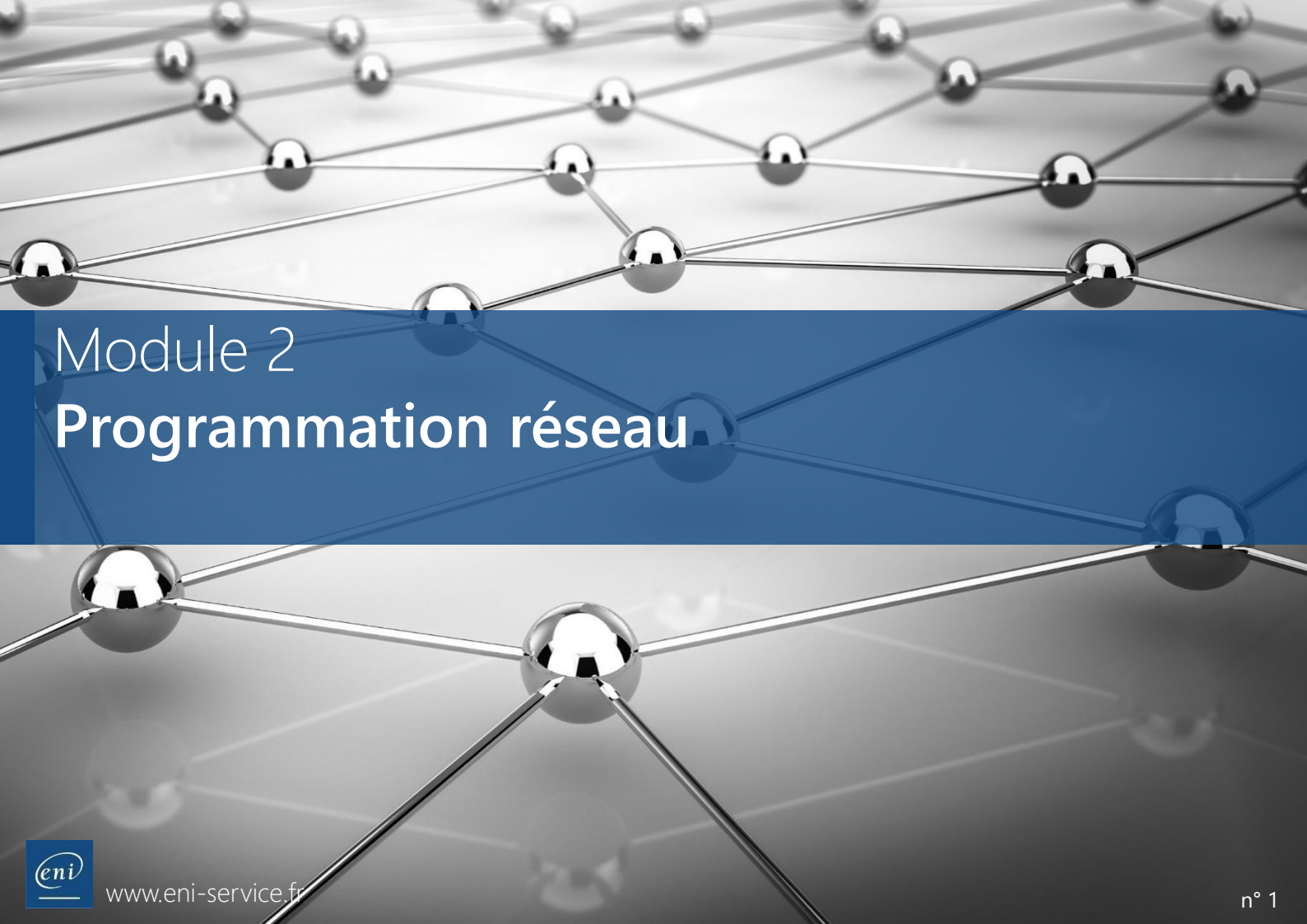


Module 2
Programmation réseau

Contenu du module

- Principes fondamentaux de programmation réseau
  - Les sockets
  - Implémentation des sockets en Python
- Les protocoles réseaux
- Implémentation des protocoles TCP/IP en Python
  - Les modules associés pour leur mise en œuvre
  - HTTP
  - FTP, SFTP
  - IMAP, SMTP, POP3
  - ...

Sockets

Les sockets

- TCP est l'acronyme de Transmission Control Protocol, soit protocole de contrôle de transmission et a été développé en 1973 et documenté au sein de la RFC 793. Il est situé dans le modèle OSI au sein de la couche transport et est répandu de par sa fiabilité.
  - Il se caractérise par la manière de mettre en place la synchronisation entre client et serveur et par le découpage en segments des octets à transmettre, chaque segment étant parfaitement identifiable et disposant d'un système de contrôle d'intégrité qui fait que celui qui reçoit un paquet peut savoir que le paquet est corrompu et le redemander.
- Le flux TCP utilise les sockets et c'est le module éponyme de Python qui permet de travailler avec TCP.
- Les données qui sont transmises d'un serveur à un client ou d'un client à un serveur sont des octets.

Implémentation des sockets en Python

- Le module `socket` permet de programmer des accès réseau bas niveau en Python.
- Dans ce module, nous présenterons uniquement l'utilisation des sockets TCP/IP mais le module `socket` permet, en utilisant la même API, de créer et d'utiliser des sockets Unix, des sockets Bluetooth…
- L'API socket suppose une architecture client/serveur.
  - Le serveur est en attente d'une connexion d'un client.
  - Des données peuvent être lues et écrites aussi bien par le client que par le serveur.

Sockets : Implémentation d'un serveur

- Pour implémenter un serveur, il suffit d'appeler la fonction `socket()` du module du même nom.
  - Les constantes `socket.AF_INET` et `socket.SOCK_STREAM` passées en paramètres permettent de spécifier respectivement que l'on veut créer une socket IP pour une transmission par paquets, c'est-à-dire une socket TCP.
  - Une socket doit être fermée après usage. Le plus souvent, la socket est crée dans une instruction `with` pour s'assurer qu'elle sera fermée à la fin du bloc.
- On appelle ensuite les méthodes suivantes :
  - `bind()`
    - Pour spécifier sous la forme d'un tuple le nom ou l'adresse de l'hôte et le port utilisés par cette socket.
  - `listen()`
    - Pour spécifier qu'il s'agit d'une socket serveur et le nombre de connexions en attente qui peuvent être tolérées avant rejet par le système.
  - `accept()`
    - Cet appel suspend l'exécution du thread jusqu'à ce qu'une connexion entrante se produise pour cette socket. Alors, le programme est réveillé et cette méthode retourne une socket de communication avec le client ainsi que l'adresse du client.

Socket Serveur

- Implémentation minimale

```python
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind(("localhost", 8000))
    s.listen(1)
    while True:
        conn, address = s.accept()
        with conn:
            buff = conn.recv(512)
            message = buff.decode('utf-8')
            conn.sendall(f"echo : {message}".encode('utf-8'))
```

- À partir de la socket de connexion du client, il est possible de lire les données reçues grâce à la méthode `recv()` et de répondre avec la méthode `sendall()`.

Sockets : Implémentation d'un client

- La même fonction `socket()` est utilisée pour créer une socket cliente.
- Une fois la socket créée, on appelle la méthode `connect()` pour se connecter au serveur en indiquant son adresse et son port.
- À partir de la socket client, il est possible :
  - D'envoyer des données avec la méthode `sendall()`
  - De lire la réponse du serveur grâce à la méthode `recv()`

Socket cliente

- Implémentation minimale

```
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect(("localhost", 8000))
    s.sendall("Hello world".encode('utf-8'))
    buff = s.recv(512)
    print(buff.decode())
```

Serveur réseau et traitements concurrents

- Dans l'implémentation d'un serveur réseau se pose la problématique de pouvoir répondre simultanément à plusieurs clients
  - La programmation concurrente et asynchrone entre en œuvre !
- L'idée étant de monopoliser un thread pour chaque demande cliente entrante
- On pourrait :
  - Créer un nouveau thread pour chaque demande
  - L'associer au traitement
- Mais :
  - Peu économe en ressources
  - On ne connait pas à l'avance le nombre de thread simultanés nécessaires
- Ne pourrait-on pas réutiliser les threads une fois qu'ils ont terminé le traitement d'une demande ?

Le « pooling » de threads

- Dans l'idée de pouvoir recycler des threads pour des traitements similaires, la notion de « pooling » est pertinente
- Il s'agit de créer un nombre de threads à l'avance et de les rendre disponible dans un ensemble : le « pool »
- Lorsqu'un traitement doit être effectué, un thread est pris dans le pool, puis remis lorsque le traitement est terminé
  - Si tous les threads sont occupés alors qu'une nouvelle demande de traitement arrive, cette demande est mise en attente
- En Python, cela est rendu possible grâce à la classe `Pool` du module `multiprocessing`
  - Elle implémente un paquet de threads créés grâce à la classe `Thread` du module `threading`
  - La méthode `apply()` du pool permet d'associer une demande entrante à l'exécution d'une fonction portée par un thread du pool

Le module `multiprocessing`

- La classe `Pool`
  - `Pool(processes=os.cpu_count(), initializer=None, initargs=None, maxtasksperchild=None)`
    - `processes` : Le nombre de threads à utiliser. Si `processes` est `None`, le nombre renvoyé par `os.cpu_count()` est utilisé
    - `initializer` : Si différent de `None`, chaque thread appellera `initializer(*initargs)` en démarrant.
    - `maxtasksperchild` : Le nombre de tâches qu'un thread peut traiter d'être supprimé et remplacé, pour permettre aux ressources inutilisées d'être libérées. Par défaut `maxtasksperchild` est `None`, ce qui signifie que le thread vit aussi longtemps que le pool.
- La méthode `apply()`
  - `apply(func, args=None)`
    - Appelle la fonction référencée par `func` avec les arguments `args`. Bloque jusqu'à ce que le résultat soit prêt.

Un serveur réseau multithread

- Dans un soucis de traitement simultané de plusieurs requêtes, le serveur réseau précédent peut être adapté de la manière suivante :

```
import multiprocessing
import socket


def handle(conn, address):
    with conn:
        buff = conn.recv(512)
        message = buff.decode('utf-8')
        conn.sendall(f"echo : {message}".encode('utf-8'))


with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind(("localhost", 8000))
    s.listen(1)
    with multiprocessing.Pool(10) as pool:
        while True:
            conn, address = s.accept()
            pool.apply(handle, (conn, address))
```

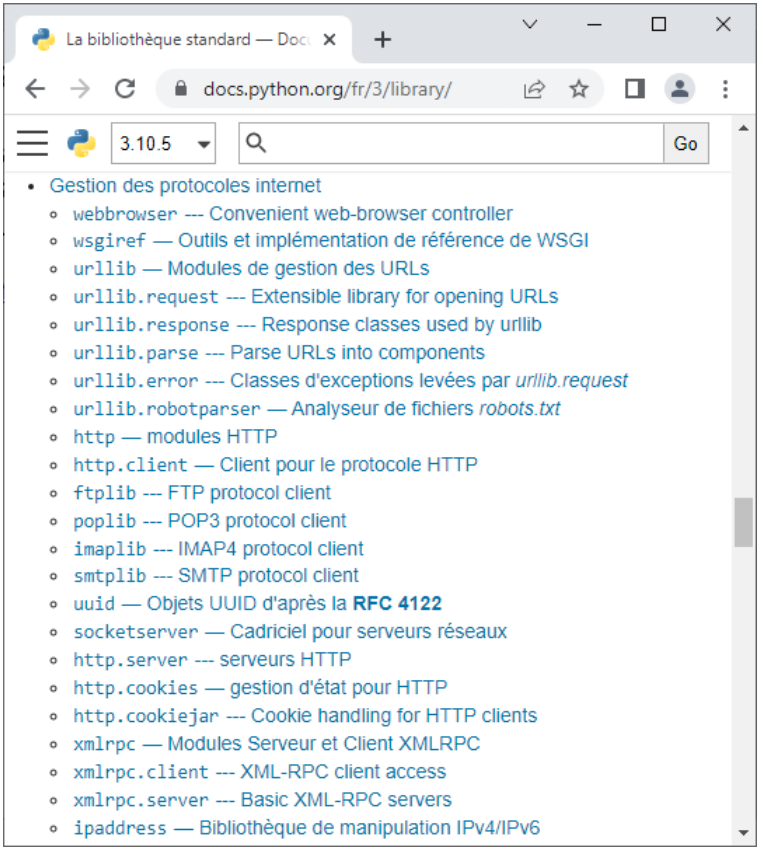
La fonction `handle()` est appliquée pour chaque demande entrante grâce à un nouveau thread pris du pool.

Cette fonction pourrait déclencher l'exécution de coroutines `asyncio` pour réaliser des traitements asynchrones !

Protocoles réseaux

Implémentation des protocoles TCP/IP en Python

- La bibliothèque standard regorge de modules prenant en charge les protocoles réseaux les plus courants
- Ils permettent d'éviter une implémentation trop bas niveau à base de sockets
- https://docs.python.org/fr/3/library/internet.html



HTTP

- Python fournit en standard de nombreux modules pour prendre en charge le protocole HTTP, aussi bien côté client que côté serveur
- Client
  - `http.client`
    - https://docs.python.org/3/library/http.client.html
    - Définit les classes qui implémentent le côté client des protocoles HTTP et HTTPS. Il est rarement utilisé directement — le module `urllib.request` l'utilise pour gérer les URL qui utilisent HTTP et HTTPS.
  - `urllib.request`
    - https://docs.python.org/3/library/urllib.request.html
    - Définit des fonctions et des classes qui aident à ouvrir des URL (principalement HTTP). Il permet de gérer l'authentification de base et digest, redirections, cookies, ...
- Serveur
  - `http.server`
    - https://docs.python.org/fr/3/library/http.server.html
    - Définit des classes permettant d'implémenter des serveurs HTTP.

Implémentations HTTP en Python

- Si l'implémentation d'un serveur HTTP en Python reste un besoin marginale, de nombreux serveurs, projets, librairies, existent déjà, il est beaucoup plus fréquent de devoir implémenter un client
  - Gestionnaire de téléchargement
  - Consommation d'API REST
  - ...
- Le module `urllib.request` peut complètement convenir à ces besoins.
- Il offre :
  - Une interface simple pour l'ouverture d'une connexion
  - Des objets pour manipuler la requête et la réponse
  - La prise en charge des standards HTTP : authentification, cookies, ...

Le module `urllib.request`

- Associé à une gestion concurrente à l'aide du module `asyncio`, `urllib.request` peut permettre de créer un client HTTP multi-threadé efficace

```python
import asyncio
import os
import urllib.request


async def download(url):
    response = urllib.request.urlopen(url)
    filename = os.path.basename(url)

    with open(filename, 'wb') as file:
        file.write(response.read())
    return f"Téléchargement de {filename} terminé."


async def crawler(urls):
    coroutines = [asyncio.create_task(download(url)) for url in urls]
    completed, pending = await asyncio.wait(coroutines)

    for item in completed:
        print(item.result())


if __name__ == '__main__':
    urls = [
        """ URLs des éléments à télécharger... """
    ]
    asyncio.run(crawler(urls))
```

Réflexions…

- Plusieurs problèmes avec l'exemple précédent :
- La fonction `download()` n'est pas vraiment une coroutine.
  - Elle n'est pas asynchrone : Pas d'usage de `await`
- `urllib` n'est pas asynchrone
  - Pour les mêmes raisons…

- Même si fonctionnel, l'exemple peut être amélioré
- Utilisation de la librairie `aiohttp`
  - Librairie cliente HTTP asynchrone
  - Elle ne fait pas partie de la libraire standard mais peut être installée via `pip`
    - `pip install aiohttp`

aiohttp

- Un client concurrent

```python
import aiohttp
import asyncio
import async_timeout
import os


async def download(session, url):
    async with async_timeout.timeout(10):
        async with session.get(url) as response:
            filename = os.path.basename(url)
            with open(filename, 'wb') as file:
                data = await response.content.read()
                file.write(data)
            return f"Téléchargement de {filename} terminé."


async def crawler(urls, loop):
    async with aiohttp.ClientSession(loop=loop) as session:
        coroutines = [asyncio.create_task(download(session, url)) for url in urls]
        completed, pending = await asyncio.wait(coroutines)

    for item in completed:
        print(item.result())

if __name__ == '__main__':
    urls = [
        """ URLs des éléments à télécharger... """
    ]
    loop = asyncio.get_event_loop()
    loop.run_until_complete(crawler(urls, loop))
```

FTP, SFTP

- `ftplib`
- https://docs.python.org/3/library/ftplib.html

- Ce module définit la classe FTP et quelques éléments associés.
- La classe FTP implémente le côté client du protocole FTP.
- Elle est utilisée pour écrire des programmes Python qui exécutent une variété de tâches FTP automatisées
  - Elle est également utilisé par le module `urllib.request` pour gérer les URL qui utilisent FTP.
- L'encodage utilisé par défaut est UTF-8

Exemple : ftplib

```
from ftplib import FTP

# Connexion à l'hôte
ftp = FTP('ftp.us.debian.org')
# Authentification : user anonymous, passwd anonymous@
ftp.login()
# Changement de répertoire
ftp.cwd('debian')
# Lister le contenu du répertoire courant
for line in ftp.retrlines('LIST'):
    print(line)
# Récupération du contenu du fichier README
with open('README', 'wb') as fp:
    ftp.retrbinary('RETR README', fp.write)

ftp.quit()
```

IMAP, SMTP, POP3

- Python propose des bibliothèques natives pour gérer les protocoles de messagerie électronique
- `poplib`
  - http://docs.python.org/py3k/library/poplib.html
  - Permet la gestion des protocoles POP3 et POP3S (non sécurisé et sécurisé) via les classes `POP3` et `POP3_SSL`
- `imaplib`
  - https://docs.python.org/3/library/imaplib.html
  - Permet de gérer le protocole IMAP grâce aux classes `IMAP4` et `IMAP4_SSL`
- `smtplib`
  - https://docs.python.org/3/library/smtplib.html
  - Permet de gérer le protocole SMTP grâce aux classes `SMTP` et `SMTP_SSL`

Travaux Pratiques