tous les moyens des données personnelles ou forcer de la publicité.

Eh bien, dans cet article, nous allons faire de la résistance. Parler de nos bonnes vieilles applications lourdes et des processus à mettre en place pour permettre de gérer une communication réseau. Mais essentiellement à visée pédagogique.

1. INTRODUCTION

1.1 Notion d'interface graphique

Une interface est ce qui permet à deux mondes d'échanger. Une interface au sens informatique du terme représente ce qui va permettre la communication homme-machine. Dans notre cas, l'idée est de dessiner – littéralement – une application qui va pouvoir nous présenter des informations (texte, zone de saisie, etc.) ainsi que des moyens d'action (boutons, curseurs, etc.) pour donner des ordres à la machine.

Lorsque l'on s'intéresse à l'architecture liée aux interfaces graphiques, on se doit de regarder au plus près le patron d'architecture MVC qui permet à plusieurs personnes visualisant les mêmes données de voir les changements en temps réel.

En ce qui nous concerne, nous n'avons pas le luxe d'entrer dans ces détails, ni même de vous montrer tout ce qu'il est possible de mettre en place avec une interface graphique.

Mais nous allons cependant vous dire que nous allons utiliser **Gtk** et vous laisser avec un tutoriel qui est à la fois court et complet **[1]**. Nous préciserons cependant que sa lecture n'est pas obligatoire pour suivre cet article, il s'agira d'un complément. Dans cet article, nous utiliserons ce qui est présenté à la fin du tutoriel officiel : **glade** et un code Python court.

1.2 Gtk et glade

Il existe de nombreuses bibliothèques permettant de réaliser des interfaces graphiques, parmi lesquelles on peut citer Gtk, sur laquelle se base **GNOME**, et qui est utilisable en Python en installant ceci :

Terminal

```
# aptitude install gobject-introspection
```

Cette simple commande se doit d'installer la bibliothèque graphique, écrite en C et C++, ainsi que les autres dépendances nécessaires et le *toolkit* pour développer une interface graphique en Python.

Avec Gtk, il est possible d'écrire un code qui va créer des composants graphiques, les paramétrer, les positionner et les faire fonctionner ensemble.

Le logiciel glade va vous permettre de – littéralement – dessiner ces mêmes composants graphiques. Donc, la même chose, mais sans effort. On se doute cependant qu'il faudra écrire un petit algorithme pour gérer l'animation de l'application. Pour l'installer :

Terminal

```
# aptitude install glade
```

Il faut ensuite l'ouvrir et découvrir l'interface. Sur la gauche, se trouve l'ensemble des différents composants qu'il est possible de créer, au centre la visualisation de l'interface et la partie de gauche est divisée en deux : en haut une vue arborescente des éléments que vous avez créés ; vous pouvez en sélectionner un et visualiser sur la partie basse l'ensemble de ses propriétés.

1.3 Créer une interface

Nous vous présentons ici une application autonome (*stand-alone*) de type « Hello World » qui sera le fil rouge de notre article. Nous allons vous guider pour créer par vous-même cette interface.

Pour commencer, nous avons créé une fenêtre (*Window*). Nous vous invitons à regarder les différentes propriétés. Cette fenêtre a un nom (`window1`) que vous pouvez changer (pour ma part, j'ai mis `fenetre_principale`). Vous voyez dans les onglets ***Général*** et ***Commun*** tout ce que vous pouvez paramétrer (beaucoup) et vous amuser à choisir différentes options pour découvrir ce que cela fait.

Ensuite, viennent les signaux. Vous avez un onglet dédié à cela, il vous présente tous les signaux liés à l'objet courant, et ces derniers sont présentés selon l'ordre d'héritage de cet objet courant. Ici, une fenêtre est une spécialisation d'un conteneur qui est lui-même un *widget* qui lui-même est un `Gobject`.

Chaque objet définissant ses propres signaux, lorsque l'on hérite d'un objet, on hérite aussi des signaux et on peut en rajouter. Nous allons ouvrir la zone ***GtkWidget*** pour aller chercher l'évènement ***delete-event*** et cliquer dans ***<Type Here>*** pour saisir le nom de la fonction à appeler lorsque ce signal se déclenche. Saisissez la lettre `o` et laissez le logiciel vous proposer le nom par défaut.

Le signal *delete-event* est déclenché lorsque l'utilisateur clique sur la croix permettant de fermer l'application. Il est impératif pour nous de capturer cet évènement pour terminer proprement l'application, nous verrons pourquoi ultérieurement.

Enfin, une fois la fenêtre paramétrée, on peut lui rajouter d'autres éléments. Pour notre part, nous avons créé deux boites, une verticale avec trois cases. La première case contient une boite horizontale qui elle-même contient un champ de saisie et un label. La seconde case contient un autre label et la dernière un bouton.

Pour le bouton, nous avons lié le signal `clicked` à un nom de méthode (celui par défaut), ce qui nous permettra de définir ce qu'il faut faire lorsque nous cliquerons sur ce bouton.

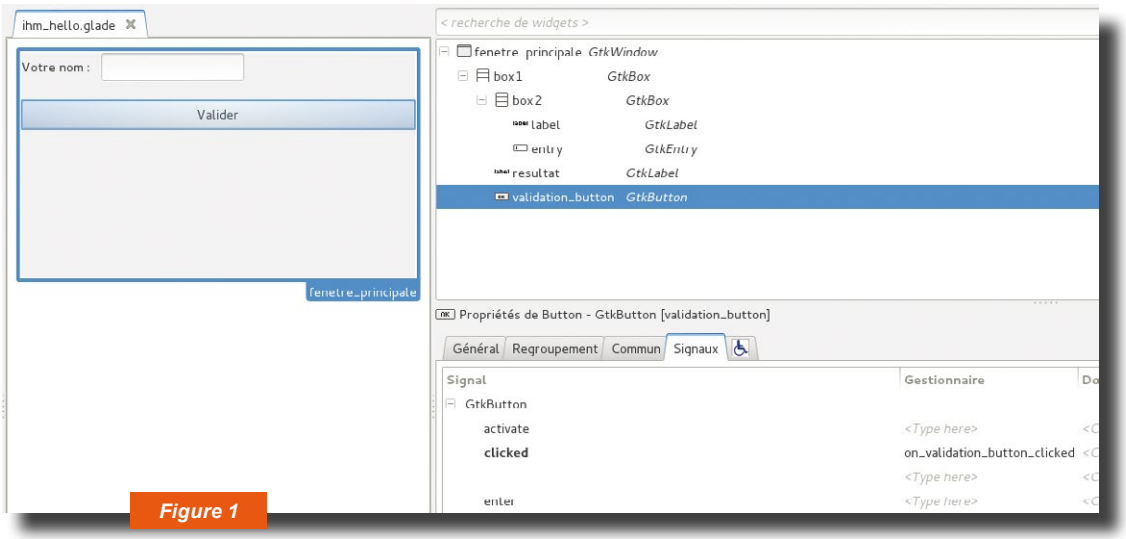


Figure 1

Les boites nous ont permis de gérer le positionnement des différents composants (vous pourrez en savoir plus dans le tutoriel et essayer des alternatives telles que la grille). Nous avons aussi choisi avec attention un nom précis pour chaque composant.

Le tout ressemble à la figure 1.

Une fois ce travail effectué, il vous faut enregistrer votre œuvre dans un fichier **hello.glade**.

1.4 Faire fonctionner l'IHM

Pour résumer, on a dessiné une interface et précisé qu'elle va gérer deux signaux particuliers.

Voici le code permettant de la faire fonctionner. Tout d'abord, commençons par le commencement :

Fichier

```
from gi.repository import Gtk
```

On va écrire maintenant une classe gestionnaire :

Fichier

```
class Handler:
    def __init__(self, builder):
        self.builder = builder
```

Comme on le verra dans quelques lignes, on utilise un objet **builder** pour charger l'interface faite avec Glade. Or, cet objet est génial, car il nous permet d'accéder à tous les éléments qui la composent très simplement. On a donc pris le parti d'agréger cet objet pour se faciliter la vie.

La classe gestionnaire va maintenant pouvoir écrire la méthode correspondant au premier signal :

Fichier

```
    def on_fenetre_principale_delete_event(self, widget, event):
        print("Bye.")
        Gtk.main_quit(widget, event)
```

On dit au revoir, parce que l'on est poli (c'est aussi une manière de vérifier que l'on est bien passé par là), mais surtout, on quitte ici la boucle Gtk dont je parlerais ultérieurement.

Pour le bouton, on gère les choses ainsi :

Fichier
```
    def on_validation_button_clicked(self, button):
        print("Hello World!")
        entry = self.builder.get_object('entry')
        resultat = self.builder.get_object('resultat')
        value = entry.get_text()
        if value:
            resultat.set_text(get_resultat(value))
        else:
            resultat.set_text('')
```

Là encore, on fait un affichage dans la console pour montrer que l'on utilise bien cette méthode, mais surtout, on va récupérer deux composants que sont **entry** et **resultat** pour lire le premier et modifier le second.

Et vous pouvez constater que nous utilisons ici le *builder* qui nous permet par l'utilisation d'une simple méthode **get_object** de récupérer n'importe quel composant, à partir de son nom. Sans cela, il faudrait se baser uniquement sur les éléments en paramètre, et naviguer dans l'arborescence des composants. Ce serait peu efficace et surtout pénible à écrire.

Voici maintenant comment on fait fonctionner le tout ensemble :

Fichier
```
builder = Gtk.Builder()
builder.add_from_file("ihm_hello.glade")
builder.connect_signals(Handler(builder))
```

On crée l'objet **Builder**, on lui fait lire l'IHM, puis on lui passe le gestionnaire pour gérer les signaux. Et le tour est joué. Il ne manque plus qu'à récupérer la fenêtre, l'afficher :

Fichier
```
if __name__ == "__main__":
    window = builder.get_object("fenetre_principale")
    window.show_all()
```

Et enfin, lancer la boucle principale de Gtk :

Fichier
```
    Gtk.main()
```

Comme vous le voyez, il s'agit d'un appel d'une fonction propre à Gtk, qui est indépendante du code précédent. Cette boucle va simplement se mettre à l'écoute d'évènements lancés par l'utilisateur (il a cliqué ici, il a tapé une lettre là, il a déplacé sa souris au-dessus de...) et faire les appels tels qu'ils ont été prévus.

C'est cette boucle dont il faut s'assurer qu'elle sera terminée proprement.

Avec ceci, se termine la courte introduction à Gtk et Glade, et nous allons reprendre cette application pour la rendre client – serveur.

2. ASPECT RÉSEAU

2.1 Gérer la connexion

L'idée d'une application réseau est que le client gère l'affichage, les signaux et soit indépendant, mais que le code métier et les données, soient gérées par le serveur.

Le gros problème d'une application *stand-alone* est qu'elle ne permet pas à deux utilisateurs de travailler en même temps sur le même jeu de données. Pour une application de gestion de projet, par exemple, c'est aberrant, cela ne fait simplement pas sens.

Nous allons donc créer un serveur qui soit capable de répondre à un besoin métier (ici un seul besoin et il est simpliste) :

Fichier

```
import socketserver

params = ('127.0.0.1', 8808)

class ExampleTCPHandler(socketserver.StreamRequestHandler):
    def handle(self):
        data = self.rfile.readline().strip()
        self.wfile.write(b"Bonjour " + data.strip() + b".\n")

if __name__ == '__main__':
    server = socketserver.TCPServer(params, ExampleTCPHandler)
    server.serve_forever()
```

Ici, la donnée reçue est simplement la donnée à traiter, la donnée renvoyée est simplement la donnée à afficher dans l'IHM.

Le client doit maintenant aller interroger le serveur pour savoir ce qu'il doit afficher :

Fichier

```
import socket
from gi.repository import Gtk

params = ('127.0.0.1', 8808)

BUFFER_SIZE = 1024

def get_resultat(value):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect(params)
        s.send("{}\n".format(value).encode())
        result = s.recv(BUFFER_SIZE)
        print('\tDonnée récupérée du serveur : {}'.format(result))
        return result.decode()
```

Cette fonction va ouvrir une connexion TCP pour aller envoyer la donnée (après l'avoir convertie en octets, car sur le réseau, rien d'autre que des octets ne peuvent circuler), puis récupère la réponse du serveur et la convertie en chaîne de caractères pour qu'elle soit utilisable.

Maintenant que la partie réseau est réglée, il faut modifier à minima notre gestionnaire :

Fichier

```
class Handler:

    def __init__(self, builder):
        self.builder = builder

    def on_fenetre_principale_delete_event(self, widget, event):
        print("Bye.")
        Gtk.main_quit(widget, event)

    def on_validation_button_clicked(self, button):
        print("Hello World!")
        entry = self.builder.get_object('entry')
        resultat = self.builder.get_object('resultat')
        value = entry.get_text()
        if value:
            resultat.set_text(get_resultat(value))
        else:
            resultat.set_text('')
```

Pour le reste du code, il n'y a pas de changements.

Par contre, bien évidemment, on n'aura jamais quelque chose d'aussi simple que cela. Le serveur n'a de sens que s'il est capable de répondre à de nombreux besoins. Il faut donc mettre en place un protocole entre le client et le serveur pour que le client indique ce dont il a besoin de manière précise.

Pour être clair, ce sont des problématiques qui sont réglées par le développement web en se basant sur l'URL. En effet, une URL permet de déterminer un contrôleur particulier et passer des données d'identification : **/blog/page?id=42** ou encore **/factures/4242**.

2.2 Utilisation d'une commande

On va utiliser ici une interface légèrement différente : deux boutons, pour deux types de messages (voir figure 2).

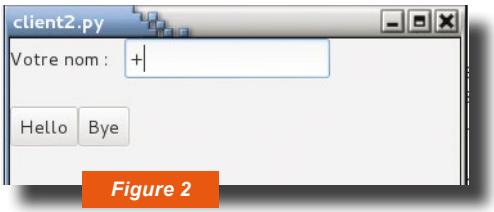


Figure 2

Une des manières les plus simples est de modifier le message pour le préfixer par une commande. On pourra utiliser un séparateur particulier (un caractère ou une séquence de caractères).

Le serveur sera ainsi modifié :

Fichier

```
class ExampleTCPHandler(socketserver.StreamRequestHandler):
    def handle(self):
        raw_data = self.rfile.readline().strip().decode().split(":")
        if len(raw_data) == 1:
            command = ""
        else:
            command, data = raw_data[0], ":".join(raw_data[1:])
        if command == "hello":
            result = "Bonjour {}.\n".format(data)
        elif command == "bye":
            result = "Au revoir {}.\n".format(data)
        else:
            result = "Erreur"
        print('>>> Reçu: ', command, ", ", data)
        self.wfile.write(result.encode())
```

Contrairement à tout à l'heure, on ne peut plus se contenter de travailler directement avec des octets, il nous faut impérativement les convertir, dans un sens, puis dans l'autre. On peut ensuite extraire la commande du message (attention, le séparateur utilisé peut parfaitement être dans le message, il faut donc le prévoir et reconstituer ce dernier proprement).

Pour le reste, on ira ensuite se baser sur la commande pour savoir quoi faire, que ce soit dans un simple branchement comme ici ou en utilisant une architecture plus complexe lorsque l'on a beaucoup de commandes.

Le client peut être modifié ainsi :

Fichier

```
def get_resultat(command, value):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect(params)
        s.send("{}:{}\n".format(command, value).encode())
        result = s.recv(BUFFER_SIZE)
        print('\tDonnée récupérée du serveur : {}'.format(result))
        return result.decode()
```

On voit ici que l'on envoie explicitement la commande suivie du message, le tout séparé par des deux-points.

Le gestionnaire sera modifié à la marge :

Fichier

```
class Handler:

    def __init__(self, builder):
        self.builder = builder

    def on_fenetre_principale_delete_event(self, widget, event):
        print("Bye.")
        Gtk.main_quit(widget, event)

    def on_hello_button_clicked(self, button):
        print("Hello World!")
        entry = self.builder.get_object('entry')
        resultat = self.builder.get_object('resultat')
        value = entry.get_text()
        if value:
            resultat.set_text(get_resultat("hello", value))
        else:
            resultat.set_text('')

    def on_bye_button_clicked(self, button):
        print("Bye World!")
        entry = self.builder.get_object('entry')
        resultat = self.builder.get_object('resultat')
        value = entry.get_text()
        if value:
            resultat.set_text(get_resultat("bye", value))
        else:
            resultat.set_text('')
```

Les deux méthodes sont ici suffisamment proches pour utiliser la même fonction. Enfin, on ne l'a pas fait ici, mais on pourrait aussi mettre en place un code de retour pour que le serveur puisse indiquer au client que quelque chose s'est mal passé, à la manière d'une commande du terminal ou, encore une fois, à la manière du Web (code **200** = OK, code autre = Erreur).

Comme il n'y a pas de standardisation à ce niveau-là, c'est à vous de mettre en place votre propre système, sachant qu'il doit être cohérent pour toute l'application et donc prévoir tous les cas possibles.

2.3 Utilisation de JSON

Si l'on en vient à complexifier les échanges entre client et serveur, un simple travail sur le découpage du message échangé ne suffira pas. En effet, le nombre d'informations à échanger ainsi que leur qualité peut changer d'une commande à une autre.

Il peut être bien, sachant cela, de se raccrocher à quelque chose de connu, de fiable et de précis. On peut s'orienter vers XML, par exemple (on le verra après) ou encore vers quelque chose de plus léger : le JSON.

L'idée est d'échanger un dictionnaire de données. Il faudra alors se mettre d'accord sur les clés, leur signification. Ici, on reste avec deux clés, une commande et un message.

Voici pour le serveur :

Fichier

```
class ExampleTCPHandler(socketserver.StreamRequestHandler):
    def handle(self):
        raw_data = json.loads(self.rfile.readline().strip().decode())
        print('>>> Reçu: ', raw_data)
        command, data = raw_data.get("command", ""), raw_data.get("message", "")
        if command == "hello":
```

```
            result = "Bonjour {}.\n".format(data)
        elif command == "bye":
            result = "Au revoir {}.\n".format(data)
        else:
            result = "Erreur"
        print('>>> Reçu: ', command, ", ", data)
        self.wfile.write(result.encode())
```

Dans l'ordre, on reçoit les données, on utilise le protocole JSON pour les récupérer sous la forme d'un dictionnaire, puis on les utilise et on renvoie ici la donnée brute (on pourrait aussi utiliser JSON pour la réponse, et mettre en place un protocole sur cet aspect également).

Voici maintenant le client :

Fichier

```
def get_resultat(command, value):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect(params)
        s.send(json.dumps({"command": command, "message": value}).encode() +
b"\n")
        result = s.recv(BUFFER_SIZE)
        print('\tDonnée récupérée du serveur : {}'.format(result))
        return result.decode()
```

On construit ici le dictionnaire de données à échanger, on le transforme en JSON, on y ajoute un caractère de fin de ligne (car le serveur utilise *readline*) et c'est la seule modification importante.

Le point important est : vous pouvez utiliser le format d'échange que vous souhaitez (mais on vous encourage à faire simple) tant que vous avez parfaitement défini ce que vous allez échanger entre client et serveur.

3. CLIENT DIRIGÉ PAR LE SERVEUR

Pour aller plus loin dans le concept client-serveur, on peut imaginer un client agnostique, qui ne soit pas prédéfini et qui soit simplement capable de suivre les instructions du serveur. Le serveur lui donnerait à la fois les interfaces à afficher ainsi que le code pour l'exploiter.

3.1 IHM envoyées par le réseau

Commençons par l'interface. On peut légitimement se demander comment faire cela. Mais on peut aussi se rendre compte que le fichier glade n'est rien d'autre qu'un fichier XML. Partant de là, pourquoi le client ne demanderait pas au serveur ce contenu XML au lieu de le lire sur le disque dur ?

On commence cette fois-ci par le client :

Fichier

```
def set_ihm(builder, ihm_name):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect(params)
        s.send("load_file:{}\n".format(ihm_name).encode())
        result = b""
        while True:
            res = s.recv(BUFFER_SIZE)
            if not res:
                break
            result += res
        builder.add_from_string(result.decode())
```

On a utilisé ici la technique qui consiste à préfixer le message échangé par une commande, afin de demander explicitement le contenu d'un fichier glade. On va ensuite récupérer ce contenu, qui sera a priori plus grand que la taille du *buffer*, puis on utilise la méthode **`add_from_string`** du *builder*.

Voici maintenant comment le serveur peut envoyer ce contenu :

Fichier

```
        if command == "load_file":
            with open("ihm/{}.glade".format(data), "rb") as f:
                self.wfile.write(f.read())
```

On ne produit ici que le code immédiatement utile, on a déjà vu comment extraire la commande et la donnée du message échangé par réseau.

Le point important est que l'on a ouvert le fichier en mode **`rb`**, donc en lecture seule, et en obtenant des octets et non une chaîne de caractères, ce qui nous évite d'avoir à convertir. La dernière ligne suffit ensuite à faire le travail, et comme vous le constatez, la complexité est minimale.

On a vu au passage comment envoyer du contenu XML par le réseau. Bien entendu, vous avez aussi, dans Python (et **lxml**) tous les outils pour générer ce XML ou le manipuler.

3.2 Envoyer du code Python par le réseau

Là, on tombe dans une problématique beaucoup plus complexe. Il s'agit de faire exécuter par le client du code Python fourni par le serveur. Un code Python peut contenir n'importe quel caractère. L'idée est donc d'utiliser *base64* pour ne pas avoir à gérer des problématiques de conversion. On va également utiliser *zlib* pour compresser le message, et ainsi gagner un peu de temps dans le transfert réseau.

On commence côté serveur :

Fichier

```
        elif command == "load_code":
            with open("code/{}.py".format(data), "rb") as f:
                self.wfile.write(base64.b64encode(zlib.compress(f.
read())))
```

On va chercher un fichier Python, que l'on lit au format octets que l'on compresse, puis transforme en base64. Ce fichier est celui-ci :

Fichier

```
import socket
from gi.repository import Gtk

params = ('127.0.0.1', 8808)
BUFFER_SIZE = 1024


global Handler

def get_resultat(value):
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(params)
    s.send("{}\n".format(value).encode())
    result = s.recv(BUFFER_SIZE)
    s.close()
    print('\tDonnée récupérée du serveur : {}'.format(result))
    return result.decode()
```

```
class Handler:

    def __init__(self, builder):
        self.builder = builder

    def on_fenetre_principale_delete_event(self, widget, event):
        print("Bye.")
        Gtk.main_quit(widget, event)

    def on_validation_button_clicked(self, button):
        print("Hello World!")
        entry = self.builder.get_object('entry')
        resultat = self.builder.get_object('resultat')
        value = entry.get_text()
        if value:
            resultat.set_text(get_resultat(value))
        else:
            resultat.set_text('')
```

Il contient un gestionnaire et tout ce dont le gestionnaire a besoin pour fonctionner. L'objet que l'on va utiliser est cette classe gestionnaire et il faut la rendre utilisable là où elle va être utilisée, autrement dit, rendre cet objet global (pour rappel, en Python, une classe est un objet, un objet de type *type*).

C'est un point important pour la suite.

Voici ce que cela donne côté client :

Fichier

```
def set_code(builder, code_name):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect(params)
        s.send("load_code:{}\n".format(code_name).encode())
        result = b""
        while True:
            res = s.recv(BUFFER_SIZE)
            if not res:
                break
            result += res
```

À cet instant, nous avons récupéré le message réseau tel qu'envoyé par le serveur. Il nous faut maintenant récupérer le code Python sous la forme d'une chaîne de caractères :

Fichier

```
        code = zlib.decompress(base64.b64decode(result))
```

Il faut maintenant exécuter ce code. Pour cela, on va mettre dans un dictionnaire les variables locales et les fournir au contexte d'exécution et exécuter ce code :

Fichier

```
        d = locals()
        exec(code, d, d)
```

À ce moment-là, le dictionnaire contient maintenant notre objet `Handler` :

Fichier

```
        Handler = d["Handler"]
```

On peut donc l'utiliser de manière très classique :

Fichier

```
builder.connect_signals(Handler(builder))
```

Pour rajouter quelques précisions, nous avons décidé qu'en l'absence de commande, on renvoyait la chaîne de type « Hello World », convertie en octets.

Nous avons maintenant les bases nécessaires pour créer un client qui soit dirigé par le serveur.

Le chemin reste encore très long, puisque la complexité de cette démarche nécessite de mettre en place un vrai format d'échange de données entre le client et le serveur comme entre le serveur et le client. Il faudra potentiellement définir une topologie particulière, voire se raccrocher à celle existant pour le développement web (contrôleurs, code d'erreurs, etc.).

CONCLUSION

Dans cet article, nous avons abordé la création d'interfaces avec glade, la manière de les faire fonctionner et la manière de faire piloter le client par le serveur.

Au passage, nous avons vu comment échanger des chaînes de caractères, des dictionnaires de données, du XML, en utilisant différents moyens de les convertir en octets (le seul type de donnée échangé par le réseau) ; on a vu l'utilisation de la compression, de base64 ou de JSON.

Ce large spectre doit vous montrer qu'il n'y a pas forcément une seule façon de répondre à une problématique, mais qu'il ne tient qu'à vous de définir votre propre mode de fonctionnement.

Ce que nous n'avons pas abordé dans cet article est la problématique MVC (qui nécessite simplement d'utiliser les différents évènements à votre disposition) ainsi que la gestion centralisée des données par le serveur. En effet, un serveur ira utiliser une ou plusieurs bases de données et il sera théoriquement le seul à y accéder.

RÉFÉRENCE

[1] Tutoriel Gtk :
http://python-gtk-3-tutorial.readthedocs.io/en/latest/introduction.html.

POUR ALLER PLUS LOIN

On peut citer une application qui va très loin dans son concept de dialogue client-serveur, puisque son serveur dirige son client : il s'agit de **Tryton**.

Si ce nom ne vous dit rien, sachez qu'il s'agit d'un fork d'**OpenERP** (maintenant **Odoo**) fait au moment où la société commençait à s'orienter vers le faux-libre.

Tryton est un serveur + un client Gtk + un client web. Il définit un protocole d'échange basé sur XML. D'un côté, vous définissez les modèles en Python, de l'autre les IHM directement en XML. Vous pourrez tester l'application et lire le code pour aller plus loin encore dans la communication client-serveur. ▉

UTILISEZ DES API REST EN PYTHON

Jean-Michel ARMAND

Comment interagir avec des services web, lorsque l'on fait du Python 3 ? Il y a toujours une bibliothèque Python pour faire le travail allez-vous me dire... Mais quand ce n'est pas le cas ?

Il n'existe plus vraiment de services web qui ne fournissent pas aujourd'hui une API REST pour que les utilisateurs dudit service puissent communiquer avec celui-ci. Et la plupart du temps, nous autres développeurs Python avons de la chance, il existe un *package* dans notre langage préféré pour nous aider. Mais quand ce n'est pas le cas ? Alors il faut se retrousser les manches et revenir aux fondamentaux, faire des appels HTTP. C'est ce que nous allons étudier dans cet article.

NOTE

Petite précision, tous les exemples de cet article ont été codés en Python 3.5. Parce que le changement et Python 3, c'est maintenant !

1. REQUESTS, LA BIBLIOTHÈQUE HTTP POUR LES HUMAINS

Requests [1] est une bibliothèque développée par Kenneth Reitz **[2]** depuis plusieurs années. Le but de Requests est d'être une bibliothèque simple, facilement compréhensible, mais qui regorge tout de même de fonctionnalités. Elle a tellement de succès qu'elle a éclipsé la plupart des autres bibliothèques Python servant à faire des appels HTTP et même la doc officielle de Python conseille de l'utiliser (nous verrons plus loin dans cet article que vu ce que fournit la bibliothèque standard comme outil, la documentation Python a bien raison de nous conseiller Requests).

1.1 Découverte de requests, installation et GET

Nous allons commençons par installer la bibliothèque. Pour cela, il nous faut un virtualenv, que vous pourrez créer avec la méthode que vous préférez (personnellement j'ai décidé d'alterner pour ce hors-série entre l'utilisation de `virtualenvwrapper` et `pipenv`).

Une fois cela fait, il vous faut lancer la commande suivante :

Terminal

```
$ pip install requests
```

Ceci vous installera `requests` en version 2.13.0 (en tout cas au moment où j'écris cet article).

Voilà, nous sommes prêts à utiliser la bibliothèque. Mais il nous faut encore trouver quelle API appeler. Nous allons faire preuve d'une grande originalité et allons travailler avec l'API de GitHub. Évidemment il existe une bibliothèque Python permettant de s'y connecter directement (que vous verrez en détail dans un autre article de ce hors-série), mais l'API de GitHub a l'intérêt de fonctionner en JSON, d'être plutôt bien *designée* et très bien documentée (la documentation se trouve à l'URL suivante : **https://developer.github.com/v3/**). Nous allons commencer par utiliser le *endpoint* listant les *endpoint* de l'API. C'est un simple appel GET à la racine de l'API. En `requests`, cela se fait ainsi :

LES ENVIRONNEMENTS VIRTUELS EN PYTHON 3

Tout au long de ce hors-série, vous allez apprendre à utiliser différentes bibliothèques, peut-être même dans différentes versions de Python. Pour ne pas que votre environnement ne devienne totalement bordélique, il serait bon que vous cloisonniez les choses. C'est là qu'interviennent les environnements virtuels Python. Un environnement virtuel est tout simplement un ensemble de bibliothèques rattaché à une version de Python bien spécifique. Un environnement virtuel s'active et vous ne pouvez en avoir qu'un seul d'actif à la fois. Cela vous permet donc de bien découper projet par projet, prototype par prototype. Vous pouvez tester différentes versions de la même bibliothèque ou différentes bibliothèques ayant le même but en gardant tout bien rangé et bien propre. Il y a plusieurs façons de faire cela en Python. Nous allons ici en présenter rapidement trois.

Tout d'abord l'historique, `virtualenv` [3] et son fidèle comparse `virtualenvwrapper` [4]. Installer virtualenv est très simple, que ce soit avec votre gestionnaire de paquets préféré (sous GNU/Debian, le paquet s'appelle `Python-virtualenv`) ou alors directement en utilisant `pip` (ici il faudra faire un `pip install virtualenv`). Bien qu'assez simple à utiliser, virtualenv est un peu rustre pour être utilisé de manière autonome. Virtualenvwrapper vous fournira une surcouche simplificatrice. Une fois celui-ci installé et après avoir rajouté deux lignes dans votre fichier de configuration de shell préféré, créer un virtualenv devient particulièrement simple, car il suffit d'utiliser la commande `mkvirtualenv NomVENV`. Activer un virtualenv se fait de manière là aussi très simple en utilisant `workon NomVENV`. Et pour sortir de l'environnement, un simple `deactivate` suffit. Petite subtilité : par défaut `mkvirtualenv` créera un virtualenv avec le Python par défaut de votre système, qui se trouve souvent être un Python 2. Pour utiliser Python 3, il faudra donner le chemin de votre interpréteur Python 3 à `mkvirtualenv` en utilisant l'option `-p` (soit `mkvirtualenv NomEnv -p /usr/bin/python3.5`)

Ensuite le « spécial » Python 3, le module `venv` [5] de Python3. Pour créer un environnement virtuel avec venv, il faut utiliser la commande `pyenv`. On peut décider que l'environnement virtuel, qui est par défaut isolé, pourra avoir accès aux bibliothèques système en ajoutant l'argument `--system-site-packages` lorsque l'on lance `pyenv`. Toujours de la même manière que pour virtualenv, une fois créé, l'environnement doit être activé. Cela se fait exactement de la même manière qu'avec virtualenv utilisé seul, c'est-à-dire qu'il faudra utiliser `source` (et il faut utiliser `deactivate` pour la désactivation du venv). Si on a crée son venv en faisant `pyenv /PATH_REP_STOCKAGE_VENV/NomVENV`, il faudra alors l'activer en utilisant `source /PATH_REP_STOCKAGE_VENV/NomVENV/bin/activate`.

Enfin, `pipenv` [6], un projet de Kenneth Reitz, encore lui, permet de mixer `pip`, les environnements virtuels et les `Pipfile`. Alors, bien que pipenv soit à l'état expérimental, il est déjà utilisable et propose quelques bonnes idées. Il est basé sur l'utilisation de `Pipfile`, un fichier de remplacement pour le `requirements.txt`. Un `pipfile` ressemble à un fichier `.ini`, se lit de manière claire et gère toutes les nouveautés de la PEP 508. Il permet de gérer différentes sections de déclaration de dépendances comme une section `dev` ou `production`. Et il permet d'être utilisé comme base pour générer un `pipfile.lock` qui, lui, va lister très précisément toutes les dépendances en profondeur (les dépendances de vos dépendances, ainsi que leurs propres dépendances, etc.) en fixant les versions. Fonctionnalité utile quand on ne veut pas se retrouver avec un projet totalement cassé parce qu'une dépendance de dépendance a introduit un changement d'API dans une montée de version mineure (et ne me dites pas que cela ne vous est jamais arrivé, je ne vous croirais pas). On crée donc son environnement virtuel avec `pipenv --three` ou `pipenv --two` (Python 3 ou 2). On installe ses paquets avec `pipenv install requests==2.13.0` ou `pipenv install django-debug-toolbar --dev` ou encore `pipenv install` si on a rempli son fichier `Pipfile`. Bien entendu, l'installation de *package* avec `pipenv install NomPaquet` modifie le `Pipfile` du projet pour le tenir à jour. Et on peut avoir un shell avec un `pipenv shell`.

Fichier

```
01: import requests
02: r = requests.get('https://api.github.com/')
03: print(r.status_code)
04: print(r.text)
05: print(r.json())
```

En première ligne, on commence par importer la bibliothèque, ensuite, ligne 2, on fait un simple appel **get** (vous l'avez deviné, un appel post se ferra avec **post**). Ligne 3, on affiche le **status_code** (ici, cela sera **200**) puis ligne 4 et 5 on affiche respectivement la réponse en texte et en json. La réponse se trouve être un gros dictionnaire listant tous les *endpoints* possibles.

Si l'on voulait lister les *repository* d'un utilisateur, on ferait alors :

Fichier

```
01: import requests
02: r = requests.get('https://api.github.com/users/kennethreitz/repos')
03: print(r.status_code)
04: print(r.text)
05: print(r.json())
```

Ici on obtient une liste (dans le cas de Kenneth, c'est une grosse liste) de dictionnaires définissant ses *repositories*. Mais quand on fait des requêtes **GET**, on aimerait bien pouvoir ajouter des paramètres, par exemple, on voudrait bien avoir la liste des repos dont Kenneth est un membre. Requests permet bien évidemment de passer des paramètres **GET**. Transformons simplement la ligne 2 en la ligne suivante :

Fichier

```
r = requests.get('https://api.github.com/users/kennethreitz/repos',
params={'type':'member'})
```

Pour passer des paramètres **GET** à un appel requests, il suffit donc de passer un dictionnaire dans l'argument **params**. Le dictionnaire en question doit être construit de la manière suivante : les clés correspondent aux noms des paramètres des arguments **GET** et les valeurs stockées dans le dictionnaire seront les valeurs desdits paramètres **GET**.

1.2 POST et authentification simple

Et si nous essayions de faire un **POST** ? Pour cela, nous allons créer un gist. Pour créer un gist, il faut simplement faire un POST avec les paramètres définis dans la documentation (**https://developer.github.com/v3/gists/**). C'est-à-dire, s'il est public ou non, la description et un ensemble de fichiers. Essayons d'en créer un en testant le code suivant :

Fichier

```
01: import requests
02: gist_data = { "description": "It is the GIST !",
03:                     "public": True,
04:                   "files": { "gist_file_1.txt": { "content": "I
don't want to talk to you no more, you empty-headed animal food
trough wiper! I fart in your general direction! Your mother was a
hamster and your father smelt of elderberries! "  }  } }
05: r = requests.post('https://api.github.com/gists', data=gist_data)
06: print(r.status_code)
07: print(r.json())
```

Si on lance ce morceau de code, on obtient la réponse suivante :

Terminal

```
$ Python create_gists.py
400
{'message': 'Problems parsing JSON', 'documentation_url': 'https://developer.
github.com/v3/gists/#create-a-gist'}
```

En effet, nous avons passé un dictionnaire directement à requests alors que l'API GitHub voulait du json. Dans les versions précédentes de requests, il fallait alors utiliser la bibliothèque json pour faire la conversion. Depuis la 2.4 de request, il y a plus simple. On peut remplacer la ligne 5 par :

Fichier

```
r = requests.post('https://api.github.com/gists', json=gist_data)
```

Alors cela fonctionne et l'on pourra récupérer un gros dictionnaire json permettant de tout connaître de notre gist. Mais bon, créer un gist en mode anonyme, ce n'est pas très marrant. On va donc se *loguer*. Requests permet de se *loguer* sur une API de plusieurs façons. Nous allons commencer par la plus simple, la paire *login* / mot de passe. Voici le code permettant de s'authentifier avec requests :

Fichier

```
01: import requests
02: from requests.auth import HTTPBasicAuth
03: r = requests.get('https://api.github.com/user',
auth=HTTPBasicAuth('LOGIN', 'PASSWORD'))
04: print(r.status_code)
05: print(r.json())
```

L'important ici c'est bien entendu les lignes 2 et 3. Sur la ligne 3, on importe la classe d'authentification et on l'utilise sur un appel get en ligne 3.

Maintenant que l'on est authentifié, on va pouvoir retenter de jouer avec les gists. Reprenons le code précédent et authentifions-nous afin de créer le gist :

Fichier

```
01: import requests
02: from requests.auth import HTTPBasicAuth
03: gist_data = {
04:   "description": "It is the GIST !",
05:   "public": True,
06:   "files": {
07:     "gist_file_1.txt": {
08:       "content": "King Arthur: What? Behind the rabbit? Tim: It
*is* the rabbit! "
09:     }
10:   }
11: }
12: r = requests.post('https://api.github.com/gists', json=gist_data,
auth=HTTPBasicAuth('LOGIN', 'PASSWORD'))
14: print(r.status_code)
14: r = requests.get('https://api.github.com/users/jmatestglm/gists')
15: print(len(r.json()))
```

Si on lance ce morceau de code, on obtient le résultat suivant :

Terminal

```
$ python create_gists_with_authent
201
2
```

On obtient tout d'abord un **201** qui signifie que la création du GIST s'est bien passée. Ensuite, on obtient le nombre de gist du compte GitHub en question, à savoir deux, celui que l'on vient de créer et un autre que j'avais créé à travers l'interface GitHub.

1.3 Autres verbes HTTP

Le protocole HTTP ne définit pas que les deux seuls verbes **GET** et **POST**. Requests permet bien entendu d'utiliser les autres verbes **HEAD**, **OPTIONS**, **PATCH**, **DELETE** ou **PUT**.

Essayons d'utiliser **PATCH** pour modifier le gist que l'on vient de créer :

Fichier

```
01: import requests
02: from requests.auth import HTTPBasicAuth
03: gist_data = {
04:   "description": "It is the GIST !",
05:   "public": True,
06:   "files": {
07:     "gist_file_1.txt": {
08:       "content": "I think it was 'Blessed are the cheesemakers'. "
09:     }
10:   }
11: }
12: r = requests.patch('https://api.github.com/gists/%s'
% '78b7f160458e7f2a7f1ca85be27b2b64', json=gist_data,
auth=HTTPBasicAuth('LOGIN', 'PASSWORD'))
13: print(r.status_code)
14: print(r.json())
```

On définit le nouveau corps du gist dans le dictionnaire et ligne 12, on appelle **PATCH** sur l'url de la ressource que l'on veut modifier.

En sortie, on va avoir (j'ai enlevé les choses inutiles) :

Terminal

```
200
{... 'description': 'It is the GIST !', 'html_url': 'https://gist.github.
com/78b7f160458e7f2a7f1ca85be27b2b64', 'comments': 0, 'url': 'https://api.
github.com/gists/78b7f160458e7f2a7f1ca85be27b2b64', 'files': {'gist_file_1.
txt': {'filename': 'gist_file_1.txt', 'content': "I think it was 'Blessed
are the cheesemakers'. ", 'truncated': False, 'size': 47, 'language':
'Text', 'type': 'text/plain'...}
```

Le **200** indique que tout s'est bien passé et ensuite parmi le json descriptif du gist, on retrouve bien la nouvelle description.

Utilisons maintenant un autre verbe HTTP, **DELETE**, pour supprimer le gist que nous venons de créer :

Fichier

```
01: import requests
02: r = requests.delete('https://api.github.com/gists/%s' %
'78b7f160458e7f2a7f1ca85be27b2b64', auth=('LOGIN', 'PASSWORD'))
03: print(r.status_code)
04: r = requests.get('https://api.github.com/users/jmatestglm/gists')
05: print(len(r.json()))
```

On aura ici le retour console suivant :

Terminal

```
$ python delete_gist
204
1
```

Le **204** indique que la suppression s'est bien passée et le **1** qu'il ne me reste donc plus qu'un seul gist. Concernant le code en lui-même, vous aurez sûrement remarqué qu'en ligne 2, j'ai simplifié l'authentification. C'est un raccourci d'écriture bien pratique que permet Requests, quand on utilise de l'authentification **Basic** HTTP. Ligne 2 toujours, j'appelle **delete** avec l'url de la ressource que je veux supprimer. Kenneth nous avait promis une bibliothèque simple d'utilisation, pour les humains... On peut dire que c'est le cas.

1.4 Autres types d'authentification

Nativement Requests gère quelques types d'authentification. Il gère la **Basic Auth** que nous avons déjà vue et il va également gérer l'authentification **Digest**. Mais comment vais-je faire si je dois gérer une authentification **Oauth** ou **JWT**, êtes-vous peut-être en train de penser ? Ne vous affolez pas, il existe des bibliothèques filles de Requests qui permettent de gérer cela. Pour tout ce qui est Oauth vous avez la bibliothèque **requests-oauthlib** [6] qui va vous permettre de gérer les différents *flow* Oauth d'une manière simple. Concernant le JWT, vous allez avoir la bibliothèque **requests-jwt** [7] qui vous permettra de mettre en place cela. Voici un exemple de connexion JWT avec définition d'un *header* pour le passage de l'authentification :

Fichier

```
01: import requests
02: from requests_jwt import JWTAuth
03:
04: response = requests.post(login_url, data={'login': USER,
'password': PASSWORD})
05: dict_login = response.json()
06: token = dict_login["token"]
07: user_id = dict_login["user"]["id"]
08: auth = JWTAuth(u"%s" % token, header_format='Bearer "%s"')
09: headers = {'Authorization': "Bearer %s" % token}
10: response = requests.get(PROTECTED_URL, headers=headers)
```

On commence par s'authentifier (ligne 4) pour récupérer un *token* (ce que l'on fait lignes 5 à 7) et ensuite on construit les informations de *token* JWT que l'on passera ensuite dans les *headers* (lignes 8 à 10).

1.5 Fonctionnalités supplémentaires

Nous n'allons pas faire le tour de Requests en un seul article, mais avant de passer à la présentation d'une autre bibliothèque, il m'a semblé intéressant de rapidement évoquer quelques petites choses sympathiques que propose Requests.

Les objets que renvoient les appels `requests` contiennent une donnée membre s'appelant `cookie`. D'une manière très logique, c'est pour gérer les *cookies*. L'objet en question se manipule comme un dictionnaire, mais c'est en fait un véritable objet qui fournit des outils pour travailler sur les domaines du *cookie* par exemple.

Par défaut Requests va faire les redirections pour tous les verbes HTTP sauf `HEAD`, de la même façon que pour `cookie`, une donnée membre de l'objet retournée par l'appel à `request`, `history` vous permettra de gérer cela.

Requests vous permet également de mettre en place un mécanisme de sessions lorsque vous faites vos requêtes. Cela vous permet de ne pas avoir à redonner certains paramètres à tous vos appels. Cela peut par exemple être utile pour tout ce qui est authentification.

Concernant l'envoi de fichiers, il est possible de mettre en place du *streaming*, mais aussi d'envoyer plusieurs fichiers dans la même requête lorsque vous faites un POST en `multipart-encoded`. Vous pourrez même définir vos verbes HTTP personnalisés !

2. UTILISATION DE LA BIBLIOTHÈQUE STANDARD PYTHON

Python est fourni avec des outils pour faire des requêtes HTTP, outil contenu dans la bibliothèque `urllib.request` [8]. Nous allons voir rapidement comment faire avec urllib une partie de ce que nous venons de faire avec requests.

2.1 Faire des requêtes GET

Nous allons refaire exactement ce que nous avons fait pour requests à savoir faire un appel `GET` sur le *endpoint* racine de l'API de GitHub et lister les *repositories* de Kenneth :

Fichier

```
01: import urllib.request
02: import urllib.parse
03:
04: op_url=urllib.request.urlopen('https://api.github.com/')
05: dir_api = op_url.read()
06:
07: params={'type':'member'}
08: data = urllib.parse.urlencode(params)
09: resp = urllib.request.urlopen('https://api.github.com/users/
kennethreitz/repos?' + data)
10: text_resp = resp.read()
```

Même si cela ressemble à ce que fait requests, on remarque assez rapidement que même sur deux appels GET aussi simples, la bibliothèque officielle est bien plus complexe. On ne sait pas explicitement que l'on fait un appel GET, il faut encoder soi-même ses paramètres et construire à la main son url. Et comme `urlopen` fournit quelque chose qui s'apparente à un fichier en retour, il faut explicitement en demander la « lecture » pour pouvoir obtenir ses résultats.

2.2 Post et authentification

Pour que urllib fasse une requête POST, il faut passer un paramètre `data` à `urlopen`. Ce paramètre contiendra les données encodées en `application/x-www-form-urlencoded`. Seul problème, GitHub, lui, veut absolument du json. On va donc devoir le spécifier. Créer un gist en urllib se présente donc de la manière suivante :

Fichier

```
01: import urllib.request
02: import urllib.parse
03: import json
04: gist_data = {
05:   "description": "It is the GIST !",
06:   "public": True,
07:   "files": {
08:     "gist_file_1.txt": {
09:       "content": "I don't want to talk to you no more, you
empty-headed animal food trough wiper! I fart in your general
direction! Your mother was a hamster and your father smelt of
elderberries! "
10:     }}}
11: params = json.dumps(gist_data).encode('utf8')
12: req = urllib.request.Request('https://api.github.com/gists',
data=params,
13:                              headers={'content-type': 'application/
json'})
14: response = urllib.request.urlopen(req)
15: page=response.read()
16: print(page)
```

Ligne 11, on transforme notre dictionnaire pour qu'il soit compréhensible, ensuite on crée un objet `Request` qui va contenir les informations nécessaires (l'url, les *datas* et les *headers*) et enfin on appelle `urlopen` qui va donc nous faire un appel POST avec des données en json.

Concernant l'authentification, urllib fonctionne avec des gestionnaires de mots de passe. Pour mettre en place une telle chose, voici le code à écrire :

Fichier

```
01: import urllib.request
02: import urllib.parse
03:
04: password_mgr = urllib.request.HTTPPasswordMgrWithDefaultRealm()
```

```
05:
06: top_level_url = "https://api.github.com"
07: password_mgr.add_password(None, top_level_url,
'LOGIN', 'PASSWORD') 08: handler = urllib.request.
HTTPBasicAuthHandler(password_mgr)
09: opener = urllib.request.build_opener(handler)
10: opener.open("https://api.github.com")
11:
12: urllib.request.install_opener(opener)
13: op_url=urllib.request.urlopen('https://api.github.com/')
```

On commence par créer le *password manager* ligne 4. On définit ensuite l'url racine (ligne 6) sur laquelle la paire *login* / mot de passe que l'on est en train de définir va s'appliquer. Ligne 7 on ajoute notre triplette *login*, mot de passe et url au manager de mot de passe. On construit ensuite un *handler* que l'on va utiliser pour construire un *opener* d'url (ligne 9). On peut alors directement utiliser cet *opener* d'url (comme en ligne 10) pour appeler des urls ou l'installer (ce que l'on fait en ligne 12) afin que tous les futurs appels faits à travers `urlopen` (et correspondant à l'url racine bien entendu) utilisent l'url *opener* avec authentification de manière transparente.

2.3 Autres verbes HTTP

Pour utiliser d'autres verbes HTTP que GET et POST il va falloir ruser et modifier un peu notre objet `request`. Basiquement, après avoir construit un objet `request`, il faut faire la chose suivante :

Fichier

```
req = urllib.request.Request(url='https://api.github.com/gists/',
data=params, headers={'content-type': 'application/json'})
req.get_method = lambda: 'PATCH'
```

Vous serez du même avis que moi je pense, on a vu mieux comme façon de faire !

3. HTTPLIB 2, ENCORE UNE BIBLIOTHÈQUE POUR FAIRE DES REQUÊTES HTTP !

Après s'être fait un peu peur en essayant de comprendre comment faire des appels HTTP avec urllib, retrouvons une sensation de normalité en étudiant une dernière bibliothèque à savoir `httplib2` [9]. Même si elle ressemble beaucoup à Requests, elle n'a pas la même façon de présenter les appels HTTP et propose un système de gestion du cache des requêtes par défaut. La documentation n'est par contre pas au niveau de celle de Requests.

3.1 Installation et appels GET

Après avoir créé un nouveau virtualenv, il vous suffira d'un petit appel à `pip` pour installer la bibliothèque :

Terminal

```
$ pip install httplib2
```

Comme pour Requests et urllib, testons un simple appel `GET` :

Fichier

```
01: import httplib2
02: h = httplib2.Http()
03: #h = httplib2.Http(".cache")
04:
05: headers, content = h.request("https://api.github.com/users/
kennethreitz/repos", "GET")
06: print(headers)
07: print(content)
```

Après avoir importé la bibliothèque, on commence par créer un objet `Http`. La ligne 3, qui est commentée permet de créer un objet Http activant la mise en place d'une politique de cache sur les appels http (les fichiers de cache seront mis dans le répertoire `.cache`). Ensuite, rentrons dans le vif du sujet avec un appel GET pour avoir la liste, encore une fois, des *repositories* de Kenneth. Contrairement à Requests, ici tous les appels se font avec la méthode `request` et on passe le nom du verbe HTTP que l'on veut utiliser en paramètre. Le retour de la méthode est un tuple, avec tout d'abord les *headers* puis le contenu de la réponse.

3.2 POST et authentification

Pour faire un POST, il va malheureusement falloir faire des choses similaires à ce que l'on a mis en place pour urllib. Voyons voir le code...

Fichier

```
01: import urllib.request
02: import urllib.parse
03: import json
04: import httplib2
05:
06: gist_data = {
07:   "description": "It is the GIST !",
08:   "public": True,
09:   "files": {
10:     "gist_file_1.txt": {
11:       "content": "I don't want to talk to you no more, you empty-
headed animal food trough wiper! I fart in your general direction!
Your mother was a hamster and your father smelt of elderberries! "
12:     }}}
13: params = json.dumps(gist_data).encode('utf8')
14: h = httplib2.Http()
```

```
15: headers, content = h.request('https://api.github.com/gists',
"POST",body=params, headers={'content-type': 'application/json'})
16: print(headers)
17: print(content)
```

Il faut bien reconnaître que l'on perd un peu en lisibilité. On commence par (ligne 13) manuellement mettre en forme le contenu de notre POST puis on fait l'appel ligne 15.

Concernant l'authentification, c'est très simple, il suffit d'ajouter les informations d'authentification à notre objet `Http()`. En clair, il faut faire la chose suivante pour que l'authentification `Basic Auth` fonctionne :

Fichier

```
h = httplib2.Http()
h.add_credentials('LOGIN', 'PASSWORD')
```

CONCLUSION

Le monde Python est plutôt bien pourvu lorsque l'on veut consommer de l'API, en tout cas de l'API REST. Il est vrai que concernant les vieilles API SOAP, c'est bien plus difficile. Mais malgré le fait qu'il existe un large nombre de bibliothèques permettant de jouer avec des API, à mon sens on peut les classer en deux catégories : Requests et toutes les autres. D'ailleurs, la plupart, pour ne pas dire la quasi-totalité des développeurs qui fournissent des bibliothèques offrant des abstractions au-dessus d'une API ne s'y trompent pas, ils utilisent Requests.

RÉFÉRENCES

[1] Requests : **http://docs.Python-requests.org/en/master/**

[2] Kenneth Reitz : **https://github.com/kennethreitz**

[3] Virtualenv : **http://www.virtualenv.org/en/latest/**

[4] Virtualenvwrapper : **http://www.virtualenv.org/en/latest/**

[5] Venv en Python 3 : **http://docs.Python.org/3/library/venv.html** et pipenv : **http://docs.pipenv.org/en/latest/**

[6] requests-oauthlib : **http://requests-oauthlib.readthedocs.io/en/latest/oauth2_workflow.html**

[7] requests-jwt : **https://pypi.Python.org/pypi/requests-jwt/0.4**

[8] urllib.request : **https://docs.Python.org/3/library/urllib.request.html**

[9] httplib2 : **https://httplib2.readthedocs.io/**

SCAPY, LE COUTEAU SUISSE PYTHON POUR LE RÉSEAU

Cyril Roelandt

Découvrez comment utiliser Scapy afin d'écrire vos propres outils réseau en Python.

Tout développeur réseau a déjà rêvé d'avoir à sa disposition une bibliothèque de manipulation de paquets, qui pourrait également être utilisée de façon interactive. C'est exactement ce que propose **Scapy**, que nous allons présenter dans cet article. Nous verrons tout d'abord son fonctionnement général, puis construirons des outils plus évolués. Il est impossible de détailler toutes les possibilités offertes par Scapy, mais à la fin de cet article, le lecteur devrait avoir toutes les bases nécessaires pour implémenter une application qui répondra à ses besoins.

1. CONCEPTS DE BASE

Voyons dans un premier temps comment prendre en main scapy en utilisant l'interpréteur. Nous utilisons ici la commande `scapy3` fournie par le paquet Debian `python3-scapy`. Notons qu'il faut lancer la commande en tant que `root`.

Terminal

```
# scapy3
WARNING: No route found for IPv6 destination :: (no default route?). This
affects only IPv6
INFO: Please, report issues to https://github.com/phaethon/scapy
WARNING: IPython not available. Using standard Python shell instead.
Welcome to Scapy (3.0.0)
>>>
```

Quelques avertissements peuvent apparaître selon la configuration de votre machine. Pensez à les relire si les exemples présentés dans la suite de l'article ne fonctionnent pas correctement.

1.1 Construire des paquets

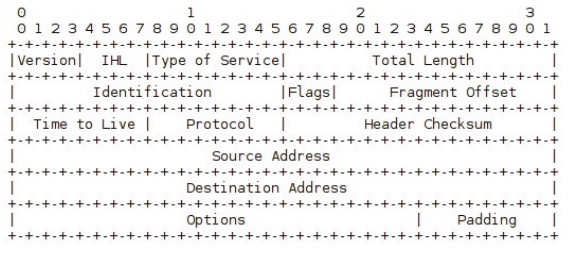
Construisons et analysons un premier paquet :

Fichier

```
>>> packet = IP()
>>> packet.show()
###[ IP ]###
  version= 4
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= ip
  chksum= None
  src= 127.0.0.1
  dst= 127.0.0.1
  \options\
```

NOTE

En affichant un paquet IP tout simple, on reconnaît les champs de l'entête IP, tels que décrits dans la RFC 791. Nous reproduisons en figure 1 le datagramme présenté en section 3.1 de cette RFC afin de rafraîchir la mémoire de nos lecteurs :



Structure de notre paquet.

Figure 1

On peut donner une valeur particulière à chacun des champs soit en passant des paramètres au constructeur de la classe `IP()`, soit en modifiant après coup les attributs :

Fichier

```
>>> packet = IP(dst='192.168.0.254')
>>> packet.ttl = 42
>>> packet.show()
###[ IP ]###
  version= 4
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 42
  proto= ip
  chksum= None
  src= 192.168.0.10
  dst= 192.168.0.254
  \options\
```

On peut ajouter des couches à notre paquet grâce à l'opérateur `/` :

Fichier

```
>>> packet = IP(dst='192.168.0.254')/ICMP()
>>> packet.show()
###[ IP ]###
  version= 4
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= icmp
  chksum= None
  src= 192.168.0.10
  dst= 192.168.0.254
  \options\
###[ ICMP ]###
     type= echo-request
     code= 0
     chksum= None
     id= 0x0
     seq= 0x0
```

On peut remarquer que les sommes de contrôle ne sont pas calculées. Elles le seront automatiquement lors de la construction du paquet, avant son envoi. Il est possible de les afficher grâce à la méthode `show2` :

Fichier

```
>>> packet.show2()
###[ IP ]###
  version= 4
  ihl= 5
  tos= 0x0
  len= 28
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= icmp
  chksum= 0xf887
  src= 192.168.0.10
  dst= 192.168.0.254
  \options\
###[ ICMP ]###
     type= echo-request
     code= 0
```

```
      chksum= 0xf7ff
      id= 0x0
      seq= 0x0
```

Nous avons donc construit un paquet similaire à ceux envoyés par l'utilitaire bien connu **ping**. Ce dernier ajoute par défaut une charge utile de 56 octets (qui peut contenir des informations d'horodatage, notamment sous **GNU/Linux**). Nous pouvons faire quelque chose de similaire :

Fichier

```
>>> packet=IP(dst='192.168.0.254')/ICMP()/Raw(load=RandString(56))
```

1.2 Envoyer des paquets

Maintenant que nous avons construit notre paquet ICMP, il nous faut l'envoyer. Plusieurs fonctions sont disponibles :

- ⇨ **send()** pour envoyer un paquet de la couche 3 ;
- ⇨ **sendp()** pour envoyer un paquet de la couche 2 ;
- ⇨ **sr1()** pour envoyer un paquet de la couche 3 et retourner la première réponse ;
- ⇨ **srp1()** pour envoyer un paquet de la couche 2 et retourner la première réponse ;
- ⇨ **sr()** pour envoyer un paquet de la couche 3 et retourner toutes les réponses ;
- ⇨ **srp()** pour envoyer un paquet de la couche 2 et retourner toutes les réponses.

Nous envoyons ici un paquet ICMP de type **echo-request**, et nous attendons à recevoir un autre paquet ICMP de type **echo-reply**. Nous pouvons donc utiliser **sr1** :

Fichier

```
>>> answer=sr1(packet, timeout=1)
>>> answer.show()
###[ IP ]###
  version= 4
  ihl= 5
  tos= 0x0
  len= 84
  id= 29709
  flags=
  frag= 0
  ttl= 64
  proto= icmp
  chksum= 0x8443
  src= 192.168.0.254
  dst= 192.168.0.10
  \options\
###[ ICMP ]###
     type= echo-reply
     code= 0
     chksum= 0x1a01
     id= 0x0
     seq= 0x0
###[ Raw ]###
        load=
'lJOHuMFbAVvLEttOTYQaF0fYGayqKLzAI8yQwvhJR0r7DQxQLT9kIf26'
```

Nous avons bien reçu la réponse attendue de la part de notre routeur (**192.168.0.254**). Si nous avions envoyé le paquet à une IP ne correspondant à aucune machine du réseau (ou ne répondant tout simplement pas au **ping**), la réponse aurait été **None** :

Fichier

```
>>> packet=IP(dst='192.168.0.253')/ICMP()/Raw(load=RandString(56))
>>> answer=sr1(packet,timeout=1)
WARNING: Mac address to reach destination not found. Using broadcast.
>>> answer is None
True
```

1.3 Écouter le réseau

Il nous est possible d'écouter le réseau, afin, par exemple, de récupérer les deux prochains paquets **ICMP** sur l'interface **eth0** :

Fichier

```
>>> pkts = sniff(filter='icmp', count=2, iface='eth0')
# La commande est bloquante. Elle retourne après avoir lancé en
# parallèle un 'ping -c 1 192.168.1.254'
>>> pkts.summary()
Ether / IP / ICMP 192.168.0.10 > 192.168.0.254 echo-request 0 / Raw
Ether / IP / ICMP 192.168.0.254 > 192.168.0.10 echo-reply 0 / Raw
```

On voit bien ici apparaître notre requête et la réponse du routeur.

2. SCAN DE PORTS

Il est assez facile, et très utile, d'écrire un programme effectuant un balayage de ports à l'aide de scapy : cela nous permet de trouver quels ports sont ouverts sur une machine distante. Il ne suffit pas d'envoyer n'importe quel paquet pour obtenir une réponse satisfaisante. Intéressons-nous à deux exemples détaillés.

2.1 Cas classique : scan TCP

Pour déterminer s'il est possible d'établir une connexion sur un port TCP donné, nous ne pouvons pas nous contenter d'envoyer un paquet forgé « au hasard » : nous risquerions de ne recevoir aucune réponse même si le port est ouvert. Essayons de contacter **www.gnulinuxmag.com** sans spécifier de drapeaux particuliers dans notre paquet TCP :

Fichier

```
>>> answer = sr1(IP(dst='www.gnulinuxmag.com')/TCP(dport=80,flags=''),
timeout=1)
>>> answer is None
True
```

Nous devons donc simuler une connexion TCP en trois étapes. Notons que la troisième étape n'est pas strictement nécessaire : recevoir un **SYN-ACK** nous indiquera que le port scanné est ouvert. Grâce à ce que nous avons appris dans la première partie de cet article, nous pouvons écrire la fonction suivante :

Fichier

```
from scapy.all import sr1, IP, TCP, conf

conf.verb = 0 # Quiet mode

SYN = 0x02
ACK = 0x10
SYNACK = SYN | ACK

def tcp_scan(host, port):
```

```
    syn_pkt = IP(dst=host)/TCP(dport=port, flags='S') # 'S' for 'SYN'
    synack_pkt = sr1(syn_pkt, timeout=1)
    if synack_pkt is None:
        print('Cannot reach host "%s" on port %d' % (host, port))
    elif synack_pkt['TCP'].flags == SYNACK:
        print("%5d OPEN" % port)
    else:
        print("%5d CLOSED" % port)
```

Ce code est relativement simple : on forge un paquet TCP comportant le drapeau **SYN**, on l'envoie à l'hôte sur le port que l'on souhaite tester, et si l'on reçoit une réponse comportant les drapeaux **SYN** et **ACK**, on déclare le port ouvert.

Il est alors possible d'utiliser cette fonction dans l'interpréteur Python, et d'avoir sous la main un outil similaire à **nmap**, bien que beaucoup moins avancé :

Fichier

```
>>> from tcp_port_scanner import tcp_scan
>>> tcp_scan('192.168.0.12', 22)
   22 OPEN
>>> tcp_scan('192.168.0.12', 80)
   80 CLOSED
```

Cette machine du réseau local fait sans doute tourner un serveur SSH, mais pas de serveur web (ou peut-être sur un port autre que le **80**).

2.2 Plus compliqué : scan OpenVPN

Dans la partie précédente, nous avons pris l'exemple bien connu de la connexion TCP en 3 étapes. Comment aurait-il fallu procéder pour tester la présence d'un autre service, comme un serveur **OpenVPN** ?

L'association **Aquilenet**, fournisseur d'accès à Internet associatif en Gironde, met à disposition de ses adhérents un VPN (**vpn.aquilenet.fr**) qui écoute sur le port **1194** (le port classique pour OpenVPN). En environnement hostile, il peut pourtant être impossible de se connecter sur ce port, et il pourrait être utile de pouvoir tester rapidement si la connexion est possible sur l'un des autres ports sur lesquels écoute OpenVPN.

On ne peut malheureusement pas se contenter d'envoyer un paquet UDP quelconque, auquel le VPN ne répondrait pas :

Fichier

```
>>> pkt = IP(dst='vpn.aquilenet.fr')/UDP(dport=1194)
>>> answer = sr1(pkt, timeout=2)
Begin emission:
.Finished to send 1 packets.
.......
Received 8 packets, got 0 answers, remaining 1 packets
>>> answer is None
True
```

Comme précédemment, nous devons donc forger un paquet valide et vérifier que le serveur OpenVPN y répond. Lançons donc la connexion dans un terminal, et regardons les paquets intéressants dans **Wireshark** (figure 2), grâce au filtre **openvpn** :

Terminal

```
# openvpn /etc/openvpn/aqn.ovpn
```

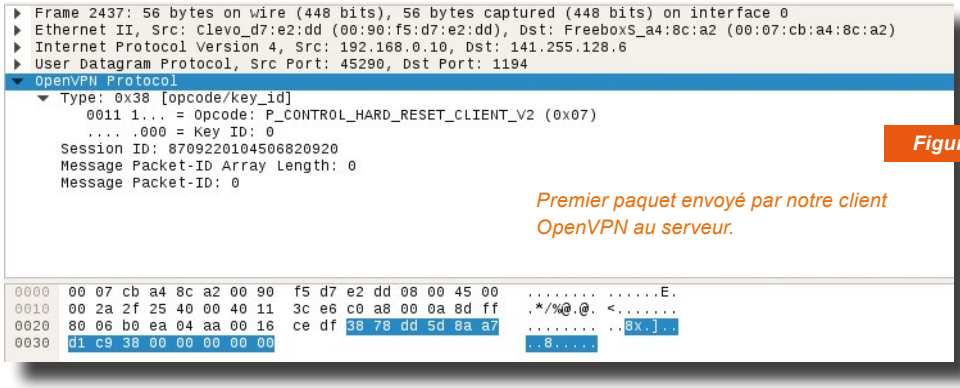


Figure 2

Premier paquet envoyé par notre client OpenVPN au serveur.

Nous pouvons alors écrire le code suivant :

Fichier

```
#!/usr/bin/env python3
from scapy.all import *

conf.verb = 0

def openvpn_udp_scan(host, port):
    print('[+] Scanning port %d... ' % port, end='')
    # La charge utile que l'on peut voir dans la figure 2
    msg = '\x38\x78\xdd\x5d\x8a\xa7\xd1\xc9\x38\x00\x00\x00\x00\x00'
    pkt = IP(dst=host)/UDP(sport=45290, dport=port)/Raw(load=msg)

    answer = sr1(pkt, timeout=3)
    if answer is None:
        print('KO')
    elif 'ICMP' in answer:
        print('KO')
    else:
        print('OK')

openvpn_udp_scan('vpn.aquilenet.fr', 1194)
openvpn_udp_scan('vpn.aquilenet.fr', 1195)
```

La seule subtilité est la gestion du cas où, le port étant injoignable, le serveur nous renverrait un paquet ICMP dont le type serait **destination unreachable**. On peut noter que nous ne vérifions pas ici le type ; la seule présence de la couche ICMP nous indique que nous n'arrivons pas à joindre le port.

Certes, ce petit programme fonctionne, mais il n'est pas très élégant de recopier la suite d'octets comme nous l'avons fait. Nous aurions préféré écrire quelque chose comme :

Fichier

```
pkt = IP(dst=host)/UDP(sport=45290, dport=port)
pkt/= OpenVPN(opcode='P_CONTROL_HARD_RESET_CLIENT_V2',
              keyid=0, ...)
```

Malheureusement, scapy ne définit pas de classe permettant de manipuler des paquets OpenVPN. Pourquoi ne pas l'écrire nous-mêmes dans la partie suivante ?

3. DÉFINISSEZ VOTRE PROPRE TYPE DE PAQUETS

Scapy permet d'implémenter un type de paquet en écrivant une classe dérivée de la classe **Packet**. Voyons cela avec l'exemple du protocole OpenVPN, défini à l'adresse suivante : **https://openvpn.net/index.php/open-source/documentation/security-overview.html**.

3.1 Une liste de champs

Un paquet est juste une liste de champs. Il suffit donc de les lister, grâce à une syntaxe particulièrement claire :

Fichier

```
class OpenVPN(Packet):
    name = 'OpenVPN'
    fields_desc = [
        BitEnumField('opcode', 1, 5, {
            1: 'P_CONTROL_HARD_RESET_CLIENT_V1',
            2: 'P_CONTROL_HARD_RESET_SERVER_V1',
            3: 'P_CONTROL_SOFT_RESET_V1',
            4: 'P_CONTROL_V1',
            5: 'P_ACK_V1',
            6: 'P_DATA_V1',
            7: 'P_CONTROL_HARD_RESET_CLIENT_V2',
            8: 'P_CONTROL_HARD_RESET_SERVER_V2',
            9: 'P_DATA_V2',
        }),
        BitField('keyid', 0, 3),
    ]
```

Nous définissons ici deux champs, que nous avons vu dans la figure 2 :

⇨ le champ **opcode**, long de 5 bits, dont la valeur par défaut est **1** ;

⇨ le champ **keyid**, long de 3 bits, dont la valeur par défaut est **0**.

Nous remarquons que le champ **opcode** est une énumération de toutes les valeurs possibles pour le champ. Il est d'ailleurs possible d'utiliser les deux syntaxes suivantes, strictement équivalentes, lors de la création d'un paquet :

Fichier

```
pkt = OpenVPN(opcode=7)
pkt = OpenVPN(opcode='P_CONTROL_HARD_RESET_CLIENT_V2')
```

Essayons maintenant de créer un paquet OpenVPN complet et de l'afficher :

Fichier

```
pkt = IP(dst='vpn.aquilenet.fr')/UDP(sport=1337,dport=1194)
pkt /= OpenVPN(opcode='P_CONTROL_HARD_RESET_CLIENT_V2')
# La charge utile est presque la même que dans la première partie :
# seul le premier octet est manquant, puisqu'il correspond aux champs
# opcode et keyid que nous définissons désormais de façon plus facile
# à lire.
pkt /= Raw(load='\x78\xdd\x5d\x8a\xa7\xd1\xc9\x38\x00\x00\x00\x00\x00')
pkt.show2()
```

En exécutant ce code, on peut voir 3 couches :

Terminal

```
###[ IP ]###
...
###[ UDP ]###
...
###[ Raw ]###
...
```

Nous aurions préféré voir une couche **OpenVPN** ainsi qu'une description lisible des champs définis dans notre classe. Il nous faut ici aider scapy en lui expliquant qu'il convient de décoder la couche qui vient après de l'UDP comme un paquet OpenVPN si le port utilisé est **1194** (le port par défaut d'OpenVPN) :

Fichier

```
bind_layers(UDP, OpenVPN, sport=1194)
bind_layers(UDP, OpenVPN, dport=1194)
```

Si nous relançons le code précédent, nous obtenons désormais un résultat bien plus facile à lire :

Terminal

```
###[ IP ]###
...
###[ UDP ]###
###[ OpenVPN ]###
        opcode = P_CONTROL_HARD_RESET_CLIENT_V2
        keyid = 0
###[ Raw ]###
...
```

On peut donc maintenant envoyer notre paquet et vérifier que nous obtenons bien une réponse, comme précédemment :

Fichier

```
answer = sr1(pkt)
answer['OpenVPN'].show2()

# La sortie :
###[ OpenVPN ]###
  opcode = P_CONTROL_HARD_RESET_SERVER_V2
  keyid = 0
###[ Raw ]###
     load = ...
```

Nous n'avons pas implémenté ici tous les champs visibles dans la figure 2 (**Session ID**, **Message Packet-ID**, etc.), ce qui nous force à spécifier une bonne partie du paquet en y ajoutant une charge utile. C'est parce que ces champs dépendent en effet de l'**opcode**. Implémenter complètement le type de paquet **OpenVPN** est un exercice un peu fastidieux dont la réalisation complète ne présenterait qu'un intérêt limité dans le cadre de cet article. Montrons toutefois comment utiliser des champs « optionnels » dans nos paquets.

3.2 Champs optionnels

Lorsque le paquet OpenVPN est envoyé en utilisant TCP, le premier champ de la couche OpenVPN doit être la taille du paquet, encodée sur 16 bits. Ce champ ne doit pas être présent lorsqu'on utilise UDP. Scapy permet heureusement de déclarer des champs optionnels :

Fichier

```
class OpenVPN(Packet):
    name = 'OpenVPN'
    fields_desc = [
        ConditionalField(ShortField("length", None),
                         lambda pkt: isinstance(pkt.underlayer, TCP)),
    ...
```

Nous indiquons ici que nous voulons insérer un **ShortField** uniquement si la fonction passée comme deuxième argument au constructeur de la classe **ConditionalField** retourne **True** pour ce paquet. On comprend aisément, en lisant le bout de code ci-dessus, que cette fonction lambda teste si la couche « du dessous » est TCP. Il nous faut maintenant donner une valeur correcte à ce champ lors de la construction du paquet : la méthode

`post_build` prend en paramètre le paquet et sa charge utile, et nous permet de modifier notre paquet :

Fichier

```
class OpenVPN(Packet):
    ...
    def post_build(self, pkt, pay):
        if isinstance(self.underlayer, TCP) and self.length is None:
            pkt = struct.pack("!H", len(pkt) - 2) + pkt[2:]
        return pkt + pay
```

Un petit tour dans la documentation du module `struct` de Python nous apprend que `!H` est la notation permettant d'obtenir un `unsigned short` en `big endian`, ce qui est exactement ce que nous voulons. Nous récupérons la longueur du paquet (moins la taille du champ `length` lui-même) et l'écrivons dans les deux premiers octets du paquet final.

Il ne nous resterait plus, pour complètement implémenter le format de paquet propre à OpenVPN, qu'à ajouter tous les champs qui dépendent de l'`opcode`, et à gérer leurs valeurs. Cela suggère bien évidemment de comprendre parfaitement le protocole OpenVPN. L'exercice est laissé au lecteur...

3.3 Sommes de contrôles

Donnons un autre exemple d'opération devant être effectuée dans la méthode `post_build` : le calcul des sommes de contrôles. Nous avons vu dans la première partie de cet article qu'il nous fallait utiliser la méthode `show2`, qui affiche un paquet après sa construction, pour pouvoir lire la valeur des sommes de contrôles de nos paquets. Voyons par exemple comment scapy gère le protocole IP :

Fichier

```
# Dans scapy/layers/inet.py
class IP(Packet, IPTools)
    ...
    def post_build(self, p, pay):
        ...
        if self.chksum is None:
            ck = checksum(p)
            p = p[:10]+chr(ck>>8)+chr(ck&0xff)+p[12:]
        ...
```

On voit ici comment, très simplement, scapy calcule la somme de contrôle d'un paquet IP et l'insère au bon endroit. Il est souvent utile d'aller fouiller dans les sources de scapy afin de trouver ce genre de code qui peut être réutilisé dans l'implémentation d'autres types de paquets.

CONCLUSION

Cette présentation de scapy s'achève, et bien qu'elle ne montre pas toutes les possibilités offertes par l'outil, le lecteur devrait désormais avoir les notions de base lui permettant de commencer à l'utiliser, et devrait pouvoir approfondir par lui-même ses connaissances afin d'écrire le code réseau de ses rêves.

Il est de toute façon particulièrement intéressant de parcourir le code de scapy (comme nous l'avons fait à la fin de la dernière partie de cet article), ou de s'intéresser aux programmes écrits grâce à scapy, afin de découvrir tout le potentiel de ce cadriciel. ▮

CRÉEZ

2 CRÉEZ...

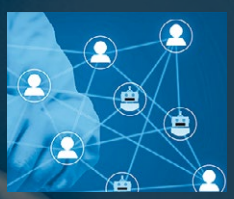
À découvrir dans cette partie...

... un système de migration des rapports de bug de GitHub à votre GitLab

Nous verrons ici comment manipuler les API de GitHub et GitLab grâce à des bibliothèques Python, et construirons un exemple didactique : la migration des rapports de bugs d'un service à l'autre. p. 58

... un driver FUSE pour Google Drive

Dans cet article, nous allons explorer la puissance de Python pour interfacer Linux et Google Drive. Nous allons décrire l'API fournie par Google, avant d'étudier un cas concret avec l'implémentation d'un driver FUSE. p. 68



... un bot IRC

IRC est l'une des plus vieilles plateformes de messagerie et nous allons voir dans cet article comment l'utiliser en Python 3.5, parfois à l'aide d'asyncio. p. 76

... un robot Slack

Dans le monde de la discussion sur Internet, Slack est la nouvelle attraction. Un joli habillage et une volonté affirmée de pouvoir se connecter ou être connecté de manière facile et rapide. Nous allons dans cet article voir comment le faire. p. 86

... un client XMPP

XMPP est le couteau suisse des messageries instantanées. Nous verrons dans cet article ce protocole qu'il faut connaître et que l'on va pouvoir utiliser en Python. p. 98

CRÉEZ UN SYSTÈME DE MIGRATION DES RAPPORTS DE BUG DE GITHUB À VOTRE GITLAB

Cyril Roelandt

Apprenez à manier les API de GitHub et GitLab afin d'effectuer des opérations courantes sur vos dépôts.

Il est de nos jours très courant pour des services web de fournir à leurs utilisateurs une API REST permettant d'effectuer diverses opérations depuis la ligne de commandes (notamment avec `curl`), ou d'écrire des applications utilisables en dehors du navigateur. Python étant un langage très populaire, de nombreuses API bénéficient d'une bibliothèque Python permettant de les manipuler. Voyons deux d'entre elles, qui permettent d'interagir avec des services d'hébergement de dépôts Git au travers de leurs API.

1. DÉCOUVERTE DE L'API GITHUB

GitHub fournit une API dont la documentation est disponible sur le Web (voir **https://developer.github.com/v3/**). Elle utilise le format bien connu `JSON`.

1.1 Premiers pas

Nous pouvons commencer par lancer une requête `GET` sur la racine de l'API :

Terminal
```
# Nous tronquons ici la sortie par souci de lisibilité, et afin de ne garder que
# les parties qui nous intéressent.
$ curl -s https://api.github.com/
{
  "current_user_url": "https://api.github.com/user",
  ...
  "repository_url": "https://api.github.com/repos/{owner}/{repo}",
  ...
}
```

On peut ensuite continuer notre exploration en utilisant les URL qui nous ont été retournées :

Terminal
```
# Récupérons des informations sur le dépôt "GLMF175" de l'utilisateur "GLMF".
$ curl -s https://api.github.com/repos/GLMF/GLMF175
{
  "id": 23825643,
  "name": "GLMF175",
  "full_name": "GLMF/GLMF175",
  ...
  "description": "Codes sources du n°175 de GNU/Linux Magazine (Octobre 2014)",
  ...
  "created_at": "2014-09-09T08:32:32Z",
  "updated_at": "2014-11-03T10:12:27Z",
  "pushed_at": "2014-10-02T07:30:06Z",
  "git_url": "git://github.com/GLMF/GLMF175.git",
  "ssh_url": "git@github.com:GLMF/GLMF175.git",
  "clone_url": "https://github.com/GLMF/GLMF175.git",
  "svn_url": "https://github.com/GLMF/GLMF175",
  ...
}
```

Certaines requêtes nécessitent une authentification :

Terminal
```
$ curl -s https://api.github.com/user
{
  "message": "Requires authentication",
  "documentation_url": "https://developer.github.com/v3"
}
```

```
$ curl -s https://api.github.com/user -u Steap
Enter host password for user 'Steap':
{
  "login": "Steap",
  "id": 416834,
  ...
  "html_url": "https://github.com/Steap",
  ...
  "repos_url": "https://api.github.com/users/Steap/repos",
  ...
  "name": "Cyril Roelandt",
  ...
}
```

Il existe plusieurs façons de s'authentifier, que nous détaillerons dans la partie suivante. Avant cela, force est de constater qu'effectuer toutes les requêtes avec `curl` va vite devenir relativement fastidieux ; il convient donc d'utiliser une bibliothèque **Python** rendant l'utilisation de l'API plus facile. Il en existe de nombreuses : nous avons choisi d'utiliser ici PyGithub (**https://pypi.python.org/pypi/PyGithub**), en version 1.32.

1.2 Authentification

Intéressons-nous à deux méthodes d'authentification : par mot de passe, et à l'aide d'un jeton privé. Notons qu'il est également possible d'enregistrer une application et de s'authentifier grâce au protocole **OAuth**, mais nous ne détaillerons pas ici cette méthode.

1.2.1 Par pseudo/mot de passe

Comme nous venons de le voir, il est possible de s'identifier en donnant tout simplement son pseudonyme (ou son adresse de courriel) et son mot de passe :

Fichier

```
>>> import github
>>> gh = github.Github('Steap', 'motdepasse')
# On peut vérifier que nous sommes bien connectés :
>>> print(gh.get_user().login)
Steap
```

1.2.2 Par jeton privé

Il est également possible d'utiliser un jeton privé, ce qui nous permet de gérer plus finement les permissions. Pour ce faire, il suffit de se rendre sur la page **https://github.com/settings/tokens/new** et d'y sélectionner les permissions requises (cf. Figure 1).

Créons un jeton en ne lui donnant pas la permission `gist`, puis vérifions qu'il est effectivement impossible de créer un `gist` :

Fichier

```
>>> import github
>>> gh = github.Github('tokensecret')
>>> gh.get_user().create_gist(True,
... {'hello.txt': github.InputFileContent('Hello GLMF')},
... 'A test gist')
Traceback (most recent call last):
...
github.GithubException.UnknownObjectException: 404...
```

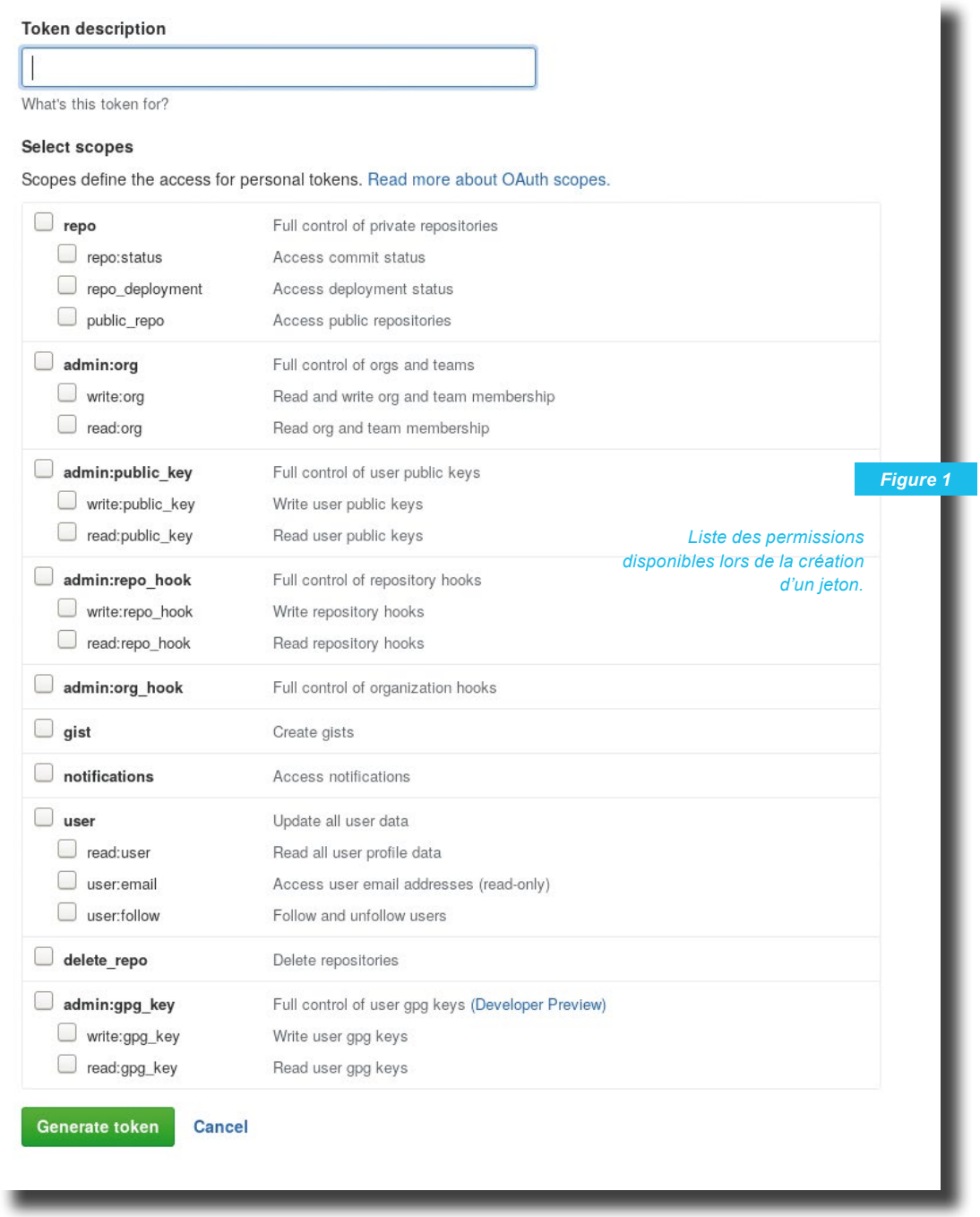


Figure 1

Liste des permissions disponibles lors de la création d'un jeton.

Créons maintenant un autre jeton, en lui donnant la permission de créer des **gists**. Nous pouvons le faire directement depuis notre console Python :

Fichier

```
>>> t = gh.get_user().create_authorization(["gist"], "Token-with-
gist")
>>> t.token
'nouveautoken'
```

Tentons à nouveau de créer un gist :

Fichier

```
>>> gh = github.Github('nouveautoken')
>>> gh.get_user().create_gist(True,
... {'hello.txt': github.InputFileContent('Hello GLMF')},
... 'A test gist')
Gist(id="a9050b4027ee16111458a6b34f959a94")
```

Cette fois-ci, la commande a fonctionné : nous pouvons le vérifier en nous rendant sur **https://gist.github.com/Steap/a9050b4027ee16111458a6b34f959a94**.

1.3 Aller plus loin

Nous n'avons pas détaillé, dans la partie précédente, les paramètres des méthodes utilisées (`create_authorization`, `create_gist`). Leur fonctionnement est relativement intuitif, et nous n'avons de toute façon pas la place d'écrire ici la documentation complète de PyGithub. Nous pouvons toutefois donner deux conseils pour appréhender ce type de bibliothèques :

- explorez les possibilités offertes par chaque objet Python dans l'interpréteur interactif : `dir()` et `help()` vous permettront de rapidement comprendre les subtilités de la bibliothèque utilisée ;
- lisez la documentation de l'API, même si vous ne comptez pas l'utiliser directement : les bibliothèques sont en général très proches de cette API, et apportent principalement une plus grande facilité d'utilisation.

Nous verrons dans la troisième partie de cet article d'autres cas d'utilisation de l'API GitHub.

2. DÉCOUVERTE DE L'API GITLAB

La documentation de l'API **GitLab** est bien évidemment disponible sur le Web (**https://docs.gitlab.com/ee/api/README.html#gitlab-api**). Montrons ici quelques cas d'utilisation, en utilisant l'instance GitLab hébergée par **Framasoft** (**https://framagit.org/**).

2.1 Authentification

Comme pour l'API GitHub, il est nécessaire de savoir s'authentifier.

2.1.1 Par pseudo/mot de passe

Sur GitLab, l'authentification par pseudo/mot de passe permet uniquement de récupérer un jeton privé (également disponible dans l'interface web) :

Terminal

```
$ curl -X POST -s "https://framagit.org/api/v3/session?login=Steap&password=xxx"
{
   "web_url" : "https://framagit.org/Steap",
```

```
    ...
    "private_token" : "secret-token",
    ...
    "username" : "Steap",
    ...
}
```

Voyons maintenant comment utiliser ce jeton privé.

2.1.2 Jeton privé

Le jeton privé que nous venons de récupérer peut maintenant être utilisé pour interroger l'API, par exemple afin de lister nos projets :

Terminal

```
$ curl -s https://framagit.org/api/v3/projects -H 'PRIVATE-TOKEN: secret-token'
[
  {
    ...
    "web_url" : "https://framagit.org/Steap/quiparrainequi",
    "id" : 13198,
    "name" : "quiparrainequi",
    ...
  }
  ...
]
```

2.1.3 OAuth2

Il est également possible d'utiliser OAuth2 afin de bénéficier d'une gestion plus fine des permissions. Une documentation complète est disponible sur le Web (**https://docs.gitlab.com/ee/api/README.html#oauth-2-tokens**), mais voyons tout de même rapidement les étapes nécessaires à l'obtention d'un jeton.

Il faut tout d'abord créer une application en passant par l'interface web de GitLab (voir figure 2).

GitLab affichera alors votre **Application ID**, votre **secret** ainsi que l'**URL** de redirection que vous avez indiquée. Utilisez ces informations pour construire l'URL suivante :

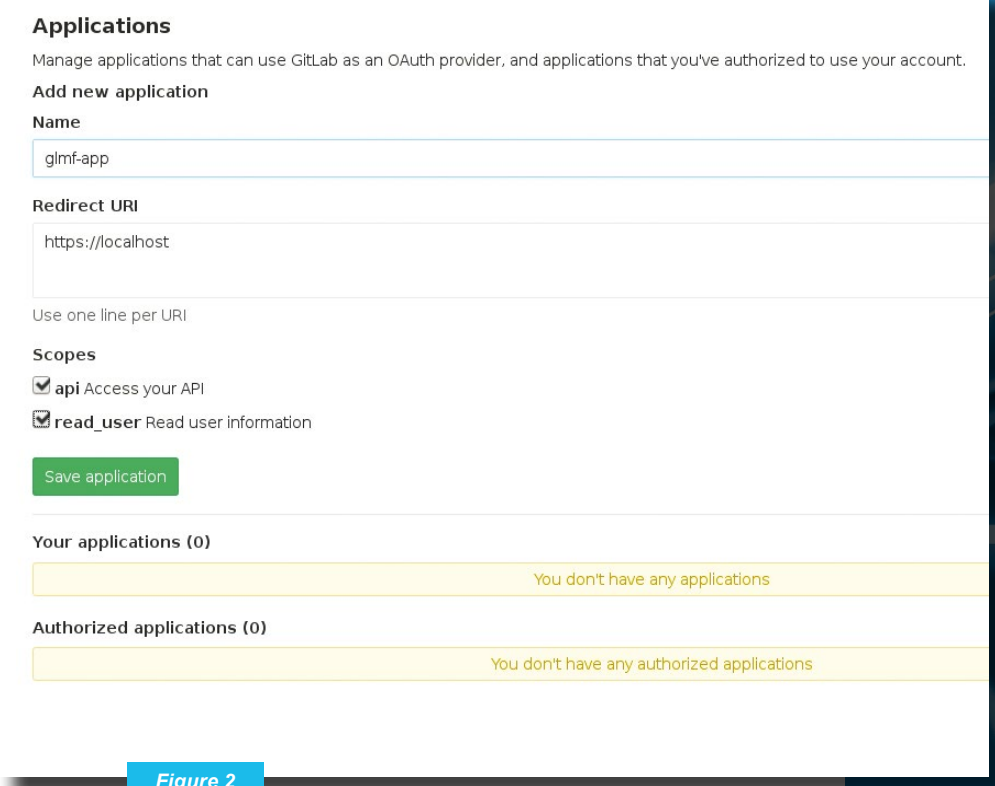


Figure 2

Création d'une application dans GitLab.

Fichier

```
>>> url = "https://framagit.org/oauth/authorize"
>>> url+= "?client_id=%s" % APP_ID
>>> url+= "&redirect_uri=%s" % REDIRECT_URI
>>> url+= "&response_type=code"
```

Dégainez votre butineur et rendez-vous sur cette page, donnez à l'application les permissions qu'elle vous demande, et vous serez redirigé vers une page dont l'URL se terminera par `code=$CODE`. Une requête vous permettra d'obtenir (enfin !) votre jeton OAuth2 :

Terminal

```
$ URL=https://framagit.org/oauth/token
$ URL="$URL?client_id=$APP_ID"
$ URL="$URL&client_secret=$SECRET"
$ URL="$URL&code=$CODE"
$ URL="$URL&grant_type=authorization_code"
$ URL="$URL&redirect_uri=$REDIRECT_URI"
$ curl -s -X POST $URL | json_pp
{
   "refresh_token" : "...",
   "scope" : "api",
   "created_at" : 1490500721,
   "token_type" : "bearer",
   "access_token" : "secret-access-token"
}
```

Vous pouvez ensuite utiliser ce jeton dans vos requêtes :

Terminal

```
$ curl -H "Authorization: Bearer secret-access-token" \
    https://framagit.org/api/v3/projects/
[
  ...
]
```

2.2 Une bibliothèque : python-gitlab

Rendons-nous la vie plus simple en utilisant une bibliothèque Python : **python-gitlab** en version 0.20. S'authentifier en utilisant un jeton privé se fait très simplement :

Fichier

```
>>> import gitlab
>>> gl = gitlab.Gitlab('https://framagit.org', 'secret-token')
```

Il est ensuite possible d'explorer la bibliothèque comme nous l'avons fait avec PyGithub :

Fichier

```
>>> for project in gl.projects.list():
...     print(project.name)
...
quiparrainequi
framadate
sf2cf
```

Voyons dans la prochaine partie comment utiliser python-gitlab pour manipuler nos projets.

3. EXEMPLE : MIGRATIONS DES RAPPORTS DE BUGS

Lorsqu'un utilisateur de GitLab veut créer un nouveau projet, il a la possibilité d'importer un projet existant depuis un autre site, notamment GitHub (voir figure 3). Le dépôt Git sera automatiquement importé, et les rapports de bugs seront copiés. Comment pourrions-nous utiliser les API de GitHub et de GitLab afin d'implémenter une fonctionnalité similaire ?

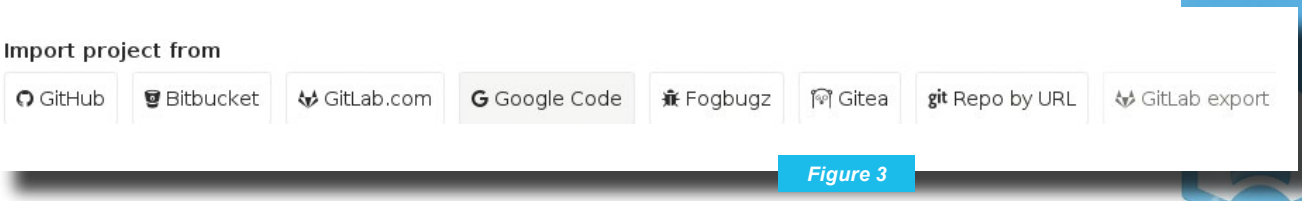


Figure 3

GitLab permet d'importer un projet depuis un autre service.

3.1 Idée générale

L'idée de base est très simple : il suffit en effet de créer un nouveau projet, puis de migrer chacun des services associés (bugs, wiki, etc.) de GitHub vers GitLab. Nous nous contenterons ici de migrer les bugs. Regardons le point d'entrée de notre code :

Fichier

```
import github
import gitlab

GITHUB_TOKEN = 'secret-github-token'
gh = github.Github(GITHUB_TOKEN)

GITLAB_URL = 'https://framagit.org'
GITLAB_TOKEN = 'secret-gitlab-token'
gl = gitlab.Gitlab(GITLAB_URL, GITLAB_TOKEN)

def migrate_gh_project(project_name):
    # Create a new project on Gitlab
    gl_project = gitlab_create_project(project_name)

    gh_repo = gh.get_user().get_repo(project_name)

    # Migrate all issues
    for gh_issue in gh_repo.get_issues():
        gl_issue = gitlab_clone_github_issue(gl_project, gh_issue)

    # TODO: Migrate everything else!

migrate_gh_project('quiparrainequi')
```

Pour l'instant, tout est facilement compréhensible. On notera bien évidemment que `gl` fait référence à GitLab, et `gh` à GitHub. Il ne nous reste plus qu'à regarder dans le détail comment définir les fonctions que nous venons d'introduire.

3.2 Créer un nouveau projet

Cette étape est sans la plus facile : elle consiste en un seul appel à l'API GitLab, et ne nécessite qu'un seul paramètre, le nom du projet :

Fichier

```
def gitlab_create_project(project_name):
    return gl.projects.create({'name': project_name})
```

Après avoir exécuté cette fonction, un nouveau projet apparaît dans notre tableau de bord GitLab.

3.3 Créer les rapports de bugs

Dans la fonction `migrate_gh_project`, nous itérons sur les rapports de bugs afin de les cloner un par un. Idéalement, nous aimerions conserver cinq informations :

- le titre du rapport ;
- la description du bug ;
- la date de création du bug ;
- l'auteur du bug ;
- les commentaires.

Comme nous pouvons le voir dans le code de la fonction `gitlab_clone_github_issue`, les trois premières informations peuvent être passées à `gl_project.issues.create`. Spécifier l'auteur du bug est plus difficile :

- il a peut-être un compte sur GitHub, mais pas sur votre instance GitLab ;
- un problème de permissions se pose : un utilisateur ne peut pas créer un bug en usurpant l'identité d'un autre.

Nous choisissons ici d'inclure le nom de l'auteur dans la description (c'est l'approche implémentée par l'outil officiel de migration fourni par GitLab).

Fichier

```
def gitlab_clone_github_issue(gl_project, gh_issue):
    body = '[Created by %s] %s' % (gh_issue.user.login, gh_issue.body)
    gl_issue = gl_project.issues.create({
            'title': gh_issue.title,
            'description': body,
            'created_at': gh_issue.created_at.isoformat()
        })


    for gh_comment in gh_issue.get_comments():
        gitlab_clone_github_comment(gl_issue, gh_comment)


    return gl_issue
```

Il ne nous reste plus qu'à cloner les commentaires.

3.4 Créer les commentaires

Créer un commentaire est une opération relativement similaire à la création d'un rapport de bug. Nous rencontrons le même problème concernant les auteurs des commentaires, que nous contournons de la même façon que précédemment :

Fichier

```
def gitlab_clone_github_comment(gl_issue, gh_comment):
    body = '[Created by %s] %s' % (gh_comment.user.login, gh_comment.
body)
    gl.project_issue_notes.create({
            'body': body,
            'created_at': gh_comment.created_at.isoformat()
        }, project_id=gl_issue.project_id, issue_id=gl_issue.id)
```

Nous avons maintenant toutes les pièces du puzzle ! Si l'on exécute la fonction `migrate_gh_project` avec un nom de projet existant sur notre compte, un projet similaire apparaîtra dans GitLab, avec les bugs reproduits (presque) à l'identique.

3.5 Améliorations possibles

Bien entendu, ce code n'est qu'un simple exemple de l'utilisation combinée des API de GitHub et GitLab. Nous pourrions y apporter de nombreuses améliorations :

- seuls les bugs sont migré (même le code du dépôt est absent de notre nouveau projet), il conviendrait de **tout** copier ;
- seuls les bugs ouverts sont copiés (`get_issues` ne retourne par défaut que ceux-ci), il faudrait également copier ceux qui ont été résolus ;
- la visibilité du projet est « privée » par défaut, mais nous pourrions corriger cela en modifiant `gitlab_create_project` ;
- afin de pouvoir migrer nos projets de n'importe quel service en ligne vers n'importe quel autre, il faudrait donner à notre code une meilleure architecture.

Tout ceci est possible grâce aux deux bibliothèques que nous avons utilisées jusqu'ici.

CONCLUSION

Nous avons montré à quel point il était facile d'utiliser les API de GitHub et GitLab grâce à des bibliothèques Python, et avons réussi à construire une application certes rudimentaire, mais tout à fait fonctionnelle, qui nous a permis de nous authentifier, de lire des données et d'en créer d'autres.

De nombreuses API fonctionnent de façon similaire, en proposant diverses méthodes d'authentification (par mot de passe, avec des jetons, avec OAuth2, etc.) et en permettant à l'utilisateur de récupérer et de modifier l'état du système. La plupart d'entre elles sont également utilisables au travers de bibliothèques Python tout aussi simples d'accès que celles dont il était question dans cet article. La méthodologie employée ici (exploration de l'API avec `curl`, authentification, découverte de la bibliothèque dans l'interpréteur Python...) aura donc sans doute l'occasion d'être réutilisée. ▉

CRÉEZ UN DRIVER FUSE POUR GOOGLE DRIVE

Sylvain Peyrefitte

Google aime beaucoup le Python et le fait savoir. Avant même de vouloir compter dans ses rangs le créateur du langage, il mettait à la disponibilité des programmeurs Python une grande partie de leurs API. Et Google Drive n'échappe pas à la règle.

Derrière l'appellation *cloud* se cache souvent beaucoup de concepts, qui à mon sens n'ont rien à y faire. À l'inverse, les services proposant une gestion de vos données multimédias disponibles partout où une connexion internet est disponible, assurant par là même une sauvegarde et redondance de vos données, constituent un réel service de *cloud*. **Google Drive** est le service de stockage de données « offert » par Google dans la limite d'un stockage de 15 Gio. Pour plus d'espace, il faudra, comme la plupart des concurrents, utiliser la carte bancaire. Mais comment faire pour intégrer un partage Google Drive à Linux ? **Python** bien sûr. Nous aborderons les dessous de l'API Python, offerte par Google, avant d'en explorer le fonctionnement. Ensuite, nous verrons comment l'intégrer de façon transparente dans Linux via l'API FUSE et son *binding* Python.

1. REST

L'API Google Drive est une API de type REST (*REpresentational State Transfer*), elle respecte donc une convention permettant l'interrogation du service au travers du protocole HTTP et, dans notre cas, HTTP2. De nombreuses applications Google reposent sur cette convention.

Mais un problème subsiste quand on implémente une API respectant une telle architecture, que l'on définit souvent sous le terme de RESTFull : c'est l'aspect « sans état ». Ceci implique que chaque requête est complètement décorrélée, et donc exclut automatiquement tout moyen d'authentification et donc d'identification des ressources auxquelles un tiers peut avoir droit. Le protocole **OAuth2** répond à cette problématique.

2. OAUTH2

L'API permettant de communiquer avec Google Drive repose, comme l'intégralité des API Google, sur le protocole **OAuth**. OAuth et son successeur OAuth2 ne sont pas à proprement parler, des protocoles d'authentification, mais il faut plutôt les voir comme des protocoles de délégation d'identité.

Une application possède de nombreuses ressources qu'il peut être intéressant de rendre disponibles, du moins en partie, à d'autres applications. C'est bien cette problématique que tente de résoudre le protocole OAuth2 (car OAuth 1 n'est plus recommandé). Il fut initié par Blaine Cook, alors ingénieur chez **Twitter**. Il était responsable de l'implémentation de **OpenID**, un service de mutualisation d'*authentication* afin d'accéder avec un compte unique à plusieurs applications web. Durant cette implémentation, il s'est rendu compte qu'il n'existait aucun standard permettent de déléguer une partie des ressources d'une application à une autre. C'est pour cela qu'il a eu l'idée d'initier le protocole OAuth. Ce projet fût très vite soutenu par Google qui y voyait un moyen de développer la communauté des développeurs autour de leurs applications et ainsi inciter les utilisateurs à utiliser indirectement les ressources mises à disposition par Google.

OAuth est donc basé sur de la délégation d'autorisations. On voit donc deux notions émerger :

- une application de type fournisseur offrant un service via une API ;
- une application de type consommatrice utilisant le service de la précédente.

Concrètement, l'utilisateur de l'application de type fournisseur va définir un périmètre fonctionnel qu'il va ensuite déléguer à l'application de type consommatrice au travers d'un *token*. Ce dernier sera donc présenté à chaque appel de l'API afin de l'identifier.

3. API PYTHON

L'API Google repose donc sur un protocole ouvert, OAuth2, et le protocole HTTP2. Cette dernière est très bien documentée et nous allons décrire les étapes afin de créer une application permettant de s'interfacer avec le compte Google Drive de n'importe quel client.

3.1 Génération du token OAuth2

Dans un premier temps, il faut créer notre *token* nous permettant de nous authentifier auprès du service. Pour ce faire, il nous faut un compte Google et ensuite nous connecter au service de développeur à l'adresse suivante **https://console.developers.google.com** (voir figure 1). Ensuite, il faut créer une nouvelle application que nous allons nommer ***Test Linux Magazine***.

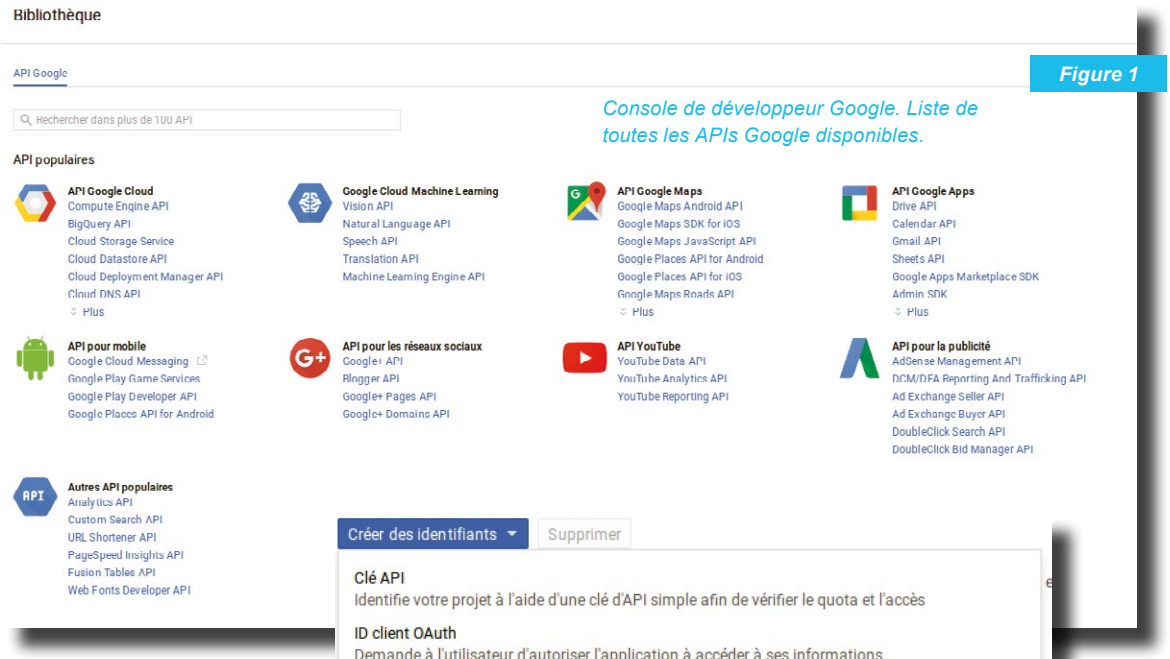


Figure 1

Console de développeur Google. Liste de toutes les APIs Google disponibles.

Figure 2

Liste des identifiants disponibles pour une application.

Il faut donc activer Google Drive. Ensuite, il faut créer notre *token* OAuth2 en sélectionnant l'onglet ***identifiant*** puis ***ID client Oauth*** (voir figure 2).

Ensuite nous allons télécharger ce *token* au format JSON pour la suite des opérations et que nous allons sauvegarder sous `oauth_token.json`.

3.2 Python

Notre code Python va donc se découper en une partie d'authentification basée sur la bibliothèque OAuth2 de Python, et enfin une partie plus spécifique utilisant un *wrapper* REST pour le Google Drive API.

Nous allons donc commencer par installer les dépendances nécessaires pour notre petite application.

Terminal

```
# pip install httplib2 google-api-python-client
```

3.2.1 Authentification

L'authentification de l'application repose entièrement sur le protocole OAuth2 :

Fichier

```
from oauth2client import client
from oauth2client import tools
from oauth2client.file import Storage

def get_credentials():
    home_dir = os.path.expanduser('~')
    credential_dir = os.path.join(home_dir, '.credentials')
    if not os.path.exists(credential_dir):
        os.makedirs(credential_dir)
    credential_path = os.path.join(credential_dir,
                                   'test_linux_magazine.json')

    store = Storage(credential_path)
    credentials = store.get()
    if not credentials or credentials.invalid:
        flow = client.flow_from_clientsecrets('oauth_token.json',
'https://www.googleapis.com/auth/drive')
        flow.user_agent = 'Test Linux Magazine'
        if flags:
            credentials = tools.run_flow(flow, store, flags)
        else:
            credentials = tools.run(flow, store)
    return credentials
```

Rien de bien extraordinaire dans ce code où toute la complexité est cachée par l'API. Ici, nous optimisons simplement les appels et autorisations que nous sauvegardons dans le répertoire caché, sous le répertoire root de l'utilisateur, `.credentials`.

Si nous exécutons cette fonction, nous verrons que notre navigateur sera automatiquement lancé sur une page d'identification Google. Après avoir rentré vos identifiants, une nouvelle page d'autorisation sera affichée, permettant donc de déléguer une partie de Google Drive à votre nouvelle application (voir figure 3).

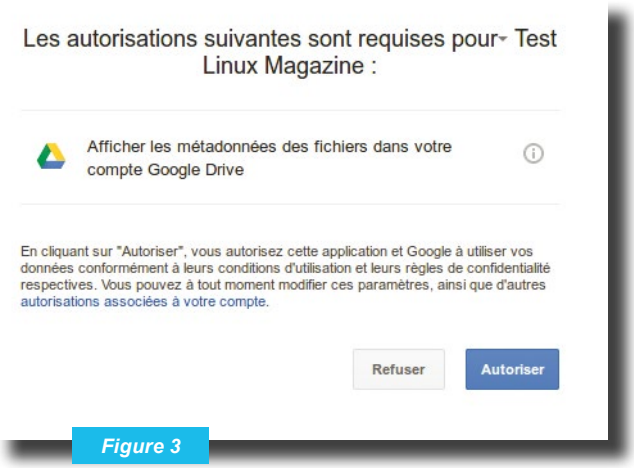


Figure 3

Fenêtre d'autorisation OAuth2 pour Google Drive.

3.2.2 REST

Nous allons donc explorer les fonctionnalités que nous offre l'API REST de Google Drive. Pour cela, Google nous propose un *wrapper* autour des requêtes et surtout des mécanismes nous permettant de manipuler simplement les résultats en Python.

Par exemple, il est simple de lister l'ensemble des fichiers disponibles pour un utilisateur :

Fichier

```
from apiclient import discovery

credentials = get_credentials()
http = credentials.authorize(httplib2.Http())
service = discovery.build('drive', 'v3', http=http)

results = service.files().list(fields="nextPageToken, files(id, name)").
execute()
items = results.get('files', [])
```

L'objet `items` contient l'ensemble des informations de vos fichiers présents dans votre Drive.

Il est bien sûr possible de télécharger vos fichiers :

Fichier

```
from apiclient import discovery, http

file_id = '0BwwA4oUTeiV1UVNwOHItT0xfa2M'
request = service.files().get_media(fileId=file_id)
fh = io.BytesIO()
downloader = http.MediaIoBaseDownload(fh, request)
done = False
while done is False:
    status, done = downloader.next_chunk()
    print("Download %d%%." % int(status.progress() * 100))
```

Il est aussi possible de créer un répertoire via l'API :

Fichier

```
file_metadata = {
  'name' : 'Images',
  'mimeType' : 'application/vnd.google-apps.folder'
}
file = service.files().create(body=file_metadata,
                                    fields='id').execute()
print('Folder ID: %s' % file.get('id'))
```

Ici nous nous sommes attardés sur des aspects de consultation, mais il est tout aussi possible d'*uploader* des fichiers. Mais bien sûr ceci requiert un *scope* différent pour l'application.

Cette API m'a donc donné une idée. Et si nous pouvions interagir avec nos données Google Drive au travers d'un explorateur de fichiers ? Nous allons donc implémenter un driver **FUSE** simple pour Google Drive écrit entièrement en Python.

4. FUSE

FUSE signifie *Filesystem in UserSpacE*. Concrètement, FUSE est une technique qui va nous permettre d'émuler un système de fichiers depuis l'espace utilisateur, et donc sans droits *root*. FUSE se décompose en une partie *driver* relayant les appels à une bibliothèque, `libfuse`, en espace utilisateur.

Il existe des *bindings* pour presque tous les langages et bien sûr Python n'échappe pas à la règle. Il existe de nombreux *bindings* et ici nous avons fait le choix de `fusepy`.

Terminal
```
# pip install fusepy
```

Nous allons ensuite reprendre le code source que nous avons testé précédemment pour initier notre « driver » en *userspace*.

Pour cela, nous allons implémenter une classe simple qui va permettre de lister nos fichiers depuis le compte Google Drive de notre client, et ensuite télécharger à la volée les fichiers demandés.

Nous allons donc initier une classe `GoogleDriveFS` qui va implémenter la méthode `init()` suivante :

Fichier
```
class GoogleDriveFS(Operations):
    def __init__(self):
        self.credentials = get_credentials()
        self.http = self.credentials.authorize(httplib2.Http())
        self.service = discovery.build('drive', 'v3', http=self.http)
        self.items = {}
        self.fh = {}
        self.next_fh = 0
```

Les *credentials* sont issus de la fonction précédemment décrite ; de plus, nous rajoutons un contexte permettant de gérer nos fichiers :

⇨ `items` pour le résultat des requêtes REST ;

⇨ `fh` qui sera tous les fichiers ouverts que nous stockerons bêtement en mémoire ;

⇨ `next_fh` est un compteur de descripteur de fichiers.

Nous allons ensuite implémenter l'interface `readdir` appelée pour lister le contenu d'un répertoire. Dans cet exemple, nous ne gérons pas les répertoires Google Drive ; tout est contenu dans la racine.

Fichier
```
    def readdir(self, path, fh):
        results = self.service.files().list(fields="nextPageToken,
files(id, name, size)").execute()
        self.items = dict([(item['name'], (item['id'], int(item.
get('size') or '0'))) for item in results.get('files', [])])
        return ['.', '..'] + list(self.items.keys())
```

Nous retrouvons ici notre requête précédente, plus une information de taille via l'attribut `size`. Nous remarquons qu'il est nécessaire d'ajouter les pseudo-répertoires `.` et `..`.

Ensuite, la méthode `getattr` est appelée à chaque fois que des méta-informations sont demandées. Ici pour nous tout est un fichier, nous rajoutons l'information de taille.

Fichier
```
    def getattr(self, path, fh=None):
        if path == '/':
            return dict(st_mode=(S_IFDIR | 0o755), st_nlink=2)
        if path[1:] not in self.items:
            raise FuseOSError(EROFS)
```

Enfin, nous allons implémenter les méthodes de lecture de fichiers. La fonction `open` est appelée pour l'ouverture d'un fichier puis la fonction `read` de façon répétée pour lire l'intégralité d'un fichier.

Fichier

```
    def open(self, path, flags):
        if path[1:] not in self.items:
            raise FuseOSError(EROFS)

        request = self.service.files().get_media(fileId=self.
items[path[1:]][0])
        fh = io.BytesIO()
        downloader = http.MediaIoBaseDownload(fh, request)
        done = False
        while done is False:
            status, done = downloader.next_chunk()

        fh_id = self.next_fh
        self.next_fh += 1
        self.fh[fh_id] = fh

        return fh_id

    def read(self, path, size, offset, fh):
        if fh not in self.fh:
            raise FuseOSError(EROFS)
        self.fh[fh].seek(offset)
        return self.fh[fh].read(size)
```

Ici nous manipulons des objets au sein de la mémoire au travers d'un `io.Bytes`.

Il suffit de lancer notre *driver* et nous pouvons interagir, de façon encore un peu simpliste, avec nos informations dans le Google Drive (voir figures 4 et 5).

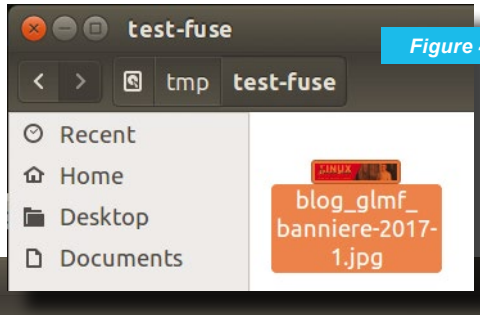


Figure 4

Intégration au sein de l'application Files sous Ubuntu.

Figure 5

Affichage d'une image présente depuis Google Drive.

CONCLUSION

100 lignes de code Python au total pour réaliser un « driver » pour Google Drive. Cela révèle la maturité de Python ainsi que son intégration dans l'écosystème Linux et réseau. Google offre une API simple pour son Drive ainsi qu'un *binding* Python adapté, associé aux contributeurs réalisant des bibliothèques telles que `fusepy` qui peuvent nous permettre d'imaginer des solutions toujours plus innovantes ! ■

CONNECT ÉVOLUE !

LISEZ CE NUMÉRO ET PLUS DE 150 AUTRES EN LIGNE !

ACTUELLEMENT SUR CONNECT :

- **CE NUMÉRO**
- **et + de 150 autres numéros de GNU/Linux Magazine**

+

- **72 numéros Hors-Séries de GNU/Linux Magazine**

TOUT CELA À PARTIR DE 199 € TTC*/AN !

* Tarif France Métropolitaine

OFFRE DÉCOUVERTE CONNECT

1 MOIS GRATUIT, RÉSERVÉE AUX PROFESSIONNELS

Appelez le 03 67 10 00 28 et donnez le code « GLMF204 » pour découvrir Connect gratuitement pendant 1 mois !

Pour tous renseignements complémentaires, contactez-nous via notre site internet : **www.ed-diamond.com**, par téléphone : **03 67 10 00 28** ou envoyez-nous un mail à **connect@ed-diamond.com** !

CRÉEZ UN BOT IRC

Jean-Michel Armand

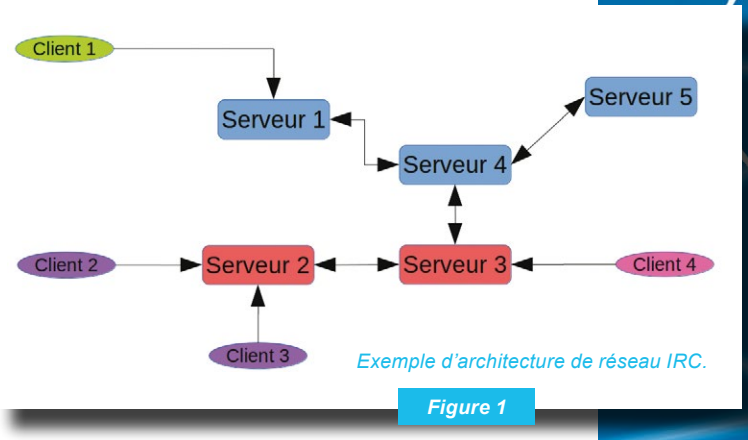
Il est possible de dialoguer avec les messageries instantanées à l'aide d'API et de créer ainsi des robots (ou bots en anglais) qui vont pouvoir réagir en fonction du type de message reçu. Dans cet article, nous nous intéresserons à IRC.

IRC reste une plateforme très utilisée, pour laquelle il est très facile de déployer un serveur, dont l'architecture est plutôt bien pensée et qui peut être utilisé pour bien plus que simplement discuter. Contrôle de *build* automatisé, vérification de bon fonctionnement de serveur, discussion entre des bots et vous-même, on peut faire plein de choses avec IRC et on peut les faire rapidement (ce qui n'est pas forcément le cas si on utilise du XMPP par exemple, mais de XMPP, nous en reparlerons plus tard).

1. IRC, PETITE PRÉSENTATION THÉORIQUE

1.1 Principe et architecture

Le protocole IRC (*Internet Relay Chat Protocol*) est un vieux protocole. La RFC s'y rapportant, la RFC 1459 a été publiée en mai 1993, dans une autre ère pour Internet. Son grand âge et ses interfaces un peu rustres font d'ailleurs que IRC est l'un des outils préférés des scénaristes de films ou séries lorsqu'ils veulent parler de pirates informatiques. Les messages qui sont envoyés entre les clients et les serveurs sont uniquement des messages textes. Il n'y pas de notions de listes d'amis, uniquement des notions d'utilisateurs et de *channel* (salon de discussion). Au niveau des statuts, il n'en existe que deux : présent ou absent. Plusieurs serveurs peuvent être connectés entre eux pour définir un maillage. Les salons de discussions sont partagés entre les serveurs. La figure 1 présente un exemple de maillage. Avec ce maillage serveur, on pourrait imaginer un salon `#linux_magazine` sur lequel se trouveraient tous les clients connectés. Un message envoyé sur ce salon passerait donc par les serveurs 1, 2, 3 et 4. Il ne passerait par contre pas par le serveur 5 qui ne gère aucun client se trouvant sur le salon `#linux_magazine`.



Exemple d'architecture de réseau IRC.

Figure 1

Dans la RFC de 1993, il n'est pas prévu de pouvoir mettre en place de connexion directe entre deux clients. Tous les messages doivent passer par un ou plusieurs serveurs. Cette possibilité de connexion client à client a toutefois été rajoutée dans la spécification CTCP (*Client to Client Protocol*) en 1994. Il est prévu par contre plusieurs types de communications client vers serveurs. Un client peut :

- envoyer un message à un serveur ;
- envoyer un message à un utilisateur, on appelle alors cela un message privé ;
- envoyer un message à tout un salon de discussion ;
- envoyer un message à tous les utilisateurs ayant un *host* correspondant à une regex ;
- envoyer un message à tous les utilisateurs du serveur.

1.2 Commandes utiles

L'un des avantages d'utiliser le protocole IRC, c'est qu'il est très simple. Il suffit de connaître quelques commandes et on peut commencer à coder. En fait, il suffit de connaître sept commandes pour pouvoir faire un bot IRC utile. Ce sont les commandes `NICK`, `USER`, `JOIN`, `PING`, `PONG`, `PRIVMSG` et `NOTICE`.

La plupart de ces commandes sont bidirectionnelles. Vous les enverrez au serveur pour lancer une action et le serveur vous les enverra pour vous informer d'une action.

La commande **NICK**, lorsque vous l'envoyez, vous permet de changer de surnom. Sa syntaxe est alors :

Fichier
```
NICK VotrNouveauNick
```

Si vous la recevez, c'est qu'un utilisateur présent sur un des salons où vous êtes a changé son surnom, sa syntaxe deviendra alors :

Fichier
```
:VieuxNick NICK NouveauNick
```

USER est une commande que vous devez lancer au tout début de votre connexion, elle sert à vous identifier auprès du serveur. C'est une identification « souple », c'est-à-dire que vous pouvez mettre ce que vous voulez, ce n'est aucunement un mécanisme d'authentification. Sa syntaxe est :

Fichier
```
USER VotreSurnom VotreHostname LeNomServeur VotreNomReel
```

Petite précision, le nom réel, vu qu'il peut contenir des espaces, doit toujours être le dernier paramètre et être préfixé par un **:** pour en définir le début.

JOIN vous sert soit à rejoindre un salon de discussion, soit à vous informer qu'un nouvel utilisateur a rejoint un salon de discussion où vous vous trouvez.

Fichier
```
JOIN #Linux_Magazine ; vous rejoignez le salon Linux_Magazine
:Mrjmad JOIN #Linux_Magazine, MrJmad rejoint le salon Linux_Magazine
```

PING et **PONG** sont deux commandes miroirs. Le serveur vous enverra des **PING** à intervalles réguliers pour vérifier que vous êtes toujours connecté. Il faudra lui répondre en lui renvoyant un **PONG**. Si vous ne le faites pas, au bout d'un certain temps le serveur vous considèrera comme déconnecté et fermera votre socket.

PRIVMSG est la commande que vous allez voir le plus souvent, c'est grâce à elle que vous recevrez et enverrez vos messages.

Fichier
```
01: PRIVMSG #Linux_Magazine :Bonjour ;
02: :MrJmad!HOST PRIVMSG #Linux_Magazine :Bonjour
03: PRIVMSG #Linux_Magazine :MrJmad: Stop les Trolls !
04: :MrJmad!HOST PRIVMSG #Linux_Magazine :BotLinuxMag: Stop les Bots !
05: PRIVMSG MrJmad : Bot toi même
06: :MrJmad!HOST PRIVMSG BotLinuxMag : espèce de singularité
```

En ligne 1 vous envoyez un message à un salon. La ligne 2 c'est **MrJmad** qui envoie un message sur un salon où vous vous trouvez. La ligne 3 vous envoyez, sur un salon, ce qu'on appelle un message direct, c'est un message lisible par tous, mais adressé précisément à quelqu'un. La ligne 4 c'est **MrJmad** qui vous envoie un message direct, sur le salon **Linux_Magazine**. La ligne 5 vous envoyez un message en privé à **MrJmad** et la ligne 6 **Mrjmad** vous répond, là aussi en privé.

NOTICE fait les mêmes choses que **PRIVMSG**, simplement en cas d'erreur le serveur ne renvoie pas de message d'erreur de manière automatique. Lorsque vous avez un programme qui va réagir de manière automatique, utiliser **NOTICE** permet de pas créer une boucle de réponses automatique.

Alors bien entendu il y a un certain nombre de commandes supplémentaires. Mais nous sommes maintenant parés pour mettre en place notre premier bot IRC.

2. PREMIÈRE VERSION, N'UTILISONS PAS DE BIBLIOTHÈQUE

Le protocole étant relativement simple, on peut tout à fait se dire qu'on va partir d'une feuille blanche et coder nous-mêmes les choses. Voici un premier exemple de ce que l'on peut faire en quelques lignes de code. Ce code exemple utilise **asyncio** de manière intense. Si vous n'êtes pas familier avec cette partie de Python, pas d'inquiétude, faites comme si les mots-clés `asyncio`, `async`, `await` et la méthode `ensure_future` n'existaient pas, vous pourrez alors comprendre le code d'exemple sans problème :

Fichier

```
01: import asyncio
02:
03: class IRCClient(asyncio.Protocol):
04:     def __init__(self, *args, **kwargs):
05:         self.lines_buffer = ''
06:         self.chan_name = "#HS_Python"
07:         self.bot_nick = "BotIRC"
08:
09:     def send_echo(self, sender, msg):
10:         nick_sender = sender.split("!")[0]
11:         message_to_send = "PRIVMSG %s :%s\r\n" % (nick_sender, msg)
12:         self.transport.write(message_to_send.encode())
13:
14:     def connection_made(self, transport):
15:         self.transport = transport
16:         user = "USER %s %s servername :%s\r\n" % ("BotIRC", "Brian", "Bot
IRC en Python")
17:         transport.write(user.encode())
18:         nick = "NICK %s\r\n" % self.bot_nick
19:         transport.write(nick.encode())
20:         join = "JOIN :%s\r\n" % self.chan_name
21:         transport.write(join.encode())
22:         send_to_channel = 'PRIVMSG %s :%s\r\n' % (self.chan_name, "Bonjour
!")
23:         transport.write(send_to_channel.encode())
24:         notice_to_channel = 'NOTICE %s :%s\r\n' % (self.chan_name, "un
message envoyé par notice")
25:         transport.write(notice_to_channel.encode())
26:         send_to_user = 'PRIVMSG %s :%s: %s\r\n' % (self.chan_name,
"MrJmad", "Ho Ho Ho ! ")
27:         transport.write(send_to_user.encode())
28:
29:     def data_received(self, data):
30:         data = self.lines_buffer + data.decode('utf-8').replace('\r', '')
31:         lines = data.split('\n')
32:         self.lines_buffer = lines.pop()
33:         for line in lines:
34:             asyncio.ensure_future(self.line_received(line))
35:
36:     async def line_received(self, line):
37:         result = await self.parse_line(line)
38:         asyncio.ensure_future(self.handle_irc_command(line, result[0],
result[1], result[2]))
39:
40:     async def parse_line(self, line):
41:         prefix = ''
42:         if line[0] == ':':
43:             prefix, line = line[1:].split(' ', 1)
44:         if ' :' in line:
45:             line, trailing = line.split(' :', 1)
46:             params = line.split()
47:             params.append(trailing)
48:         else:
49:             params = line.split()
50:         command = params.pop(0)
```

```
51:
52:         return [prefix, command, params]
53:
54:     async def handle_irc_command(self, line, prefix, command, params):
55:         if command == 'PING':
56:             message = line.replace('PING', 'PONG')
57:             self.transport.write(message.encode())
58:             return
59:         if command == "PRIVMSG" and params[0] ==self.bot_nick:
60:             self.send_echo(prefix, params[-1])
61:             return
62:         print(line)
63:
64:
65: def connection_lost(self, exc):
66:     print('server closed the connection')
67:     asyncio.get_event_loop().stop()
68:
69:
70: loop = asyncio.get_event_loop()
71: coroutine = loop.create_connection(IRCClient, "irc.freenode.net", 6667)
72: loop.run_until_complete(coroutine)
73: loop.run_forever()
74: loop.close()
```

Notre premier bot ne fait pas grand-chose : il commence par initialiser l'*event loop* asyncio ligne 70, crée la connexion et lance la boucle « pour toujours ». Une fois connecté au serveur, on lance la méthode `connection_made` (ligne 14) qui nous permet de récupérer l'objet transport qui nous permettra de parler au serveur IRC. On envoie directement quelques commandes irc pour se présenter au serveur puis on rejoint un *channel* et on envoie des messages de différentes façons, respectivement un message sur tout le chan, une notice sur tout le chan et un message direct en partant du principe qu'il y a un utilisateur ayant `MrJmad` pour surnom sur le salon `#HS_Python` (lignes 16 à 27).

Ensuite, on va simplement laisser tourner l'*event loop* qui appellera automatiquement `data_received` (ligne 29) à chaque fois que l'on recevra des données en provenance du serveur.

`data_received` va se contenter de très rapidement découper les données reçues en lignes et puis demandera un traitement « dans le futur » pour chacune des lignes (lignes 29 à 34). Le traitement est fait par la méthode `line_received` (ligne 36) qui va *parser* la ligne pour la découper en quelque chose de plus intelligible puis passer la main à la méthode `handle_irc_command` (ligne 54) qui, elle, s'occupera de vraiment traiter les lignes. Cette méthode ne traite que deux cas. Tout d'abord on vérifie si la ligne que l'on reçoit n'est pas un `PING`, si oui alors on renvoie le message sous forme de `PONG` pour être sûr de ne pas finir par se faire déconnecter. Enfin, on regarde si quelqu'un nous envoie un message et si oui on le renvoie à l'envoyeur en utilisant la méthode `send_echo` définie ligne 9. Voilà, en moins de 80 lignes, on a déjà quelque chose qui fonctionne.

3. LA BIBLIOTHÈQUE IRC

Il existe un certain nombre de bibliothèques permettant de construire des bots IRC. Nous allons dans cet article nous limiter à en étudier deux. La première sera la bibliothèque **IRC** [3]. Pourquoi ce choix ? J'ai utilisé trois critères lors de ma sélection :

- ⇨ que le code soit maintenu, avec des *commits* récents ;
- ⇨ que le code tourne en Python 3 ou soit en cours de portage ;
- ⇨ que l'utilisation n'en soit pas trop lourde, ne force pas à rentrer dans un carcan trop étroit.

La bibliothèque IRC a plusieurs points très intéressants. Tout d'abord elle vous permettra de vous connecter à plusieurs serveurs IRC en même temps. Enfin les actions CTCP sont bien supportées. C'est une bibliothèque orientée évènements. Vous devrez donc composer avec une boucle principale et définir des *callbacks*. Par contre, ce qui est intéressant c'est que l'on peut se détacher de la boucle principale de la bibliothèque et utiliser sa propre boucle d'action. En effet, on peut récupérer les sockets de communication et donc les intégrer dans un appel `select()` du programme à qui l'on veut donner un accès à IRC. Une fois cette courte présentation faite, passons aux choses sérieuses, son utilisation.

Commençons par créer le virtualenv et installer la bibliothèque :

Terminal

```
$ mkvirtualenv irc35 -p /usr/bin/python3.5
$ pip install irc
```

La documentation pour cette bibliothèque est assez courte. Le mieux pour comprendre le fonctionnement reste encore de plonger dans le code et les exemples. Quelques classes sont particulièrement intéressantes. La classe `Reactor` définie dans le module `client.py` est la classe qui va collecter les évènements irc d'une ou plusieurs connexions serveur et qui les passera ensuite aux classes qui les traiteront. On peut avec cette simple classe faire un mini bot irc :

Fichier

```
01: from irc import client
02:
03: def on_privmsg(connexion, e):
04:     print ("on_privmsg : Target %s , Source %s , Arguments %s" %
(e.target, e.source, e.arguments))
05:     connexion.privmsg(e.source.split("!")[0], e.arguments[0])
06:
07: def on_publicmsg(connexion, e):
08:     print ("on_publicmsg : Target %s , Source %s , Arguments %s"
% (e.target, e.source, e.arguments))
09:     connexion.privmsg(e.target, e.arguments[0])
10:
11: mini_client = client.Reactor()
12: server = mini_client.server()
13: server.add_global_handler("privmsg", on_privmsg)
14: server.add_global_handler("pubmsg", on_publicmsg)
15: server.connect("irc.freenode.net", 6667, "BotBrian")
16: server.join("#HS_Python")
17: server.privmsg("#HS_Python", "Hello !")
18: mini_client.process_forever()
```

Voyons ce que fait ce petit bout de code. On définit tout d'abord deux fonctions de *callback* `on_privmsg` et `on_publicmsg`. La bibliothèque fait en effet la différence entre un message privé et un message reçu à partir d'un *channel*. Les fonctions *callback* ont toutes la même signature, elles prennent une connexion et un évènement en paramètre. Un évènement (modélisé par la classe `Event` définit dans `irc/client.py`) est constitué d'un type (la liste des types étant définie dans le fichier `irc/events.py`), une source, une cible, une liste d'arguments et une liste de tags. Chaque fois qu'un serveur envoie des données à notre bot, celles-ci seront modélisées avec un évènement par la bibliothèque irc. Après avoir défini les deux fonctions de *callback*, on instancie la classe `Reactor()`. On va ensuite récupérer une connexion vers un serveur ligne 12. À ce moment-là, la connexion n'est pas encore active. Elle le deviendra bientôt (à la ligne 15) mais on peut déjà enregistrer les *callbacks* que l'on a définis en utilisant la méthode `add_global_handler`. C'est ce que l'on fait ligne 13 et 14. La connexion que l'on récupère ligne 12 est une instance de la

classe `ServerConnection` (elle aussi définie dans le fichier `irc/client.py`). C'est elle qui contient tout le code métier qui permet de décortiquer les lignes que le serveur envoie et qui s'occupe ensuite d'appeler les fonctions de *callback* que l'on a défini. On se connecte enfin au serveur irc à la ligne 15. Avec la ligne 16, on rejoint un serveur, puis à la ligne 17 on envoie un message sur le salon que l'on vient de rejoindre. Enfin, on lance la boucle principale ligne 18.

L'enregistrement de *callback* comporte quelques petites subtilités que nous allons détailler dans le code suivant :

Fichier

```
01: from irc import client
02:
03: def on_privmsg1(connexion, e):
04:     print("on_privmsg : Target %s , Source %s , Arguments %s" %
(e.target, e.source, e.arguments))
05:
06: def on_privmsg_and_public_2(connexion, e):
07:     print("Deuxieme callback !")
08:
09: def on_publicmsg(connexion, e):
10:     print("on_publicmsg : Target %s , Source %s , Arguments %s"
% (e.target, e.source, e.arguments))
11:     return "NO MORE"
12:
13: mini_client = client.Reactor()
14: server = mini_client.server()
15: server.add_global_handler("privmsg", on_privmsg1, 2)
16: server.add_global_handler("privmsg", on_privmsg_and_public_2, 10)
17: server.add_global_handler("pubmsg", on_publicmsg, 2)
18: server.add_global_handler("pubmsg", on_privmsg_and_public_2, 10)
19: server.connect("irc.freenode.net", 6667, "BrianBis")
20: server.join("#HS_Python")
21: server.privmsg("#HS_Python", "Hello Bis Bis !")
22: mini_client.process_forever()
```

Si on lance ce petit script et qu'avec un client irc on parle en privé au bot et sur le salon où il se trouve, voilà ce qui va se passer :

Terminal

```
$ python ircbot_chain_callback.py
on_privmsg : Target BrianBis , Source MrJmad!uid12663@gateway/web/irccloud.
com/x-ukfpdwjmpbqtfcci , Arguments [u'Test message priv\xe9']
Deuxieme callback !
on_publicmsg : Target #HS_Python , Source MrJmad!uid12663@gateway/web/irccloud.
com/x-ukfpdwjmpbqtfcci , Arguments [u'Message public']
```

On remarque plusieurs choses : tout d'abord dans le code on enregistre plusieurs *callbacks* pour le même évènement (ligne 15 à 18). On ajoute également un argument, une priorité à l'appel de la fonction d'enregistrement. Cette priorité permettra de trier l'ordre dans lequel seront appelées les fonctions *callback*. Les fonctions avec la plus petite priorité seront appelées en premier.

Enfin, si vous observez la sortie console, vous remarquez qu'alors que pour un message en privé les deux *callbacks* sont bien appelés, dans le cas d'un message sur un chan, il n'y a que le premier *callback* qui est exécuté. Il est en effet possible d'interrompre la liste d'appel des fonctions *callback*. Lorqu'un *callback* veut stopper les exécutions des *callbacks* suivants, il suffit qu'il renvoie la chaîne de caractères `NO MORE`, ce qui est fait à la ligne 11.

Avant de changer de bibliothèque, mettons le tout sous forme objet en utilisant la classe-cadre fournie par la bibliothèque pour faire des bots, à savoir `SimpleIRCClient` définie dans le fichier `irc/client.py` :

Fichier

```
01: import irc.client
02: import sys
03:
04: class HsPythonBot(irc.client.SimpleIRCClient):
05:     def __init__(self, target):
06:         irc.client.SimpleIRCClient.__init__(self)
07:         self.target = target
08:
09:     def on_welcome(self, connection, event):
10:         if irc.client.is_channel(self.target):
11:             connection.join(self.target)
12:         else:
13:             self.send_it()
14:
15:     def on_join(self, connection, event):
16:         connection.privmsg(self.target, 'Hello !')
17:
18:     def on_disconnect(self, connection, event):
19:         sys.exit(0)
20:
21:     def on_privmsg(self, connexion, event):
22:         print ("on_privmsg : Target %s , Source %s , Arguments
%s" % (event.target, event.source, event.arguments))
23:         connexion.privmsg(event.source.split("!")[0], event.
arguments[0])
24:
25:     def on_pubmsg(self, connexion, event):
26:
27:         print ("on_publicmsg : Target %s , Source %s , Arguments
%s" % (event.target, event.source, event.arguments))
28:         connexion.privmsg(event.target, event.arguments[0])
29:
30: if __name__ == "__main__":
31:     c = HsPythonBot("#HS_Python")
32:     c.connect("irc.freenode.net", 6667, "BrianLeBot")
33:     c.start()
```

En fait, on fait à peu près la même chose que précédemment. Mais c'est tout de même plus lisible et plus élégant de le faire sous forme d'objets. Les *callbacks* ne sont plus des fonctions, ce sont devenus des méthodes et il y a un peu de magie pour nous faciliter la tâche. En effet, toute méthode commençant par la chaine `on_` puis d'un nom d'évènement IRC reconnu par la bibliothèque sera alors automatiquement considéré comme une *callback* pour ledit évènement. C'est ce que l'on utilise avec les *callbacks* `on_privmsg()` à la ligne 21 et `on_pubmsg()` à la ligne 15. Rien de bien difficile dans le reste du code. La boucle principale se lance juste avec la méthode `start()` de notre classe de bot, à la ligne 33 alors qu'à la ligne précédente (32), on se connecte au serveur irc.

4. LA BIBLIOTHÈQUE IRC3

Les concepteurs de bibliothèques IRC ont vraiment une imagination débordante pour trouver des noms à leurs bibliothèques. Avant d'aller nous amuser avec d'autres réseaux de messagerie instantanée, nous allons étudier une dernière bibliothèque, la bibliothèque

IRC3 qui fonctionne avec un principe intéressant de *plugins* de bot. On définit un ou plusieurs *plugins* en Python et ensuite on construit un bot en lui ajoutant les différents *plugins*.

Nous commençons par créer le virtualenv et installer la bibliothèque en faisant un tout simple :

Terminal
```
$ pip install irc3
```

Nous sommes parés pour utiliser IRC3. Et là, nous n'allons pas coder ! Nous allons utiliser un système de génération de fichiers de configuration. Voici les choses à faire :

Terminal
```
$ mkdir bot
$ cd bot
$ python -m irc3.template hspythonbot
```

Si on fait maintenant un `ls` dans le répertoire `bot`, on va trouver deux fichiers :

Terminal
```
$ ls
.  ..  config.ini  hspythonbot_plugin.py
```

Il s'agit d'un squelette de notre bot et d'un fichier de configuration. Le fichier de configuration va permettre de configurer le serveur sur lequel on se connecte, les *channels* en « auto-join » ainsi que la liste des *plugins* que l'on veut activer. Dans le mien, il y a par exemple :

Fichier
```
includes =
    irc3.plugins.command
#    irc3.plugins.uptime
#    irc3.plugins.ctcp
    hspythonbot_plugin
```

SI je prends le fichier du bot et que je le modifie un peu, j'obtiens le code suivant :

Fichier
```
01: from irc3.plugins.command import command
02: import irc3
03:
04:
05: @irc3.plugin
06: class Plugin:
07:
08:     def __init__(self, bot):
09:         self.bot = bot
10:
11:     @irc3.event(irc3.rfc.JOIN)
12:     def say_hi(self, mask, channel, **kw):
13:         """Say hi when someone join a channel"""
14:         if mask.nick != self.bot.nick:
15:             self.bot.privmsg(channel, 'Hi %s!' % mask.nick)
16:         else:
17:             self.bot.privmsg(channel, 'Hi!')
18:
19:     @irc3.event(irc3.rfc.MY_PRIVMSG)
20:     def on_message(self, mask=None, event=None, target=None,
data=None, **kw):
21:         self.bot.privmsg('#HS_Python', 'Yo, je suis Brian !')
22:
```

```
23:     @command(permission='view')
24:     def echo(self, mask, target, args):
25:         """Echo
26:
27:             %%echo <message>...
28:          """
29:         yield ' '.join(args['<message>'])
```

On définit trois *callbacks*. Le premier `say_hi` ligne 12 va simplement dire bonjour quand le bot arrive et dire bonjour aux nouveaux venus. Le second `on_message` , ligne 20 va faire que le Bot va répondre quelque chose si on parle sur un *channel* en le nommant. Enfin le dernier `echo` ligne 24 est un *callback* de commande. Si quelqu'un sur le chan tape une phrase commençant par `!echo`, le bot va alors la répéter. L'utilisation du caractère `!` est définie dans le fichier de configuration `config.ini` et les règles permettant de définir qui a le droit d'activer les commandes aussi. Ici on voit que toutes les personnes ayant la permission « view » peuvent lancer la commande `echo`. Bien entendu, irc3 permet de mettre en place des bots sans utiliser de configuration sous forme de fichier ini. La configuration est alors mise en place par un dictionnaire de la manière suivante :

Fichier

```
01:     config = {
02:         'loop': loop,
03:         'nick': 'Brian',
04:         'channel': '#HsPython',
05:         'host': 'irc.freenode.net',
06:         'debug': True,
07:         'verbose': True,
08:         'raw': True,
09:         'includes': ['hspythonbot_plugin'],
10:     }
11: brian_bot = irc3.IrcBot.from_config(config)
```

Tout le sel de l'utilisation d'irc3 est au final de définir des morceaux de bot proposant différentes fonctionnalités, un peu comme des *mixins* de bot et de les assembler ensuite pour faire le bot que l'on veut en fonction de la situation. C'est une approche ma foi assez intéressante et je trouvais qu'elle méritait d'être présentée ici.

CONCLUSION

J'espère que ces quelques pages vous auront donné envie de faire rejoindre IRC à vos programmes Python. IRC de par sa simplicité et sa facilité d'accès est en plus la plateforme parfaite pour s'amuser à détourner son but premier de lieu de discussion pour l'utiliser afin de contrôler ou monitorer vos processus. Après tout au final, les serveurs IRC ce ne sont que des tuyaux pour faire passer de l'information. Alors, pourquoi pas imaginer de les utiliser de cette manière-là ? Pourquoi ne pas faire un HTML over IRC ? Et puis, je pense qu'on est loin de voir arriver la mort des serveurs IRC, même si elle est annoncée assez régulièrement. Après tout, Slack n'est-il pas bâti sur des serveurs IRC ? ▮

RÉFÉRENCES

[1] La RFC IRC : **http://www.irchelp.org/irchelp/rfc/**

[2] Page Pypi de la bibliothèque irc : **https://pypi.python.org/pypi/irc**

[3] La bibliothèque IRC3 : **https://github.com/gawel/irc3**

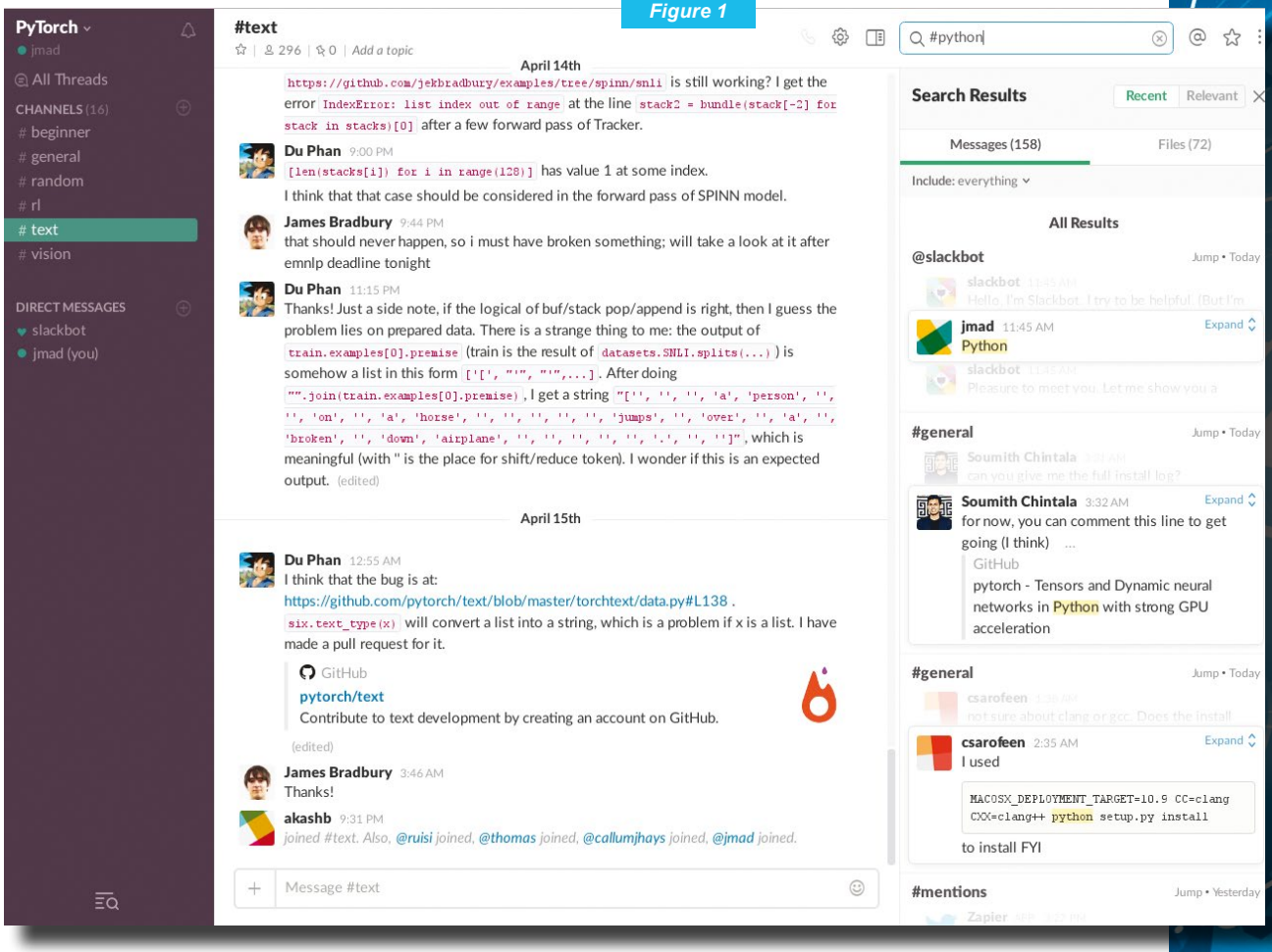
CRÉEZ UN ROBOT SLACK

Jean-Michel Armand

Pour discuter sur Internet, il existe de nombreuses solutions. Slack fait partie de celles qui ont le vent en poupe et nous allons voir dans cet article comment utiliser son API.

Slack [1] est donc une nouvelle plateforme de discussion en ligne. Son concept est simple, vous créez une équipe où vous allez inviter des gens. Suivant les paramètres que vous avez définis pour votre équipe, d'autres personnes pourront venir se créer des comptes sur votre *team* Slack et discuter avec les autres utilisateurs. Dans une instance Slack, il peut y avoir des *channels*, plein de *channels* qui seront publics ou privés. Il pourra aussi y avoir des discussions entre deux ou plusieurs utilisateurs. Une discussion entre plusieurs utilisateurs ressemble à un *channel* privé sauf qu'elle n'a pas de nom. Depuis peu, il existe une notion de « thread » qui permet de démarrer une conversation chaînée à partir de n'importe quelle phrase d'un *channel*. Bien qu'un peu confuse au départ cette notion de *thread* permet de bien « ranger » les discussions et de s'y retrouver bien mieux qu'avant. On peut également partager des fichiers, des posts ou des *snipets*.

Figure 1



Page d'accueil (oui, il y a beaucoup de channels en anglais...).

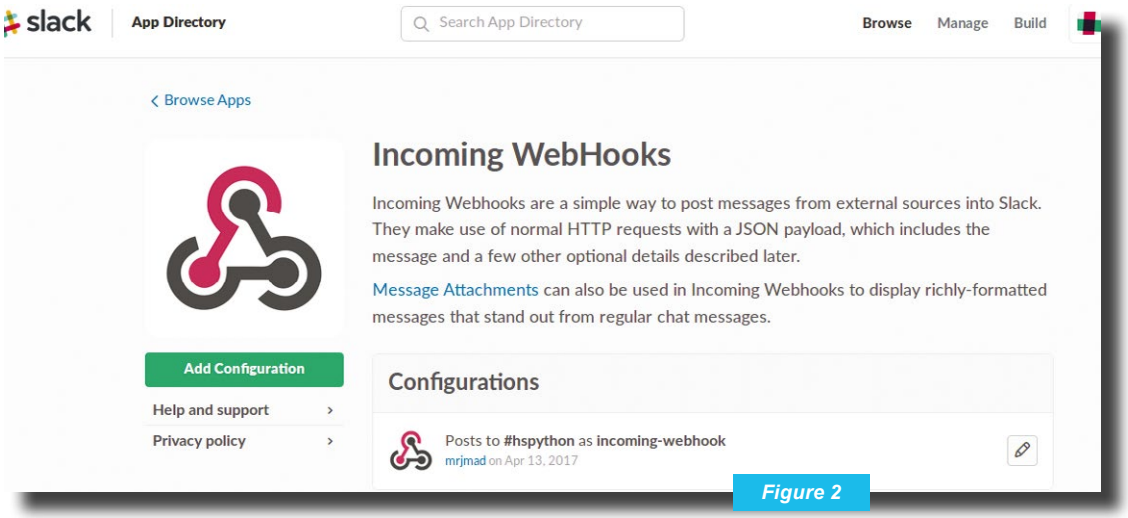
Voilà en quelques mots ce qu'est Slack. Mais ce qui à mon avis fait que Slack est autant utilisé, ce sont ses possibilités d'intégration avec des services tiers, possibilités que justement nous allons tester dans cet article.

Il y a plusieurs possibilités pour interagir avec Slack et pour commencer nous allons voir comment utiliser les *incoming webhooks*. Le principe en est simple, Slack vous définit une URL qui va vous permettre de POSTER des données qui seront intégrées comme des phrases dans Slack.

1. DÉCOUVERTE DE SLACK

1.1 Incoming Webhooks

Pour configurer les *incomings webhooks*, il faut que vous alliez dans la partie intégration de la configuration [2] de votre instance Slack. Le plus simple à partir de là est d'utiliser la barre de recherche et d'y taper « incoming webhooks ». Vous atteindrez alors la page de configuration des *webhooks* (page que vous pouvez voir sur la figure 2).



Page de configuration des webhooks.

Il vous suffit alors de cliquer sur le bouton ***Add configuration*** et de choisir le *channel* sur lequel vous allez vouloir agir de l'extérieur de Slack. Et voilà, vous allez avoir une belle URL qui va vous servir de point d'entrée. Reprenons **Request** que nous avons découvert il y a de cela quelques articles et faisons quelques tests. Par convention et pour rendre le code plus lisible, dans tous les exemples que je vais donner, je partirai du principe que l'url de *callback* Slack est stockée dans une constante `URL` dans votre code.

Fichier

```
01: paylod = {'text' : 'Bonjour, premier message du bot'}
02: requests.post(URL, json=payload)
03:
04: payload = {'text' : 'Bien entendu, on peut ajouter des *liens*
<https://boutique.ed-diamond.com|Editions Diamond>'}
05: requests.post(URL, json=payload)
06:
07: payload = {'text' : 'Bonjour, premier message du bot, on peut
ajouter des liens email <mailto:grandchef@ed-diamong.com|Le Grand
Chef>'}
08: requests.post(URL, json=payload)
09:
10: payload = {'text' : 'Aller sur le chan <#C4YFZE3JP>'}
11: requests.post(URL, json=payload)
12:
```

```
13: payload = {'text' : "Configurons la façon d'afficher qui émet le
texte", 'username':'Le bot python', icon_emoji: ':robot_face:'}
14: requests.post(URL, json=payload)
15:
16:
17: payload = {'text' : "On peut écrire sur un autre channel que
celui prévu", 'channel': 'javabien'}
18: requests.post(URL, json=payload)
```

Premier test, on se contente d'envoyer un simple message. On utilise, comme on l'a vu plus tôt les facilités proposées par request : on lui passe en argument notre dictionnaire et requests le convertira tout seul en json.

Deuxième exemple à la ligne 4, on ajoute un lien. Slack utilise les caractères `< >` comme indiquant une chaîne qu'il faut interpréter. Ici on lui donne un lien avec ensuite, simplement séparé par un `|`, le label que l'on veut avoir pour ce lien. Vous aurez remarqué, on utilise la syntaxe `* MOT *` pour mettre un mot en gras. Vous allez en effet pouvoir ajouter un peu de style dans vos messages, avec donc les `*` pour mettre en gras, les `-` pour barrer des mots, le `` ` `` pour afficher du code, etc. Slack vous met à disposition une console de test de rédaction de message qui pourra vous aider à construire ceux-ci [3].

Ligne 7, on fait la même chose, mais avec un lient `mailto :`. Ligne 10, un simple petit exemple d'utilisation des entités gérées par Slack. Ici on parle d'un channel, mais on aurait pu faire la même chose avec un utilisateur.

Ligne 13, on configure non plus seulement le message que l'on envoie, mais aussi qui l'envoie. On peut en effet changer le *username* qui va être utilisé pour afficher le message, mais aussi l'icône qui sera attachée à ce *username*. Au niveau des icônes disponibles, vous allez pouvoir choisir parmi toutes les émoticônes de l'application, y compris celles que vous avez peut-être rajoutées.

Avant d'aller plus loin et d'utiliser la bibliothèque client Slack, une dernière chose. Si vous utilisez Slack et que vous avez déjà configuré des intégrations, par exemple celles de GitHub, vous avez dû remarquer que les messages envoyés par GitHub ne sont pas de simples messages textes classiques. Il faut pour cela utiliser les attachements de Slack. Chaque message peut en comporter jusqu'à cent.

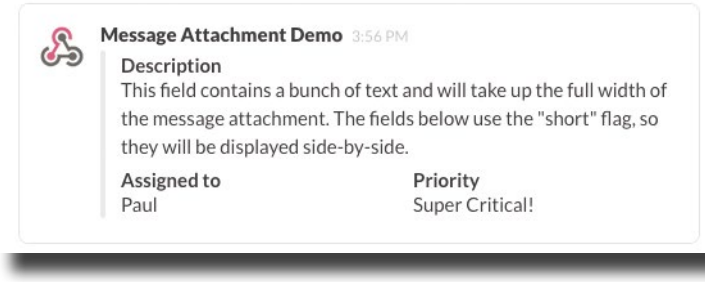


Figure 3

Exemple d'attachement de Slack.

Voici un exemple de code pour configurer les attachements :

Fichier

```
01: {
02:     "attachments": [
03:         {
04:             "fallback": "Required plain-text summary of the attachment.",
```

```
05:             "color": "#36a64f",
06:             "pretext": "Optional text that appears above the
attachment block",
07:             "author_name": "Bobby Tables",
08:             "author_link": "http://flickr.com/bobby/",
09:             "author_icon": "http://flickr.com/icons/bobby.jpg",
10:             "title": "Slack API Documentation",
11:             "title_link": "https://api.slack.com/",
12:             "text": "Optional text that appears within the
attachment",
13:             "fields": [
14:                 {
15:                     "title": "Priority",
16:                     "value": "High",
17:                     "short": false
18:                 }
19:             ],
20:             "image_url": "http://my-website.com/path/to/image.jpg",
21:             "thumb_url": "http://example.com/path/to/thumb.png",
22:             "footer": "Slack API",
23:             "footer_icon": "https://platform.slack-edge.com/img/
default_application_icon.png",
24:             "ts": 123456789
25:         }
26:     ]
27: }
```

Voyons les différentes clés de configuration :

- la valeur `fallback` permet de configurer l'affichage pour les clients Slack non natifs (il existe en effet par exemple des *bridges* pour se connecter sur Slack avec un client **irc**) ;
- `color` permet de définir la couleur qui sera utilisée pour afficher un liseré sur le bord gauche du message ;
- `pretext` permet d'afficher un texte avant l'affichage complet de l'attachement ;
- ensuite viennent plusieurs informations concernant l'auteur du message ;
- `title` permet de décider ce qui sera affiché comme titre de l'attachement (en mode gras et une police plus grande que pour le reste du message) ;
- `title_link` est un paramètre optionnel qui rend le titre cliquable ;
- `test` définit le texte principal de l'attachement ;
- `fields` définit un ensemble d'informations sous forme de liste de dictionnaires qui vont s'afficher après le texte. Chaque champ est défini par trois valeurs, le titre du champ, la valeur à afficher et un booléen pour indiquer si la valeur du champ est suffisamment « petite » pour être affichée à côté d'une autre valeur ;
- `image_url` est un paramètre facultatif qui permet d'afficher une image dans l'attachement ;
- `thumb_url` est elle, une petite image qui sera affichée à droite de l'attachement ;
- des variables de configuration pour mettre en place un pied d'attachement si nécessaire ;
- `ts`, un *integer* correspondant à une valeur en « Epoch Time ». Si `ts` est fourni, cela affichera dans le pied de page, une donnée date supplémentaire.

Jusqu'à présent, nous avons simplement vu comment envoyer des messages sur Slack en partant du principe que tout se passait bien. Mais il peut arriver que des problèmes arrivent. Dans ce cas-là, vous pouvez avoir à gérer des erreurs.

Vous allez pouvoir recevoir sept erreurs différentes :

- `invalid_payload` indiquera que vous avez envoyé des données invalides à Slack. Typiquement si vous voulez essayer, il vous suffit d'utiliser l'argument `data` dans votre post à la place de `json` ;
- `user_not_found` et `channel_not_found` que vous recevrez si vous essayez de communiquer avec un *channel* ou un utilisateur qui n'existe pas ;
- `channel_is_archived` qui indique que le *channel* sur lequel vous voulez parler est archivé, et que donc il est possible d'ajouter des messages dans ce *channel* ;
- `action_prohibited`, indique comme son nom le définit clairement, que vous avez essayé de faire quelque chose qui vous est interdit ;
- `posting_to_general_channel_denied` est une erreur très spécifique. Elle indique que votre code essaie de poster un message sur le *channel* **#general** alors que le *channel* est restreint ou que vous n'avez pas les droits de poster sur ce *channel*. Cette erreur sera accompagnée d'un statut `403` ;
- `too_many_attachments` indique que vous avez été trop gourmand et que vous avez voulu configurer plus de cent attachements à un de vos messages.

Nous en avons fini avec cette première partie de découverte de Slack. Avec ce que nous venons de voir, vous êtes déjà tout à fait capable de mettre en place une connexion entre l'un de vos services web et Slack. Mais ce n'est que le début. Nous allons maintenant creuser un peu plus profond et découvrir la bibliothèque Python qui permet d'interagir avec Slack sans avoir à forger ses propres requêtes.

2. LA BIBLIOTHÈQUE PYTHON-SLACKCLIENT

2.1 Installation et premières utilisations de l'API

Il existe une bibliothèque officielle pour communiquer avec une instance Slack, directement en Python. Il s'agit de **python-slackclient [4]**. Pour pouvoir l'utiliser, il va falloir faire deux choses. Premièrement l'installer et ensuite générer un *token* d'authentification.

Pour installer python-slackclient, rien de plus simple. Après avoir créé un virtualenv spécialement dédié à slack, il vous suffira de taper :

Terminal

```
$ pip install slackclient
```

Concernant le *token*, on va commencer par générer simplement un *token* de test. Pour cela, vous devez aller sur la page des « legacy token » **[5]** et demander la génération d'un token. Bien entendu, cette génération de *token* n'est là que pour vous aider à construire votre app. Une fois cela fait, il faudra mettre en place un système utilisant **Oauth2**, mais pour développer, un *token* de test suffira.

Comme pour la partie précédente, dans les morceaux de code que je vous proposerais, je prendrais comme convention le fait qu'il existe une variable `slack_token` qui contient votre *token* d'authentification.

Commençons par tester si tout se passe bien.

Fichier

```
from slackclient import SlackClient
slack_client = SlackClient(slack_token)
slack_client.api_call("api.test")

# Retour : {'args': {'token': 'VOTRE_TOKEN'}, 'ok': True}
```

Si tout se passe bien, vous devriez avoir comme retour un dictionnaire contenant votre token et une clé '`ok`' ayant pour valeur `True`.

Si tout fonctionne, nous allons pouvoir demander la liste des *channels* existants sur notre instance :

Fichier

```
channels_call = slack_client.api_call("channels.list")
if channels_call.get('ok'):
    print(channels_call['channels'])
```

Cela va vous renvoyer une liste de dictionnaires dont voici un exemple :

Fichier

```
[{'is_org_shared': False, 'topic': {'creator': '', 'last_set': 0,
'value': ''}, 'is_channel': True, 'purpose': {'creator': '', 'last_
set': 0, 'value': ''}, 'members': ['U4XTFFP97'], 'created': 1492040473,
'creator': 'U4XTFFP97', 'name_normalized': 'djangotoutbeau', 'id':
'C4YFZE3JP', 'is_member': True, 'is_general': False, 'is_archived':
False, 'num_members': 1, 'name': 'djangotoutbeau', 'previous_names':
[], 'is_shared': False}]
```

On voit que l'on a toutes les informations nécessaires : le nom du *channel*, son identifiant, s'il est archivé ou pas, le nombre d'utilisateurs présents, etc.

Par défaut, la liste inclura les *channels* archivés. Si vous ne voulez pas qu'ils apparaissent, il faudra ajouter l'argument `exclude_archived=1`.

Maintenant que nous avons l'identifiant d'un *channel*, nous pouvons demander à avoir les informations détaillées qui contiennent par exemple aussi la dernière chose dite sur le *channel*.

Fichier

```
>>> channel_id='C4YFZE3JP'
>>> channel_info = slack_client.api_call("channels.info", channel=channel_id)
>>> print(channel_info['channel'])

{'is_org_shared': False, 'latest': {'ts': '1492472835.107909', 'user':
'U4XTFFP97', 'text': 'Django 1.11 est sorti , youhou !', 'type': 'message'},
'last_read': '1492472835.107909', 'topic': {'creator': '', 'last_set': 0,
'value': ''}, 'is_channel': True, 'purpose': {'creator': '', 'last_set': 0,
'value': ''}, 'unread_count_display': 0, 'members': ['U4XTFFP97'], 'created':
1492040473, 'creator': 'U4XTFFP97', 'name_normalized': 'djangotoutbeau', 'id':
'C4YFZE3JP', 'is_member': True, 'unread_count': 0, 'is_general': False, 'is_
archived': False, 'name': 'djangotoutbeau', 'previous_names': [], 'is_shared':
False}
```

On voit ici que le dictionnaire `latest` nous permet d'afficher la dernière phrase dite sur le *channel*.

On peut aussi parler sur le *channel*. Reprenons un des exemples de la première partie pour le reproduire avec python_slacklient :

Fichier

```
01: slack_client.api_call(
02:         "chat.postMessage",
03:         channel=channel_id,
04:         text="Configurons la façon d'afficher qui emet le texte",
05:         username='Le bot python',
06:         icon_emoji=':robot_face:'
07:     )
```

On le voit, c'est très similaire à la méthode un peu plus « rustre » utilisant directement l'API. La seule différence notable est le fait que l'on définit le type d'action que l'on veut faire en premier argument (ligne 2).

Avant de voir les autres actions que vous pouvez faire avec cette méthode `api_call`, une dernière information concernant le fait de poster un message. Dans Slack, il existe donc depuis peu les *threads* qui permettent de chaîner des messages pour avoir plusieurs discussions en parallèle, dans le même *channel*, sans que cela soit trop compliqué de suivre (et en ayant une zone d'affichage *thread* par *thread* spéciale). Pour que votre bot puisse répondre à un *thread*, il suffit d'ajouter l'argument `thread_ts` à votre appel. Si vous voulez démarrer un *thread*, il faudra mettre dans `thread_ts` le `ts` du message que vous voulez utiliser comme racine de votre *thread*. Si votre bot veut répondre à un *thread* existant, il faudra qu'il indique dans `thread_ts`, le `ts` du message contenu dans le *thread* et auquel vous répondez.

2.2 Autres actions possibles

On peut, heureusement, faire plus que d'envoyer des messages avec l'api. Nous allons voir, exemple après exemple, ce que nous pouvons faire.

On peut supprimer un message avec l'action `"chat.delete"` :

Fichier

```
01: slack_client.api_call(
02:   "chat.delete",
03:   channel=channel_id,
04:   ts="1492472835.107909"
05: )
06: # {'channel': 'C4YFZE3JP', 'ts': '1492472835.107909', 'ok': True}
```

Ligne 2, on définit l'action de suppression, puis on donne l'identifiant du channel et enfin le timestamp du message que l'on veut supprimer. Si tout se passe bien, vous recevrez un dictionnaire récapitulant les informations de suppression.

On peut ajouter un émoji à un message. Visuellement, une liste de tous les émojis ajoutés sur un message se construit en dessous du message en question. Voici le code pour ajouter un tel émoji :

Fichier

```
01: slack_client.api_call(
02:   "reactions.add",
03:   channel=channel_id,
04:   name="grinning",
05:   timestamp="1492473150.180856"
06: )
07: {'ok': True}
```

On doit donc définir deux informations, dont le nom de l'émoji en question. Attention, c'est bien le nom et non le code de l'émoji. Dans l'exemple, j'ai donc mis `grinning` et non pas `:grinning:`. Comme à chaque fois que l'on agit sur un message déjà existant, il faut ensuite définir le *timestamp* du message.

Dans le cas où vous mettez un nom d'émoji invalide (par exemple si vous avez mis `:grinning:` à la place de `grinning`, vous allez recevoir le dictionnaire suivant :

Fichier
```
{'error': 'invalid_name', 'ok': False}
```

De la même façon que l'on peut ajouter un émoji, on peut retirer un émoji. La seule chose qui change est alors l'action que l'on utilise :

Fichier
```
slack_client.api_call(
  "reactions.remove",
  channel=channel_id,
  name=":grinning:",
  timestamp="1492473150.180856"
)
```

Il est également possible de rejoindre et de quitter un *channel*. Cela se fait avec les commandes `channels.join` et `channels.leave`. En voici les exemples :

Fichier
```
slack_client.api_call(
  "channels.join",
  channel="C4XSMPDTJ"
)


slack_client.api_call(
  "channels.leave",
  channel="C4XSMPDTJ"
)
```

L'une des fonctionnalités majeures de Slack est de tout bêtement pouvoir éditer ses messages. Oui cela peut paraître trivial, mais ce n'est par exemple pas possible sur un serveur IRC classique. L'édition de messages est possible avec la commande `chat.update` :

Fichier
```
01: slack_client.api_call(
02:     "chat.update",
03:     channel=channel_id,
04:     text="on peut modifier ses messages !",
05:     timestamp="1492473150.180856"
06: )
```

Comme pour les autres commandes modifiant un message, il faut indiquer le *timestamp* permettant d'identifier le message. Si vous vous trompez dans un des paramètres (ou que vous essayez de modifier un message que vous avez déjà supprimé, vous aurez le message d'erreur suivant :

Fichier
```
{'error': 'message_not_found', 'ok': False}
```

Enfin, il est possible d'obtenir la liste des utilisateurs inscrits sur l'équipe Slack avec l'action `users.list` :

Fichier

```
>>> slack_client.api_call("users.list")
{'members':  {'status': None, 'updated': 0, 'is_owner': False, 'tz': None,
'is_restricted': False, 'is_primary_owner': False, 'is_admin': False,
'is_ultra_restricted': False, 'team_id': 'T4YHY9UH0', 'profile': {'image_24':
'https://a.slack-edge.com/0180/img/slackbot_24.png', 'image_48': 'https://a.
slack-edge.com/2fac/plugins/slackbot/assets/service_48.png', 'avatar_hash':
'sv1444671949', 'last_name': '', 'real_name': 'slackbot', 'always_active':
True, 'fields': None, 'image_32': 'https://a.slack-edge.com/2fac/plugins/
slackbot/assets/service_32.png', 'image_72': 'https://a.slack-edge.com/0180/
img/slackbot_72.png', 'first_name': 'slackbot', 'real_name_normalized':
'slackbot', 'image_512': 'https://a.slack-edge.com/1801/img/slackbot_512.png',
'image_192': 'https://a.slack-edge.com/66f9/img/slackbot_192.png'}, 'deleted':
False, 'tz_label': 'Pacific Daylight Time', 'real_name': 'slackbot', 'id':
'USLACKBOT', 'color': '757575', 'tz_offset': -25200, 'name': 'slackbot',
'is_bot': False} , .....], 'ok': True, 'cache_ts': 1492475746}
```

Les informations récupérées sur les utilisateurs sont assez complètes, permettant de récupérer les informations personnelles, les différentes images ainsi que les informations se rapportant aux types de comptes. Il est également possible de demander des informations sur un utilisateur bien particulier en utilisant `users.info`. Dans ce cas-là, il faut donner l'identifiant de l'utilisateur dont l'on veut les informations.

Fichier

```
slack_client.api_call(
  "users.info",
  user='U4XTFFP97'
)
```

Cette action est d'autant plus intéressante que, par exemple, la liste des membres donnés dans les informations détaillées d'un *channel* n'est qu'une liste d'identifiants d'utilisateurs et qu'il faut donc utiliser `users.info` pour avoir plus d'informations sur les utilisateurs.

Je l'ai dit en introduction, il est possible de créer des « directs messages » entre deux ou plusieurs utilisateurs. On peut également le faire avec l'api en utilisant les commandes `im.open` et `im.close`. La commande d'ouverture de message privé vous retournera un identifiant qui vous permettra ensuite, en utilisant `chat.postmessage` d'envoyer des messages dans le message direct :

Fichier

```
slack_client.api_call(
  "im.open",
  user='U4XTFFP97'
)
{'no_op': True, 'already_open': True, 'ok': True, 'channel': {'id':
'D4YFZ55D1'}}
```

Lorsque vous voulez fermer votre discussion privée, rien de plus simple :

Fichier

```
slack_client.api_call(
  "im.close",
  channel='D4YFZ55D1'
)
```

Ce qui fait la puissance des messages privés de Slack, c'est la possibilité d'être à plus que deux dans un message privé. C'est ce que Slack appelle les *multiparty direct messages*. On va retrouver les commandes similaires à celle des messages directs, mais au lieu d'être

préfixées par `im`, les commandes seront préfixées par `mpim`. Pour le reste, les commandes `open` et `close` sont aussi disponibles, par contre `open` ne va plus attendre un seul identifiant d'utilisateur, mais une liste d'utilisateurs séparés par des virgules.

Fichier

```
slack_client.api_call(
  "mpim.open",
  user='U4XTFFP97,U3456789012'
)
```

Une information importante : s'il n'existe pas déjà un message pluri-parties avec exactement les participants listés, un nouveau message pluri-parties sera créé. Si par contre un message direct pluri-parties existe déjà avec les participants listés, c'est ce message-là qui sera renvoyé.

2.3 Et pour recevoir des messages ?

Nous n'avons pas du tout parlé de comment recevoir des messages. C'est parce que dans ce mode bien précis, l'api ne permet pas de recevoir des messages. Pour pouvoir recevoir des messages, il vous faudra configurer un *outgoing webhook*. C'est-à-dire qu'il faudra que vous mettiez en place un mini serveur http dans lequel vous définirez une url acceptant des requêtes `POST`. Une fois que votre serveur est prêt, il faudra donc aller configurer un *webhook* dans la partie configurations de Slack. Une fois cela fait, à chaque fois qu'un événement aura lieu, Slack enverra un POST sur votre url. Cette requête post comportera plusieurs données :

- un *token* (dans l'attribut `token`) qui vous permettra de vérifier que c'est bien Slack qui vous envoie ce POST ;
- `channel_name` qui contiendra le nom du *channel* sur lequel l'événement s'est passé ;
- `user_name` qui vous dira quel utilisateur a généré l'événement ;
- `text` qui contiendra le texte du message ou de l'événement en question.

Il est vrai que cette méthode de réception de message n'est pas la plus souple à mettre en place qui soit. Ce que l'on voudrait ici c'est quelque chose qui ressemble aux autres plate-formes de messagerie instantanées et qui permettrait à une app d'envoyer et de recevoir les informations par le même canal. C'est ici qu'apparaît la partie *Real Time Messaging* de Slack que nous allons voir pour finir cet article.

3. REAL TIME MESSAGING

Dans ce mode de communication avec Slack, on lance une boucle événementielle qui nous permet de recevoir les événements qui arrivent sur un *channel* et d'y réagir.

Fichier

```
01: if slack_client.rtm_connect():
02:     slack_client.rtm_send_message(channel_id, "message test RTM")
03:     while True:
04:         msg = slack_client.rtm_read()
05:         print(msg)
06:         time.sleep(1)
07: else:
08:     print("Erreur de connexion")
```

Comme on le voit, les choses sont assez simples. Ligne 1, on essaie de se connecter. Si cela marche, on envoie un message sur un *channel* en ligne 2 puis on part sur une boucle infinie dans laquelle on récupère les différents événements. Voici un exemple des événements que l'on va pouvoir recevoir :

Fichier

```
[{'type': 'hello'}]
[{'channel': 'C4YFZE3JP', 'user': 'U4XTFFP97', 'text': 'Ho HO Ho',
'type': 'message', 'team': 'T4YHY9UH0', 'ts': '1492478530.207125',
'source_team': 'T4YHY9UH0'}]
[{'channel': 'C4YFZE3JP', 'event_ts': '1492478533.953874',
'num_mentions': 0, 'mention_count_display': 0, 'num_mentions_
display': 0, 'type': 'channel_marked', 'unread_count': 0, 'ts':
'1492478530.207125', 'unread_count_display': 0, 'mention_count': 0}]
```

L'événement `hello` est un événement automatique que vous recevrez toujours en tant que premier élément juste après votre connexion. Le deuxième élément correspond à un message envoyé par un utilisateur et donc reçu par votre boucle infinie. Le dernier est un peu spécial. Il correspond à une modification de la ligne de lecture du *channel*. Cela arrive quand votre programme utilise un compte de connexion identique à celui d'un utilisateur réel et que l'utilisateur réel en question lit le *channel* sur lequel votre programme se trouve.

Le principal problème de l'utilisation de *Real Time Messaging* avec la bibliothèque officielle c'est que l'on se retrouve à faire des attentes actives sur la *websocket* ouverte. Mais rien ne nous empêche de modifier cela et de faire son propre mécanisme de connexion *websocket*.

CONCLUSION

Slack en plus d'être une plateforme intéressante pour les utilisateurs est une plateforme qui offre beaucoup de possibilités lorsque l'on se coiffe de la casquette de développeur. Cela n'a rien d'étonnant, le succès de Slack étant en partie dû à cette grande facilité avec laquelle on interagit avec la plateforme et qui a permis, en quelques semaines à peine après le lancement de Slack, de voir émerger un nombre impressionnant de bots et d'applications de connexion. En fait l'un des seuls reproches que l'on pourrait faire à Slack, c'est qu'il ne soit pas libre. Et il est vrai que c'est un reproche de grande taille. Heureusement pour nous, un certain nombre de clones existent et commencent à devenir des solutions de remplacement viables. À voir maintenant si ces alternatives libres proposent également un large panel de solutions d'intercommunication. Mais ceci est une autre histoire qui fera d'ailleurs peut-être l'objet d'un nouvel article. ■

RÉFÉRENCES

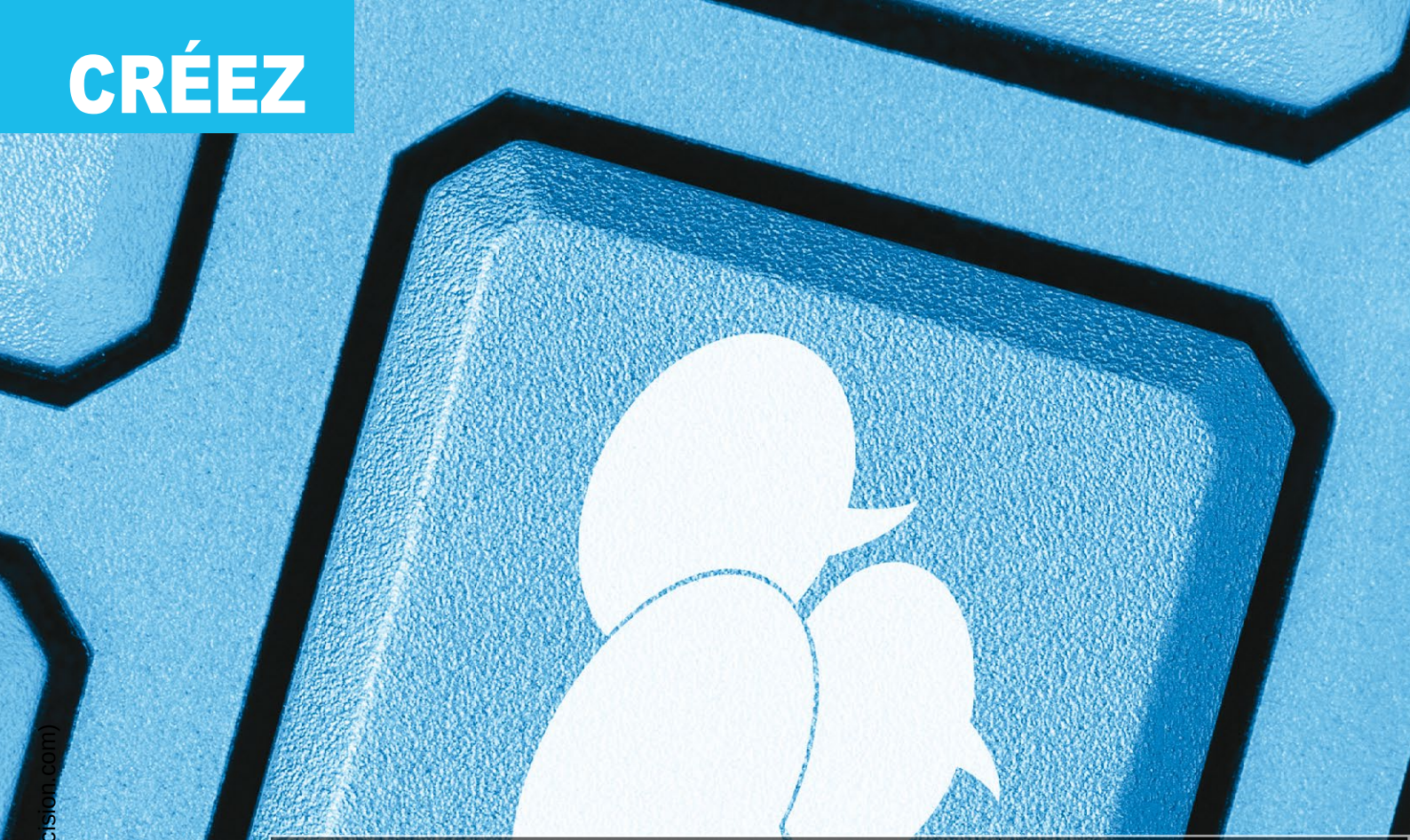
[1] Slack : **https://slack.com/**

[2] Url de la page de configuration des intégrations : **https://VOTRENOMDETEAM.slack.com/apps**

[3] Aide au formatage des messages : **https://api.slack.com/docs/messages/builder**

[4] python-slackclient : **https://github.com/slackapi/python-slackclient**

[5] *Token* generator : **https://api.slack.com/custom-integrations/legacy-tokens**



CRÉEZ UN CLIENT XMPP

Jean-Michel Armand

XMPP est le couteau suisse des messageries instantanées. C'est pour moi le protocole de messagerie instantanée qui devrait écraser tous les autres. Le limiter ainsi à un protocole de messagerie instantanée alors que c'est en fait un protocole d'échange d'informations, c'est déjà ne pas le juger à la hauteur de sa valeur. XMPP devrait, au vu de tous ses avantages et de sa puissance, faire jeu égal en importance avec HTTP et se tailler la part du lion dans tout ce qui est Internet des objets. Et pourtant, ce n'est pas le cas. Et pas une année ne passe sans que je me demande si au final XMPP ne va pas avoir le même destin que les cassettes Betamax. Mais en attendant, heureusement pour nous, le XMPP existe encore et nous allons pouvoir nous amuser avec, en Python, bien entendu.

XMPP [1], de son vrai nom *Extensible Messaging and Presence Protocol* est bien plus qu'un protocole de messagerie instantanée. C'est en fait un ensemble de protocoles standards ouverts définis par l'IETF **[2]** qui permettent de gérer bien entendu l'envoi de messages au sens large entre utilisateurs, mais aussi de mettre en place une réelle authentification, de gérer les notions de présence, de statut et de liste de contacts. Les avantages de l'XMPP sont multiples. C'est un protocole décentralisé, extensible et qui supporte le chiffrement. Une extension connue, **Jingle**, permet également le support de la voix et de la vidéo. Le protocole propose aussi la possibilité de déployer des passerelles. Soit entre serveurs, soit vers d'autres protocoles de messagerie instantanée. C'est-à-dire qu'en vous connectant sur votre serveur **jabber** habituel, vous pourrez discuter avec vos amis qui sont sur **IRC** ou **ICQ**. Chaque pièce ayant deux faces, cette extensibilité, cette modularité et ce foisonnement de fonctionnalités engendrent en contrepartie une certaine lourdeur du protocole. Et le choix de l'**XML** comme base pour les messages conduit à ajouter de la verbosité aux choses. En fait, XMPP, c'est un vrai couteau suisse permettant de tout faire, mais au prix d'une certaine lourdeur.

1. PRÉSENTATION DU PROTOCOLE XMPP

1.1 Notions utiles

La première notion à comprendre c'est la notion de **XEP [3]**. Une XEP c'est une *XMPP Extension Protocol*. En clair c'est un document qui décrit une extension au protocole XMPP. En fait dans sa version sans XEP le protocole XMPP est assez léger. Il va définir comment se connecter, s'authentifier, signaler sa présence et construire sa liste d'amis. Mais c'est tout. Par exemple, il n'est pas possible de faire de salons de discussions sans utiliser une XEP, la XEP 45. Il y a donc un grand nombre de XEP et il s'en rajoute continuellement. La dernière en date, la XEP-390 date du 28 février 2017 et modifie la façon qu'ont les entités XMPP de parler de leurs capacités (qui sont définies dans la XEP-115).

Tous les messages sont en XML, récupérer des infos à l'intérieur est donc assez simple. Par contre, envoyer un message devient plus compliqué que dans le cas d'autres messageries instantanées comme irc par exemple.

NOTE

Les capacités sont ce qu'une entité XMPP est capable de faire ou de gérer.

En plus de la notion de Message, XMPP met en place la notion **IQ**. Un IQ représente une *info/query*. Un IQ c'est un peu pour le XMPP les GET / POST du HTML. Vous allez en effet utiliser un IQ lorsque vous allez demander une information ou en modifier sa valeur. Il y a quatre types d'IQ : **get**, **set**, **result** et **error**.

Lorsque l'on fait du XMPP, on ne parle pas d'*id*, mais plus de **jid**, le `j` venant de Jabber. Un jid est composé de trois parties : l'utilisateur, le domaine et la ressource qui est optionnelle. L'utilisateur et le domaine sont des notions classiques. La ressource est quelque chose de très spécifique à XMPP. Un utilisateur peut en effet se connecter plusieurs fois en même temps sur le même serveur. La ressource lui servira à fournir un moyen de différencier ses différentes connexions. Vous pouvez avoir les ressources `work`, `home`, `gajim`, `phone`, etc.

2. NE RÉINVENTONS PAS LA ROUE, UTILISONS UNE BIBLIOTHÈQUE

Autant il est imaginable pour certaines messageries instantanées (comme IRC) de commencer par faire des tests sans utiliser une bibliothèque (c'est d'ailleurs ce que nous faisons dans l'article qui parle de connexion avec un serveur IRC), autant pour XMPP ce n'est pas une solution réellement envisageable. Nous allons

donc directement utiliser une bibliothèque qui nous facilitera un peu la tâche. Et au moment de choisir une bibliothèque à utiliser, je me suis rendu compte que décidément, XMPP n'était pas dans la meilleure des formes. Il y a en effet très peu de bibliothèques encore activement maintenues qui sont utilisables. En fait, il va rester principalement trois possibilités à savoir :

- **wokkel [4]** qui est une surcouche à **Twisted** proposant des fonctionnalités XMPP. Un portage Python 3 serait en cours, mais malheureusement rien de fonctionnel et il n'y a pas eu de commit depuis six mois sur GitHub ;
- **Sleekxmmp [5]**, la bibliothèque qui est a priori majoritairement utilisée quand on veut faire du Python en XMPP. Elle est encore maintenue et il y a des commits, de temps en temps avec même des *pull requests* ;
- **slixmpp [6]** qui est une réécriture du cœur de Sleekxmmp pour en extirper toute notion de *thread*.

J'ai donc décidé que nous allions passer quelques heures à nous amuser avec Sleekxmpp.

Avant toute chose, commençons par créer un nouveau virtualenv, toujours en python 3.5 et par y installer la bibliothèque :

Terminal

```
$ mkvirtualenv sleekxmpp -p /usr/bin/python3.5
$ pip install sleekxmpp
```

La version que j'ai installée se trouve être la version 1.3.2. Au vu du nombre de commits, il ne devrait pas y avoir de nouvelles versions entre le moment où j'écris ces lignes et le moment où vous allez vouloir tester, mais il vaut mieux préciser les choses, juste au cas où.

Une fois cela fait, commençons donc à coder :

Fichier

```
01: import sleekxmpp
02:
03: class XmppBot1(sleekxmpp.ClientXMPP):
04:
05:     def __init__(self, jid, password):
06:         sleekxmpp.ClientXMPP.__init__(self, jid, password)
07:         self.add_event_handler("session_start", self.start)
08:         self.add_event_handler("message", self.message)
09:
10:     def start(self, event):
11:         self.send_presence()
12:         self.get_roster()
13:
14:     def message(self, msg):
15:         print(msg)
16:         if msg['type'] in ('chat', 'normal'):
17:             msg.reply(u"%s Si tu as du temps, au lieu de discuter,
viens coder des Bots en Python !" % msg['from']).send()
18:
19: if __name__ == '__main__':
20:     from constants import jid2, password2
21:     linuxmagbot1 = XmppBot1(jid2, password2)
22:     if linuxmagbot1.connect(('jabber.fr', 5222)):
23:         linuxmagbot1.process(block=True)
24:         print("Die like the rest")
25:     else:
26:         print("Unable to connect.")
```

Voilà un premier programme qui va se contenter de répondre une phrase, toujours la même, aux utilisateurs qui vont venir parler en privé à notre bot. On commence, dès la création de notre bot, par configurer les *handlers* (lignes 7 et 8). Ici on va se contenter de réagir lors de la connexion et à chaque réception de message. Notre fonction `start` (lignes 10 à 12), va informer les gens de notre présence et va récupérer notre liste d'amis. Notre fonction `message`,

elle, (lignes 14 à 17) va afficher le message que l'on reçoit. Ensuite en fonction de son type on agira ou pas. Dans le cas où c'est bien un message de chat, on se contentera de renvoyer une phrase bateau contenant le nom de la personne qui nous a parlé (avec le `msg['from']`). Vous remarquez ici qu'il existe une notion de réponse que l'on utilise à la ligne 17 et qui nous facile grandement la vie. Ce n'est d'ailleurs pas qu'une méthode d'aide fournie par la bibliothèque. Les messages XMPP contiennent un attribut `thread` qui permet en effet de les chaîner (comme vous pourrez le vérifier dans l'affichage du message XMPP dans quelques lignes).

Les types d'un message peuvent être `chat`, `normal`, `groupchat`, `error`, ou `headline`. En plus de l'attribut `from`, on va avoir les paramètres `body`, `to` ainsi que `subject` pour un message, `subject` étant une donnée optionnelle.

Ensuite, lignes 20 à 22 on configure la connexion puis on l'établit. On lance enfin la boucle principale en ligne 23. Avant de passer à un bout de code permettant de se rendre sur un groupe de discussion, voici ce que donne le `print` de la ligne 15 :

Terminal

```
<message from="mrjmad@jabber.fr/9c8b0378-4e61-4a1a-82af-269e073aa765"
id="purpled6637cbb" to="linuxmag2@jabber.fr" type="chat"><active xmlns="http://
jabber.org/protocol/chatstates" /><body>Hello Bonjour !</body><html
xmlns="http://jabber.org/protocol/xhtml-im"><body xmlns="http://www.
w3.org/1999/xhtml"><p><span style="color: #252020;"><span style="font-size:
medium;">Hello Bonjour !</span></span></p></body></html></message>
```

Petite précision quand vous allez essayer de faire la même chose chez vous, l'identifiant pour s'authentifier sur un serveur jabber est l'identifiant complet. Donc par exemple `thierry@jabber.fr` et pas juste `thierry`. Rappelez-vous en, ça pourra vous faire économiser quelques minutes d'incompréhension (oui, c'est du vécu).

Voyons maintenant un exemple un peu plus complet et qui va permettre à notre bot de parler sur un salon.

Fichier

```
01: import sleekxmpp
02:
03: class XmppBot2(sleekxmpp.ClientXMPP):
04:
05:     def __init__(self, jid, password, room, nick):
06:         sleekxmpp.ClientXMPP.__init__(self, jid, password)
07:
08:         self.room = room
09:         self.nick = nick
10:         self.add_event_handler("session_start", self.start)
11:         self.add_event_handler("groupchat_message", self.muc_message)
12:         self.add_event_handler("muc::%s::got_online" % self.room,
13:                                self.muc_online)
14:         self.add_event_handler("message", self.message)
15:
16:
17:     def start(self, event):
18:         self.get_roster()
19:         self.send_presence()
20:         self.plugin['xep_0045'].joinMUC(self.room, self.nick, wait=True)
21:
22:     def muc_message(self, msg):
23:         print (msg)
24:         if msg['mucnick'] != self.nick and self.nick in msg['body']:
25:             self.send_message(mto=msg['from'].bare,
26:                               mbody="Je ne comprend pas ce que tu me dis
%s." % msg['mucnick'],
27:                               mtype='groupchat')
28:
29:     def muc_online(self, presence):
30:         if presence['muc']['nick'] != self.nick:
31:             self.send_message(mto=presence['from'].bare,
```

```
32:                                 mbody="Bonjour, %s" % presence['muc']['nick'],
mtype='groupchat')
33:
34:     def message(self, msg):
35:         if msg['type'] in ('chat', 'normal', 'groupchat'):
36:             if msg['mucnick'] == '':
37:                 msg.reply("Tu me parles, je te réponds").send()
38:
39:
40: if __name__ == '__main__':
41:     from constants import jid2, password2
42:     muc_chan = "SalonHSPython@chat.jabberfr.org"
43:     xmpp = XmppBot2(jid2, password2, muc_chan, "BlackKnight")
44:     xmpp.register_plugin('xep_0045')
45:     if xmpp.connect():
46:         xmpp.process(block=True)
47:     else:
48:         print("Unable to connect.")
```

Au niveau de l'initialisation de notre bot entre les lignes 5 à 14, on rajoute un nouveau *handler* en ligne 11 qui va permettre de récupérer les messages qui proviennent des salons où l'on est connecté. Lignes 12 et 13, on ajoute un handler qui va permettre de réagir quand quelqu'un arrive sur le salon. Ce *handler* se lancera aussi pour chaque utilisateur présent sur le salon au moment de notre connexion. Vous remarquerez la syntaxe `muc::NomSalon::got_online` permettant de ne récupérer que les événements d'un seul salon précisément. Ensuite dans notre fonction `start`, il n'y a qu'une nouveauté, on utilise le *plugin* gérant le XEP des groupes de discussion pour justement rejoindre un salon, en ligne 20.

La fonction `muc_message` (lignes 22 à 27) permet de gérer ce que l'on fait quand on reçoit un message provenant d'un salon. Ici on se contente de regarder si le message que l'on vient de recevoir nous est destiné et si oui, on y répond.

La fonction `muc_online`, est le *handler* que l'on a accroché à l'événement `got_online`. Elle va vérifier si l'événement ne nous concerne pas (le serveur envoyant à tout le monde et donc y compris à nous un événement `got_online` quand on se connecte) et ensuite souhaiter la bienvenue au nouveau venu.

La fonction `message()` lignes 34 à 37 n'est là que pour une chose, vous montrer que les messages de groupe vont bien avoir pour effet de lancer cette fonction en plus de la fonction `muc_message()`. Faites bien attention à ne pas l'oublier. Dans le même type, la ligne 24 ne sert pas qu'à tester si quelqu'un sur le salon de discussion s'adresse à nous. Elle permet également de vérifier que nous ne traitons pas un message que nous venons nous même d'envoyer. En effet, là encore le serveur envoie à tous les présents, y compris les émetteurs, les messages qui s'échangent sur le salon. Le test se fait avec l'attribut `munick` du message qui est un attribut en lecture seule qui contient le *nick* de l'émetteur du message. Un point fonctionnel intéressant de la gestion des salons de discussion par XMPP : un utilisateur peut avoir un *nickname* par salon. Lignes 25 à 27, vous voyez qu'on construit la réponse que l'on va envoyer sur le salon. Pour récupérer les informations nous permettant de répondre, on utilise `msg['from'].bare` à la ligne 31. Cela va nous donner l'identifiant à qui on adressera le message (ici, c'est directement le salon). En fait `msg['from']` renvoie une instance de `JID` (définie dans `sleekxmpp/jid.py`). La classe `JID` possède quelques attributs intéressants comme donc `bare`, `resource`, `full`, `username`, `host`, `server`, mais aussi `local` que l'on utilise pour n'avoir que la partie locale du JID de l'utilisateur. Ici, nous sommes sur un salon, donc le `JID` en question correspond au salon. Le reste du code ressemble beaucoup à notre premier test. Une chose importante tout de même, ne pas oublier l'activation du *plugin* qui permet de gérer la XEP 45 à la ligne 44.

Voici un exemple de message XMPP que recevra votre bot lorsque quelqu'un parlera sur un salon où il se trouve :

Terminal

```
<message id="purpled6637cd6" type="groupchat" from="salonhspython@chat.
jabberfr.org/mrjmad" to="linuxmag2@jabber.fr/16dcc5ad-a491-442e-bb46-
1f926bd3c146"><body>hello BlackKnight</body></message>
<message type="groupchat" from="salonhspython@chat.jabberfr.org/
BlackKnight" xml:lang="en" to="linuxmag2@jabber.fr/16dcc5ad-a491-442e-bb46-
1f926bd3c146"><body>Je ne comprends pas ce que tu me dis mrjmad.</body></message>
```

Le premier message XML correspond à un message qui est envoyé sur le salon par un autre utilisateur. Vous remarquerez que la balise `from` correspond bien à un `JID` qui modélise le salon comme un utilisateur et la personne qui parle sur le salon comme au final une ressource. Le deuxième message est en fait la réponse envoyée par le bot.

2.1 Google, Facebook et le XMPP

Gtalk fonctionne avec XMPP. Le service de discussions de **Facebook** aussi. Est-ce que votre bot pourra s'y connecter ?

Pour Facebook, il faudra que vous dotiez votre robot d'un compte. Ensuite pour se connecter, le nom d'utilisateur sera votre nom d'utilisateur Facebook (pas l'e-mail qui vous sert à vous connecter, mais bien votre nom d'utilisateur). Quant au serveur XMPP, il est tout simple : `chat.facebook.com`. Par contre, il y a plus de problèmes pour récupérer sa liste d'amis et leur surnom.

Pour **Google** et Gtalk, c'est un peu plus compliqué. Le certificat SSL utilisé ne pointe pas sur le bon domaine. Il vous faudra donc un peu modifier votre code. Rien de bien méchant. Dans la fonction `__init__()` de votre bot vous devrez enregistrer un *handler* de plus :

Fichier

```
self.add_event_handler("ssl_invalid_cert", self.invalid_cert)
```

Il vous faudra ensuite définir la fonction *handler* correspondante :

Fichier

```
def invalid_cert(self, pem_cert):
    der_cert = ssl.PEM_cert_to_DER_cert(pem_cert)
    try:
        cert.verify('talk.google.com', der_cert)
        logging.debug("CERT: Found GTalk certificate")
    except cert.CertificateError as err:
        log.error(err.message)
        self.disconnect(send_close=False)
```

Et voilà, votre robot pourra aller parler à ses amis présents sur **Google+**.

2.2 Pour finir, un peu de RPC

On peut faire plus que de la discussion avec XMPP, je n'arrête pas de le répéter à longueur d'article. Nous allons donc voir une autre utilisation du protocole XMPP : la mise en place de commande RPC. J'aurais aussi pu vous présenter du HTTP over XMPP, mais le RPC, depuis que je tape sur un clavier, cela me fascine. On va donc avoir besoin d'un serveur et d'un client.

Tout d'abord voici le code du serveur :

Fichier

```
01: from sleekxmpp.plugins.xep_0009.remote import Endpoint, remote, Remote, ANY_ALL
02: import threading
03:
04:
05: class LinuxMagRPC(Endpoint):
06:
07:     def __init__(self):
08:         self.answer = 0
09:         self._event = threading.Event()
10:
11:     def FQN(self):
12:         return 'LinuxMagRPC'
13:
14:     @remote
15:     def get_answer_to_everything(self):
16:         return self.answer
17:
18:     @remote
19:     def release(self):
20:         self._event.set()
21:
22:     def wait_for_release(self):
23:         self._event.wait()
24:
25:     @remote
26:     def set_answer_to_everything(self, answer):
27:         self.answer = answer
28:
29:
30: def main():
31:     session = Remote.new_session('LOGINSERVEUR@jabber.fr/rpc', 'PASSWORD')
32:     linuxmag = session.new_handler(ANY_ALL, LinuxMagRPC)
33:     linuxmag.wait_for_release()
34:     session.close()
35:
36: if __name__ == '__main__':
37:     main()
```

Puis le code du client :

Fichier

```
01: from sleekxmpp.plugins.xep_0009.remote import Endpoint, remote, Remote
02: import time
03:
04:
05: class LinuxMagRPC(Endpoint):
06:
07:     def FQN(self):
08:         return 'LinuxMagRPC'
09:
10:     @remote
11:     def get_answer_to_everything(self):
12:         return NotImplemented
13:
14:     @remote
15:     def release(self):
16:         return NotImplemented
17:
18:     @remote
19:     def set_answer_to_everything(self, answer):
20:         return NotImplemented
21:
22:
23: def main():
24:
25:     session = Remote.new_session('LOGINCLIENT@jabber.fr/rpc', 'PASSWORD')
26:
27:     linux_mag_rpc = session.new_proxy('LOGINSERVEUR@jabber.fr/rpc', LinuxMagRPC)
```

```
28:
29:     print(' Answer : %s ' % linux_mag_rpc.get_answer_to_everything())
30:     linux_mag_rpc.set_answer_to_everything(42)
31:     print(' Answer : %s ' % linux_mag_rpc.get_answer_to_everything())
32:     linux_mag_rpc.release()
33:     time.sleep(4)
34:     session.close()
35:
36: if __name__ == '__main__':
37:     main()
```

Étudions cela un peu plus en détail. Une précision, j'utilise rapidement du *threading* et des *events*, c'est simplement pour simuler le comportement du serveur. Ce n'est bien entendu que pour l'exemple. Un vrai serveur de RPC ne ferait pas cela ainsi. Maintenant que c'est dit, voyons voir le code.

Au niveau du serveur, on commence par déclarer notre classe serveur. La méthode **FQN** est un passage obligé : elle permet d'identifier de manière unique notre serveur de RPC. Ensuite on définit plusieurs méthodes que l'on va décorer avec le décorateur **@remote**, ce qui aura pour effet de les rendre accessibles. On peut changer le nom d'appel rpc de la fonction ou rendre la fonction publique ou pas grâce aux paramètres de **@remote**. Une fois notre classe serveur prête, on se contente de se connecter à la ligne 32 puis de déclarer que l'on attend les demandes, de la part de tout le monde. Sleekxmmp gère plutôt bien les ACL sur les connexions rpc, mais pour l'exemple j'ai voulu faire simple.

Passons maintenant au code du client. Tout d'abord on définit le proxy du serveur. C'est simplement une classe présentant des méthodes avec les mêmes signatures que les méthodes serveur (ligne 5 à 20). Ensuite on se connecte au serveur jabber (ligne 25) puis on demande un proxy sur notre composant serveur.

Une fois qu'on a effectivement notre proxy, il ne nous reste plus qu'à appeler les différentes méthodes du serveur et finir par le libérer. Et voilà, nous avons mis en place un mécanisme RPC en quelques dizaines de lignes !

CONCLUSION

XMPP est vraiment un excellent protocole d'échange de données. Certaines des applications qui ont un jour fonctionné en l'utilisant (ici je pense à **Wave** de Google, l'une des meilleures applications de discussion et de partage que j'ai pu utiliser, quel dommage vraiment...) ont su tirer parti de toute la richesse du protocole. J'espère que ce rapide aperçu vous aura donné envie d'aller plus loin et de tester des choses avec XMPP. Alors oui la barrière à l'entrée est plus importante et oui la courbe d'apprentissage est assez ardue, mais le jeu en vaut la chandelle... enfin, tant qu'il y aura des serveurs XMPP en ligne en tout cas. ■

RÉFÉRENCES

[1] XMPP : **http://xmpp.org/**

[2] IETF : **http://www.ietf.org/**

[3] Liste des XEP : **http://xmpp.org/xmpp-protocols/xmpp-extensions/**

[4] wokkel : **https://github.com/ralphm/wokkel**

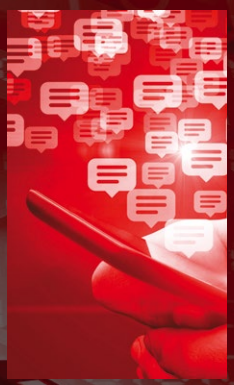
[5] SleekXMPP : **https://github.com/fritzy/SleekXMPP**

[6] slixmpp : **https://pypi.python.org/pypi/slixmpp/1.1**

PROGRESSEZ

3 PROGRESSEZ...

À découvrir dans cette partie...



Envoyez des SMS avec un Raspberry Pi et Python

Vous avez un Raspberry Pi, une clé 3G et une carte SIM ? Pourquoi ne pas envoyer des SMS directement depuis votre Raspberry Pi en Python ? Il faudra être patient en ce qui concerne l'installation, mais il est possible de réaliser cette opération très simplement. p. 108

Un exemple concret de serveur HTTP servant des fichiers

Dans cet article, nous étudierons le script woof permettant de servir des fichiers et comment les différentes problématiques inhérentes à cette tâche ont été résolues. Cela nous permettra de nous en inspirer par la suite pour éventuellement développer notre propre serveur... p. 116

ENVOYEZ DES SMS AVEC UN RASPBERRY PI ET PYTHON

Jean-Michel Armand

On peut faire beaucoup de choses en Python, y compris envoyer des SMS ! Et du coup, pourquoi ne pas en profiter pour réaliser cela sur un mini-ordinateur, un Raspberry Pi ? Une clé 3G, une carte SIM et c'est parti !

Le Raspberry Pi est une machine merveilleuse. À chaque fois que je traîne sur Internet, je découvre une nouvelle façon de l'utiliser. Console de retrogaming, centrale de domotique, ferme à bots pour messagerie instantanée ou réseaux sociaux, mini serveur web, boîte à son, j'en passe et des meilleures. Aujourd'hui, nous allons voir comment utiliser un petit Rpi pour envoyer des SMS. Attention toutefois, lors de mes multiples tests, j'ai utilisé trois clés USB 3G différentes et quatre cartes SIM de trois opérateurs (**Orange**, **Bouygues** et **Free**). J'ai malheureusement eu un taux de non-fonctionnement important, dépassant même la moitié des combinaisons que j'ai testées (voir tableau ci-dessous). Au final, il semblerait que le plus simple soit encore d'utiliser un vrai téléphone que l'on passerait en mode USB, mais n'ayant que mon i**Phone** sous la main, je n'ai malheureusement pu tester cette solution-là.

	Bouygues	Free	Orange Entreprise
Huawei E1750	✓	✓	✗
Pumpkin 3G	✓	✗	✗
4G System Clé XS Stick P10	✗	✗	✗

1. LA PARTIE COMPLIQUÉE, PRÉPARATION DU RASPBERRY

J'ai utilisé un Raspberry Pi 3 Modèle B. Je pense qu'il n'y aurait pas de problème à utiliser un modèle Rpi 2 Modèle B, mais mon dernier Rpi 2 ayant malheureusement grillé il y a quelques mois, je n'avais plus que des 3 sous la main. La consommation de la clé USB 3G n'étant pas anodine, il faut par contre prévoir une alimentation de qualité pour votre petite framboise (de toute façon comme les mauvaises alimentations sont les causes les plus fréquentes de mort des Rpi, je vous conseille vivement de ne pas lésiner sur les quelques euros supplémentaires que coûte une bonne alimentation). Après tout un Rpi entier, cela coûte de toute façon plus cher.

Concernant la distribution, j'ai opté pour une **Ubuntu MATE** spéciale Raspberry **[1]**, mais je pense qu'une **Raspbian [2]** fonctionnerait sans problème. L'avantage de la Ubuntu est qu'elle est fournie directement avec Python 3.5 que je voulais utiliser pour cet article. Je pensais également qu'ayant une Ubuntu sur mon *laptop*, avoir une Ubuntu sur mon Rpi me permettrait de me sortir de situations compliquées en reproduisant les choses sur mon *laptop* si nécessaire. En fait, ce ne fut d'aucune utilité, les erreurs que j'ai eues sur une de mes plateformes n'ayant pas été reproductibles sur l'autre.

Une fois la distribution installée, il faut installer ce qui va nous servir à communiquer avec le téléphone/modem 3G.

L'outil que tout le monde utilise maintenant se nomme **gammu [3]** et coup de chance, il existe un *binding* python pour l'utiliser, *binding* qui s'appelle donc **python-gammu [4]**. Je pensais donc qu'un simple `apt-get` ferait l'affaire... mais que nenni ma bonne dame. En effet, la dernière version de python-gammu se trouve être la 2.7 et python-gammu a bien entendu besoin de gammu. Mais du fait du changement d'API, python-gammu 2.7 a obligatoirement besoin de Gammu >= 1.37.90. Sauf que dans les dépôts officiels d'Ubuntu Mate, ce n'est pas cette version-là qui est installable, mais une version plus vieille. Il faut donc installer semi-manuellement un Gammu qui convienne. J'aurais également pu essayer de faire fonctionner les choses en python-gammu 2.6, mais la documentation, déjà réduite pour la 2.7 n'existe que peu pour la 2.6 et puis quitte à mettre en place un système de

SMS à travers un article, autant ne pas utiliser de bibliothèques en version dépassée. Pour installer mon Gammu, il m'a du coup fallu ajouter un ppa. Ce que j'ai fait avec les deux commandes suivantes :

Terminal

```
# add-apt-repository ppa:nijel/ppa
# apt-get update
# apt-get install gammu
```

Une fois cela fait, mon gammu était installé. Il fallait ensuite le rendre fonctionnel. Tout d'abord, il faut brancher son périphérique 3G. Une fois qu'il est branché, on va lancer la commande suivante :

Terminal

```
root@minimousse:~# dmesg | grep "ttyUSB*"
[12892.959618] usb 1-1.2: GSM modem (1-port) converter now attached to ttyUSB0
[12892.960338] usb 1-1.2: GSM modem (1-port) converter now attached to ttyUSB1
[12892.960922] usb 1-1.2: GSM modem (1-port) converter now attached to ttyUSB2
[12892.961486] usb 1-1.2: GSM modem (1-port) converter now attached to ttyUSB3
```

Cela va vous permettre de connaître sur quel port USB votre modem USB est connecté. Si la commande ne vous renvoie rien, ne vous découragez pas trop vite. Il est en effet possible que vous ne deviez pas chercher **ttyUSB** comme nom de port. Suivant les plateformes que j'ai utilisées, j'ai par exemple dû utiliser **/dev/ttyACM0**.

Une fois que vous savez sur quel port est connecté votre équipement 3G, on va pouvoir configurer gammu. Pour cela, il faut lancer la commande suivante qui va vous ouvrir une belle interface Ncurses (voir figure 1).

Terminal

```
$ gammu-config
```

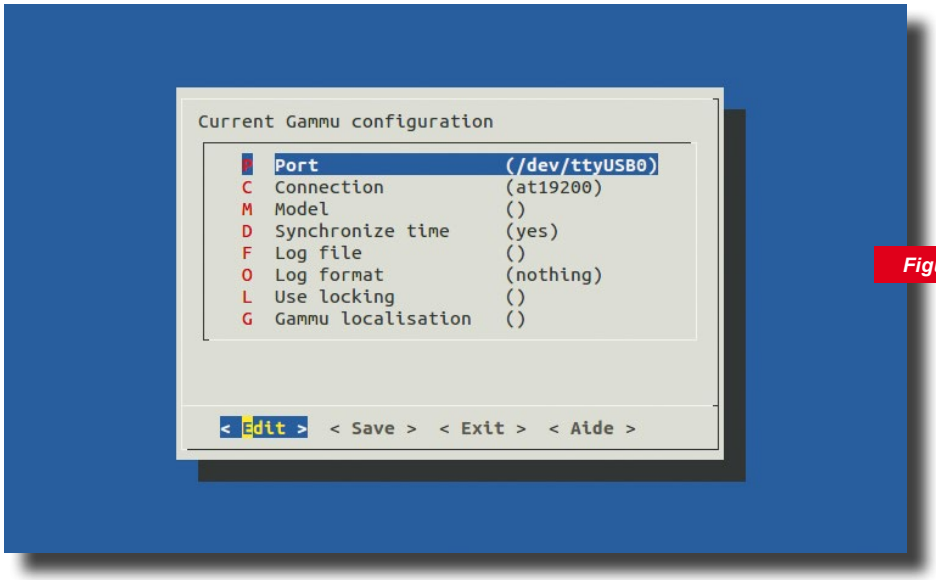


Figure 1

Interface Ncurses de Gammu.

Ici vous n'avez besoin, en tout cas ce fut le cas pour moi, que de définir le port. Vous voyez sur la capture 1 que j'ai déjà renseigné mes informations. Une fois cela fini, sauvez la configuration et sortez. Gammu-config vous a généré un fichier **.gammurc** à la racine de votre *home*. En voici une partie.

Fichier

```
01: [gammu]
02:
03: port = /dev/ttyUSB0
04: model =
05: connection = at19200
06: synchronizetime = yes
07: logfile =
08: logformat = nothing
09: use_locking =
10: gammuloc =
```

On voit que l'on ouvre une section **[gammu]**, que l'on définit ensuite le port, les informations de connexion ainsi que d'autres options. **synchronizetime** permet de synchroniser l'heure du modem sur celle du PC, et **use_Locking** permet de bloquer l'utilisation du modem pour une seule application.

Un autre utilitaire, **gammu-detect** permet aussi de trouver des informations concernant les périphériques 3G. Lorsque je le lance, j'ai la sortie suivante :

Terminal

```
(sms) jmad@minimousse:~$ gammu-detect
; Fichier de configuration généré par gammu-detect.
; Merci de consulter le manuel de Gammu pour plus d'informations.

[gammu]
device = /dev/ttyUSB0
name = Téléphone sur le port USB série Qualcomm__IncorporatedQualcomm_CDMA_
Technologies_MSM
connection = at

[gammu1]
device = /dev/ttyUSB1
name = Téléphone sur le port USB série Qualcomm__IncorporatedQualcomm_CDMA_
Technologies_MSM
connection = at

[gammu2]
device = /dev/ttyUSB2
name = Téléphone sur le port USB série Qualcomm__IncorporatedQualcomm_CDMA_
Technologies_MSM
connection = at

[gammu3]
device = /dev/ttyUSB3
name = Téléphone sur le port USB série Qualcomm__IncorporatedQualcomm_CDMA_
Technologies_MSM
connection = at

[gammu4]
device = 54:92:BE:B7:53:A6
name = B2100
connection = blueat
```

Pour faire bonne mesure, j'ai ajouté les informations données par **gammu-detect** dans mon fichier **.gammurc** et j'ai ajouté les sections supplémentaires. Une autre possibilité de configuration concernant le fichier **.gammurc** est de le mettre dans **/etc** et de l'appeler dans ce cas-là **gammurc**. J'ai pour ma part préféré laisser les configurations de mon *home*.

Une fois les configurations faites, on peut vérifier si tout fonctionne en lançant **gammu-identify** qui doit vous renvoyer des informations sur votre modem ou votre téléphone :

Terminal

```
(sms) jmad@minimousse:~$ sudo gammu --identify
Périphérique       : /dev/ttyUSB0
Fabricant          : Qualcomm
Modèle             : unknown (HSDPA Modem)
Firmware           : M7227-1.1  1  [Aug 17 2011 05:00:00]
IMEI               : 864319030956642
```

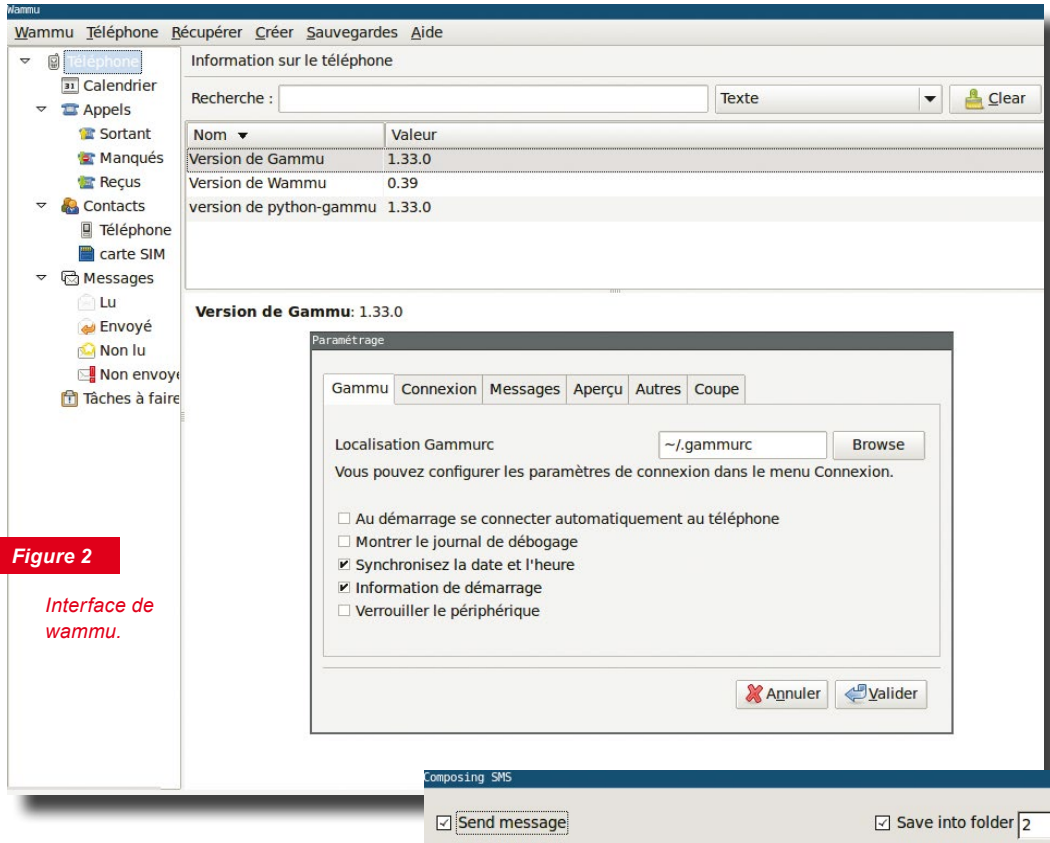


Figure 2

Interface de wammu.

Une fois arrivé ici, cela veut dire que le modem répond. On va pouvoir passer à la partie Python. Avant cela, deux dernières choses. Il existe un outil de configuration pour gammu qui s'appelle **wammu**. Il s'installe également grâce à votre gestionnaire de paquets et il est effectivement plutôt bien fait. On peut demander une recherche automatique des périphériques, puis finir les configurations et ensuite utiliser toutes les possibilités offertes par gammu, mais avec une jolie interface graphique. Pour que vous voyiez à quoi cela ressemble, voici deux captures d'écran, la

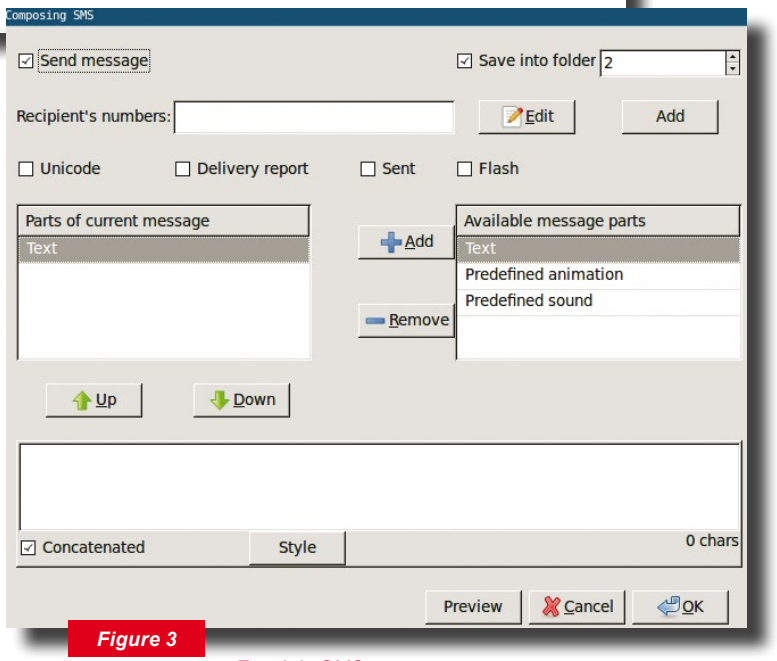


Figure 3

Envoi de SMS avec wammu.

première concernant l'interface de wammu (figure 2) et la seconde concernant la fonctionnalité d'envoi de SMS de wammu (figure 3).

On va d'ailleurs, avant de passer à la partie Python, tester si les choses fonctionnent. Pour cela, il n'y a qu'à essayer d'avoir l'heure de notre périphérique 3G en tapant :

Terminal
```
$ gammu --getdatetime
```

Bon pour le coup, avec mes clés 3G, cela ne fonctionnait pas, la preuve en image :

Terminal
```
(sms) jmad@minimousse:~$ gammu --getdatetime
Erreur à l'ouverture du périphérique: vous n'avez pas les bons droits.
(sms) jmad@minimousse:~$ sudo gammu --getdatetime
Fonctionnalités non supportées par votre téléphone.
```

Premier essai, un problème de droit. C'est d'ailleurs problématique, je pensais pouvoir utiliser gammu en utilisateur, mais impossible d'y arriver. Normalement, il faut ajouter l'utilisateur en question au groupe **dialout** ou **plugdev**, ce que j'ai fait avec un simple :

Terminal
```
# adduser jmad dialout
```

Mais non, pas moyen de faire avancer les choses. Même en tentant d'écrire des règles spéciales udev pour mes clés 3G (comme détaillé dans la documentation ici : **https://wammu.eu/docs/manual/config/**), je n'y suis pas arrivé. J'ai du me résoudre à utiliser un **sudo** et là, patatras, je n'ai pas le droit à l'heure. Bon du coup, testons directement en envoyant un SMS. La commande est facile (bon c'est encore plus simple avec wammu, mais pour ne rien vous cacher, j'ai essayé wammu après avoir fini la totalité de la configuration « à la main » avec gammu).

Et donc en lançant la commande :

Terminal
```
$ echo "On dit Gnu Linux ! Attention Gnu Linux Mag te surveille" | gammu
--sendsms TEXT 06XXXXXX
```

Et je pensais que ça allait fonctionner. Mais non. Là, je dois bien vous avouer que j'ai failli en rendre mon tablier. La plupart des périphériques que j'avais essayé ne m'avait pas permis d'arriver jusque-là. Et là, alors que je pensais avoir réussi, la chausse-trappe finale. En fouillant la documentation, je découvris que certains appareils pouvaient avoir besoin d'un code PIN. Cela semble aller de soi quand on utilise un téléphone, mais moi avec mes clés 3G, je n'y avais pas pensé. J'entrepris donc de vérifier si c'était le cas.

Terminal
```
(sms) jmad@minimousse:~$ sudo gammu getsecuritystatus
En attente du PIN.
```

Et voilà, la dernière difficulté avant de pouvoir réussir à envoyer un SMS. Sans un doute, je lançais donc les commandes permettant de saisir le PIN et de vérifier que c'était bon.

Terminal
```
(sms) jmad@minimousse:~$ sudo gammu entersecuritycode PIN 1234
(sms) jmad@minimousse:~$ sudo gammu getsecuritystatus
Rien à faire entrer.
```

Et ce fut le moment, où croisant les doigts, je relançais la commande pour envoyer un SMS.

Terminal

```
$ echo "On dit Gnu Linux ! Attention Gnu Linux Mag te surveille" | gammu
--sendsms TEXT 06XXXXXX
Si vous désirez interrompre, appuyez sur Ctrl + C...
Envois du SMS 1 1 / 1 2.... Attente d'une réponse réseau

..erreur 500, message de référence=-1 1
Erreur inconnue.
```

Je vous ai dit que les clés 3G cela ne captent pas vraiment très bien ? A priori en tout cas, les miennes n'étaient pas dans leurs meilleures dispositions.

Après un déplacement généralisé de mon Raspberry, mon *laptop* et moi-même vers l'extérieur, je lançais une nouvelle fois la fatidique commande.

Et là, miracle !

Un SMS part vraiment. Et il est même reçu. Ça fonctionne. Pour le coup d'ailleurs, ce fut un vrai soulagement. Vu le nombre d'essais en terme de matériel que j'ai du faire pour y arriver, j'ai vraiment cru que j'allais devoir expliquer à monsieur le rédac' chef qu'en fait cet article allait se résumer à un 'désolé, mais en fait, ça ne marche pas'. Je pense que j'aurais eu quelques problèmes.

2. PASSONS AU PYTHON

Maintenant que la partie « main dans le cambouis » est finie, on va pouvoir commencer à jouer en Python. Tout d'abord, on commence par créer un virtualenv en python 3.5 (le mien s'appelle **sms**, vous avez sûrement déjà dû le noter si vous avez lu mes extraits de console).

Ensuite, on y lance l'installation de python-gammu :

Terminal

```
$ mkvirtualenv sms -p /usr/bin/python3.5
$ pip install python-gammu==2.7.0
```

Pour rappel, moi j'ai du faire cela en root parce que je ne suis pas arrivé à faire fonctionner les choses avec mon utilisateur « normal ». Mais peut-être que vous aurez plus de chance que moi.

Nous allons maintenant commencer par refaire ce que nous avions fait avec la ligne de commandes, à savoir envoyer un SMS. Voici le code pour le faire :

Fichier

```
01: import gammu
02:
03: sm = gammu.StateMachine()
04: sm.ReadConfig()
05: sm.Init()
06:
07: message = {
08:     'Text': 'n dit Gnu Linux ! Attention Gnu Linux Mag te surveille',
09:     'SMSC': {'Location': 1},
10:     'Number': TEL_NUMBER_IN_STRING,
11: }
12:
13: sm.SendSMS(message)
```

Et voilà. Assez facile. On commence par initialiser le tout avec les lignes 3, 4 et 5. Puis on définit les informations du SMS et enfin on l'envoie. Le dictionnaire en question peut contenir pas mal d'informations. La documentation [5] liste le tout. Mais on peut noter que l'on peut, en modifiant l'attribut `Class` (par défaut il est à `1`), et en ajoutant des informations dans le dictionnaire `SMSC` changer le numéro ou le label de l'expéditeur. `SMSC` correspond en fait aux informations de l'émetteur. En définissant donc une `Class` à `1` et donnant une valeur à la clé `NUMBER` du dictionnaire `SMSC`, on change le numéro d'expédition.

Même s'il est vrai qu'envoyer des SMS est le sujet principal de cet article, il ne faut pas croire que gammu et python-gammu se limitent à cela.

Vous allez en effet pouvoir démarrer des appels téléphoniques, lire les événements présents dans les agendas, les *todos* ou des contacts contenus dans le téléphone, mais aussi créer dés événements, des *todos* ou des contacts.

Vous allez également pouvoir gérer les *backups*, extraire ou encoder des Vcard. En bref, vous allez pouvoir contrôler à peu près toutes les fonctions classiquement « téléphoniques » de votre téléphone.

CONCLUSION

Le sujet de l'article était de pouvoir envoyer des SMS en Python, à partir d'un Raspberry. Même si cela ne fut pas sans douleur, au final les choses fonctionnent. Et cela permet déjà d'imaginer énormément de façons de mettre en pratique une telle fonctionnalité. Mais ce n'est pas tout. En effet gammu, quand il s'installe ne s'installe pas seul. Il amène avec lui **Gammu SMS Daemon** qui permet lui de recevoir des SMS. Gammu SMS Daemon qui s'utilise également en Python, toujours à travers python-gammu. Et là d'un coup, les potentialités se décuplent. Vous pouvez imaginer envoyer un SMS à votre Rpi pour qu'il allume le chauffage quand vous rentrez de vacances, pour qu'il fasse un ensemble de vérifications sur vos serveurs, qu'il démarre ou stoppe des choses. Ou pourquoi pas pour qu'il mette en route automatiquement la cafetière quand vous vous levez (même si là effectivement, je pense qu'une communication IP directe serait plus efficace). Une dernière chose avant de vous laisser imaginer révolutionner le futur de la domotique, il existe une distribution Linux Raspberry Pi qui est spécialement faite pour envoyer et recevoir des SMS. Elle s'appelle **RaspiSMS [7]**. Et elle fait tout ce que je viens de vous montrer et même plus. Mais bon, tout est fait en PHP et où est le plaisir si on ne peut pas refaire soi-même les choses ! ■

RÉFÉRENCES

[1] Ubuntu MATE pour Raspberry : **https://ubuntu-mate.org/raspberry-pi/**

[2] Raspbian : **https://www.raspbian.org/**

[3] gammu : **https://wammu.eu/gammu/**

[4] python-gammu : **https://wammu.eu/python-gammu/**

[5] SMS Object in gammu : **https://wammu.eu/docs/manual/python/objects.html#sms-obj**

[6] Gammu SMS Daemon : **https://wammu.eu/smsd/**

[7] RaspiSMS : **https://github.com/RaspbianFrance/RaspiSMS**

UN EXEMPLE CONCRET DE SERVEUR HTTP SERVANT DES FICHIERS

Sébastien CHAZALLET

Nous allons ici réutiliser ce que nous avons auparavant présenté dans l'article sur les concepts de base en essayant d'aller plus loin et en prenant comme fil rouge l'application woof.

L'application **woof** est une petite merveille d'application qui, en à peine plus de 500 lignes de code, commentaires et documentation compris, vous permet de partager un fichier sur le réseau. Analyser ce code permettra de créer un serveur de fichiers que nous pourrons adapter à différents besoins.

1. PRÉSENTATION DE WOOF

L'installer est assez basique :

Terminal
```
# aptitude install woof
```

Il s'agit d'un seul fichier :

Terminal
```
$ which woof
/usr/bin/woof
```

Et il suffit de l'éditer pour lire son code, écrit en **Python** :

Terminal
```
$ vim `which woof`
```

L'utiliser est très simple, il vous suffit de préciser le fichier à partager :

Terminal
```
$ woof fichier.py
Now serving on http://192.168.1.9:8080/
```

Lorsque vous lancez cette commande, vous créez en fait un mini-serveur HTTP qui sert le fichier. Par défaut, ce serveur tourne en boucle locale, sur le port **8080**, mais des options peuvent vous permettre de changer cela :

Terminal
```
$ woof -h
```

De plus, vous avez à votre disposition un fichier de configuration dans **~/.woofrc** qui devrait vous permettre d'adapter la configuration que vous souhaitez de manière permanente.

Une fois woof lancé, vous pouvez prendre cette url et l'envoyer par messagerie à celui qui va la télécharger. Cette personne pourra alors copier l'url dans son navigateur et il téléchargera le fichier. Dans le terminal, woof affichera l'IP de celui qui télécharge et se stoppera lorsque le téléchargement sera terminé, terminant ainsi le serveur.

On peut aussi partager à plusieurs personnes :

Terminal
```
$ woof -c 8 fichier.py
```

Dans cet exemple, les huit premiers à envoyer la requête seront les huit premiers servis.

Enfin, on peut également envoyer des répertoires entiers, en les compressant au passage (et choisir le format de compression).

Enfin, on terminera en précisant que woof est écrit en Python2, ce qui fait qu'il y a de légères différences avec ce que vous connaissez déjà, et qu'il utilise un code partiellement obsolète (comme le recours à **getopt** au lieu de **argparse**, pour ceux qui connaissent). Ceci dit, ces éléments sont hors sujet par rapport à ce qui nous intéresse : le réseau.

1.1 Déterminer l'IP du serveur

Et la première question à laquelle nous allons répondre est celle-ci : comment woof détermine-t-il quelle est l'IP de la machine courante ?

A priori, on pourrait y répondre à l'aide d'un appel système ou d'une fonctionnalité du langage Python, mais ce n'est pas vraiment nécessaire.

En fait, lorsque l'on crée une socket, on doit lui préciser une IP et un port. Cependant, l'IP réellement utilisée va être déterminée par la socket, qui ira chercher l'IP réelle de votre machine. Ainsi, vous pouvez faire, dans un terminal Python, ceci :

Fichier

```
>>> s = socket.socket (socket.AF_INET, socket.SOCK_DGRAM)
>>> s.connect (("192.0.2.0", 80))
>>> print(s.getsockname())
('192.168.1.9', 48541)
```

On retrouve bien mon adresse IP réelle ainsi qu'un port quelconque, ce dernier point n'étant pas important du tout. Notez que l'on peut faire la même chose avec une autre IP :

Fichier

```
>>> s.close ()
>>> s = socket.socket (socket.AF_INET, socket.SOCK_DGRAM)
>>> s.connect (("198.51.100.0", 80))
>>> print(s.getsockname())
('192.168.1.9', 58913)
```

Et une dernière :

Fichier

```
>>> s.close ()
>>> s = socket.socket (socket.AF_INET, socket.SOCK_DGRAM)
>>> s.connect (("203.0.113.0", 80))
>>> print(s.getsockname())
('192.168.1.9', 42894)
>>> s.close ()
```

Ces IP ne sont pas choisies au hasard. Il s'agit d'IP appartenant à des plages particulières [1]. Ces adresses IP appartiennent à des plages réservées, portant les noms de **TEST-NET**, **TEST-NET-2** et **TEST-NET-3**.

On peut donc trouver, dans woof, le code suivant (expurgé des commentaires) :

Fichier

```
def find_ip ():
    candidates = []
    for test_ip in ["192.0.2.0", "198.51.100.0", "203.0.113.0"]:
        s = socket.socket (socket.AF_INET, socket.SOCK_DGRAM)
        s.connect ((test_ip, 80))
        ip_addr = s.getsockname ()[0]
        s.close ()
        if ip_addr in candidates:
            return ip_addr
        candidates.append (ip_addr)
    return candidates[0]
```

En gros, on teste les IP que l'on obtient par la méthode que nous venons de voir et si l'on trouve au moins deux fois la même, alors on l'utilise. Sinon, on prend la première d'entre elles.

2. SERVIR UN FICHIER

Nous allons maintenant présenter la manière de servir un fichier. Pour cela, woof utilise un *handler* tout ce qu'il y a de plus classique :

Fichier
```
class FileServHTTPRequestHandler (http.server.BaseHTTPRequestHandler):
    server_version = "Simons FileServer"
    protocol_version = "HTTP/1.0"
```

Il en profite pour surcharger quelques attributs et ainsi personnaliser un peu le serveur. Utiliser le protocole HTTP/1.0 au lieu du 1.1 permet d'assurer une compatibilité avec de (très) vieux navigateurs, dans un contexte où la version 1.1 n'apporte rien de particulier, quoi qu'il en soit.

Il rajoute ensuite un attribut à la classe, qui est le nom du fichier, mais le laisse vide, car à cet instant, on ne le connaît pas. Pour rappel, le *handler* va gérer le fait de servir un fichier à une personne. Donc, si le fichier est partagé *n* fois, il y aura *n* instances de ce *handler* qui seront créées et à chaque fois, le même fichier sera servi.

Le fait que le nom de fichier soit un attribut de classe (donc identique pour toutes les instances) a donc du sens :

Fichier
```
    filename = "-"
```

On notera que le tiret a ici une signification : ce qui est envoyé est issu de *stdin* - woof peut aussi servir à envoyer ce que vous tapez dans votre terminal.

On ne va maintenant se focaliser que sur ce qui est important, le fait de servir un fichier (pour cela, on va retirer du code tout ce qui concerne des sujets autres, y compris le fait de servir un répertoire et donc la partie sur la compression - ceci compliquerait inutilement le propos).

Lorsque l'on va dans navigateur et que l'on tape une URL et qu'on la valide, on envoie une requête utilisant la méthode GET. Il nous faut donc répondre spécifiquement à cette méthode :

Fichier
```
    def do_GET (self):
```

Comme on a pu le présenter dans l'article précédent, on dispose d'un certain nombre d'informations à propos du client, en particulier la notion de chemin. On veut ici récupérer un chemin propre et pour cela, on utilise séquentiellement **`urllib.unquote`** puis **`urllib.quote`** :

Fichier
```
        self.path = urllib.quote (urllib.unquote (self.path))
```

On détermine ensuite la localisation du fichier à partager par rapport au chemin de base, c'est-à-dire par rapport à l'endroit où l'on a lancé le serveur :

Fichier
```
        location = "/" + urllib.quote (os.path.basename (self.filename))
```

Si ces deux données ne correspondent pas, alors on renvoie le navigateur vers la bonne URL, en utilisant le code erreur **302** [2], qui correspond à une redirection temporaire.

Ce qu'il faut percevoir, ce sont les conséquences concrètes de l'utilisation de ce code. Associé au code, on a un entête **Location** qui contient l'URL où l'on pourra trouver la ressource et le navigateur va alors demander cette dernière sans que cette URL n'apparaisse dans la barre d'adresse.

Voici le code correspondant :

Fichier

```
        if self.path != location:
            txt = """\
                    <html>
                       <head><title>302 Found</title></head>
                       <body>302 Found <a href="%s">here</a>.</body>
                    </html>\n""" % location
            self.send_response (302)
            self.send_header("Location", location)
            self.send_header("Content-Type", "text/html")
            self.send_header("Content-Length", str (len (txt)))
            self.end_headers()
            self.wfile.write(txt)
            return
```

Vous noterez cependant que l'on rajoute aussi un contenu HTML qui est simplement un lien vers la bonne URL. Si ce code s'affiche, vous devrez alors cliquer sur ce lien pour télécharger le fichier.

La question que l'on peut se poser à ce point est celle-ci : à quoi cela sert-il ? Et si vous avez la moindre connaissance de la manière dont HTTP fonctionne, vous pouvez probablement être choqué de voir cela.

Il y a beaucoup de raisons à tout cela, mais la principale est due à l'histoire du Web et de HTTP, à la difficulté que nous connaissons tous à faire évoluer les choses (grâce à ce que l'on peut joliment nommer « la résistance au changement »). Et c'est là l'illustration parfaite, s'il en est, de cette problématique.

En effet, si vous avez un navigateur qui ne respecte pas les normes, alors il ne saura pas quoi faire avec un **302** et il a besoin d'un contenu pour que l'utilisateur puisse lui-même cliquer sur le bouton pour remplacer la redirection temporaire. Mais même Lynx, un navigateur en console, va comprendre la redirection et vous éviter cet affront.

Et pour être encore plus précis, j'ai passé sous silence certains faits lorsque je vous ai dit qu'il n'y avait pas de différence entre HTTP/1.0 et 1.1 car, s'il n'y en a pas dans ce cas précis, si la requête avait été un POST, il y avait deux manières de la rediriger : en conservant la méthode POST pour la nouvelle localisation, ou en passant sur un GET. D'où les nouveaux codes **303** et **307** introduits par HTTP/1.1.

En bref, créer un serveur web nécessite de maîtriser tous ces aspects, mais aussi, potentiellement, de prévoir que des personnes vont utiliser des clients qui ne respecteront peut-être pas les normes. Bon courage !

Nous terminerons enfin par la partie réellement intéressante, celle qui va envoyer le fichier au client. Et on commence avec les entêtes :

Fichier

```
        self.send_response (200)
        self.send_header ("Content-Type", "application/octet-stream")
        self.send_header ("Content-Length", os.path.getsize (self.
filename))
        self.end_headers ()
```

On précise ici qu'il s'agit d'une réponse **200**, pour dire au client que tout va bien, puis on précise un *content-type* qui est certes plus que vague, mais qui n'est pas une information qui sera enregistrée chez le client : elle est détectée à chaque fois. De plus, votre navigateur saura parfaitement le détecter par lui-même.

Ensuite, on se doit de renseigner la taille de l'envoi, ce qui se fait par **os.path.getsize**, tout simplement.

Enfin, on envoie réellement le fichier :

Fichier

```
        try:
           with open(self.filename, 'rb') as datafile:
               shutil.copyfileobj (datafile, self.wfile)
        except Exception as e:
           print(e)
           print("Connection broke. Aborting" , file=sys.stderr)
```

Et là, on utilise tout simplement **shutil**, qui est le module de haut niveau pour faire, si on résume très fortement, tout ce que les commandes élémentaires de **bash** peuvent faire. Oui, c'est très résumé. Certains diront exagéré et ils n'auront pas tort, mais l'idée est là.

Bien entendu, il faut rajouter un peu de code pour compter le nombre de téléchargements effectifs (et donc éteindre le serveur lorsque son travail est terminé), ce que j'ai retiré du code présenté dans cet article.

Il ne nous reste maintenant plus qu'à voir comment créer le serveur. Comme on peut servir plusieurs fichiers en même temps, il nous faut un serveur multitâche, ce dont nous avons déjà parlé dans le précédent article :

Fichier

```
class ThreadedHTTPServer(socketserver.ThreadingMixIn, http.server.
HTTPServer):
   """Handle requests in a separate thread"""
```

Voici maintenant la fonction qui met en place le serveur :

Fichier

```
def serve_files (filename, maxdown = 1, ip_addr = '', port = 8080):
```

Elle prend en paramètre le nom du fichier à servir, le nombre de fois qu'il faut le servir ainsi que l'IP et le port fourni par la ligne de commandes ou, à défaut, le fichier de configuration.

La première chose à faire est de passer le nom du fichier à télécharger au *handler* :

Fichier

```
   FileServHTTPRequestHandler.filename = filename
```

On rappelle qu'il s'agit d'un attribut de classe.

Ensuite, on démarre le serveur, d'une manière légèrement différente de ce que nous avons vu :

Fichier

```
   try:
      httpd = ThreadedHTTPServer ((ip_addr, port),
FileServHTTPRequestHandler)
   except socket.error:
      print >>sys.stderr, "cannot bind to IP address '%s' port %d" %
(ip_addr, port)
      sys.exit (1)
```

Évidemment, il faut gérer la possibilité que l'IP fournie par la ligne de commandes (ou le port) ne puisse être accessible.

Il faut ensuite déterminer l'IP à partager. Pour cela, on va utiliser la fonctionnalité présentée au tout début de cet article :

Fichier

```
if not ip_addr:
   ip_addr = find_ip ()
if ip_addr:
   print "Now serving on http://%s:%s/" % (ip_addr, httpd.server_port)
```

Pour les plus attentifs d'entre vous, vous aurez noté que l'on détermine l'IP après avoir créé la socket, mais uniquement si cette IP est une chaîne vide. Comment cela se fait-il ?

Fichier

```
>>> import socket
>>> s = socket.socket (socket.AF_INET, socket.SOCK_DGRAM)
>>> s.connect (("", 80))
>>> print(s.getsockname())
('127.0.0.1', 35088)
>>> s.close ()
```

Une IP vide signifie que l'on va tourner en boucle locale. On ne peut donc pas partager cette adresse, puisqu'elle est inaccessible aux autres. C'est à ce moment-là que l'on va partager une IP qui peut l'être. Les règles réseaux feront que, tout naturellement, en tapant sur **192.168.1.9** (dans mon exemple), on ira redescendre dans ma boucle locale.

Pour terminer, il suffit de démarrer le serveur :

Fichier

```
httpd.serve_forever ()
```

3. COMPRESSER UN RÉPERTOIRE ET L'ENVOYER

Pour commencer, il est important de savoir si le chemin donné par l'utilisateur correspond à un fichier ou un répertoire, auquel cas, nous allons rajouter, lors de la redirection, une extension :

Fichier

```
if os.path.isdir (self.filename):
    if compressed == 'gz':
        location += ".tar.gz"
    elif compressed == 'bz2':
        location += ".tar.bz2"
    elif compressed == 'zip':
        location += ".zip"
    else:
        location += ".tar"
```

La variable **compressed** contient le nom de l'utilitaire de compression à utiliser.

Ensuite, lors du téléchargement effectif, nous allons à nouveau devoir déterminer s'il s'agit d'un fichier ou d'un répertoire :

Fichier

```
type = None
if os.path.isfile (self.filename):
    type = "file"
elif os.path.isdir (self.filename):
    type = "dir"
elif self.filename == "-":
    type = "stdin"
if not type:
    print("can only serve files, directories or stdin. Aborting." ,
file=sys.stderr)
    sys.exit(1)
```

La particularité ici est que l'on accepte aussi de servir le contenu que tapera l'utilisateur, au lieu d'un fichier ou d'un répertoire. Nous verrons cela ultérieurement, à la fin de ce chapitre.

Pour la suite, rien ne change lorsque l'on écrit les entêtes, si ce n'est qu'on ne précise la taille de ce que l'on envoie que s'il s'agit d'un fichier.

Fichier

```
self.send_response (200)
self.send_header ("Content-Type", "application/octet-stream")
if os.path.isfile (self.filename):
    self.send_header ("Content-Length", os.path.getsize (self.filename))
self.end_headers ()
```

Ce qui est plus intéressant est le code qui est utilisé pour envoyer un contenu compressé, que ce soit en **`tar`**, en **`tar.gz`** ou en **`tar.bz2`**, lesquels utilisent la même bibliothèque.

D'une part, on ouvre l'archive en écriture, avec le bon mode, et en utilisant directement l'attribut **`wfile`** comme fichier dans lequel écrire l'archive, puis rajouter le contenu dans l'archive :

Fichier

```
with tarfile.open (mode='w|%s' % compressed, fileobj=self.wfile) as tfile:
    tfile.add (self.filename, arcname=os.path.basename(self.filename))
```

C'est tout simplement génialement simple et efficace, court et lisible, puissant et facile.

Après, si on veut être complet, il existe aussi le format ZIP et là, les choses sont un peu moins directes. En effet, ce format, calé pour **Windows** et très différent des autres nécessite un peu plus de travail. Il faut tout d'abord créer un *wrapper*, c'est-à-dire un objet n'étant pas un fichier, mais se comportant comme tel et donc utilisable en tant que tel et qui, par conséquent, dispose des méthodes **`tell`**, **`seek`** et **`write`**. Et rajoutons que pour qu'il soit fonctionnel, il faut qu'il respecte la norme du format ZIP, autant dire qu'il faut connaître cette norme pour comprendre réellement ce qui se passe ici :

Fichier

```
class EvilZipStreamWrapper(TM):
   def __init__ (self, victim):
      self.victim_fd = victim
      self.position = 0
      self.tells = []
      self.in_file_data = 0

   def tell (self):
      self.tells.append (self.position)
      return self.position
```

```
    def seek (self, offset, whence = 0):
       if offset != 0:
          if offset == self.tells[0] + 14:
             self.write ("PK\007\010")
          elif offset == self.tells[1]:
             self.tells = []
             self.in_file_data = 0
          else:
             raise "unexpected seek for EvilZipStreamWrapper"

    def write (self, data):
       if self.in_file_data == 0:
          if data[:4] == zipfile.stringFileHeader:
             hdr = list (struct.unpack (zipfile.structFileHeader, data[:30]))
             hdr[3] |= (1 << 3)
             data = struct.pack (zipfile.structFileHeader, *hdr) + data[30:]
             self.in_file_data = 1
          elif data[:4] == zipfile.stringCentralDir:
             hdr = list (struct.unpack (zipfile.structCentralDir, data[:46]))
             hdr[5] |= (1 << 3)
             data = struct.pack (zipfile.structCentralDir, *hdr) + data[46:]

       self.position += len (data)
       self.victim_fd.write (data)

    def __getattr__ (self, name):
       return getattr (self.victim_fd, name)
```

Je suis donc parfaitement en accord avec vous si vous me reprochez d'écrire des choses violentes, parce que c'est ici manifestement le cas. D'autant plus que je ne vais pas spécialement m'attarder là sur ces problématiques bas niveau, mais plutôt enchaîner avec la manière d'utiliser ce *wrapper*, qui est nettement plus intéressante. Tout d'abord, on crée l'objet *wrapper* autour de **wfile** :

Fichier

```
ezfile = EvilZipStreamWrapper (self.wfile)
```

Puis on l'utilise dans cette boucle :

Fichier

```
with zipfile.ZipFile (ezfile, 'w', zipfile.ZIP_DEFLATED) as zfile:
    stripoff = os.path.dirname (self.filename) + os.sep
    for root, dirs, files in os.walk (self.filename):
        for f in files:
            filename = os.path.join (root, f)
            if filename[:len (stripoff)] != stripoff:
                raise RuntimeException, "invalid filename assumptions,
please report!"
            zfile.write (filename, filename[len (stripoff):])
```

Cette boucle va parcourir récursivement tous les répertoires et fichiers contenus dans le répertoire à compresser et compresser chaque fichier en lui donnant son chemin depuis le répertoire initial.

Maintenant que vous avez vu ça, vous devriez apprécier d'autant plus le code de haut niveau que nous avons produit pour gérer les fichiers tar.

4. ENVOYER UN CONTENU SAISI À PARTIR DU TERMINAL

Enfin, pour terminer, voici le code pour envoyer du contenu saisi depuis le terminal :

Fichier

```
shutil.copyfileobj (sys.stdin, self.wfile)
```

Vous vous attendiez à quelque chose de plus complexe ? Vous voilà rassuré. Grâce au fait qu'en Python, un fichier et une sortie ou entrée standard partagent une même interface (au sens pythonique du terme), les choses restent simple à utiliser.

Du coup, ça ne valait pas la peine de faire une partie entière de l'article sur cette seule ligne de code !

Sauf si, évidemment, l'auteur souhaitait mettre une emphase sur la simplicité avec laquelle on peut faire quelque chose qui, de premier abord, semble vraiment complexe.

CONCLUSION

On peut noter que le code qui est présenté ici a été légèrement retouché pour correspondre à la syntaxe de Python 3. Le vrai code est en Python 2 et utilise malheureusement quelques anti-patterns dont il serait préférable de se débarrasser et quelques modules qui sont obsolètes. D'un autre côté, grâce à la compatibilité ascendante, il fonctionne toujours avec la dernière version de Python2 et peut tourner sur des machines qui ont une version très ancienne de Python 2, comme la 2.3, par exemple.

Cela illustre, encore une fois, le dilemme qui se pose aux développeurs lorsqu'ils doivent faire le choix entre un code à jour, profitant des dernières innovations et permettre à leur application d'être utilisée par le plus grand nombre. Certains placent le curseur à un extrême et d'autres à l'autre extrême, mais je ne suis pas certain qu'il y ait une réponse satisfaisante pour toutes les parties sur cette problématique. Et la problématique concerne aussi bien les versions de Python à supporter que le respect des diverses normes réseau, internet ou autre RFC qui pullulent dans tous les domaines.

Cependant, si on se recentre d'un point de vue réseau, ce programme démontre sans ambiguïté la facilité avec laquelle il est possible de faire quelque chose de complexe en seulement quelques lignes de Python.

Je vous invite donc à lire le code entier de ce programme pour vous faire une opinion plus éclairée et également vous en inspirer pour vos propres développements. ■

RÉFÉRENCES

[1] Adresses IP réservées :
https://en.wikipedia.org/wiki/Reserved_IP_addresses

[2] Codes HTTP :
https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP

VISITEZ NOTRE NOUVELLE BOUTIQUE ET DÉCOUVREZ NOS GUIDES !

LINUX MAGAZINE / FRANCE | GNU LINUX PRATIQUE ESSENTIEL | MISC Multi-System & Internet Security Cookbook | HACKABLE MAGAZINE | Open Silicium

ET VOUS ?
COMMENT LISEZ-VOUS VOS MAGAZINES PRÉFÉRÉS ?

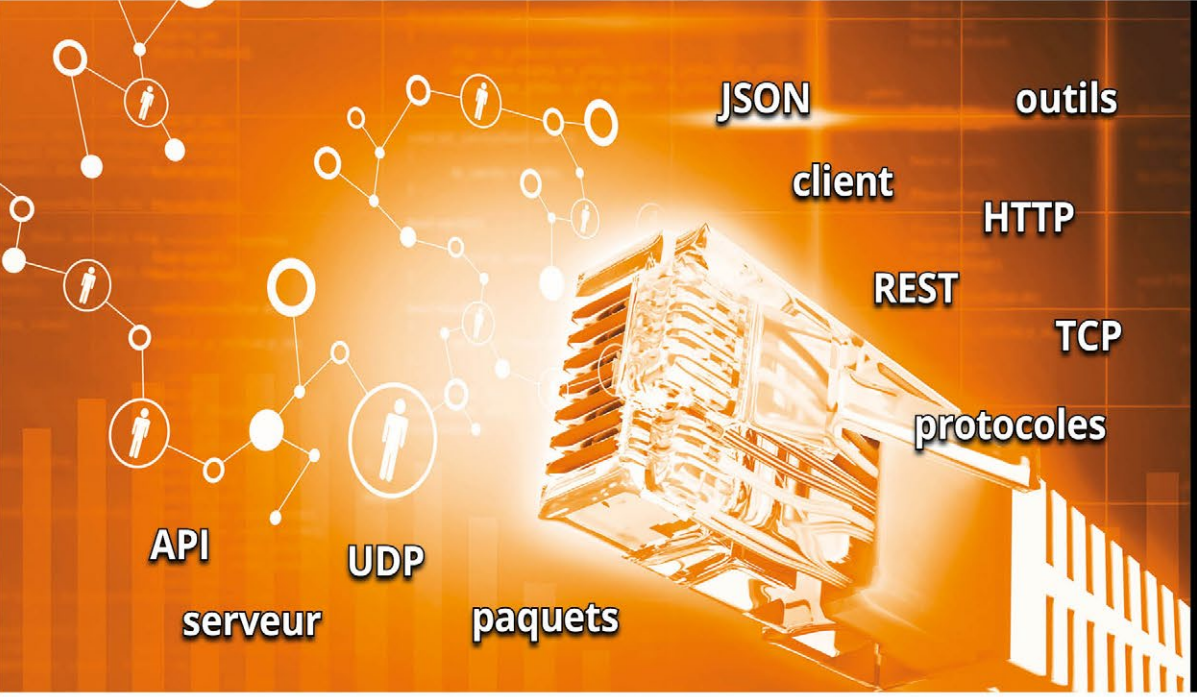
« Moi, je les lis en version **PAPIER** ! »

« Moi, je les lis en version **PDF** ! »

« Moi, je consulte la **BASE DOCUMENTAIRE** ! »

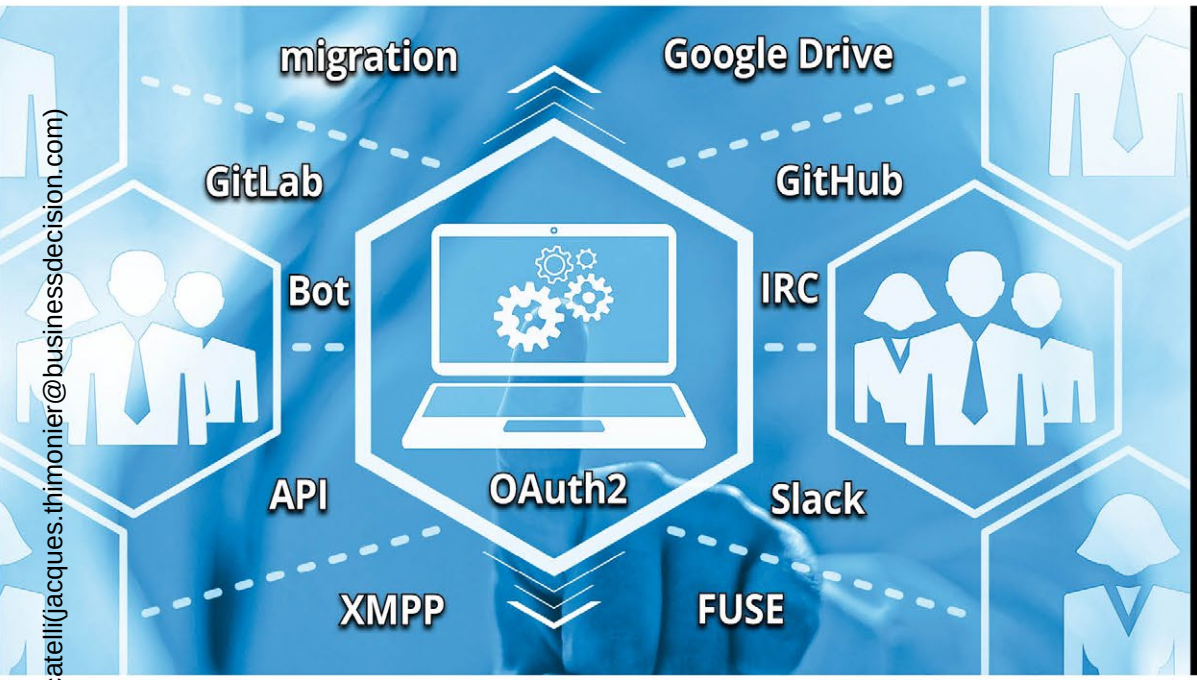
RENDEZ-VOUS SUR **www.ed-diamond.com**

POUR DÉCOUVRIR TOUTES LES MANIÈRES DE LIRE VOS MAGAZINES PRÉFÉRÉS !



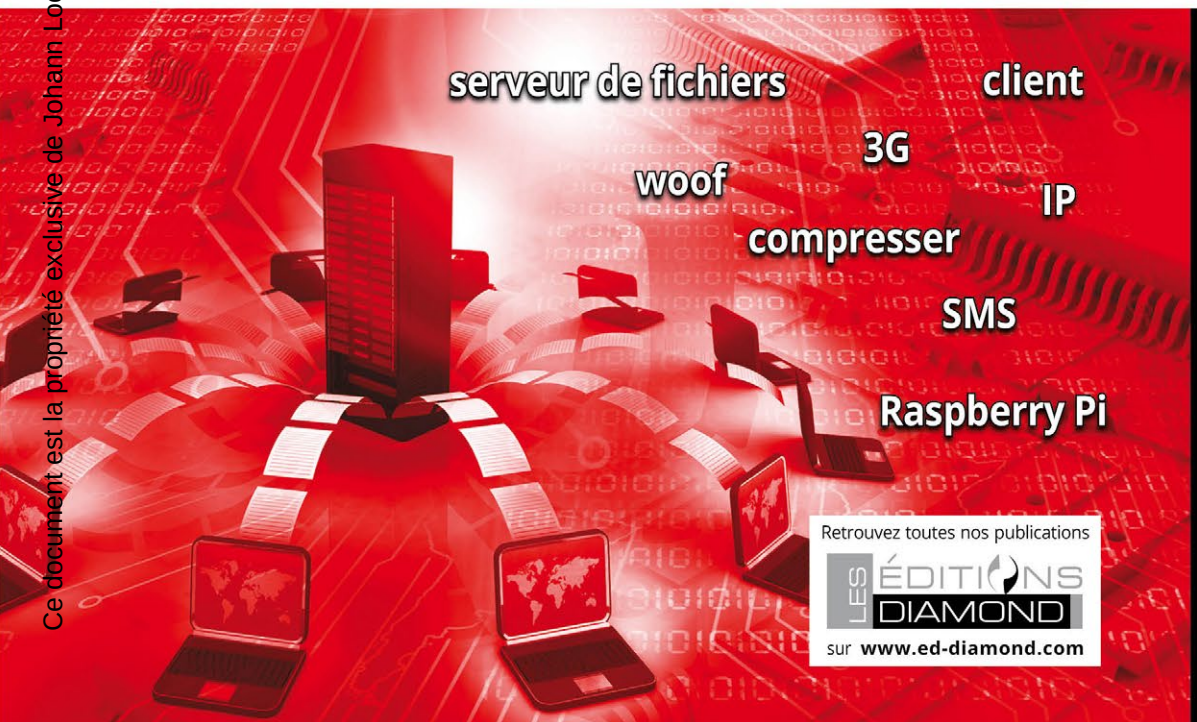
DÉMARREZ...

la programmation réseau en Python avec les modules essentiels



CRÉEZ...

- un système de migration des rapports de bug de GitHub à votre GitLab
- un driver FUSE pour Google Drive
- un bot IRC
- un robot Slack
- un client XMPP



PROGRESSEZ...

en créant un script communiquant par SMS et en analysant un serveur de fichiers