UE PE Web

Adrien Luxey-Bitri[1,2] and Boris Baldassari[2,3]

[1]Université de Lille
[2]Association Deuxfleurs
[3]Eclipse Foundation Europe

Version : 11 avril 2024

Table des matières

Introduction

Ce module-projet enseigne la conception d'un serveur web en Python, en demandant un minimum de connaissances préalables. Par ce projet, l'objectif pédagogique est de faire une première expérience de plusieurs domaines de l'informatique, notamment le génie logiciel, les systèmes d'exploitation et le réseau.

Un serveur web est un logiciel fondamental d'Internet dont le rôle est de transmettre des données avec le protocole HyperText Transfer Protocol (HTTP), typiquement des pages HyperText Markup Language (HTML), feuilles de style Cascading Style Sheets (CSS), scripts JavaScript, et des média.

Le serveur web écoute le réseau, attend des requêtes de clients, et y répond avec les informations demandées ou un code d'erreur approprié.

Objectifs pédagogiques

- Acquérir les principes du développement logiciel, ses outils et méthodes.
  - Utiliser le paquetage de logiciels en Python.
  - Utiliser le versionnage de code source.
  - Identifier les bonnes pratiques du développement logiciel : conventions de codage & de nommage, documentation, tests, intégration continue.
- Manipuler le système d'exploitation Linux.
- Acquérir les concepts fondamentaux du réseau, en particulier les protocoles TCP et HTTP.
- Découvrir le déploiement de logiciels en réseau sur un site distant.
- Travailler en petit groupe : communiquer efficacement, répartir et prioriser le travail.
- Gérer son apprentissage en autonomie : identifier les difficultés/lacunes, trouver des ressources et solutions (voir appendice A).

Méthode de travail

- Groupes de travail de 1 à 2 personnes.
- 2h de travail par semaine hors séance.
- Chaque semaine, vous devrez rendre individuellement l'URL du dernier commit du projet en cours.

Évaluation

Ce module de découverte n'a pas pour objectif d'évaluer vos connaissances et compétences. L'évaluation sera par contre sans pitié face au plagiat et au *free-riding* (e.g. laisser tout le travail au binôme et se l'approprier). Tout simplement : *soyez présent·e, participez, travaillez régulièrement, et vous aurez une bonne note.* Nous évaluerons :

- L'**assiduité** : venir en classe, essayer, progresser.
- La **prise d'initiative** et l'**autonomie** : la réponse à votre question est certainement dans la documentation, le manuel ou sur Internet, voir appendice A.
- Plusieurs **livrables** , évalués individuellement.
- La **participation individuelle** au développement, qui prendra en compte l'historique des commits et les résultats d'éventuelles interrogations de cours.

Méthode de développement

Parce que le module propose de réaliser un projet de développement logiciel, nous apprendrons et appliquerons les méthodes et bonnes pratiques du génie logiciel :

- L'outil de versionnage git sera utilisé tout au long du projet *via* la forge universitaire `gitlab-etu.univ-lille.fr`.
- Notre projet sera empaqueté comme un module Python professionnel, avec l'outil PDM.
- Chaque fonctionnalité devra être validée par un test unitaire approprié.

— Nous utiliserons l'intégration continue ou Continuous Integration (CI), pour effectuer les tests et assurer la qualité de notre logiciel à chaque publication du code sur la forge.
— Chaque projet (et chacune de ses fonctions) devra contenir une documentation en anglais de préférence.

1 Mise en place d'un projet Python

1.1 Prise en main du système d'exploitation

> On ne peut vaincre sa destinée.
>
> Jean Racine

L'intégralité du module sera réalisé sur le système d'exploitation (Operating System (OS)) Linux. Un OS est l'interface entre les composants physiques d'un ordinateur et les applications qui fonctionnent dessus (comme représenté en figure 1). Il a de nombreux rôles : le partage équitable des ressources (stockage, mémoire, processeur, réseau...) entre applications, la transmission des événements du matériel aux applications (clic de la souris, réception d'un message sur le réseau...), l'orchestration du démarrage et de l'extinction...

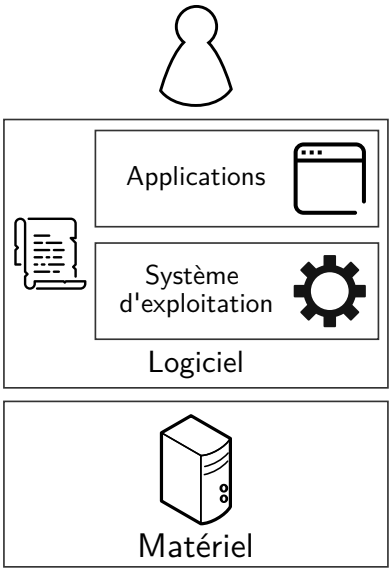


FIGURE 1 – Architecture d'un système d'exploitation

Système de fichiers La plupart des systèmes d'exploitation organisent tout le contenu des périphériques de stockage (disques ou SSD) grâce à un *système de fichiers*, constitué de dossiers (ou répertoires) et fichiers.

Chaque dossier ou fichier y est rangé dans une hiérarchie de sous-dossiers, qui remonte jusqu'à la *racine* : le dossier ultime contenant tous les autres fichiers et dossiers. Sur Windows, chaque disque a droit à sa racine : C:, D:... Sur Linux, il existe une racine unique : `/` (*slash*), chaque fichier ou dossier du système en est un descendant.

Chaque dossier ou fichier dispose d'une *adresse* unique (aussi appelée *chemin*), constituée (de droite à gauche) du nom du fichier ou dossier en question, et de la liste de tous ses dossiers *parents* jusqu'à la racine. On sépare les dossiers de l'adresse par un anti-slash \ sur Windows, et par un slash `/` sur Linux.

Ainsi, sur Linux, l'adresse `/home/adrien/Images/chat.jpg` fait référence à un fichier `chat.jpg`, rangé dans le dossier `Images`, appartenant lui-même à `adrien` (le dossier personnel de l'utilisateur Adrien), qui appartient au répertoire `home`, dans la racine `/`.

Terminal Aussi appelé *invite de commande*, le terminal permet d'interagir avec l'OS et les applications *via* une interface textuelle en exécutant des commandes. Bien qu'il existe sur Windows, le terminal est particulièrement fondamental chez Linux. Il y existait avant les gestionnaires de bureau (qui affichent les applications sous forme de fenêtres cliquables) ; toutes les applications reposent dessus.

Un terminal s'exécute toujours depuis un dossier, appelé *dossier courant* ou *working directory*. On démarre toujours dans notre *répertoire personnel* ou *home*, représenté par un Tilde ~, et qui se trouve généralement à l'adresse `/home/monNom`. Pour afficher le dossier courant, on utilise la commande `pwd` (pour *Print Working Dir*).

Exercice 1.1 : Lancer un terminal et afficher le dossier courant

1. Lancez l'application Terminal. Par exemple en appuyant sur la touche Windows ⊞, en tapant `terminal`, et en appuyant sur Entrée ↵.
2. Tapez `pwd` suivi de Entrée ↵. La ligne affichée est votre répertoire courant.

Laissez votre terminal affiché entre les exercices.

Une autre commande courante est `ls` (pour *LiSt*). Elle affiche le contenu du dossier courant.

Exercice 1.2 : Afficher le contenu du répertoire courant

1. Tapez `ls` suivi de Entrée ↵. Le contenu de votre répertoire personnel s'affiche.

Chemins relatifs et absolus On peut exécuter `ls CHEMIN` pour afficher le contenu du répertoire chemin (on dit qu'on passe `CHEMIN` *en paramètre* à `ls`). Mais, quand je suis dans mon répertoire personnel, et que je veux afficher le contenu du sous-dossier `Images`, je n'ai pas du tout le courage d'écrire : `ls /home/adrien/Images` (ce qui est le chemin *absolu* dudit dossier). Les informaticien·nes sont fainéant·es comme moi, iels ont donc inventé les chemins *relatifs* : si je suis dans `/home/adrien`, je n'ai qu'à taper `ls Images` pour arriver à mes fins (notez bien l'absence de slash `/` devant `Images`). Car `Images` est le chemin *relatif* de `/home/adrien/Images` par rapport à `/home/adrien`.

On va utiliser la commande `cd` (*Change Directory*) qui sert à changer de répertoire courant, pour expérimenter les chemins relatifs.

Exercice 1.3 : Se déplacer avec le terminal

1. Rendez-vous dans un sous-dossier de votre répertoire personnel en tapant `cd CHEMIN` où `CHEMIN` est le nom du dossier choisi (par exemple `Documents`).
2. Tapez `pwd` pour vérifier votre répertoire courant.

On utilise un point `.` pour représenter le répertoire courant. `ls .` est donc équivalent à `ls` tout seul. deux points à la suite `..` pour représenter le dossier parent. On enchaîne les groupes de deux points séparés par des slash `/` pour remonter plus haut dans la hiérarchie. Par exemple, `../..` représente le dossier parent du dossier parent. Enfin, le caractère étoile `*`, appelé *wildcard*, signifie « tout le contenu d'un dossier ». `../*` fait donc référence à tous les fichiers et dossiers du répertoire parent.

Exercice 1.4 : Utilisation de `..`

1. Tapez `pwd` pour connaître votre dossier actuel.
2. Utilisez `cd CHEMIN` pour aller dans le dossier `/home` en utilisant judicieusement `..` dans `CHEMIN`.
3. Tapez `pwd` pour vérifier que vous êtes bien dans `/home`.
4. Utilisez `cd CHEMIN` avec le bon `CHEMIN` pour retourner dans votre dossier personnel.

En bref : Les chemins relatifs et absolus

— Un chemin absolu commence par un slash `/`, un chemin relatif, non.
— On utilise `.` pour symboliser le répertoire courant, et `..` pour symboliser le répertoire parent.
— Le caractère étoile `*` appelé *widlcard* signifie « tout le contenu du répertoire ».

L'auto-complétion est une preuve supplémentaire de la paresse des informaticien·nes : on peut très souvent compléter une commande ou un paramètre en appuyant sur Tab ↹ après avoir tapé les quelques premiers caractères. Dans l'exemple précédent, j'aurais pu taper seulement `ls Im`, appuyer sur Tab ↹, admirer `Images` s'écrire tout seul, et appuyer sur Entrée ↵.

Il est recommandé d'user et d'abuser de l'auto-complétion, car c'est notamment elle qui donne à l'interface *a priori* aride du terminal son efficacité et son charme.

Références : Pour aller plus loin

— Le système de fichiers et le terminal, cours en ligne, Univ Lille, 2023.
— Utilisation d'Unix — compléments, cours en ligne, Univ Lille, 2023.
— Initiez-vous à Linux, cours en ligne, OpenClassrooms.

Commandes de base Voici une liste des commandes les plus utiles pour ce module. Elles sont suivies de leurs paramètres éventuels, entre crochets `[...]` s'ils sont facultatifs. Ces paramètres, séparés par des espaces, modifient le comportement de la commande comme indiqué. Tous les chemins décrits peuvent représenter un chemin relatif ou absolu. On affiche des points de suspensions après un paramètre `CHEMIN...` pour désigner qu'on peut le renseigner plusieurs fois (séparés par des espaces) : `CHEMIN1 CHEMIN2`. Les paramètres commençant par un tiret `-` qui ne sont pas suivis d'un paramètre sont appelés des *options*, et peuvent être mis à la suite : `ls -l -a` est équivalent à `ls -la`.

— `pwd` : affiche le répertoire courant.
— `ls [-l] [-a] [CHEMIN...]` : liste le contenu d'un ou plusieurs répertoires.
  — `CHEMIN...` : s'il n'est pas fourni, liste le contenu du répertoire courant. S'il est fourni, celui du ou des répertoires désignés par `CHEMIN...`
  — `-l` : affiche des informations plus détaillées
  — `-a` : affiche les fichiers et dossiers cachés (commençant par un `.`)
— `touch FICHIER...` : crée un ou plusieurs fichiers vides à l'adresse `FICHIER...`.
— `mkdir [-p] CHEMIN...` : crée un ou plusieurs dossiers vides à l'adresse `CHEMIN...`.
  — `-p` : crée des répertoires parents (dossiers intermédiaires) si nécessaire
— `cd [CHEMIN]` : change le répertoire courant.
  — `CHEMIN` : s'il n'est pas fourni, nous déplace dans le répertoire personnel `~`. S'il est fourni, nous déplace dans le répertoire désigné par `CHEMIN`.
— `cp [-r] SOURCE... DESTINATION` : copie le ou les fichiers désignés par `SOURCE...` dans le répertoire `DESTINATION`.
  — `-r` : copie *récursivement*, ce qui autorise `SOURCE...` à être un dossier
  — `SOURCE...` : un ou plusieurs chemins relatifs ou absolus à copier
  — `DESTINATION` : un chemin relatif ou absolu vers un dossier
— `mv SOURCE... DESTINATION` : déplace le ou les fichiers ou dossiers désignés par `SOURCE...` dans le répertoire `DESTINATION`.
— `cat FICHIER` : affiche le contenu de FICHIER dans le terminal.
— `unzip FICHIER [-d DESTINATION]` : Extrait une archive ZIP désignée par `FICHIER`.
  — `-d DESTINATION` : Si le paramètre n'est pas renseigné, extrait `FICHIER` dans le répertoire courant. Sinon, extrait dans le répertoire désigné par `DESTINATION`.
— `man COMMANDE` : affiche le manuel utilisateur de `COMMANDE`.

En bref : Lisez de la doc

Pour maîtriser le terminal, prenez l'habitude de chercher de l'aide (voir appendice A) :
— Aller sur Internet pour chercher comment réaliser une tâche,

— Lire l'aide des commandes accessibles *via* le paramètre `--help`,
— Lire le manuel des commandes *via* la commande `man COMMANDE`.

Exercice 1.5 : Télécharger le zip et le dézipper au bon endroit

1. Créez un répertoire `~/dev` avec `mkdir`.
2. Téléchargez le fichier `intro_python.zip` sur Moodle. Il devrait être sauvegardé par exemple dans votre dossier `~/Downloads`.
3. Extrayez l'archive `intro_python.zip` dans votre répertoire `~/dev` avec la commande `unzip`.
4. Vérifiez qu'un dossier `~/dev/intro_python` a bien été créé avec `cd` et `ls`.

1.2 Git ou le versionnage des projets logiciels

Git est un gestionnaire de version omniprésent dans le monde informatique. Nous l'utiliserons tout au long du module, et vous rendrez à la fin de chaque semaine le dernier *commit* (point de sauvegarde) de votre travail sur Moodle.

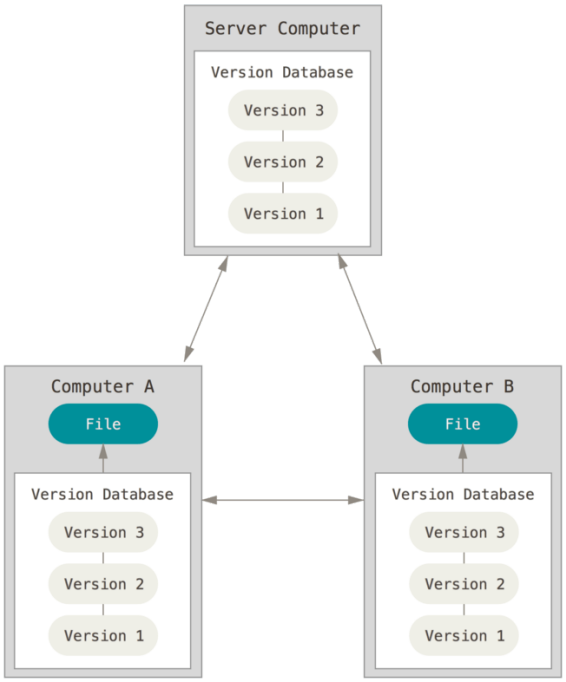


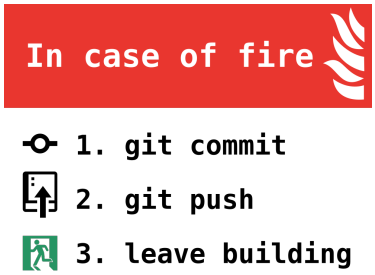
FIGURE 2 – Architecture d'un système d'exploitation (CC-NC-SA 3.0 Git book)

Comme montré en figure 2, , git permet à plusieurs développeurs de coordonner leur développement, en enregistrant pour chaque fichier ses différentes versions , puis en partageant l'historique des version entre les développeurs (« Computer A & B »), et avec un serveur appelé *forge* (« Server Computer »).

Le développement d'un logiciel est à l'échelle d'un dossier, qui contient une myriade de fichiers. Ce dossier versionné s'appelle un *dépôt*. Quand on développe, on enregistre régulièrement son travail sur l'intégralité du projet dans des *commits*. De temps en temps, on récupère (*pull*) la dernière version du dépôt, avant d'envoyer (*push*) nos dernières modifications.

```
# On commence par récupérer la dernière version du code
$ git pull
# On développe, on développe...
# Régulièrement, on vérifie l'état du travail ...
$ git status
# ... et on enregistre nos modifications, en ajoutant un message décrivant notre contribution
$ git commit -am "amélioration de l'interface utilisateur"
# Quand on a fini, on récupère une éventuelle nouvelle version du code, avant d'envoyer notre contribution
$ git pull
$ git push
```

En bref : Il ne faut pas croire tout ce qu'on voit écrit (il manque `git pull`)



Un message que vous pourrez lire dans certaines salles du M5.

Dans la pratique, on va créer un *dépôt* git sur nos machines dans notre répertoire de développement, ce qui permettra de versionner tous les fichiers qui en font partie. Puis, nous initialiserons un dépôt vide sur la forge git de la FST, avant d'y envoyer notre code.

Exercice 1.6 : Initialisation du dépôt local

- À l'aide du terminal et de `cd`, rendez-vous dans le répertoire `intro_python` que nous avons dézippé précédemment.
- Lancez la commande `git init` pour initialiser un dépôt git vide.
- Lancez `git add .` pour ajouter tout le répertoire courant au dépôt.
- Faites un premier commit : `git commit -am "Premier commit"`

Exercice 1.7 : Création d'un dépôt vide sur `gitlab`

1. Rendez-vous sur https://gitlab-etu.fil.univ-lille.fr/
2. Cliquez sur New project > Blank project.
3. Entrez « UEPE-web-tp1 » comme nom de projet.
4. **Décochez** « Initialize repository with a README ».
5. Cliquez sur Create project.
6. Ajoutez votre binôme et le prof en membres de votre dépôt :
   (a) Cliquez sur Manage > Members.
   (b) Cliquez sur Invite members.
   (c) Cherchez l'utilisateur à l'identifiant « luxey » et ajoutez-le en « Maintainer ».

(d) Faites de même pour votre binôme.

Gardez ouvert l'onglet de navigateur de votre dépôt, il resservira bientôt.

Communiquer avec la forge On ne rentre pas dans une forge comme dans un moulin. Il faut être authentifié. Pour cela, on utilise le protocole Secure SHell (SSH), qui authentifie l'utilisateur avec une paire de clés SSH. On envoie ensuite notre clé publique à la forge, pour qu'elle nous reconnaisse quand on fera des `git pull/push`.

Exercice 1.8 : Création et import sur la forge d'une clé SSH

1. Dans un terminal, lancez la commande `ssh-keygen` pour créer un couple de clés SSH.
   Appuyez plusieurs fois sur Entrée ↵ pour garder les paramètres par défaut.
2. Tapez `cat ~/.ssh/id_rsa.pub` pour afficher votre clé publique.
3. Sélectionnez le texte obtenu à la souris, et tapez Ctrl+⇧+C pour copier la clé dans le presse-papiers.
4. Rendez-vous sur https://gitlab-etu.fil.univ-lille.fr/-/profile/keys, cliquez sur Add new key, collez (Ctrl+V) votre clé dans le champ « Key », et cliquez sur Add key.

Exercice 1.9 : Envoi du code du dépôt local à la forge

1. Ouvrez un terminal, et rendez vous dans le répertoire `intro_python` avec `cd`.
2. Sur la page web de votre dépôt, copiez-collez les lignes de la section « Git global setup » dans votre terminal. Ces commandes donnent votre identité à git sur votre machine.
3. Dans la section « Push an existing folder », copiez-collez la ligne commençant par `git remote add origin`. Elle informe votre dépôt local de l'adresse de la forge.
4. Il ne reste plus qu'à lancer `git push`.
   Git n'a pas l'air content. En effet, la première fois qu'on pousse une branche locale, il faut indiquer avec quel serveur on veut la synchroniser.
5. Suivez les recommandations de git, en copiant-collant la ligne suggérée dans le terminal.
6. Une fois la commande lancée, on nous demande si on veut faire confiance à `gitlab-etu`. Entrez « yes » puis Entrée ↵.
7. Rechargez la page web de votre dépôt avec F5. Vous devriez y voir le contenu du dossier.

1.3 Prise en main de Python

Le dossier `intro_python` que nous avons téléchargé contient un projet logiciel incomplet. Il dispose de tests qui ne passent pas. Votre mission durant ce premier TP sera de réaliser le code permettant de les accomplir.

Paquetage et gestion des dépendances Le projet empaqueté avec l'outil de paquetage et de gestion des dépendances PDM. C'est un outil de gestion des dépendances utilisé pour le développement de paquets Python : il permet d'installer automatiquement tous les paquets nécessaires à l'exécution ou

au développement d'une application. Dans cette section, nous vous fournissons un projet déjà initialisé avec PDM. Pour notre prochain module, vous aurez à faire vous-même cette étape.

Le paquetage est le fait d'« emballer » un projet logiciel de façon à ce qu'il soit facilement ouvrable, installable et adaptable aux différentes machines sur lesquelles il pourrait être déployé.

La gestion des dépendances est cruciale dans la mesure où aucun logiciel n'existe en isolation : on dépend toujours du logiciel de quelqu'un d'autre, d'autant plus quand on utilise des outils de développements professionnels pour bien empaqueter son code. Logiciels qu'on appelle les *dépendances* d'un projet. Ces dépendances ayant elles-mêmes les leurs, on parle d'*arbre* de dépendances. Gérer cet arbre étant étonnamment compliqué, cette tâche mérite donc son gestionnaire.

Les environnements virtuels En fonction du projet Python sur lequel on travaille, on peut avoir des arbres de dépendances conséquents, avec des versions de dépendances différentes pour chaque projet. Afin de ne pas interférer avec les dépendances Python du système, ou avec les autres projets de développement que nous avons en cours, il est recommandé d'utiliser un *environnement virtuel*. On entend par là le fait d'installer nos dépendances localement à notre projet en cours, dans le dossier même du projet. On peut par exemple utiliser le module Python `venv` [1]. Il permet d'activer (et désactiver) à la demande l'environnement de développement du projet et toutes ses dépendances.

Exercice 1.10 : Création et activation d'un environnement virtuel

1. Exécutez `python3 -m venv venv` pour initialiser un dossier `venv` contenant notre environnement virtuel.
2. Activez-le avec `source venv/bin/activate`

Votre *prompt* (la partie avant le texte de votre ligne de commande) indique désormais que vous vous trouvez dans l'environnement :

```
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $
```

Lorsque nous avons fini de travailler et souhaitons revenir dans un environnement système classique, il suffit de taper la commande `deactivate`, de quitter le terminal ou de taper Ctrl+D. Pareillement, la prochaine fois que nous voulons travailler sur notre projet, il suffira de taper la commande ci-dessus (`source venv/bin/activate`) pour retrouver notre environnement complet, avec toutes ses dépendances.

Attention

À chaque fois que vous démarrez un terminal pour travailler sur votre projet, il faut penser à activer l'environnement virtuel avec `source venv/bin/activate` !

Installation des dépendances Nous devons maintenant installer l'outil de gestion de dépendances PDM. Dans votre environnement virtuel, exécutez la commande `pip` pour installer le package `pdm` :

1. Voir la documentation officielle du module `venv` : https://docs.python.org/3/library/venv.html

Exercice 1.11 : Installez le module pdm

```
(venv) $ pip install pdm
  [...]
(venv) $ pdm
Usage: pdm [-h] [-V] [-c CONFIG] [-v | -q] [-I] [--pep582 [SHELL]] ...
  [...]
```

Nous disposons désormais de PDM installé uniquement dans notre dossier de développement. Comme PDM connaît la liste de toutes les dépendances du projet, nous allons lui demander de les installer pour nous au moyen de la commande `pdm install`.

Exercice 1.12 : Installez les dépendances du projet

```
(venv) $ pdm install
```

Nous pouvons vérifier que toutes les dépendances ont été installées correctement en utilisant la commande `pdm list`. Nous devrions notamment y retrouver `pytest`, `pylint`, `black`, `pdoc`.

Exercice 1.13

1. Exécutez la commande `pdm list`
2. Vérifiez que les dépendances nommées ci-dessus sont présentes. Si oui, notre environnement de développement est prêt. Sinon, faites appel à l'enseignant.

1.3.1 A propos des tests

Les tests sont des scripts Python qui exécutent le code de nos scripts, et s'assurent que le résultat est correct. Ce sont eux qui garantissent, s'ils sont bien faits, que l'application fonctionne et rend les résultats escomptés. L'intérêt d'un test est qu'il soit reproductible n'importe où, y compris de manière automatique, sur d'autres machines, et dans des contextes différents.

Les tests nous permettent de savoir à tout moment si un code (scripts, fonctions) est fonctionnel dans différentes situations (paramètres, variables d'environnement, fuseau horaire, système utilisé, etc.). Nous les utiliserons également pour savoir si une modification du code affecte d'autres fonctionnalités (tests de régression) : tant que les tests passent, la fonctionnalité est réputée bonne pour le contexte ciblé.

Il est donc important d'avoir des tests, et de les rendre autonomes – c'est à dire qu'ils puissent être exécutés n'importe où facilement. Il existe des librairies qui facilitent cette démarche, pour tous les langages et toutes les techniques : jUnit, pytest, selenium par exemple. Des outils d'intégration continue (CI) sont également disponibles pour automatiser et optimiser les tests, et les distribuer sur un parc de machine : Jenkins, GitLab CI, GitHub CI, Travis par exemple.

Des tests sont déjà présents dans le dépôt. Bien entendu, ils ne passent pas, et votre mission durant ce TP sera de modifier le code jusqu'à ce qu'ils fonctionnent.

Exercice 1.14 : Première passe de tests

Exécutez les tests au moyen de la commande `pytest` à la racine du projet.

Cette commande recherche tous les fichiers de test (i.e. commençant par `test_`) dans le répertoire et ses sous-répertoires, puis les exécute. Ci-après la sortie que vous devriez obtenir :

```
(venv) $ pytest
====================================== test session starts ======================================
platform darwin -- Python 3.11.7, pytest-7.4.4, pluggy-1.3.0
rootdir: /home/boris/ue_pe_web/intro_python
configfile: pyproject.toml
collected 5 items

tests/test_intro_python.py FFFFF                                                        [100%]

========================================== FAILURES ==========================================
_____________________________________ test_add_ints _____________________________________

    def test_add_ints():
        """ Tests add_integer with various inputs.
        """
>       assert target.add_ints(2, 3) == 5
E       assert 0 == 5
E        +  where 0 = <function add_ints at 0x10212c360>(2, 3)
E        +    where <function add_ints at 0x10212c360> = target.add_ints

tests/test_intro_python.py:27: AssertionError
_____________________________________ test_format_date _____________________________________
[...]
================================== short test summary info ==================================
FAILED tests/test_intro_python.py::test_add_ints - assert 0 == 5
FAILED tests/test_intro_python.py::test_format_date - AssertionError: assert 1 == ''
FAILED tests/test_intro_python.py::test_get_path_from_http_get[GET /index.html HTTP/2-/index.html] -
AssertionError: assert '' == '/index.html'
FAILED tests/test_intro_python.py::test_get_path_from_http_get[GET /img/logo.svg HTTP/1.1-/img/logo.
svg] - AssertionError: assert '' == '/img/logo.svg'
FAILED tests/test_intro_python.py::test_get_path_from_http_get_exception - Failed: DID NOT RAISE <c
lass 'Exception'>
==================================== 5 failed in 0.05s ======================================
```

Dans la première section, `pytest` nous indique le nom des fichiers de test trouvés, et le nombre de tests réussis ou échoués. Le fichier de test `tests/test_intro_python.py` contient ainsi 5 tests, tous échoués (marqués `F`, comme `False`). Lorsqu'un test est réussi, le `F` est remplacé par un point (`.`).

Ensuite dans la section principale (`FAILURES`), pour chaque test échoué `pytest` nous donne **la trace de l'exécution**, ce qui permet de savoir où l'erreur s'est produite, et **la comparaison des deux résultats** (attendu, ce que l'on souhaite obtenir, et obtenu, ce que l'on obtient réellement).

Le dernier paragraphe (`short test summary info`) nous propose un résumé de tous les tests échoués.

Pas de panique ! Ce résultat est normal, car les fonctions ou leurs tests ne sont pas encore implémentés. C'est à vous de les écrire dans le module et dans le fichier de test, de manière à ce que tous les tests passent. Nous allons voir ensemble, test par test, comment faire passer l'intégralité de nos tests et ainsi développer un module complet et testé.

1.3.2 Implémentation de la fonction d'addition

Le premier test échoué est celui de `test_add_ints`, qui teste la fonction `add_integer`. Sans grande surprise, cette fonction permet d'ajouter deux nombres entiers – voir la section 1.4 page 19 sur les conventions de codage.

En ouvrant le fichier `src/intro_python/intro_python.py`, on trouve une fonction `add_ints(a, b)`, qui prend en entrée deux entiers (notez l'indice de type `variable: type`) et retourne un entier correspondant à la somme des deux paramètres d'entrée.

```
def add_ints(a: int, b: int):
    """
    Adds two integers and returns the outcome.

    Returns an integer

    Parameters:
    - `a`: First integer to add.
    - `b`: Second integer to add.
    """

    return 0
```

Exercice 1.15 : Implémentez la fonction `add_ints()`

Remplacez le code de la fonction par votre propre code, qui lit les deux données passées en entrée, puis les additionne et retourne le résultat. Quand cela est fait, ré-exécutez la commande `pytest`, qui doit maintenant afficher un résultat positif pour la fonction `add_ints`.

Si le test ne passe toujours pas, continuez d'éditer votre fonction et ré-exécutez les tests, jusqu'à ce que le test passe.

Lorsque le test de la première fonction trouvée dans le fichier passe, la sortie doit ressembler à cela :

```
(venv) $ pytest
====================================== test session starts ======================================
platform darwin -- Python 3.11.7, pytest-7.4.4, pluggy-1.3.0
rootdir: /home/boris/ue_pe_web/intro_python
configfile: pyproject.toml
collected 5 items

tests/test_intro_python.py .FFFF                                                        [100%]

========================================== FAILURES ==========================================
_____________________________________ test_format_date ______________________________________
[...]
================================== short test summary info ===================================
FAILED tests/test_intro_python.py::test_format_date - AssertionError: assert 1 == ''
FAILED tests/test_intro_python.py::test_get_path_from_http_get[GET /index.html HTTP/2-/index.html]
- AssertionError: assert '' == '/index.html'
FAILED tests/test_intro_python.py::test_get_path_from_http_get[GET /img/logo.svg HTTP/1.1-/img/logo
.svg] - AssertionError: assert '' == '/img/logo.svg'
FAILED tests/test_intro_python.py::test_get_path_from_http_get_exception - Failed: DID NOT RAISE <
class 'Exception'>
================================== 4 failed, 1 passed in 0.06s ===================================
```

Attention : Enregistrez votre travail avec `git commit`

Vous venez d'apporter des modifications validées à votre dépôt. Il est temps d'enregistrer votre travail, de sorte à ne pas le perdre, et à pouvoir comparer vos futures modifications avec ce point de synchronisation.

1. Exécutez `git status` pour voir les fichiers modifiés depuis le commit initial.
2. Exécutez `git diff` pour voir le détail des modifications. Vous vous retrouvez en mode paginé (comme avec la commande `less`). Pour revenir au terminal, appuyez sur Q.
3. Vous êtes satisfait·e ? Alors commitez : `git commit -am "REMPLASSEZ PAR UN MESSAGE APPROPRIÉ"`
4. Pour envoyer vos modifications, faites d'abord `git pull` (au cas où des modifications aient été apportées par un·e camarade), puis `git push`

1.3.3 Implémentation de la fonction de formatage de date

Dans cet exercice, la fonction `format_date()` est déjà implémentée dans le code, et nous allons devoir écrire, ou plutôt modifier, le test `test_format_date()`.

A propos des formats de date Lorsque nous voulons manipuler des données de temps, telles que des dates ou des heures, il existe différentes manières de décrire une date (par exemple "21 décembre 1995" ou "1995/12/21", qui référencent la même date) ; notons dans le cas présent deux conventions récurrentes en informatique : le timestanp et la norme ISO 8601.

Le *timestamp* est le nombre de secondes écoulées depuis un certain événement, en général le 1er janvier 1970 à minuit (00h00m00s). Il prend la forme d'un nombre, comme : `959347394` pour le `26 mai 2000 13:23:14 GMT`. ou `1705674994` pour le `19 janvier 2024 14:36:34`.

La norme ISO 8601 définit plusieurs formats standards (ou profiles) pour représenter les dates. Le format de date utilisé dans les en-têtes HTTP est le profil RFC 2616, qui prend la forme suivante : `<day-name>, <day> <month> <year> <hour>:<minute>:<second> GMT`. Quelques exemples de dates utilisant ce format sont : `Fri, 26 May 2000 13:23:14 GMT` ou `Fri, 19 Jan 2024 14:36:34 GMT`.

Écrire le test Dans le fichier `tests/test_intro_python.py`, on trouve la fonction de test `test_format_date()` :

```python
def test_format_date():
    """ Tests the date_format function.

    TODO: This test function (and documentation) should be updated.
    """
    assert 1 == ""
```

Exercice 1.16 : Écrivez le test `test_format_date()`

L'assertion `1 == ""` est évidemment fausse. Il nous appartient de faire appel à la fonction testée, `format_date()` pour convertir un timestamp que l'on connaît, et comparer le résultat à son équivalent au format ISO 8601.

Modifiez le code de la fonction de test en vous inspirant de l'exercice précédent, puis exécutez les tests. Lorsque le deuxième test passe, c'est bon.

La sortie de `pytest` doit ressembler à quelque chose comme cela :

```
(venv) $ pytest
====================================== test session starts ======================================
platform darwin -- Python 3.11.7, pytest-7.4.4, pluggy-1.3.0
```

```
rootdir: /home/boris/ue_pe_web/intro_python
configfile: pyproject.toml
collected 7 items

tests/test_intro_python.py ..FFF                                        [100%]

========================================= FAILURES =========================================
_________________ test_get_path_from_http_get[GET /index.html HTTP/2-/index.html] _________________
[...]
=================================== short test summary info ====================================
FAILED tests/test_intro_python.py::test_get_path_from_http_get[GET /index.html HTTP/2-/index.html]
- AssertionError: assert '' == '/index.html'
FAILED tests/test_intro_python.py::test_get_path_from_http_get[GET /img/logo.svg HTTP/1.1-/img/logo
.svg] - AssertionError: assert '' == '/img/logo.svg'
FAILED tests/test_intro_python.py::test_get_path_from_http_get_exception - Failed: DID NOT RAISE <c
lass 'Exception'>
================================== 3 failed, 4 passed in 0.06s ==================================
```

Nous pouvons constater que les deux premiers tests passent maintenant avec succès. Il ne nous reste plus qu'à traiter les tests relatifs à la dernière fonction de notre code : `get_path_from_http_get()`.

1.3.4 Implémentation de la fonction d'extraction de chaînes

Il arrive régulièrement, lorsque l'on développe, d'avoir à traiter des chaînes de caractères : que ce soit pour les analyser et extraire une partie de leur contenu, ou pour les formater. Dans cet exercice, nous analyserons la chaîne de caractère envoyé par un client HTTP pour en extraire le deuxième mot, comme dans l'exemple suivant :

Entrée	Sortie attendue
`"GET /index.html HTTP/2"`	`"/index.html"`
`"GET /img/logo.svg HTTP/1.1"`	`"/img/logo.svg"`

La dernière fonction de notre code, `get_path_from_http_get()`, fournit une fonctionnalité qui nous sera fort utile lors de nos manipulations réseau : extraire une chaîne de caractères donnée d'une requête `HTTP GET`.

```python
def get_path_from_http_get(http_request: str):
    """
    Parses a typical HTTP request [1] and returns the path [2] part.
    [1] "GET /prise_en_main/ HTTP/2"
    [2] "/prise_en_main/"

    TODO: Update this function so it passes the associated test.

    Returns the path part of the request as a string.
    Throws a ValueError exception if the input string is malformed.

    Parameters:
    - `http_request`: The HTTP request sting to parse.
    """
    out = ""

    return out
```

La fonction renvoie pour l'instant systématiquement une chaîne de caractères vide ; vous devez l'éditer pour qu'elle extraie le chemin de la requête et la renvoie.

A propos des exceptions La plupart des langages de programmation proposent un moyen de gérer des exceptions, c'est-à-dire l'interruption immédiate de l'exécution du code courant lorsque quelque chose ne va pas. Il existe des exceptions standard pour les cas les plus communs, qui peuvent être aisément utilisées et comprises car leur sémantique est connue. Par exemple :

— Si l'on doit faire une division et que le dénominateur est nul, alors Python (ou le développeur) pourra lancer une exception `ZeroDivisionError`.
— Si l'on essaie de calculer la somme d'une chaîne de caractère, Python lancera une exception de type `TypeError`.
— Si le fichier que l'on veut lire n'existe pas, Python lancera une exception de type `FileNotFoundError`.

L'intérêt des exceptions est qu'elles peuvent être récupérées et traitées dans le code. Cela permet notamment de rattraper des erreurs afin d'éviter au programme de planter complètement, ou de trouver une solution alternative : si on veut lire un fichier qui n'existe pas, on peut s'en rendre compte et demander à l'utilisateur de nous fournir un autre nom. Le code suivant montre un exemple de gestion d'exception pour une erreur de type `ZeroDivisionError`.

```python
def my_division(num: int, denom: int):
    """
    Returns the division of `num` by `denom`.
    Raises a `ZeroDivisionError` if `denom` equals zero.

    Parameters:
    - `num`: int - the numerator of the division
    - `denom`: int - the denominator of the division
    Returns:
    - float - the result of the division
    """

    if denom == 0:
        raise ZeroDivisionError("denom equals zero. Are you trying to break reality?")
    return num / denom
```

Références : La gestion d'exceptions en Python

— Découvrez la gestion des exceptions avec Python, cours en ligne, Open Classrooms.
— Erreurs et exceptions, tutoriel, documentation Python.

Exercice 1.17 : Écrivez la fonction `get_path_from_http_get()`

La fonction `get_path_from_http_get()` prend en argument un paramètre, la chaîne de caractères à analyser, et doit retourner le deuxième mot. Vous pouvez utiliser la fonction `str.split()` pour découper une chaîne et en extraire chaque élément individuellement.

Lorsque vous aurez traité la partie extraction de chaîne, les deux premiers tests devraient passer avec succès. Bravo ! Nous souhaitons maintenant émettre une exception `ValueError` lorsque la chaîne de caractères d'entrée est malformée, c'est-à-dire si elle ne contient pas trois mots, ou si le premier mot n'est pas `GET`. Améliorez la fonction existante afin que le dernier test passe avec succès également.

Lorsque vous avez terminé, ré-exécutez les tests. S'ils passent tous avec succès, vous avez réussi !

```
(venv) $ pytest .
====================================== test session starts ======================================
platform darwin -- Python 3.11.7, pytest-7.4.4, pluggy-1.3.0
rootdir: /home/boris/ue_pe_web/intro_python
configfile: pyproject.toml
collected 7 items

tests/test_intro_python.py .......                                                        [100%]

======================================= 7 passed in 0.01s =======================================
```

Notez au passage que l'on peut obtenir plus d'information sur les tests grâce à l'option `verbose`, ou `-v` :

```
(venv) $ pytest . -v
====================================== test session starts ======================================
platform darwin -- Python 3.11.7, pytest-7.4.4, pluggy-1.3.0 -- /home/boris/ue_pe_web/intro_python/
venv/bin/python
cachedir: .pytest_cache
rootdir: /home/boris/ue_pe_web/intro_python
configfile: pyproject.toml
collected 7 items

tests/test_intro_python.py::test_add_ints PASSED                                           [ 14%]
tests/test_intro_python.py::test_format_date PASSED                                        [ 28%]
tests/test_intro_python.py::test_date_format_parameterized[959347394-Fri, 26 May 2000 13:23:14 GMT]
PASSED [ 42%]
tests/test_intro_python.py::test_date_format_parameterized[1705674994-Fri, 19 Jan 2024 14:36:34 GMT
] PASSED [ 57%]
tests/test_intro_python.py::test_get_path_from_http_get[GET /index.html HTTP/2-/index.html] PASSED
[ 71%]
tests/test_intro_python.py::test_get_path_from_http_get[GET /img/logo.svg HTTP/1.1-/img/logo.svg] P
ASSED [ 85%]
tests/test_intro_python.py::test_get_path_from_http_get_exception PASSED                   [100%]

======================================= 7 passed in 0.01s =======================================
```

1.3.5 Génération de la documentation

Les `docstrings` rédigés en commentaire peuvent être extraits pour créer automatiquement une documentation HTML décrivant le code et ses fonctions. En Python, on peut utiliser l'outil `pdoc` pour lire tout fichier Python – script ou module – et générer un site local simple, mais utile. Par défaut `pdoc` sert les pages localement, comme dans la commande suivante où nous analysons notre nouveau module :

```
$ pdoc intro_python
```

Cette commande doit ouvrir un navigateur web, avec quelques pages HTML décrivant notre fichier `intro_python` et ses différentes fonctions. Il est possible d'écrire les pages web sur disque, afin de l'utiliser comme documentation pour notre projet. Dans l'exemple suivant, nous utilisons `pdoc` sur les répertoires `src/` et `tests/` pour générer la documentation de notre code et de nos tests dans le répertoire `docs/`.

```
$ pdoc src/ tests/ -o docs/
```

Vérifiez que le répertoire `docs/` a été créé, et qu'il contient bien la documentation de nos fonctions et de leurs tests.

1.4 Bonnes pratiques de qualité logicielle

La qualité logicielle englobe tout ce qui permet d'obtenir un *bon* projet, qui soit facilement utilisable, qui fonctionne bien (e.g. rapide, solide, compacte), qui pourra être maintenu (par soi-même ou par d'autres personnes). En fonction du domaine d'application, les contraintes de la qualité logicielle seront différentes : si le projet doit calculer la descente d'un avion ou être embarqué dans un satellite, les contraintes ne seront pas les mêmes que dans le cas d'un utilitaire de conversion d'images pour le bureau, par exemple.

Quel que soit le langage de programmation utilisé, il existe des pratiques et des outils pour nous aider à mieux développer.

Les pratiques :

— L'adoption d'un *style de programmation* cohérent permet à n'importe quelle personne qui connaît le Python de lire notre code, le comprendre et le modifier aisément. Cela passe par exemple par une écriture uniforme, des longueurs maximales de lignes, et une *convention de nommage* (identifier rapidement à quoi sert une variable, si elle est locale ou globale, etc.). Pour cela nous avons à notre disposition plusieurs outils, tels qu'un *linter* et un *formatter*. Nous allons les aborder dans la prochaine section.

— Écrire et maintenir *la documentation* : un nouveau venu, débutant ou expert du sujet, doit avoir suffisamment d'information pour comprendre ce que fait le logiciel, comment l'utiliser, quelles seront ses limites.

— Écrire et maintenir *des tests*, afin de s'assurer que le programme est toujours correct. Lorsque l'on veut aller plus loin, il est recommandé de les exécuter automatiquement en intégration continue, ou CI.

1.4.1 Être élégant·e

Un code élégant est un code qui :

— se lit bien, est bien commenté, et respecte un style de programmation reconnu,

— est robuste, car il a été testé avec de nombreux cas de tests variés,

— est correctement optimisé quant à la consommation de ressources et la vitesse d'exécution.

Il est de l'intérêt de tous, des autres comme de nous-même, d'écrire un code élégant : cela facilite la maintenance et l'évolution du code, permet à d'autres de s'y intéresser, et... on en est fier, ce qui n'est pas rien.

Le linter Un *Linter* analyse le code et propose automatiquement des améliorations sur le style et les bonnes pratiques. Nous utiliserons `pylint`, un utilitaire classique en Python, pour contrôler et améliorer notre code. Exemple d'utilisation :

```
(venv) $ pylint intro_python.py
************* Module intro_python
intro_python.py:40:49: C0303: Trailing whitespace (trailing-whitespace)
intro_python.py:1:0: C0114: Missing module docstring (missing-module-docstring)
intro_python.py:46:4: W0621: Redefining name 'args' from outer scope (line 90) (redefined-outer-name)
intro_python.py:84:4: W0101: Unreachable code (unreachable)

-----------------------------------
Your code has been rated at 5.71/10
```

Le formatter Un *Formatter* reformate automatiquement le code en utilisant une convention choisie, ce qui permet de mieux se concentrer sur le contenu plutôt que sur la forme. Nous utiliserons pour nos exercices `black`, un formatter Python largement apprécié pour son style clair et concis, compatible avec la norme de convention Python PEP 8.

Black modifie les fichiers *sur place* : il écrase votre fichier avec la version améliorée. Exemple d'utilisation :

```
(venv) $ black src/intro_python/intro_python.py
reformatted src/intro_python/intro_python.py

All done!
1 file reformatted.
```

Structure de projet Quelque soit le type de développement adopté pour notre projet, il y a des éléments qui reviendront toujours : de la documentation, du code, et des tests par exemple.

Il existe certaines conventions qui permettent de retrouver rapidement ces éléments dans n'importe quel projet. En particulier, nous voulons ajouter un fichier `README.md` à la racine de notre dépôt qui décrit rapidement ce que fait notre projet, liste les dépendances, décrit comment tester et exécuter le code, ou fournit des liens vers plus de contexte.

1.4.2 Documentation

La documentation d'un projet logiciel est fondamentale, pour ses développeurs autant que pour les utilisateurs. Sans elle, il sera très difficile de comprendre ce que fait l'application, de la maintenir, ou même de l'utiliser – ce qui rend l'application, et tous vos efforts de développement, totalement vains.

Pour les utilisateurs, il importe de décrire clairement ce que fait l'application, quelles sont ses limites, et comment la mettre en œuvre – avez-vous déjà essayé d'utiliser un logiciel, ou une librairie, mal documentés ? On essaie quelques heures, on passe un mauvais moment, et on finit généralement par abandonner en maudissant le développeur : pourquoi avoir passé du temps à écrire ce logiciel si on ne peut pas l'utiliser ? Donc rendez votre travail réellement utile, et écrivez une documentation utilisateur correcte.

La documentation de développement est importante pour vous, et pour toute personne qui s'intéressera à votre code pour le comprendre, vous aider à le développer ou de manière générale participer au projet. Cela s'applique également à vous-même, développeur·se, lorsque vous reprendrez votre propre code une semaine, un mois ou un an plus tard : ne pensez pas que vous retrouverez vos petits, cela ne fonctionne pas : on oublie les détails. Chaque fois que vous écrivez du code, pensez à mettre par écrit (en commentaire, dans un README ou dans un document dédié) tous les éléments que vous avez dans la tête, et qui vous permettront plus tard de comprendre ce que vous avez fait. Damian Conway, l'un des pères fondateurs de l'informatique, disait : « La documentation est une lettre d'amour écrite à votre futur moi. » Il avait raison.

1.4.3 Tests

Les tests sont des scripts Python qui exécutent le code de nos scripts, et s'assurent que le résultat est correct. Ce sont eux qui garantissent, s'ils sont bien faits, que l'application fonctionne et rend les résultats escomptés. L'intérêt d'un test est qu'il soit reproductible n'importe où, y compris de manière automatique, sur d'autres machines, et dans des contextes différents.

Les tests sont utiles pour de nombreuses raisons : outre qu'ils nous permettent de valider à tout moment si un code (scripts, fonctions) rend le service attendu (tests *fonctionnels*), ils assurent également la portabilité et fiabilité du code dans différentes situations, sur différentes machines, pour différentes personnes : paramètres, variables d'environnement, fuseau horaire, langue de l'utilisateur, système d'exploitation utilisé, etc.). Ils permettent également de savoir si une modification du code affecte d'autres fonctionnalités (tests de *régression*), ailleurs dans le code, ou développée par une autre personne – cela devient en fait incontournable sur des développements complexes ou à plusieurs. On peut *a priori* assumer que tant que les tests passent, la fonctionnalité est réputée bonne pour le contexte ciblé.

Il est donc important d'avoir des tests, et de les rendre autonomes – c'est à dire qu'ils puissent être exécutés n'importe où facilement. Il existe des librairies qui facilitent cette démarche, pour tous les langages et toutes les techniques : jUnit, pytest, selenium par exemple.

Enfin, des outils d'intégration continue (CI) sont également disponibles pour automatiser et optimiser les tests, et les distribuer sur un parc de machine : Jenkins, GitLab CI, GitHub CI, Travis par exemple.

Exercice Bonus B1.1 : Appliquer les bonnes pratiques de développement

1. Exécutez le linter, et notez les violations listées dans votre README.
2. Corrigez certaines violations, puis ré-exécutez le linter, jusqu'à ce que toutes les violations aient disparu.
3. Exécutez la commande `black` sur votre script et sur les tests. Comparez le résultat dans les fichiers avant et après exécution de la commande, et notez vos impressions dans le README.
4. Enregistrez les fichiers mis à jour dans un commit (`git commit -am "Exercice xxx."` , et poussez les modifications sur le serveur (`git push`).

2 Initiation au réseau

> L'informatique est la science des abstractions.
>
> Inconnu

D'aucun·es disent que les données numériques sont l'or noir du XXIe siècle. Qu'iels aient raison ou non, il n'y aurait pas d'échange de données numériques possible sans réseaux informatiques. Le réseau fournit l'infra-structure qui rend possible la société digitale que nous connaissons. Si ce domaine n'est pas aussi sensationnel qu'un robot conversationnel qui semble intelligent, il est pourtant fondamental. Nous vous proposons donc un premier tour d'horizon des réseaux informatiques, avec pour double objectif de démystifier ce nébuleux OVNI, et de préparer le terrain pour la suite du module. Pour commencer, on présentera quelques éléments théoriques en partie 2.1, avant d'attaquer la pratique : d'abord avec la ligne de commande Linux (section 2.3), puis avec Python en section 2.3.

2.1 Généralités

Le modèle client-serveur On pense souvent les interactions en réseau avec le modèle client-serveur, représenté en figure 3. Un serveur fournissant un service attend des connexions de clients (phase `listen`). Chaque client initie une connexion pour demander à accéder au service (phase `connect`). Le serveur accepte la connexion (phase `accept`) : la communication peut s'effectuer en fonction des besoins du service (phase `connected`). À noter que le serveur continue d'attendre des connexions pendant qu'il sert des clients, en parallèle.

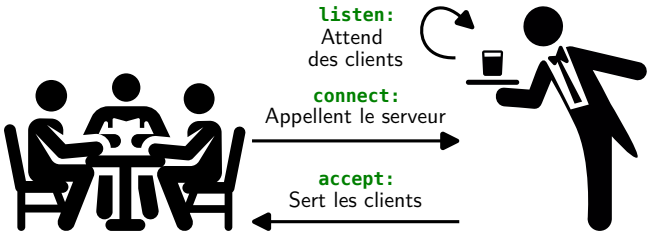


FIGURE 3 – Modèle de communication client-serveur

Parfois, les rôles ne sont pas aussi bien définis : dans le cas d'une conversation, il n'y a pas de hiérarchie client-serveur, plutôt des *pairs* qui discutent sur un pied d'égalité—d'où l'expression *pair-à-pair* ou Peer-to-peer (P2P). Dans ce cas comme dans les autres, on représente quand même les échanges avec ce modèle, en considérant simplement que chaque pair tient à la fois le rôle de serveur et de client.

Mais qui sont donc ces clients et ces serveurs, finalement ? Il s'agit toujours de programmes (ou applications), qui s'exécutent sur des ordinateurs (un *smartphone* ou une console n'étant jamais qu'un ordinateur spécialisé). Toute communication est une suite de messages, qui voyagent à grande vitesse dans des câbles ou à l'air libre. Détaillons la vie aussi brève que trépidante d'un message envoyé à travers Internet : il prend diverses formes, et passe entre les « mains » de nombreuses machines. Souvent sous forme d'ondes radio électromagnétiques (e.g. Wi-Fi) de l'ordinateur du client jusqu'au routeur ou à l'antenne qui le relie au net ; puis sous forme de signal électrique dans les fils de cuivre [2] qui relient les nombreux routeurs qui constituent Internet ; jusqu'à sa destination : une machine, où s'exécute le programme serveur qu'il souhaitait joindre.

FIGURE 4 – Les messages réseau sont *encapsulés* à la manière de poupées russes.

Encapsulation Afin que chaque message trouve son chemin, on lui ajoute de multiples en-têtes (ou *headers*) fournissant des informations de *routage*, qui seront lues et modifiées par divers acteurs durant le trajet du message. Ces en-têtes sont ajoutés récursivement, comme si l'on mettait le message original

2. De plus en plus, on a recours à la fibre optique plus rapide, au sein de laquelle les messages prennent la forme d'un signal lumineux.

dans plusieurs enveloppes successives, ou à la manière de poupées russes (voir figure 4). Ce processus appelé *encapsulation* est fondamental en réseau, et suit des conventions très précises (au bit près).

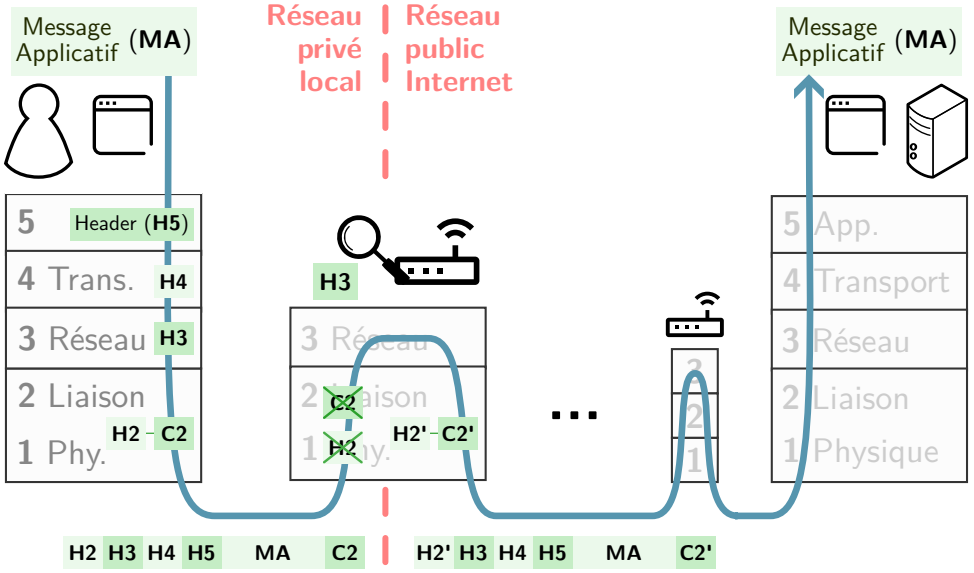


FIGURE 5 – Encapsulation et décapsulation d'un message applicatif sur le réseau, à travers les couches d'abstraction du modèle IP. Le message applicatif **MA** est encapsulé de haut en bas, en y ajoutant des en-têtes à chaque couche par l'ordinateur expéditeur. C'est donc un message préfixé de nombreux en-têtes qui transite sur le réseau, dont certains seront lus et modifiés en chemin. Entièrement décapsulé de bas en haut par l'ordinateur destinataire, **MA** pourra finalement être lu par l'application cible.

Pile de protocoles du modèle IP La figure 5 représente le trajet qu'effectue un message à travers les cinq *couches d'abstraction* du modèle IP [3] : de l'application émettrice au support de communication physique, et du support à l'application cible, en passant par les routeurs sur son chemin. On parle de couches d'*abstraction*, car c'est d'abord un modèle mental (*abstrait*) : que l'on utilise pour comprendre les mécanismes du réseau, pour séparer en blocs distincts les multiples tâches qui interviennent dans la communication numérique. Néanmoins, un protocole et des en-têtes bien précis régissent le fonctionnement de chaque couche, ce qui les rend pour l'ingénieur réseau aussi concrètes que des murs en brique. Détaillons pour chaque couche, leur rôle, leurs protocoles et leurs en-têtes :

1. Couche **physique** : Son rôle est d'assurer la transmission physique des messages, c'est à dire de convertir sans erreur le signal (électrique, lumineux, radio...) en bits. Utilisant généralement le protocole Ethernet (pour un support cablé) ou 802.11 (pour du Wi-Fi), cette couche partage en quelque sorte son en-tête avec la couche suivante.
2. Couche **liaison de données** : En charge d'assurer la liaison entre ordinateurs directement reliés par un lien physique. À la création d'un message, cette couche lui ajoute un en-tête (noté **H2** en figure 5), qui contient les adresses matérielles (adresses Media Access Control (MAC)) de l'émetteur et du récepteur sur le lien. Elle lui ajoute aussi un code correcteur (noté **C2**) en fin de message, qui sert à vérifier l'absence d'erreur de transmission. On dit que la couche liaison *encapsule* le message dans **H2** et **C2**. En-tête et code correcteur seront supprimés ou *désencapsulés* à chaque fois que le message passe entre les mains d'un équipement réseau sur sa route, pour être remplacés par de nouvelles valeurs (**H2'** et **C2'**) contenant les adresses MAC des équipements du lien suivant.
3. Couche **réseau** : Assurant le routage des messages sur Internet, cette couche est le royaume indisputé du protocole Internet Protocol (IP) (version 4 ou 6). Son en-tête **H3** contient l'adresse

3. On présente généralement les abstractions réseau avec les sept couches du modèle OSI. La couche Application (7) y est précédée des couches Session (5) et Présentation (6). C'est par souci de concision et pour ne pas alourdir l'exposé qu'on a préféré présenter le modèle IP. Le modèle OSI est plus complet et précis.

IP de l'émetteur initial et celle du récepteur final du message, similairement aux adresses postales d'émission/destination qu'on trouve sur le courrier physique. Les routeurs sur la route lisent cet en-tête pour décider de la direction que doit prendre le message sur Internet, afin de le rapprocher de sa destination. Il n'est généralement pas modifié par les routeurs, sauf dans le cas particulier (mais très courant) d'un sous-réseau traduisant des adresses IP privées en IP publiques (protocole Network Address Translation (NAT)). **H3** sera désencapsulé après **H2** lors de sa réception par le destinataire final.

4. Couche **transport** : En charge de distribuer chaque message reçu par l'ordinateur récepteur vers la bonne application cible. Ses deux protocoles principaux sont User Datagram Protocol (UDP) et Transmission Control Protocol (TCP). UDP se contente d'envoyer des lettres à travers le net en mode « advienne que pourra », tandis que TCP s'assure que les messages soient tous reçus, et dans l'ordre, par l'application cible.
   Les deux informations principales inscrites dans l'en-tête **H4** sont *le numéro de port* de l'émetteur et du récepteur. Une interface réseau d'ordinateur dispose de 65 536 ports TCP (et autant en UDP), ce qui fait de nombreux canaux de communication disponibles pour ses applications.
   Sauf dans le cas du NAT, **H4** n'est ni lu ni désencapsulé par les routeurs en chemin, et sera seulement désencapsulé par le destinataire après **H3**.
5. Couche **application** : Interne à l'application qui effectue la communication des deux côtés (émetteur et récepteur), son en-tête **H5** fournit des informations dites *métier* : qui dépendent du cas d'usage. Par exemple, un navigateur et un serveur web utiliseront le protocole HTTP, qui est précisément défini dans sa spécification, la RFC9110. L'en-tête **H5** sera décapsulé en dernier, après **H4**.

La plus petite des poupées russes contient souvent un message applicatif (**MA**) représenté en figure 5. Dans le cadre de ce cours, nous manipulerons tout particulièrement les protocoles TCP et HTTP.

Références : Pour aller plus loin

— Andrew Tanenbaum et al., Réseaux (6e édition), livre, Éditions Pearson, 2022.
— Les réseaux de zéro, cours en ligne, Zestes de Savoir, 2022.

2.2 Pratique du réseau avec Linux

Dans cet exercice, nous allons utiliser les utilitaires système Linux pour manipuler et observer les connexions réseau sur notre machine. Commençons par créer le projet Git du second TP afin d'y noter nos premiers résultats dans son README.

Exercice 2.1 : Initialisation du TP2

1. Créez un dépôt local :
   (a) Ouvrez un terminal et rendez-vous dans le répertoire `~/dev` avec `cd`.
   (b) Créez un dossier appelé `initiation_reseau` avec `mkdir`.
   (c) Rendez-vous dans le dossier nouvellement créé avec `cd`.
   (d) Initialisez votre dépôt local avec `git init`.
2. Créez un README et faites un premier commit :
   (a) Créez et lancez l'édition d'un fichier `README.md`, par exemple avec la commande `gedit README.md &`. [a]
   (b) Éditez ce fichier Markdown avec en titre le nom du projet, puis le nom des deux binômes, et la date. Pour des rappels sur la syntaxe Markdown, voir les références ci-dessous.

(c) Ajoutez le README au dépôt, et faites votre premier commit (voir exercice 1.6).

3. Créez un dépôt sur la forge nommé « UEPE-web-tp2 » en reprenant les instructions de l'exercice 1.7.
4. Renseignez le serveur distant dans votre dépôt local avec :
   `git remote add origin git@URL_DE_VOTRE_DEPOT_DISTANT`.
   (N'oubliez pas de remplacer l'URL ci-dessus par l'URL de votre project GitLab !)
5. Poussez votre premier commit sur la forge : `git push -u origin main`.

a. `gedit` est l'éditeur de texte par défaut sur Ubuntu. L'esperluette `&` en fin de ligne sert à lancer un programme en tâche de fond, de sorte que vous conserviez l'usage de votre terminal pendant que `gedit` s'exécute.

Attention : Enregistrez votre travail avec `git commit`

N'oubliez pas de faire un commit après chaque exercice pour sauvegarder votre travail ! Nous utiliserons la liste de vos commits pour juger de votre avancée.

Pour chaque commit, indiquez dans le message le numéro de l'exercice, ainsi qu'une petite phrase pour expliciter brièvement ce que contient le commit. Par exemple :

```
git commit -am "Exercice 2.1 : Initialisation du TP2"
```

A la fin de chaque session (ou plus régulièrement), faites un `git push` pour envoyer vos modifications (commits) vers le dépôt Git de l'université.

Références : Rédaction de texte avec Markdown

— Produire des documents avec Markdown, cours en ligne, Univ Lille, 2023.
— Bernard Pochet, Markdown & vous, livre, ULiège Library, 2023.
— Sarah Simpkin, Débuter avec Markdown, cours en ligne, Programming Historin, 2015.

2.2.1 Première requête HTTP avec netcat

L'exercice suivant propose d'interroger manuellement un serveur web avec le protocole HTTP, pour qu'il nous renvoie la page d'accueil du site `http://example.org`. [4]

L'utilitaire `netcat` est le couteau suisse du réseau, disponible sur la plupart des OS. Il permet d'effectuer des communications en TCP ou en UDP (couche 4, transport), en mode client ou serveur, directement depuis la ligne de commande. Voilà sa syntaxe :

```
netcat [OPTIONS] [IP OU NOM DE DOMAINE] [PORT]
```

D'après la spécification du protocole applicatif HTTP, ce dernier nécessite de s'appuyer sur un protocole de transport « fiable ». C'est le cas de TCP, que nous allons donc utiliser pour discuter avec le serveur web. Par ailleurs, par convention, un serveur HTTP écoute (`listens`) sur son port TCP numéro 80.

4. `example.org` est un domaine réservé pour les tests et l'enseignement. Voir la documentation officielle des domaines réservés sur le site de l'organisme de standardisation IANA.

TCP, un protocole fiable Nous avons déjà dit que TCP s'assurait que tous les messages qu'il transporte soient reçus, et dans l'ordre. C'est pourquoi on dit de TCP qu'il crée des *connexions* fiables. Le terme de connexion est encore une vue de l'esprit (ou abstraction) : *du point de vue de la couche applicative*, TCP crée un canal de communication bidirectionnel, temps-réel, et sans perte. On peut comparer une connexion TCP avec un appel téléphonique : tout ce qui est dit d'une part et d'autre est instantanément transmis de l'autre côté. À l'inverse, UDP ne fournit pas une telle abstraction. Une application reposant sur UDP a plutôt l'impression d'envoyer et de recevoir des lettres par La Poste, sans garantie qu'elles arrivent, et encore moins dans l'ordre. UDP n'est donc pas fiable. Par contre, son fonctionnement est plus proche de la réalité physique, puisque dans les deux cas ce sont bien des messages qui transitent sur le réseau à la façon de la figure 5.

Exercice 2.2 : Raconter n'importe quoi à un serveur web

1. Ouvrez un terminal, et lancez la commande suivante pour ouvrir une connexion TCP avec le serveur hébergeant `univ-lille.fr` sur son port 80 :

   ```
   netcat univ-lille.fr 80
   ```

   Votre terminal a sauté une ligne, et semble attendre vos instructions.
2. Tapez n'importe quel texte suivi de la touche Entrée ↵.
   Du texte apparaît : c'est la réponse du serveur web !
3. Appuyez sur Ctrl+C pour quitter `netcat`.
4. Détaillez dans le README du TP2 les éléments qui vous permettent de déduire que le serveur web n'était pas satisfait de votre requête. [a]

a. **Une attention particulière sera prêtée à la forme de votre rendu.** Il n'est pas demandé dans ce module de réaliser l'intégralité des exercices, mais si vous faites quelque chose, *faites-le bien.*

Parlons HTTP Bien qu'on lui ait envoyé n'importe quoi, le serveur web nous a répondu dans sa langue, le HTTP. On observe que :

- C'est un protocole *textuel* : les caractères échangés sont des caractères d'imprimerie relativement interprétables (quoiqu'en anglais).
- La réponse est constituée de deux parties séparées par une ligne vide. La première partie est l'en-tête applicatif **H5**, la seconde message applicatif **MA**
- L'en-tête **H5** débute avec une ligne contenant version de HTTP, code de retour et sa signification. S'en suivent une information par ligne, au format `Clé: valeur`.
- Le message applicatif **MA** est écrit en HTML, le langage prédominant du web.

Nous allons désormais effectuer une demande valide : une *requête* GET. En HTTP, c'est de cette façon qu'on demande une ressource à un serveur web (sa page d'accueil, dans notre cas). Elle ne contiendra qu'un en-tête suivi d'une ligne vide, car une requête GET n'a pas besoin de message (tout est dans l'en-tête). Notre en-tête sera constitué en deux lignes :

1. La première sera la requête à proprement parler. Elle se compose du type de requête (`GET`), de la ressource concernée (`/`, la page d'accueil), et de la version du protocole (`HTTP/1.1`). Chaque champ doit être séparé d'un espace. [5]
2. La deuxième ligne précisera le site que l'on veut joindre au format `Host: site_demandé.org`.

On terminera l'en-tête en sautant deux lignes avec ↵.

5. Cela devrait vous rappeler un exercice précédent.

Exercice 2.3 : Demander la page d'accueil d'un site à un serveur web

1. Ouvrez une connexion TCP avec le serveur hébergeant `example.org` sur le port 80, de la même manière que dans l'exercice précédent.
2. Composez les deux lignes de votre requête en suivant les indications ci-dessus.
3. Tant que vous n'obtenez pas un code `200 OK` et la page HTML espérée, rééssayez. Pressez Ctrl + C pour quitter `netcat`, puis flèche du haut ↑ pour ré-afficher votre commande précédente, puis Entrée ↵.
4. Indiquez dans votre README la requête envoyée, ainsi que l'en-tête renvoyé par le serveur, sous forme de blocs de code source Markdown.

On peut observer dans la réponse du serveur qu'il y a plusieurs lignes d'en-têtes. La première est la réponse classique HTTP (`HTTP/1.1 200 OK`), indiquant que la requête a été reçue, comprise, et va être complétée. Les lignes suivantes sont des en-têtes HTTP décrivant la réponse, par exemple : dates d'exécution de la requête (`Date:`) et de dernière mise à jour du document (`Last-Modified:`), le nom du logiciel serveur utilisé (`Server: ECS ..:`), ou la longueur de la réponse (`Content-length:`). Ici aussi, une ligne vide précède le code HTML de la page demandée.

2.2.2 Des applications serveur sur notre machine

On a peut-être eu l'impression de rentrer dans Fort Knox en discutant avec le serveur de `example.org`, mais en fait, des application serveurs, il s'en trouve sur à peu près tous les ordinateurs. Nous allons vérifier cela avec un nouvel utilitaire qu'on trouve sur Linux, nommé `ss`, acronyme de *socket statistics*. `ss` dispose de nombreuses options :

- `-l` (*listen*) : permet de lister les applications en écoute ;
- `-n` (*numeric*) : permet d'afficher les numéros de ports, plutôt que le protocole auxquels ils font référence (e.g. afficher 80 plutôt que http) ;
- `-p` (*process*) : affiche le nom de l'application (ou processus) qui utilise un port ;
- `-t` (*TCP*) : affiche les ports TCP ;
- `-u` (*UDP*) : affiche les ports UDP.

Exercice 2.4 : Observation des *sockets* en fonctionnement sur notre machine

1. Exécutez la commande `ss -lnpt`, et observez la sortie qu'elle nous renvoie.
2. Notez dans votre README la signification des options que vous avez passées à `ss`. Combien de lignes ont été renvoyées ? Décrivez les colonnes dont vous comprenez le sens.
3. Exécutez `ss -npt` (sans `-l`). La commande vous renvoie d'autres résultats.
4. Notez dans votre README le sens des résultats observés.

Les sockets On a dit que `ss` signifiait *socket statistics* : chaque ligne renvoyée par `ss` dans l'exercice 2.4 décrivait donc vraisemblablement une socket. Le terme *socket* signifie en réalité *prise*, ou *connecteur*.[6] En réseau, c'est l'objet manipulé par les applications quand elles utilisent un port réseau (TCP, UDP ou autre). Il ne peut exister qu'un connecteur par port. Il est prêté par le système à une application qui en fait la demande. Sans quoi, il est déconnecté, et n'apparaît pas dans la liste renvoyée par `ss`.

6. Malheureusement rien à voir avec les chaussettes, finalement.

2.2.3 Création d'un serveur avec netcat

Comme susmentionné, `netcat` permet également d'ouvrir un port en mode serveur (écoute ou `listen`), au moyen de l'option `-l` . Dans cette configuration, `netcat` attend les connexions de clients. Puis, comme vu dans la section 2.2.1, il transmettra tout ce qu'on tapera au client, et affichera tout ce que le client envoie dans le même terminal. Essayons donc de créer notre propre serveur.

Dans les exercices suivants, nous utiliserons deux terminaux pour :

— d'un côté exécuter netcat en mode serveur pour ouvrir un port en écoute,

— dans l'autre terminal se connecter sur ce port en mode client pour établir une connexion.

Exercice 2.5 : Ecouter sur un port en mode serveur

1. Utilisez la commande suivante pour ouvrir le port 12345 en écoute : `netcat -l 12345` [a]
   Si vous obtenez une erreur « Address already in use », cela veut sans doute dire que le port `12345` est déjà utilisé par un autre processus sur votre système. Choisissez un port libre en vous servant de la sortie de `ss -lnpt` , et recommencez. Adaptez le port à celui que vous aurez choisi dans les exercices suivants.
2. Laissez tourner ce terminal jusqu'à la fin des exercices de cette section.
3. Sur un autre terminal, exécutez à nouveau la commande `ss -lnpt` .
4. Notez dans votre README les différences que vous constatez par rapport à l'exercice 2.4.

a. Cette commande est *bloquante* : tant que l'on n'interrompt pas le programme (e.g. en tapant Ctrl+C), netcat restera en écoute sur ce port.

Comme nous venons d'ouvrir un port, on peut voir notre `netcat` écoutant dans la liste des sockets retournée par la commande `ss` vue précédemment. En lançant à nouveau la commande `ss -lnpt` , on trouve une ligne correspondant à la socket ouverte sur le port en question :

```
$ ss -lntp | grep 12345
tcp      LISTEN    0      1     0.0.0.0:12345      0.0.0.0:*          users:(("nc",pid=64721,fd=3))
```

Exercice 2.6 : Se connecter sur le port ouvert en mode client

Dans un deuxième terminal, utiliser la commande netcat pour ouvrir une connexion sur votre propre machine (`127.0.0.1`), sur le port 12345. [a]

```
$ netcat 127.0.0.1 12345
```

Cette commande est également *bloquante* : netcat ouvre la connexion, puis attend que nous lui fournissions les éléments à envoyer. Taper dans ce terminal quelques mots, par exemple : `Je suis le client` , puis appuyer sur ↵.

a. L'adresse IP `127.0.0.1` est appelée adresse de *loopback* et signifie « moi-même ». On l'utilise quand on veut qu'une machine parle à elle-même (c'est plus courant que vous ne le croiriez).

Nous pouvons observer que le texte tapé sur le client apparaît dans le terminal du serveur. Notre socket serveur a donc bien écouté sur le port 12345, reçu une communication, et lu le message transmis. Mais les sockets sont à double sens : nous pouvons tout aussi bien envoyer un message du serveur vers le client en utilisant la même connexion.

Exercice 2.7 : Envoyer un message du serveur vers le client

1. **Dans le terminal serveur**, celui où nous avons lancé la commande d'écoute `nc -l 12345`, tapez quelques mots, par exemple : `Je suis le serveur`, puis appuyez sur ↵.
2. Notez dans votre README l'échange des deux terminaux.

Cette fois, nous pouvons observer que le texte tapé sur le serveur apparaît dans le terminal du client. Nous avons donc bien une communication à double sens, qui peut être utilisée pour échanger entre deux machines pour, par exemple, avoir une discussion entre utilisateurs ou transférer des fichiers.

Exercice Bonus B2.1 : Discuter avec ~~une autre machine~~ un·e collègue

1. Trouvez votre adresse IP avec la commande `ip a`. Cette commande est bavarde. L'adresse IP commence par `172`, par `192` ou par `10`. Notez-la.
2. Utilisez la machine d'un·e collègue pour vous connecter à votre serveur TCP, et échangez des messages.
3. Prenez note des commandes utilisées dans votre fichier README.

Exercice Bonus B2.2 : Transférer un fichier

1. Que réalise la commande suivante ?

```
$ netcat -l 12345 > unfichier
```

2. Et que se passerait-il si vous lanciez dans un second terminal, cette commande ?

```
# autrefichier doit exister
$ netcat 127.0.0.1 12345 < autrefichier
```

3. Vous avez compris ? Alors utilisez cette méthode pour envoyer votre README à vos collègues. Sinon, faites une recherche sur les « redirections Linux », ou passez à la suite.
4. Notez ce que vous avez compris dans votre README.

Attention : Vous êtes sûr·e d'avoir commité et poussé votre travail ?

Allez sur https://gitlab-etu.fil.univ-lille.fr, et assurez-vous que la dernière version de votre TP2 y est.

Sinon, commencez par sauvegarder votre travail sur la forge, et tâchez de ne plus oublier de commit/push entre chaque exercice.

2.3 Communication TCP avec Python

Maintenant que nous savons manipuler des sockets TCP avec la ligne de commande, nous allons développer un module Python qui permet d'écouter sur un port, et d'interagir avec les connexions entrantes.

2.3.1 Initialisation du projet

Lors de la première séance, un projet déjà initialisé vous a été fourni, avec toutes les dépendances. Dans ce chapitre, nous repartons de zéro en initialisant :

— un dépôt Git pour stocker nos fichiers (fait précédemment),
— un environnement virtuel (venv) Python dédié à ce TP,
— un projet PDM.

Exercice 2.8 : Initialisation de l'environnement virtuel

1. Créez un environnement virtuel : `python3 -m venv venv/`
2. Activez-le : `source venv/bin/activate`
3. Installez `pdm` : `pip install pdm`

Comme décrit dans le premier chapitre, PDM permet de gérer les dépendances d'un projet ; mais il offre également des fonctionnalités de *build*, de test et de paquetage (e.g. si l'on souhaite envoyer notre développement sur le serveur officiel de dépendances Python, PyPi).

Exercice 2.9 : Initialisation du projet PDM

1. Initialisez l'environnement PDM : `pdm init`
2. Vous pouvez accepter la plupart des valeurs par défaut en appuyant sur Entrée ↵ sauf pour le nom du projet (*Project name*), qui doit s'appeler `sockets`.
3. Ajoutez les fichiers créés par PDM à votre dépôt en faisant `git add .` avant de commiter/pousser.

```
$ pdm init
Creating a pyproject.toml for PDM...
Please enter the Python interpreter to use
0. /home/boris/dev/initiation_reseau/python/venv/bin/python (3.11)
1. /usr/bin/python3.11 (3.11)
Please select (0):
Project name (python): sockets
Project version (0.1.0):
Do you want to build this project for distribution(such as wheel)?
If yes, it will be installed by default when running `pdm install`. [y/n] (n):
License(SPDX name) (MIT):
Author name (boris):
Author email (boris@deuxfleurs.fr):
Python requires('*' to allow any) (==3.11.*): >3.9
Project is initialized successfully
```

PDM nous a créé les fichiers suivants :

— `.gitignore` liste les fichiers à ne pas inclure dans le dépôt Git (par exemple des fichiers temporaires).
— `pyproject.toml` est le fichier de configuration principal du paquetage.
— Deux répertoires `src` et `tests`, que nous avons vus précédemment et qui contiennent respectivement le code source et les tests.

À partir de maintenant, nous pouvons utiliser PDM pour gérer nos dépendances : si l'on souhaite installer `pytest`, et l'ajouter comme dépendance au projet, on exécute la commande `pdm add pytest`. Cela a pour effet d'installer le paquetage dans notre environnement virtuel, et de l'ajouter automatiquement à la liste des dépendances de notre module.

Exercice 2.10 : Ajout des dépendances dont nous aurons besoin

1. En utilisant la syntaxe précédente, ajoutez les librairies suivantes au projet : `pytest`, `pylint`, `black`, `pdoc`.
2. Vérifiez dans le fichier `pyproject.toml` que ces librairies ont bien été ajoutées.

Une fois que nous avons installé toutes nos dépendances, nous pouvons créer le fichier lock, qui va contenir cette liste et permettra lors des futures exécutions d'installer directement toutes les dépendances dans les bonnes versions. La commande `pdm lock` lit les dépendances définies dans pyproject.toml, les résout (pour obtenir les versions compatibles exactes) et les inscrit dans le fichier `pdm.lock`.

Exercice 2.11 : Figer les dépendances et les ajouter au projet

1. Utilisez la commande ci-dessus pour figer les dépendances, puis vérifier que le fichier `pdm.lock` a bien été créé.
2. Ajoutez le nouveau fichier au dépôt Git avec la commande `git add pdm.lock`, puis faire un commit.

Nous avons initialisé notre environnement correctement. Lorsque quelqu'un·e voudra utiliser ou maintenir notre projet, il ou elle n'aura qu'à suivre les actions que nous avons vues lors du chapitre précédent (`pdm install` dans l'exercice 1.11).

2.3.2 Initialisation du code

Nous vous fournissons un squelette pour le code et pour les tests, à télécharger sur Moodle. Il ne contient que les signatures des fonctions, c'est à vous de les implémenter.

Exercice 2.12

1. Téléchargez et décompressez l'archive `tp2_sockets.zip` dans le dossier de votre choix (sauf dans le dossier `initiation_reseau`).
2. Copiez-collez les fichiers de l'archive au bon endroit :
   — `src/sockets/__init__.py` et `src/sockets/sockets.py` sont à mettre dans le dossier `src/sockets/`.
   — `tests/test_sockets.py` est à mettre dans le dossier `tests/`.

Nous allons créer un utilitaire semblable à `netcat -l`, qui créera un serveur TCP sur le port qu'on lui aura indiqué. On souhaite l'utiliser comme suit :

```
$ python3 src/sockets/sockets.py -p 12345
Démarrage du serveur TCP sur le port 12345...
```

Problème : Nous ne savons pas, en Python, récupérer les *paramètres* passés en ligne de commande—le numéro du port TCP sur lequel faire écouter le serveur, en particulier.

Récupération des paramètres de ligne de commande Python fournit une bibliothèque pour traiter les paramètres de la ligne de commande, nommée `argparse` (pour « *argument parser* », car paramètre se dit aussi *argument*). Pour l'utiliser, on commence par créer un ArgumentParser : `parser = argparse.ArgumentParser(...)`. On définit ensuite les futurs arguments de notre programme. Pour chacun d'eux, on apelle la fonction `parser.add_argument(...)` : ses deux premiers paramètres doivent être les noms de l'argument au format court (e.g. `-p`) et long (e.g. `--port`) ; il convient ensuite de reseigner sa description avec le paramètre nommé[7] `help='...'`. Lorsque tous nos arguments sont définis, on analyse les paramètres passés en ligne de commande avec `parser.parse_args(argv)`.

```
parser = argparse.ArgumentParser(
  prog="Program name",
  description="Quick description of the module.")
parser.add_argument(
  '-m', '--myoption', # Deux façons de nommer l'argument
  help='Quick description of the option.', # Aide de l'argument
)
args = parser.parse_args(argv)
```

Après la dernière ligne, l'objet `args` contiendra tous les arguments de la ligne de commande. Par exemple si le module a été appelé avec l'option `--myoption 12345` ou `-m 12345`, alors `args.myoption` contiendra la chaîne `12345`.

⚠ Attention : Importer son projet avec `pip` pour qu'il soit trouvé par `pytest`

Pour que la commande `pytest` trouve le module qu'on est en train de développer, vous devez exécuter la commande suivante :

```
pip install --editable .
```

Cela va installer votre projet (le module « sockets ») dans votre environnement virtuel, et permettra à `pytest` de bien l'importer et le tester.

Exercice 2.13 : Traiter les arguments de la ligne de commande

1. Complétez la fonction `parse_args(argv: list[str])` en reprenant l'exemple ci-dessus pour que votre programme récupère un numéro de port en argument de ligne de commande :
   (a) Renseignez le nom du programme et sa description ;
   (b) Ajoutez l'argument `-p|--port` ainsi que sa description ;
   (c) Vérifiez que passent les tests validant le fonctionnement de `parse_args(...)`.
2. Vérifiez dans la fonction `main(...)` que le port est bien renseigné :
   (a) Assurez-vous que l'attribut `port` existe dans l'objet `args` : `hasattr(args, 'port')` ;
   (b) Vérifiez qu'il ait une valeur non-nulle : `args.port is not None` ;
   (c) Si `args.port` est invalide, quittez le programme avec un code d'erreur non nul : `exit(2)`.

7. En python, les paramètres *nommés* ont une clé, séparée de leur valeur par le caractère égal `=`.

3. Récupérez le numéro de port, et passez-le en argument à la fonction `server(...)`, en pensant à le convertir en entier avec `int(...)`.
4. Dans la fonction `server(...)`, informez l'utilisateur de la valeur du port avec `print(...)`.
5. Testez l'exécution de votre programme : `python src/sockets/sockets.py`.
6. Vérifiez que vous obtenez le comportement attendu en essayant différentes combinaisons de paramètres. Notez vos résultats dans le README.
7. Enregistrez votre travail sur la forge.

2.3.3 Implémentation d'un serveur TCP

Il ne nous reste plus qu'à remplir la fonction `server(...)`, pour créer un serveur TCP écoutant sur le port passé en paramètre. Nous commencerons par renvoyer toujours le même message aux clients qui se connectent, avant de fermer la connexion. Si le temps vous le permet, des exercices bonus proposent d'aller plus loin : jusqu'à un serveur de discussion instantanée entre plusieurs clients connectés simultanément.

Références

— Guide pratique : programmation avec les *sockets*, tutoriel en ligne, Documentation Python, 2024
— socket — Gestion réseau de bas niveau, documentation, Documentation Python, 2024
— socket (2), documentation du système Linux en langage C, Manuel Linux

Création d'une *socket* TCP en mode serveur La création d'une socket est une histoire de négociation entre un programme et le système d'exploitation. Il existe de nombreuses bibliothèques pour camoufler les arcanes de cette négociation, mais nous sommes ici pour apprendre. Nous utiliserons donc l'interface de communication avec le système (ou Application Programming Interface (API)) du plus bas-niveau : la librairie Python socket, qui appelle directement l'API socket du système Linux en langage C, et en emprunte mot pour mot le verbiage. Voilà les différentes étapes de création d'une *socket* TCP serveur (en écoute) :

1. On importe d'abord de la bibliothèque : `import socket`.
2. On crée l'objet *socket* : `s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`.

   On crée ici une variable `s`, à laquelle on affecte le résultat de la fonction `socket.socket(...)`, qu'on appelle avec deux paramètres pour le moins cryptiques. Vous n'y entendez rien ? C'est normal. Ces paramètres sont issus du système Linux, comme on peut le voir dans la page de manuel du module Linux *socket* :
   — `socket.AF_INET` signifie que l'on souhaite obtenir une *socket* pour un protocole reposant sur IPv4 (il existe d'autres types de *sockets*).
   — `socket.SOCK_STREAM` demande une *socket* en mode *stream*, ou flux, ou connecté : c'est à dire TCP dans le cas d'une communication Internet.

   À ce stade, on a effectué une opération système mais elle est *non bloquante*—car on n'a rien demandé d'autre au système que notre intention de réserver une *socket*.
3. On acquiert un port TCP : `s.bind((<IP>, <PORT>))`.

   Ici, on attache (*bind*) réellement notre *socket* à l'adresse `<IP>` et sur le port TCP `<PORT>`.
   — `<PORT>` : on renseignera le port qui nous a été passé en argument de ligne de commande dans les exercices précédents.

— `<IP>` : laquelle ? Celle de notre ordinateur, puisqu'on souhaite créer un serveur : c'est à la « porte » de *notre* « maison » qu'on souhaite se poster. Mais il se trouve que des portes, notre maison en a plusieurs : au moins une par *carte* ou *interface réseau* (Ethernet + Wi-Fi, par exemple). Sans oublier l'interface *loopback* (signifiant « rebouclage » ou « moi-même ») à l'adresse `127.0.0.1`.
N'ayant pas envie de choisir, on va renseigner l'adresse ultime : `"0.0.0.0"`, adresse IP *wildcard* (ou joker) signifiant dans cette situation « toutes les IPs disponibles sur la machine ».

Cet appel système peut planter notre programme, si l'on essaye de réserver un port déjà utilisé par un autre processus. Cela fera l'affaire, puisqu'on ne souhaite pas que le programme continue son exécution s'il n'arrive pas à créer de *socket*.

4. On informe le système que l'on souhaite faire de notre *socket* un serveur avec : `s.listen()`.
   À ce stade, on peut enfin voir notre *socket* dans la sortie de `ss -ltpn`.
5. Enfin, on attend la connexion d'un client, en se préparant à accepter leur requête avec : `conn, addr = s.accept()`.
   Cet appel système est *bloquant*, c'est à dire que l'exécution de notre programme va s'interrompre tant que l'on n'aura pas reçu de connexion : au client d'envoyer un `connect(...)`, maintenant (voir figure 3).
   Quand un client se sera connecté, l'exécution reprendra à la ligne suivante, et nous aurrons accès à deux variables de plus :
   — `conn` : la *socket* de la connexion TCP dont on dispose désormais avec ledit client. Elle occupe en fait un nouveau port sur notre machine, afin que `s` puisse retourner attendre d'éventuels nouveaux clients : on pourrait vouloir en recevoir plusieurs en parallèle (qui sait, pour faire un serveur de discussion peut-être ?). Nous nous intéresserons au fonctionnement de `conn` très prochainement.
   — `addr` : c'est un simple *tuple* contenant l'adresse du client distant, au format `(<IP>, <PORT>)`.
   On pourra l'utiliser par exemple pour afficher son adresse avec `print(...)`.

Voilà déjà un gros morceau d'expliqué alors commençons par réaliser notre programme jusqu'ici, en nous assurant que le port TCP choisi soit bien lié à notre processus avec la commande `ss`.

Exercice 2.14 : Serveur TCP, première partie

1. Implémentez la fonction `server(port: int)` afin qu'elle acquierre une *socket* sur le port TCP `port` en mode serveur, en suivant les instructions ci-dessus.
2. Exécutez votre programme avec `python3 src/sockets/sockets.py`. L'exécution devrait bloquer après le `s.accept()`.
3. Ouvrez un autre terminal, et lancez `ss -ltpn`. Vous devriez voir une ligne concernant votre *socket* : avec votre couple IP/port dans la colonne « Local Address », et marqué « python3 » dans la colonne « Process ».
4. N'oubliez pas de sauvegarder votre travail sur la forge.

Utilisation de la *socket* client Nous nous sommes arrêté·es dans notre code au moment où le serveur acceptait la connexion d'un client, et obtenait les deux objets `conn` et `addr`. Comme déjà dit, `addr` contient le couple `(<IP>, <PORT>)` de l'adresse TCP du client. Quant à `conn`, il s'agit d'une *socket* client avec laquelle on peut intéragir de deux façons :

— **Envoyer** : on utilise pour ce faire la fonction `conn.send(buf)`, où `buf` (pour *buffer* ou tampon) est une chaîne d'octets (voir ci-dessous).

— **Recevoir** : cela se fait avec la fonction `buf = conn.recv(BUF_SIZE)`. La fonction de réception a besoin de connaître la taille maximum du tampon qu'on est prêt à recevoir, en octets. Pour l'exercice, on peut mettre `BUF_SIZE` à `1024`. De même, `buf` contiendra une chaîne d'octets.

Quand on a fini de l'utiliser, on doit fermer la *socket* avec `conn.close()`.

Octets et caractères L'API `socket` manipule des chaînes d'*octets*, alors que nous sommes habitué·es à utiliser des chaînes de *caractères*. La différence peut paraître subtile quand on ne manipule que des caractères latins qui sont dans la table ASCII : chacun de ces caractères est représenté à l'aide d'un unique octet. La lettre `a` est *encodée* par la valeur décimale `97`, `b` par `98`, etc. Mais un octet n'a que $2^8 = 256$ valeurs possibles, alors qu'il existe des myriades d'alphabets (cyrillique, chinois, coréen...), une pléthore d'émojis et des caractères accentués en pagaille. La plupart des caractères du monde sont en fait encodés sur plusieurs octets.

— Un tampon `buf` reçu avec `conn.recv(...)` est une chaînes d'octets. Pour l'afficher correctement avec `print(...)`, il faut le décoder avec `buf.decode()`.

— Si l'on souhaite envoyer un texte `txt`, il faut d'abord l'encoder : `conn.send(txt.encode())`.

Voilà un petit exemple dans l'interpréteur Python pour en avoir le cœur net :

```
>>> caracteres = "é" # une chaîne de caractères
>>> type(caracteres)
<class 'str'>
>>> octets = caractere.encode() # une chaîne d'octets
>>> type(octets)
<class 'bytes'>
>>> for octet in octets:
...   print(int(octet))
...
195
169
```

Exercice 2.15 : Serveur TCP, deuxième partie

1. Terminez l'implémentation de votre serveur TCP :
   (a) Un hébergeur web (connecté à Internet) doit légalement garder une trace du passage des clients sur son serveur : affichez avec `print(...)` un message informant de la connexion d'un client et son adresse.
   (b) Dans la fonction `server(...)`, écrivez sur la *socket* client le message renvoyé par la fonction `answer()`, en suivant les instructions ci-dessus.
   (c) Fermez la *socket* client et celle du serveur.
2. Exécutez votre programme dans un terminal, et connectez-vous à votre serveur depuis un deuxième terminal en exécutant la commande : `netcat 127.0.0.1 <PORT>`.
3. Sauvegardez votre travail.

Vous devriez obtenir un résultat ressemblant au suivant :

```
# Server-side
$ python src/sockets/sockets.py -p 12345
Got port option set to: 12345
Starting server...
Got connection from ('127.0.0.1', 52553)
Bye bye.
```

```
# Client-side
$ netcat 127.0.0.1 12345
Gotcha.
```

Exercice 2.16 : Servir plusieurs clients d'affilée

Actuellement, notre serveur ne travaille pas longtemps : il sert un client, et termine son service (le serveur s'éteint). Un serveur web doit tourner *indéfiniment*, tant qu'il n'est pas tué (par exemple avec Ctrl+C).

Ajoutez une boucle `while(True):` qui permette d'accepter un nouveau client une fois que le premier est parti.

Félicitations, vous venez d'utiliser les API du système en Python, et vous avez avec succès créé un serveur TCP capable d'accepter la connexion d'un client, et de lui écrire ! Ce n'est pas rien.

Il vous reste du temps ? Alors voilà quelques exercices d'approfondissement.

Exercice Bonus B2.3 : Modifier la fonction de réponse

La fonction de réponse de notre code `answer()` retourne pour l'instant toujours la même chaîne de caractères : « Gotcha. » C'est super, mais nous aimerions avoir une réponse plus dynamique.

1. Modifiez la fonction `answer()` en y ajoutant un paramètre, pour retourner une chaîne qui contient l'adresse IP et le port du client.
2. Exécutez votre code serveur, et relancez (dans un terminal différent) la commande client `netcat`. Vous devriez obtenir quelque chose de similaire à la sortie suivante :

```
$ netcat 127.0.0.1 12345
Hello 127.0.0.1! You are using port 52656. Have a nice day!
$
```

3. Faites un commit/push de vos dernières modifications (code et README).

Exercice Bonus B2.4 : Recevoir un message du client avant de le quitter

Nous n'avons expérimenté que l'envoi de données sur notre *socket* `conn`. Quid de la réception ?

1. Après avoir envoyé des données au client dans la fonction `server(...)`, ajoutez une ligne de code pour recevoir un message avec `conn.recv(...)`, comme expliqué dans le paragraphe sur l'utilisation de la *socket* client.
2. Affichez le message du client, en n'oubliant pas que vous avez reçu une chaîne d'octets et non de caractères.
3. Lancez votre serveur, et braquez comme d'habitude `netcat` dessus pour tester votre programme. Écrivez un court message sur le terminal faisant tourner `netcat`, et appuyez sur Entrée ↵. Vous devriez voir ce message s'afficher sur le terminal du serveur.
4. Sauvegardez votre travail.

Vous venez de réaliser une communication bi-directionnelle entre un client et un serveur TCP de votre cru ! Sur la route pour réaliser le prochain réseau social qui vous fera multi-milliardaire—vous avez fait le plus dur. Bravo à vous.

2.3.4 Bonus : Développement d'un logiciel de discussion instantanée

Tout bon réseau social a un système de *chat* (discussion instantanée). Puisque vous êtes en avance, voyons donc comment en réaliser un ! Pour commencer, nous allons développer un client TCP simple.

Nous verrons ensuite comment faire pour qu'un serveur puisse s'occuper de plusieurs clients *en même temps*. Enfin, nous retournerons à notre client pour qu'il puisse recevoir des messages du réseau ou de son utilisateur simultanément.

Simple client TCP en Python On voudrait utiliser le même code pour développer serveur et client. On va utiliser pour cela les paramètres de ligne de commande, comme le fait `netcat`. Et tant que nous y sommes, regardons comment récupérer des messages de l'utilisateur, afin de les envoyer au serveur—cela nous sera utile par la suite. Voilà ce qu'on souhaite obtenir :

```
# Server-side
$ python src/sockets/sockets.py -l -p 12345
Got server flag: -l
Got port option set to: 12345
Starting server...
Got connection from ('127.0.0.1', 52553)
Sent: Hello, client.

Received: Hello and goodbye, server!
Bye bye.
```

```
# Client-side
$ python src/sockets/sockets.py -p 12345 -s 127.0.0.1
Got port option set to: 12345
Got IP option set to: 127.0.0.1
Starting client...
Connected to server ('127.0.0.1', 12345)
Received: Hello, client.
To send: Hello and goodbye, server!
Sent.
Bye bye.
```

Du côté des paramètres :

— Il faut maintenant ajouter le paramètre `-l|--listen` pour que `sockets.py` prenne le rôle de serveur. Ce paramètre n'attend pas de valeur à sa suite, c'est une option booléenne (comme dans `netcat`). On parle de paramètre drapeau (ou *flag*). Pour réaliser cela avec `argparse`, on passe l'argument `action='store_true'` à la fonction `add_argument(...)` [8].

— On a besoin d'un nouveau paramètre `-s|--server` pour récupérer l'adresse IP du serveur distant. Ce paramètre sera uniquement utilisé si l'on est en mode client (sans `-l`).

Quant au déroulement de l'échange :

1. Le serveur envoie un premier message prédéfini, qui est affiché par le client ;
2. Le client demande à son utilisateur de rentrer une chaîne de caractères, grâce à la fonction bloquante `input` : `msg = input('To send: ')` ;
3. `msg` est envoyé au serveur, qui l'affiche.
4. Les deux parties terminent leur exécution.

Exercice Bonus B2.5 : Implémenter un client en TCP en Python

1. Modifiez votre code pour qu'il puisse se comporter comme un client *ou* comme un serveur, comme expliqué ci-dessus.
   (a) Modifiez votre fonction `parse_args` afin qu'elle récupère les nouveaux paramètres ;
   (b) Créez une fonction `client(ip: str, port: int)` vide (elle doit contenir une seule ligne : `pass`) ;
   (c) Modifiez votre `main` afin qu'il appelle la fonction `server` ou la fonction `client` suivant la valeur du paramètre `-l`, en leur passant les arguments dont elles ont besoin ;
   (d) Implémentez les fonctions `client` et `server` pour reproduire le fonctionnement ci-dessus.
2. Lancez deux terminaux, et testez la communication entre votre serveur et votre client Python.

8. Voir la section « Le paramètre *action* » de la documentation d'`argparse`.

Cet échange est assez rigide : le client et le serveur s'envoient des messages selon un ordre défini. Comment faire pour que le client puisse recevoir un message de son utilisateur (`input`) *ou* du serveur (`recv`) ? Il faudrait que le client puisse être en attente de deux fonctions bloquantes à la fois. Ce n'est pas réalisable avec un seul fil d'exécution.

Un serveur qui gère plusieurs clients en parallèle Dans le code précédent, le serveur agit *séquentiellement* : il ne peut traiter qu'un client à la fois. Ainsi lorsque l'on reçoit une connexion, les attentes de réponse sont bloquantes (i.e. il ne se passe plus rien tant que le client n'a pas répondu), et nous devons attendre que la connexion soit complètement terminée avant d'en traiter une autre.

Nous souhaitons maintenant être capable de gérer plusieurs connexions simultanées, c'est-à-dire que nous voulons être prêts à écouter et recevoir une connexion sur le socket serveur *avant* d'avoir terminé le dialogue avec la connexion précédente. Pour cela nous allons utiliser les fils d'exécution ou *threads*, qui permettent de gérer plusieurs actions en parallèle dans un même programme. Tout script ou module contient au moins un *thread*, correspondant au programme principal. En utilisant les librairies de *multi-threading*, nous pouvons lancer une fonction, et continuer à exécuter notre code *pendant* que la fonction s'exécute de son côté.

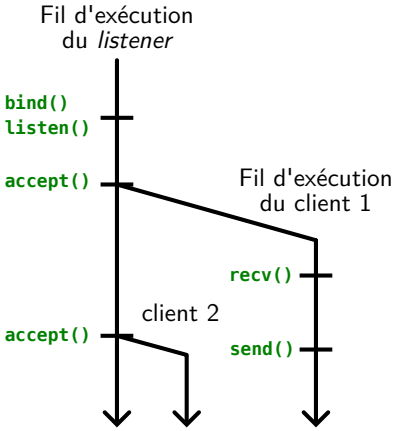


FIGURE 6 – Exécution du serveur sur plusieurs fils

Nous utiliserons la bibliothèque Python bas-niveau `threading` (`import threading`) pour gérer le multi-tâches. Notez qu'il existe d'autres librairies permettant l'exécution simultanée de code ou de connexion, mais (encore une fois), nous sommes là pour apprendre. Vous pouvez vous aider de cet article en français pour la trame du multi-threading.

Concrètement dans notre cas, notre fonction `server(...)` va lancer la *socket* serveur (`bind` & `listen`), puis, dans une boucle infinie :

1. Attendre une nouvelle connexion : `conn, addr = s.accept()` ;
2. Lancer une fonction pour gérer le client dans un nouveau *thread* :

```
t = threading.Thread(target=gestion_client, args(conn, addr))
t.start()
```

Pendant que le fil d'exécution de la fonction `server(...)` retourne attendre une autre connexion, la fonction `gestion_client` s'occupe d'écrire et de lire dans la socket du nouveau client.

Exercice Bonus B2.6 : Implémenter un serveur multi-utilisateurs en TCP en Python

1. Rédigez une fonction `gestion_client(conn, addr)` qui envoie un message sur la *socket* `conn`, attend un message en retour, affiche l'adresse du client et son message avec `print`, puis ferme la *socket*.
2. Modifiez votre fonction `server(...)` afin qu'elle appelle `gestion_client` dans un nouveau *thread*, comme vu ci-dessus.
3. Lancez un terminal pour exécuter votre serveur, afin qu'il écoute sur un port de votre choix.
4. Sur deux terminaux suplémentaires, lancez votre programme en mode client, pour tester la communication avec votre serveur.
5. Notez les échanges entre le serveur et ses clients dans votre README.
6. Et n'oubliez pas de faire un commit de vos modifications, et de les pousser vers le dépôt Git !

Une bonne pratique lorsque l'on fait du multi-tâches est de maintenir une liste des tâches en cours,

ou dans notre cas une liste des connexions ouvertes. Une ressource utile (et en français) pour cela peut être trouvée sur le site de developpez.com.

Transférer les messages reçus d'une *socket* à l'autre Notre serveur sait désormais gérer plusieurs connexions à la fois, mais ne sait pas encore transmettre les messages d'un utilisateur à l'autre. Il faudrait trouver une méthode pour que la réception d'un message dans un fil exécutant `gestion_client` soit retransmis sur les autres *sockets* client... On comprend vite qu'il va falloir commencer par stocker la liste des connexions quelque part, par exemple en modifiant la fonction `server(...)` :

```
def server(port: int):
  connections = dict()
  # [...]
  while (True):
    conn, addr = s.accept()
    # On stocke les connexions dans un dictionnaire, pour que chaque connexion
    # soit identifiée par l'adresse du client à laquelle elle est connectée
    connections[addr] = conn
    # [...]
```

On imagine déjà une fonction en charge de transmettre un message `msg` reçu d'un client `sender` à toutes les connexions *sauf celle de l'expéditeur* : `forward_message(connections, msg, sender)`.

Très bien, mais qui exécute `forward_message` ? Le fil de `gestion_client` n'a pas que ça à faire : il doit se concentrer sur la réception des messages de son clients. Il va falloir faire un autre *thread*. Il existe au moins deux solutions, la naïve et la bonne :

1. **Solution naïve :** On passe le dictionnaire `connections` à chaque fonction `gestion_client`, et on lance un fil pour exécuter `forward_message` à chaque réception d'un message.
2. **Solution *thread-safe* :** On utilise une file ou *queue*, qui est un objet *thread-safe* (capable d'être appelé par plusieurs fils d'exécution sans faute), et on utilise un *unique thread* pour gérer toutes les transmissions de messages dans une boucle.

En effet, le dictionnaire `connections` est une variable *partagée par plusieurs threads*, que ce soit pour y ajouter des connexions (dans la fonction `server`) ou pour itérer dessus à l'envoi d'un message (dans `forward_message`). Il risque de se passer des choses étonnantes si `connections` est modifié pendant qu'il est lu dans un autre fil d'exécution. C'est pourquoi, en programmation parallèle ou *multi-threads*, on essaye de minimiser les fils qui accèdent à des variables partagées. On a pour cela recours à des objets *thread-safe* qui manipulent des messages, comme la file présentée ci-dessus.

D'autant qu'on n'a pas traité un cas important : si un client se déconnecte, comment supprimer sa *socket* du dictionnaire `connections` ?

Exercice Bonus B2.7 : Serveur de discussion

1. Implémentez la retransmission des messages des clients avec la solution de votre choix.
2. Testez votre programme en exécutant plusieurs clients. Faites partir et revenir certains clients, pour voir comment votre serveur supporte ces connexions/déconnexions.
   Si vous observez des comportements intéressants, notez-les dans votre README.
3. Implémentez la suppression des connexions closes si ce n'est pas déjà fait.
4. Testez votre programme. Notez dans votre README ce qu'il se passe.
5. Si vous observez des bugs, essayez de proposer des solutions dans votre README.
6. Sauvegardez votre travail.

Vous avez mis le nez sur un des problèmes fondamentaux de l'informatique : il y a quasi-toujours plusieurs acteurs qui souhaitent accéder aux mêmes informations. En Licence Informatique, vous apprendrez des méthodes théoriques et appliquées pour y remédier.

3 Réalisation d'un serveur web

L'heure est venue pour nous de fabriquer notre propre serveur web, ou serveur HTTP.

Comme on a déjà pu le voir, son rôle est d'attendre les requêtes HTTP de ses clients, et de leur transmettre en retour les *ressources* web qu'ils ont demandé—comme des fichiers HTML, CSS, Javascript (JS), des media (images, audio, vidéo), mais pas seulement. Avant d'attaquer la pratique, nous commencerons encore une fois par un peu de théorie.

3.1 Généralités

Nous traiterons d'abord de l'historique et des propriétés de HTTP en général, avant de présenter, dans le cadre du TP, comment va fonctionner notre serveur web.

Historique Le protocole HTTP fut proposé entre 1989 et 1991 par Tim Berners-Lee et son équipe de recherche au CERN. L'équipe proposa dans le même temps le protocole HTML, le premier navigateur, et bien sûr le permier serveur web. Le protocole a connu assez peu de versions depuis 30 ans qu'il existe :

- La version beta 0.9 (HTTP/0.9) existe depuis 1991, suivie de HTTP/1.0 en 1996, et rapidement de HTTP/1.1 en 1997.
  En 2014, après 17 ans d'existence, la spécification de HTTP/1.1 (la RFC 2616) a été augmentée et scindée en huit, sans mise à jour fonctionnelle du protocole. Rien que de la clarification !
- En 2015, c'est HTTP/2 qui fait son entrée, poussé par Google, décidé à améliorer la performance des échanges sur le web. Pour ce faire, le serveur réutilise notamment la même connexion TCP avec son client pour répondre à plusieurs requêtes d'affilée, plutôt que d'ouvrir une connexion par ressource demandée.
- Enfin, HTTP/3 est officiellement validé en 2022, toujours sous l'impulsion de Google, et toujours dans un but de performance. C'est carrément la dépendance de HTTP à TCP qui est abandonnée dans cette version, en faveur du nouveau protocole de transport QUIC, basé sur UDP.

Références

- Stéphane Bortzmeyer, Présentation de la RFC2616 (HTTP/1.1), billet de blog, 2007.
- Stéphane Bortzmeyer, La norme HTTP 1.1, nouvelle rédaction, billet de blog, 2014.
- L'évolution du protocole HTTP, documentation, Mozilla Developer Network (MDN), 2023.

Ressource Il est temps d'avouer que *ressource* n'est pas un simple synonyme de *fichier*. En effet, les fichiers sont statiques (ils ne changent pas à chaque requête), tandis que de nombreuses ressources web sont *dynamiques*. Une simple recherche sur votre moteur de recherche préféré en fera la démonstration.

Mais qu'est-ce donc qu'une ressource, si on définit à la fois par ce terme fichiers statiques et résultats de recherche dynamiques ? Faute de pouvoir être plus précis, on définit une ressource web comme étant **tout objet ou entité qui a une identité : qui est identifiable par une *adresse web*.**

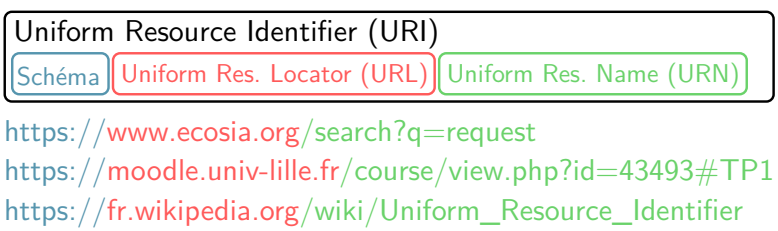


FIGURE 7 – Anatomie d'une adresse web, ou plus précisément d'une Uniform Resource Identifier (URI).

Adresse web ou URI On connaît les adresses web ou Uniform Resource Identifier (URI) au moins pour avoir déjà utilisé la barre d'adresses de notre navigateur Internet. On les utilise pour se rendre sur un site Internet, ou pour partager un contenu web. Une URI identifie l'adresse d'une *ressource*, de façon *universelle* et de préférence *unique*.

Concernant HTTP, sa morphologie est notamment spécifiée dans la section 4 de la RFC9110. Comme représenté en figure 7, on peut découper une URI en trois parties :

- **Schéma :** On commence toujours par définir la famille ou schéma de l'URI. Le bien connu `https://` indique que l'on définit une adresse de la famille HyperText Transfer Protocol Secure (HTTPS). Il en existe d'autres, comme `ftp://` pour accéder à une ressource utilisant le protocole File Transfer Protocol (FTP).
- **Uniform Resource Locator (URL) :** C'est la partie de l'URI qui sert à *localiser* le fournisseur de la ressource. En HTTP, c'est donc le nom du domaine ou site web qui héberge la ressource identifiée. Le protocole Domain Name System (DNS) a notamment pour rôle de convertir les URL (interprétables par les humain·es) en adresses IP (interprétables par les routeurs du net). C'est donc un protocole très important sur Internet.
  Par abus de langage, on utilise souvent le terme « URL » pour désigner une URI.
- **Uniform Resource Name (URN) :** Identifie une ressource au sein d'un domaine. On a déjà utilisé l'URN `/`, la *racine* du domaine, en fabriquant la requête HTTP `GET / HTTP/1.1`. L'URN peut contenir plusieurs parties :
  - L'identifiant de la ressource à proprement parler ;
  - Ses *paramètres* après un point d'interrogation `?`, pour préciser la requête ;
  - Une *ancre* commençant par un dièse `#`, pour désigner un endroit précis au sein de la ressource.

  On voit dans la figure 7 que sur le moteur de recherche `www.ecosia.org`, la ressource `/search` prend en paramètre `q=request`. La variable `q` contient donc la requête de recherche de l'utilisateur·ice. Sur le lien Moodle, on voit une ancre : `#TP1` ; l'URI représente une section précise d'un cours.

Les URI couplées aux *hyperliens* HTML forment le système *hypertexte*, qui fait toute la superbe de la navigation sur Internet—comparé par exemple à la navigation par références bibliographiques dans des livres. Le plaisir que l'on peut prendre à « se perdre sur Wikipédia » tient pour beaucoup à ce mode d'exploration qui encourage la sérendipité [9].

Requête Comme nous l'avons déjà vu, une requête HTTP envoyée par un client est constituée en premier lieu d'une ligne d' en-tête, par exemple `GET /index.html HTTP/1.1`. Cette ligne se décompose en trois partis : le « verbe » HTTP, l'adresse de la *ressource* requêtée, et la version de HTTP utilisée par la requête. Dans le cadre de ce TP, nous ne verrons que le verbe `GET`, qui indique qu'on cherche à récupérer une ressource—mais il en existe d'autres, comme `POST` (pour envoyer des informations) ou `DELETE` (pour en supprimer). Le serveur web étant en charge d'un domaine (identifié par une URL), il ne reçoit de l'adresse de la ressource que son URN.

Après cette première ligne d'en-tête, viennent divers paramètres qui ajoutent de l'information ou spécifient la requête, dont :

`Host:` Déjà abordé, il spécifie le domaine que souhaite joindre le client (plusieurs domaines pouvant être hébergés sur une même machine).

`User-Agent:` Ce champ apporte de l'information sur le client : son navigateur et sa version, son système d'exploitation et sa version, etc. Très important dans les années 2000, à l'époque ou des navigateurs comme Internet Explorer et Firefox n'interprétaient pas du tout HTML de la même manière ; la grande quantité d'information partagée par le `User-Agent` pose aujourd'hui des problèmes de vie privée.

Il est commun d'inclure la valeur de ce champ dans les *logs* conservés par le serveur, puisqu'il apporte des informations supplémentaires sur l'origine de la requête.

`Accept-Language:` Ce champ renseigne la langue du contenu attendue par le client. C'est grâce à ce champ que les pages web sont parfois automatiquement traduites dans notre langue (celle qui

9. On peut définir « sérendipité » comme le fait de faire d'heureuses découvertes en se promenant au hasard.

est configurée sur notre système).

`Accept:` Ce champ renseigne le type de contenu attendu par le client. On ne va pas en faire grand cas ici, mais on peut souhaiter recevoir la même information formatée dans différents formats : HTML si l'on est dans un navigateur, ou JSON/XML si l'on utilise un logiciel pour accéder au même service. Par exemple, l'application de communication WhatsApp est accessible *via* un navigateur web, et *via* une application mobile.

Réponse La réponse du serveur démarre elle aussi par un en-tête, qui commence par une ligne du type `HTTP/1.1 200 OK`. Après la version de HTTP parlée par le serveur, on voit apparaître le *code de statut*, qui décrit comment s'est passée la requête. On connaît souvent l'inénarable code d'erreur « *404 not found* ».

Le consortium Internet Assigned Numbers Authority (IANA) [10] définit formellement la liste des codes de réponse correspondant aux différents scenarii possibles. Pour une version plus agréable de cette liste, on peut se référer à la documentation de Mozilla sur le sujet. Ces codes ont été repris dans les diverses spécifications de HTTP et sont classés par type de réponse : succès (2xx), redirections (3xx), erreurs client (4xx) et erreurs serveur (5xx). Les codes les plus courants sont :

`200 - OK` Tout s'est bien passé, la page a été trouvée et est retournée avec la requête.

`401 - Unauthorized` Non autorisé. La resource nécessite des privilèges que la requête ne fournit pas [11].

`403 - Forbidden` Interdit. L'accès à la resource est interdit. Utiliser une authentification n'aidera pas.

`404 - Not Found` La resource demandée n'a pas été trouvée.

`418 - I'm a Teapot` Le serveur refuse de faire du café, étant, et de façon permanente, une théière.

`500 - Internal Error` Le serveur a rencontré un problème qui l'empêche de traiter la requête.

`501 - Not implemented` Le serveur ne fournit par le service demandé.

La suite de l'en-tête contient divers champs, dont les deux plus importants sont :

`Content-Length` La taille de la ressource en octets. Le client en a impérativement besoin pour savoir quand arrêter de lire sur sa *socket* TCP.

`Content-Type` Comme nous le verrons lors du TP, HTTP attend que l'on renseigne le type des ressources transmises, sans quoi le client ne sait pas quoi en faire.

Fonctionnement d'un serveur web Un serveur HTTP est en charge de la réception et du traitement des requêtes client au sein d'un domaine identifié par une URL (i.e. d'un site web). Il reçoit uniquement des URN, en première ligne de chaque requête HTTP (juste après le type de requête, e.g. `GET`). Il peut renvoyer une variété de réponses au client, notamment pour gérer des cas d'erreur. Du contenu est souvent renvoyé en plus de l'en-tête de réponse—parfois dynamique, comme nous l'avons vu avec l'exemple du moteur de recherche.

Le fonctionnement général du traitement de requêtes par un serveur web est représenté en figure 8. **Dans le cadre du TP**, nous allons réaliser un « simple » serveur de fichiers. Sauf exception, l'exposé va donc recommencer à utiliser « ressource » et « fichier » indifféremment.

Détaillons les étapes présentés en figure 8 :

1. **Recevoir un message** Comme vu au TP2, un serveur web doit être à l'écoute de nouvelles connexions TCP, accepter celle de son nouveau client, et lire la *socket* pour récupérer sa requête.
2. **Interpréter la requête** Une fois la requête reçue, encore faut-il la comprendre. Le serveur, connaissant le protocole, s'attend à lire une requête HTTP bien formée. Si tel est le cas, il en extrait une représentation utile à la suite de son programme (on utilisera un `dict()` Python). Sinon, il passe directement à l'étape 4, et prépare un message d'erreur.

10. Autorité très sérieuse, et à la vie sans aucun doute *excitante*. Les nombres sont très importants dans nos sociétés. Il est néanmoins vrai qu'ils sont responsables de la disparition du code fondamental `418 I'm a teapot`.

11. En général, cela implique d'utiliser un en-tête *Authorization*.

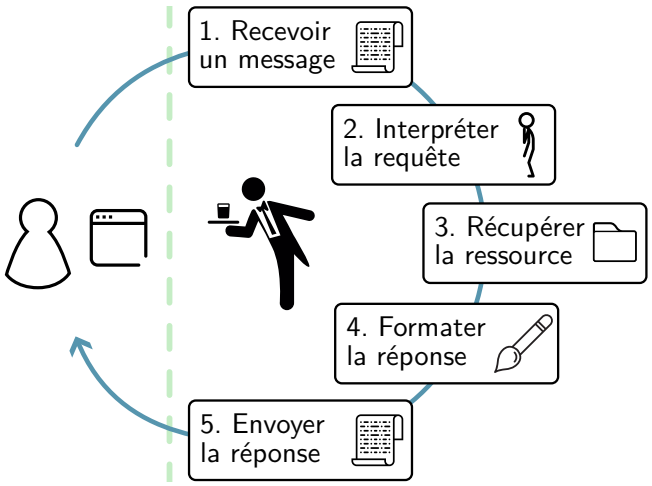


FIGURE 8 – Étapes générales du traitement d'une requête par un serveur web

3. **Récupérer la ressource** Notre serveur ne sachant que fournir des fichiers, il ne répondra qu'aux requêtes `GET`, et ira chercher sur son système le fichier correspondant à l'URN requêtée. Cette étape d'accès au système de fichiers comporte quelques difficultés :
   — Traduire la bonne adresse du fichier *sur disque* à partir de l'URN ;
   — Gérer les cas d'erreurs : fichier non existant, ou à l'accès interdit.
   Un serveur plus avancé que notre serveur de fichiers pourrait ici effectuer d'autres opérations, comme *ajouter* des informations sur les disques ou les bases de données du serveur (un nouveau billet de blog, une transaction bancaire...), ou encore générer un contenu *dynamique* en interrogeant ses bases de données.
4. **Formater la réponse** Une fois le fichier récupéré (ou l'erreur émise), il ne reste plus qu'à l'emballer dans un beau message HTTP, en utilisant le code de statut et les champs d'en-tête appropriés.
5. **Envoyer la réponse** Il ne reste plus qu'à envoyer, dans la *socket* de la connexion avec le client : l'en-tête que l'on vient de fabriquer, une ligne vide, et la ressource. Dans notre cas, on ferme ensuite la *socket*, pour retourner attendre de nouvelles connexions.
   Rappelons que c'est une façon de procéder très inefficace : mettre en place une connexion TCP prend du temps, et il serait donc plus pertinent de conserver cette *socket* ouverte en prévision de nouveaux échanges avec le même client [12]. Nous expliquions plus haut que la version HTTP/2 avait notamment pour but d'améliorer les performances de la sorte.

3.2 Préparation du projet

Nous vous fournissons un projet Python nommé « `myserver` ». Il s'agit d'un projet *exécutable*, c'est à dire qu'on doit le lancer avec la commande suivante :

```
$ python3 -m myserver PARAMÈTRES
```

Le projet contient des tests et des fichiers sources, dont la plupart sont incomplets. Ce sera à vous de les remplir, en implémentant les fonctions vides. Dans l'immédiat les fichiers complets sont les suivants :

— `__main__.py` est le point d'entrée du programme quand il est lancé comme exécutable (comme ci-dessus). Il ne contient qu'un import, et un appel à l'analyse de la ligne de commande avec `cli.main()`.

— `__init.py__` définit les fonctions du module disponibles quand on importe notre projet sous forme de bibliothèque (`import myserver`) : aucune, puisque nous écrivons un exécutable, pas une bibliothèque.

12. En effet, une requête HTTP ne vient (presque) jamais seule : une fois que notre client aura récupéré une page HTML, nul doute qu'il voudra obtenir les fichiers CSS & JS associés, ainsi que toutes les images dont nous avons ponctué notre page.

— `cli.py` contient la mécanique pour exécuter notre serveur en ligne de commande. Cela revient à traiter les paramètres de ligne de commande avec `argparse`, puis à appeler la suite du programme : ici `server.serve(...)`.
— `date.py` contient l'utilitaire de gestion de dates, vu dans le TP1.
— `http.py` contient des informations spécifiques au protocole HTTP, dont on aura besoin par la suite.

Vous aurez à compléter, tout au long de ce TP, les fichiers suivants :

— `server.py` pour la mécanique de gestion des connexions avec les clients. Seule la fonction `serve(...)` est déjà rédigée : elle crée une *socket* de serveur, et accepte les connexions client.
— `file.py` pour la gestion des opérations sur le système de fichier.
— `http_request.py` pour l'analyse des requêtes. Fonctions à implémenter : `parse_request`, `parse_request_head`, `parse_request_params`.
— `log.py` pour consigner sur la sortie standard les actions effectuées (requêtes services, ressources fournies, etc.).

Exercice 3.1 : Préparer le projet

1. Téléchargez l'archive contenant l'arborescence de projet, et extrayez-la dans un nouveau sous-répertoire de `~/dev/` nommé « `serveur_web` ».
2. Initialisez votre projet comme vu dans les chapitres précédents :
   (a) Créez et activez l'environnement virtuel :
      — `python3 -m venv venv/`
      — `source venv/bin/activate`
   (b) Installez `pdm`, et initialisez des dépendances :
      — `pip install pdm`
      — `pdm install`
   (c) Installez le projet de développement dans votre environnement : `pip install --editable .`
   (d) Vérifiez que `pytest` est installé et exécutez les tests.

En exécutant `pytest`, on voit que de nombreux tests ne passent pas :

```
tests/test_file.py FFFF                                  [ 12%]
tests/test_http_request.py FFF.FFFFFFFFFF                [ 54%]
tests/test_log.py FFFFFFF                                [ 75%]
tests/test_server.py .FFFFFFF                            [100%]
```

Exercice 3.2 : Utiliser un navigateur pour se connecter

1. Dans un terminal, *en prenant soin d'avoir activé votre environnement virtuel*, placez-vous dans le répertoire contenant votre projet et exécutez le module :

   ```
   (venv) $ python -m myserver
   usage: myserver [-h] -p PORT -r ROOT
   myserver: error: the following arguments are required: -p/--port, -r/--root
   ```

   La commande d'aide (définie dans le fichier `cli.py`, allez y jeter un oeil) nous indique quelles sont les options à passer au module pour qu'il s'exécute correctement. Pratique !
2. Ré-exécutez la commande en spécifiant le port 8000, et en fournissant en tant que racine (*root*)

le répertoire `tests/resources/www/`.

Lancez votre navigateur préféré (e.g. Mozilla Firefox) et pointez-le sur l'adresse http://localhost:8000/. Que voyez-vous ?

3. Le navigateur ouvre une connexion, et reste en attente d'un contenu. C'est normal, car la fonction `server.serve()` se contente d'accepter la connexion (`socket.accept()`) et délègue son traitement à la fonction `handle_client()`, qui ne fait rien pour l'instant (sa seule instruction est `pass`).
4. Notez vos observations dans votre fichier README (sans oublier de faire un commit, bien sûr !).

La plupart des navigateurs permettent de visualiser les requêtes transmises et reçues. Pour Firefox par exemple, la fonctionnalité « Web Developer Tools » affiche beaucoup d'informations sur le traitement de la page, et offre notamment un onglet « Réseau » permettant de voir les requêtes associées à la page affichée. À ce stade, nous pouvons voir qu'il n'y a pas grand chose, et notamment pas de code de statut.

3.3 Gestion des connexions

La première chose à faire pour que le navigateur ne reste pas à attendre indéfiniment, c'est de fermer la *socket* que le serveur ouvre avec le client.

Exercice 3.3 : Implémentation de la gestion des connexions

1. Dans la fonction `handle_client` du fichier `server.py`, fermez la *socket* client `c`.
2. De la même manière qu'à l'exercice précédent, relancez votre serveur, et pointez votre navigateur pour qu'il envoie une requête au serveur.
3. La plupart des navigateurs permettent de visualiser les requêtes transmises et reçues.

   Pour Firefox par exemple, la fonctionnalité « Web Developer Tools » affiche beaucoup d'informations sur le traitement de la page, et offre notamment un onglet « Réseau » permettant de voir les requêtes associées à la page affichée.

   Qu'affiche l'onglet « Réseau » de la console de développement de votre navigateur ?
4. Notez vos observations dans le README, et commitez votre travail.

On voit dans la console que la requête semble bien partie, mais qu'aucun résultat n'est reçu :

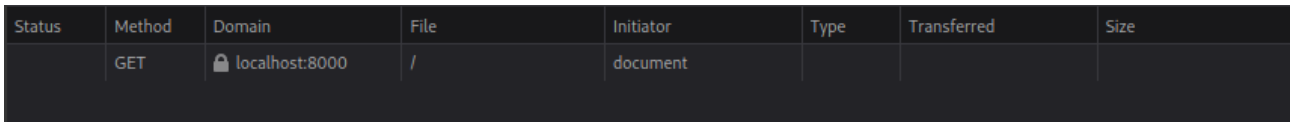


FIGURE 9 – Liste des requêtes dans l'onglet « Réseau » de la console de développement de Firefox.

Nous allons maintenant utiliser la connexion client `c` : d'abord pour lire la requête que le navigateur nous envoie, et après, pour envoyer une réponse.

3.4 Interprétation des requêtes

Commençons par analyser et interpréter les requêtes reçues du client (étape 2. de la figure 8). Avant d'interpréter la requête, cherchons déjà à l'afficher.

Exercice 3.4 : Lecture de la requête

1. Dans la fonction `handle_client` du fichier `server.py`, *avant de fermer la socket*, lisez son contenu dans un *buffer* `buf`, et affichez le résultat avec `print(buf)`.
   Référez-vous au TP précédent si vous ne vous souvenez plus comment lire le contenu d'une *socket*.
2. Relancez votre serveur, pointez votre navigateur dessus, et notez dans votre README ce qui s'affiche dans le terminal où s'exécute le serveur.
   On obtient plus d'*output* qu'on s'y attendait, non ? Votre navigateur, ne voyant pas de réponse arriver à sa requête, semble insister auprès du serveur.
   Combien de fois le navigateur essaye-t-il de se connecter au serveur avant d'abandonner ?
3. Faites un commit, et poussez votre travail sur la forge.

Nous allons maintenant interpréter cette requête, en générant une structure de données dictionnaire `dict()` à partir du *buffer* `buf`.

Structure de données retournée par `parse_request` Elle devra comprendre deux sous-dictionnaires :
— celui à la clé `head` qui définit l'action à mener (typiquement un `GET`) et la ressource demandée ;
— celui à la clé `params` qui liste les options envoyées par le client.

La fonction `parse_request` retournera par exemple le dictionnaire suivant après avoir interprété une requête d'un navigateur classique (Firefox sous Debian) [13] :

```
{
  'head': {
    'verb': 'GET',
    'resource': '/index.html'
  },
  'params': {
    'Host': 'localhost:8000',
    'User-Agent': 'Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0',
    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8',
    'Accept-Language': 'en-GB,en;q=0.5',
    'Accept-Encoding': 'gzip, deflate, br',
    'Connection': 'keep-alive'
  }
}
```

Le sous-dictionnaire `'head'` contient deux informations issues de la première ligne de la requête : le « verbe » et l'identifiant de la ressource (son URN). Comme déjà expliqué section 3.1, nous ne traiterons que les commandes `GET`, qui permettent de récupérer du contenu—typiquement une page web, avec ses images, ses feuilles CSS, et ses scripts.

Le sous-dictionnaire `'params'` contient les paramètres issus des lignes suivantes de la requête. Ces paramètres apportent de l'information sur le client, ou précisent la requête, comme expliqué précédemment.

Dans les exercices restants de la section 3.4, vous n'avez pas besoin de relancer le serveur ni le navigateur : **votre travail consiste à implémenter les fonctions du fichier `http_request.py` de sorte à ce que les tests du fichier `tests/test_http_request.py` passent.** Nous commencerons par les fonctions `parse_request_head` et `parse_request_params`, qui s'occupent respectivement

13. Nous avons enlevé quelques lignes par souci de visibilité.

d'interpréter la première ligne de la requête, et les suivantes. Nous pourrons ensuite nous intéresser à `parse_request`, la fonction qui sera directement appelée par `handle_client` du fichier `server.py` pour transformer la requête brute du client en un dictionnaire interprétable par notre code.

3.4.1 Interpréter la première ligne de la requête

La fonction `parse_request_head`, dont l'unique paramètre `line` est la première ligne d'une requête, doit l'interpréter, afin de retourner un dictionnaire contenant deux champs `'verb'` et `'resource'`. Ces deux champs ont respectivement comme valeur le « verbe » HTTP de la requête (en majuscules), et la ressource demandée.

Notre premier objectif est que le test `test_parse_request_head` passe. Il est contenu dans le fichier `tests/test_parse_request.py`, et effectue deux tests :

— La chaîne `line` contenant `'GET / HTTP/1.1'` doit donner en résultat :
`{'verb': 'GET', 'resource': '/'}`

— Celle contenant `'options /assets/style.css HTTP/1.1'` doit donner :
`{'verb': 'OPTIONS', 'resource': '/assets/style.css'}`
Notez que l'entrée contient `options` en minuscules : elle doit être transformée en majuscules (`OPTIONS`) par notre fonction.

Notre second objectif est que les requêtes invalides retournent une erreur, ce qui sera vérifié par le test `test_parse_request_head_invalid`. Il faudra notamment s'assurer que la chaîne `line` contienne bien 3 parties (le verbe, la ressource, et la version de HTTP). Nous allons pour ce faire **lever une erreur** quand `line` est invalide.

Exercice 3.5 : Implémentation de la fonction `parse_request_head(line: str)`

1. Étant donné la ligne `line`, nous avons tout d'abord besoin de la découper en ses trois segments séparés par des espaces : le verbe, la ressource, et la version de HTTP (que nous n'utiliserons pas).
   Nous avons déjà vu comment faire lors du TP1, dans l'exercice 1.17 : en utilisant la fonction `str.split(SEP)`, où `SEP` est le *séparateur* autour duquel on souhaite découper les segments (ici `' '`).
   Utilisez la méthode[a] `split` pour récupérer la liste des champs de `line` dans une variable appelée `fields`.
2. Assurez-vous que le champ « verbe » soit en majuscule, avec la méthode `str.upper()`.
3. Instanciez un dictionnaire contenant une clé `'verb'` qui a pour valeur le verbe HTTP en majuscules, et une clé `'resource'` qui a pour valeur la ressource demandée.
4. Faites retourner ce dictionnaire par la fonction.
5. Enfin, *juste après avoir découpé* la ligne `line`, effectuez un test et levez une `ValueError` quand elle ne contient pas 3 champs :

   ```
   if len(fields) != 3:
       raise ValueError(f"Request header is invalid: {line}")
   ```

6. Exécutez les tests. Pour voir le détail des tests qui passent, activez le mode *verbose* (verbeux) de `pytest` avec son paramètre `-v` :

   ```
   $ pytest -v
   ```

7. Enregistrez votre travail sur la forge git.

a. Une méthode est une fonction attachée à un objet, et qui agit dessus.

Vous aurez réussi l'exercice précédent quand passeront les deux tests `test_parse_request_head` et `test_parse_request_head_invalid`. Vous devriez voir affichées les trois lignes suivantes en sortie de la commande `pytest -v` :

```
tests/test_http_request.py::test_parse_request_head[GET / HTTP/1.1-expected0] PASSED
tests/test_http_request.py::test_parse_request_head[options /assets/style.css HTTP/1.1-expected1] PASSED
tests/test_http_request.py::test_parse_request_head_invalid PASSED
```

Gestion d'erreurs Vous n'êtes pas coutumier·es de renvoyer vos propres erreurs (aussi appelées *exceptions*)—vous avez plus souvent l'habitude de les subir. *Lever* une exception avec `raise` interrompt l'exécution d'une fonction, et retourne cette erreur à la fonction appelante de la même manière : en interrompant son exécution. Si la fonction appelante a entouré l'appel à notre fonction d'un bloc `try/except`, elle peut continuer son exécution en passant dans le segment `except`. Sinon, elle interrompt elle aussi son exécution pour lever l'exception. Si aucune fonction appelante n'a traité le cas d'erreur avec un bloc `try/except`, l'exception va remonter la pile d'invocation (ou *call stack*) jusqu'au `main`, et arrêter l'exécution de notre programme en affichant notre exception et la pile d'invocation remontée.

Pour désigner ce comportement qu'ont les exceptions de remonter la pile d'exécution jusqu'en haut, on dit en anglais « *to bubble up* », à la manière d'une bulle remontant à la surface, pour enfin éclater—notre programme.

3.4.2 Interpréter les paramètres de la requête

Passée la première ligne d'une requête HTTP se trouvent ses paramètres, au format `CLÉ: VALEUR`. Les clés que l'on peut trouver sont généralement issues de la liste des « en-têtes » HTTP officiels. Les valeurs sont des chaînes de caractères, systématiquement terminées par un saut de ligne. Notre but est de générer, à partir d'une requête, un dictionnaire ayant une entrée par paramètre, dont la clé est le champ `CLÉ`, et la valeur le champ `VALEUR` (logique). On utilisera ici aussi la méthode `str.split(SEP)` —à vous de déterminer quelle valeur donner à `SEP`.

Nous allons pour ce faire implémenter la fonction `parse_request_params` du fichier `http_request.py`.

Exercice 3.6 : Implémentation de `parse_request_params(lines: list[str])`

1. Commencez par créer un dictionnaire vide nommé `params`.
2. La fonction a en paramètre une *liste* de chaînes de caractères, contenant chacune une *ligne* de la requête (sauf la première, déjà traitée). Il va falloir itérer sur cette liste pour traiter les lignes une à une.

   Rédigez une boucle `for` pour traiter les lignes itérativement. Au sein de cette boucle :

   (a) Séparez les deux champs à l'aide de la méthode `str.split(SEP)`, en choisissant judicieusement la valeur de `SEP`.

   (b) Testez la validité de vos champs :
   - Ils doivent être deux ;
   - Chacun d'entre eux doit contenir quelque chose (leur longueur ne doit pas être égale à zéro).

   Si une de ces conditions n'est pas validée, levez une exception `ValueError` :

   ```
   raise ValueError(f"Request line is not a valid key/value pair: {l}")
   ```

   (c) Enregistrez le premier champ dans une variable `cle` après vous être assuré·e que ledit champ ne contient pas d'espace inutile avec la méthode `str.strip()`.

   (d) De même, enregistrez le deuxième champ dans une variable `valeur`, en veillant à supprimer les espaces avant et après.

(e) Ajoutez ce couple clé/valeur au dictionnaire `params` :

```
params[cle] = valeur
```

3. Une fois la boucle terminée, faites retourner votre dictionnaire à votre fonction.
4. Assurez-vous que les tests passent avec `pytest -v`. Référez-vous à l'*output* des tests affiché après l'exercice.
5. Vérifiez que les tests passent, et sauvegardez votre travail sur la forge.

Vous devriez notamment voir affichée la sortie suivante à l'exécution de `pytest -v` :

```
tests/test_http_request.py::test_parse_request_params[params0-expected0] PASSED
tests/test_http_request.py::test_parse_request_params[params1-expected1] PASSED
tests/test_http_request.py::test_parse_request_params[params2-expected2] PASSED
tests/test_http_request.py::test_parse_request_params_invalid[params0] PASSED
tests/test_http_request.py::test_parse_request_params_invalid[params1] PASSED
tests/test_http_request.py::test_parse_request_params_invalid[params2] PASSED
```

Nous avons réussi à interpréter chaque ligne des requêtes qui pourront nous être envoyées par nos futurs clients HTTP.

3.4.3 Interprétation d'une requête HTTP complète

Il ne reste pour ainsi dire qu'un bout de scotch à mettre, pour ficeler tout cela et l'incorporer à notre code serveur. Mais avant, revenons sur une notion déjà abordée, très importante dans notre cas.

Retour sur la différence entre octets (`bytes`) et chaîne de caractères (`str`) Nous avons déjà mentionné en page 35 la différence entre le type Python `bytes` et le type `str`. Nous avons tout intérêt à y revenir, car la lecture d'une *socket* renvoie un tableau d'octets (`bytes`), tandis que nous avons jusqu'ici travaillé à l'interprétation de notre requête en utilisant des chaînes de caractères (`str`). Il faut dire qu'une *socket* peut être utilisée pour n'importe quel type de contenu : textuel (comme un en-tête HTTP) ou binaire (comme une image, de l'audio, une vidéo, une archive...). Or, pour le moment, nous souhaitons uniquement interpréter le contenu d'une requête HTTP, et nous avons déjà mentionné que HTTP était un protocole *textuel*. Nous pouvons donc sans risque décoder les octets reçus de la *socket* en caractères, car nous savons que l'on nous a uniquement envoyé du texte.

C'est la fonction `parse_request(buf: bytes)`, que nous allons maintenant implémenter, qui doit s'occuper de cette conversion. Vous constatez en effet qu'elle accepte en paramètre un tableau d'octets `buf` directement issu de la *socket*—et vous aurez compris qu'elle doit faire appel aux fonctions implémentées précédemment, qui elles attendent des `str` en paramètre. Il nous faut donc commencer par transformer le contenu du *buffer* `buf` lu depuis la *socket* en une chaîne de caractères.

```
buf_str = buf.decode('utf-8')
```

La ligne de code ci-dessus *décode* le tableau d'octets `buf` (de type `bytes`) en une chaîne de caractères (`str`) encodée en UTF-8, qui est un format pour représenter numériquement des caractères (et le seul que nous utiliserons).

Avant cela néanmoins, assurons-nous que le *buffer* ne soit pas vide, pour lever une `ValueError` dans le cas contraire. Pour ce faire, nous pourrions vérifier la longueur de `buf` avec `len(buf)`, et la comparer à `0`. Mais profitons de l'occasion pour vous présenter `b''`, la chaîne d'*octets* vide. Pour tester si la chaîne `buf` est vide, on peut aussi faire comme suit :

```
if buf == b'':
    raise ValueError("Received empty request")
```

En fait, pour rédiger à la main un tableau d'octets, il suffit de mettre un `b` devant les guillemets (qu'ils soient simples, doubles, ou multilignes), comme ceci :

Chaîne d'octets simple :

```
>>> a = b"coucou"
>>> a
b'coucou'
>>> type(a)
<class 'bytes'>
```

Chaîne multilignes :

```
>>> a = b"""Je suis sur
... plusieurs lignes"""
>>> a
b'Je suis sur\nplusieurs lignes'
>>> type(a)
<class 'bytes'>
```

La parenthèse refermée, retournons à nos moutons.

Exercice 3.7 : Implémentation de la fonction `parse_request(buf: bytes)`

1. Commencez par vérifier si la chaîne d'octets `buf` est vide, auquel cas remontez une `ValueError`, comme expliqué ci-dessus.
2. Décodez `buf` en une chaîne de caractères `buf_str` (ici aussi comme vu ci-dessus).
3. On veut maintenant découper notre `buf_str` en un tableau de *lignes*. On pourrait utiliser la désormais connue `str.split(SEP)`, mais une fonction `str.splitlines()` existe pour ce cas particulier (et pour de bonnes raisons). Utilisons-là plutôt, mais pas avant d'avoir supprimé les espaces et sauts de lignes surnuméraires du *buffer* :

   ```
   lines = buf_str.strip().splitlines()
   ```

   On vient d'enchaîner deux méthodes sur la même ligne, ce qui peut surprendre. La méthode `striplines()` s'exécute en fait *après* `strip()`, sur la valeur que retourne cette dernière (la requête privée de ses fameuses deux dernières lignes vides).
4. La variable `lines` étant un tableau qui contient un élément par ligne de la requête, on va pouvoir différencier la première ligne de la requête (`lines[0]`) des lignes suivantes (`lines[1:]`).

   (a) Faites appel à `parse_request_head(line: str)` en lui passant la première ligne de la requête en paramètre, en enregistrez le résultat dans une variable `req_head`.

   (b) Appelez `parse_request_params(lines: list[str])` en lui passant les lignes suivantes de la requête, et enregistre ce qu'elle renvoie dans la variable `req_params`.
5. Il ne vous reste plus qu'à retourner un dictionnaire contenant les deux champs `'head'` et `'params'` sus-mentionnés, dont les valeurs seront respectivement `req_head` et `req_params`.
6. Testez la fonction avec `pytest`.

   Vous devriez voir passer tous les tests du fichier `test_parse_request.py`.
7. N'oubliez pas de sauvegarder votre travail.

Pour terminer cette section d'interprétation de requête, nous allons vérifier le bon fonctionnement de nos fonctions avec une vraie requête, en ajoutant du code à `handle_client`, de sorte à afficher la requête du navigateur proprement interprétée, dans la sortie du terminal du serveur.

Exercice 3.8 : Utilisation de `parse_request` dans la fonction `handle_client`

1. Dans la fonction `handle_client` du fichier `server.py`, *juste après avoir lu la socket* dans

la variable `buf`, appelez la fonction `parse_request` en lui passant le *buffer*, et en récupérant le résultat dans une variable `req`.

2. Remplacez la ligne `print(buf)` par `print(req)`.
3. Lancez votre serveur, braquez votre navigateur dessus, et comparez la sortie du terminal du serveur à ce que vous trouvez dans l'onglet « Réseau » de la console de développement du navigateur. La requête envoyée par le navigateur devrait correspondre à celle reçue par le serveur.
4. Notez les similarités et différences observées dans votre README, et sauvegardez.

Vous verrez apparaître la même ligne de multiples fois dans votre terminal. L'explication est simple : comme le navigateur n'obtient pas de réponse à sa requête, il ré-essaie automatiquement un certain nombre de fois avant d'abandonner.

Vous pouvez vous féliciter : vous venez de réaliser un interpréteur de requête machine, une grande étape dans la réalisation d'un serveur applicatif !

3.5 Construction d'une réponse

Nous allons maintenant renvoyer du contenu au navigateur. D'abord sans nous soucier du protocole HTTP ni de sa requête, puis en en intégrant progressivement des fonctionnalités, jusqu'à arriver à un respect rigoureux du protocole. Il ne restera plus, dans la section suivante, qu'à lire nos fichiers locaux sur disque, pour les renvoyer au client, et nous aurons un serveur de fichier HTTP complet.

3.5.1 Renvoi de réponses prédéfinies

Commençons par réaliser un serveur « simple » : toujours content, répondant toujours la même chose. Il va nous falloir une « page » web à retourner. En voilà une (le fameux « hello world »), en-tête compris :

```
_REPLY_200 = b"""HTTP/1.1 200 OK
Server: RegardeMamanJeFaisUnServeurWeb/0.1
Date: Wed, 20 Mar 2024 16:36:42 GMT
Content-type: text/plain
Content-Length: 14

Hello, world!
"""
```

Notez le `b` en début de la chaîne multilignes, signifiant que la variable `REPLY_200` est une chaîne d'octets de type `bytes`. On remarque aussi l'en-tête, décrivant le serveur, et la réponse. Comme indiqué dans le champ `Content-Length`, le contenu retourné pèse 14 octets en comptant le saut de ligne final. Son type (`Content-type`) est `text/plain`, un bête « fichier » texte, et non du HTML.

Exercice 3.9 : Implémentation d'un serveur « simple »

1. Dans le fichier `server.py`, copiez-collez le code ci-dessus après la ligne 25 (après la définition de `_SERVER_ADDR`) pour faire de `_REPLY_200` une variable globale interne au fichier.
2. Dans la fonction `handle_client` du même fichier, après avoir lu et affiché le contenu de la *socket*, répondez en écrivant le contenu de `_REPLY_200` dans la *socket*.
   Référez-vous à la page 34 pour un rappel sur l'écriture sur une *socket*.
3. Assurez-vous que vous fermez toujours bien la *socket* avant la fin de la fonction.
4. Réexécutez le serveur, puis pointez à nouveau votre navigateur vers l'URL servie, par exemple `http://127.0.0.1:8000/` si vous utilisez toujours le port 8000.

Que voyez-vous ? Notez vos observations dans votre README.

5. Affichez la console de développement du navigateur à l'onglet « Réseau ». Quelles sont les requêtes envoyées par le navigateur ? Quels sont les codes et les types de réponse fournis ? Répondez dans votre README.
6. Demandez une autre page depuis votre navigateur, comme `http://127.0.0.1:8000/admin/login`.
   Inspectez de la même manière les réponses reçues, et notez vos observations dans votre README.
7. Sauvegardez votre travail.

La figure 10 montre l'onglet « Réseau » de la console de développement de Firefox, quand on lui demande d'afficher une page avec notre serveur « simple ». On observe qu'une réponse HTTP valide est retournée à chaque fois. Par contre, c'est toujours la même : *favicon*[14] comprise.

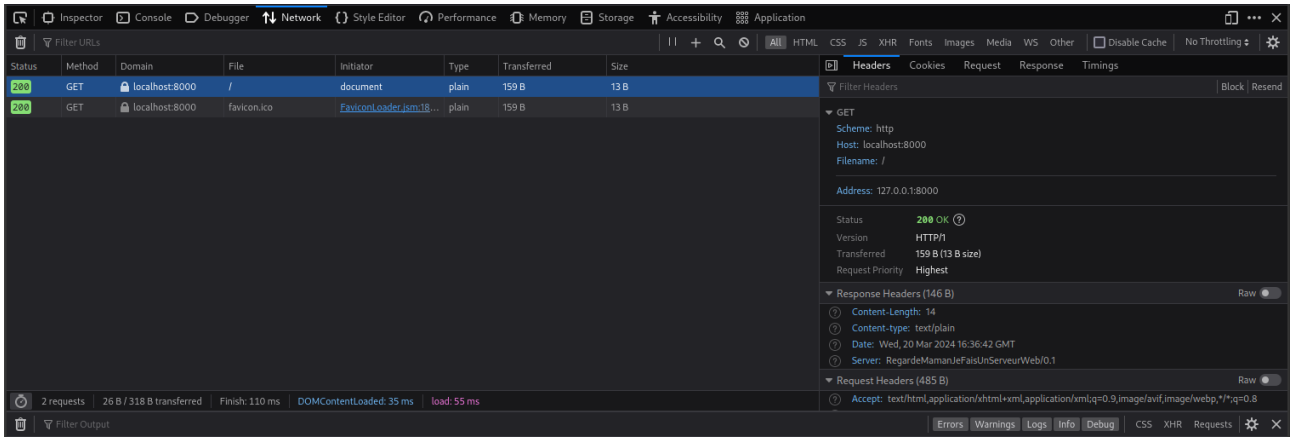


FIGURE 10 – Liste des requêtes dans Firefox avec les status codes et les headers.

Vous pouvez noter au passage, dans le terminal d'exécution du module, que le navigateur n'essaie plus de se connecter frénétiquement : comme nous répondons à sa requête, il ne la renvoie pas et nous n'avons qu'une ligne qui apparaît dans le log.

Nous renvoyons pour l'instant toujours la même chose, quelque soit la requête. Pour améliorer cela, nous allons utiliser les fonctions développées dans l'exercice précédent pour analyser ce que le client nous envoie, et adapter notre réponse en fonction.

Utilisation de la requête dans le choix de la réponse Pour améliorer sensiblement notre serveur, nous allons utiliser les fonctions développées dans l'exercice précédent : analyser la demande du client, et adapter notre réponse en fonction.

Nous aurons pour ce faire besoin d'autres « pages »—d'erreur, cette fois-ci. Voilà deux variables contenant l'intégralité d'une réponse `404 Not Found` et `501 Not Implemented`, deux codes de réponse primordiaux pour tout serveur web qui se respecte :

```
_REPLY_404 = b"""HTTP/1.0 404 Not Found
Server: RegardeMamanJeFaisUnServeurWeb/0.1
Date: Wed, 20 Mar 2024 16:36:42 GMT
Content-type: text/plain
Content-Length: 14

404 Not Found
```

14. Une *favicon* est une petite image toujours demandée par les navigateurs : elle sera affichée comme icône à côté de l'onglet du navigateur.

```
"""

_REPLY_501 = b"""HTTP/1.0 501 Not Implemented
Server: RegardeMamanJeFaisUnServeurWeb/0.1
Date: Wed, 20 Mar 2024 16:36:42 GMT
Content-type: text/plain
Content-Length: 20

501 Not Implemented
"""
```

Exercice 3.10 : Analyse de la requête

1. Dans le fichier `server.py`, copiez-collez les définitions des deux variables `_REPLY_404` et `_REPLY_501`.
2. Dans la fonction `handle_client`, remplacez la ligne où vous envoyiez systématiquement `_REPLY_200` par un bloc `if/elif/else`, où vous répondrez différemment en fonction du contenu de la requête contenu dans le dictionnaire `req` :
   (a) Si la requête n'a pas pour « verbe » `GET`, écrivez une erreur `501` préformatée (`_REPLY_501` dans la `socket`.
   Cette erreur (du serveur) signifie que la requête est sans doute valide, mais que le serveur n'a pas implémenté cette partie de la spécification. De fait, comme nous ne comptons créer qu'un serveur de fichiers, il ne comprendra qu'une seule méthode de requête (ou verbe) : `GET`.
   (b) Sinon, si la requête demande la ressource `/`, renvoyez notre « page » `200`.
   (c) Sinon, renvoyez l'erreur `404` fournie.
   Cette erreur (du client) est renvoyée quand le client demande une page non existante. On considère donc ici que seule la page d'accueil ou *racine* `/` est définie sur notre serveur.
3. Après le bloc `if`, n'oubliez pas de fermer la *socket*.
4. Lancez votre serveur, et accédez à ces deux pages avec votre navigateur (dans l'hypothèse où vous avez lancé votre serveur sur le port `8000`) :
   — http://127.0.0.1:8000/
   — http://127.0.0.1:8000/journal_intime.html
   Vérifiez ce qu'il se passe dans l'onglet « Réseau » de la console de navigateur. Qu'est-il arrivé à la requête pour la *favicon* ? A-t-on pu vérifier l'affichage de la page d'erreur `501` ? Pourquoi ? Décrivez vos observations dans le README.
5. Sauvegardez votre prose sur git.

Nous disposons d'un serveur interprétant basiquement la requête utilisateur, mais qui renvoie des réponses prédéfinies en variables globales. La prochaine étape consiste à fournir des réponses appropriées à la requête, et surtout, à fournir des en-têtes correspondant à la réponse.

3.5.2 Génération dynamique de l'en-tête

Champs d'en-tête à générer Les en-têtes de réponses préformatées précédentes étaient composées de plusieurs champs. Souhaitant désormais générer un en-tête qui correspond aux requêtes demandées, nous allons devoir générer la plupart *dynamiquement*. Détaillons les champs que nous souhaitons traiter :

Un code de réponse Nous avons déjà vu page 43 que les codes de statut étaient précisément définis. Ils sont dictés par le code *métier*, c'est à dire par la requête, l'état, et le comportement du serveur.

Le fichier `http.py` fournit une fonction `get_http_code(code: int)` qui contient pour quelques codes, un dictionnaire avec : un champ `'head'` contenant la chaîne de caractères à mettre dans l'en-tête retour, ainsi qu'un champ `'html'` contenant une page HTML à renvoyer en contenu.

`Server:` Le nom du serveur. Nous laisserons `RegardeMamanJeFaisUnServeurWeb/0.1`, faute de plus d'inspiration.

`Date:` Censé contenir la date au moment de l'envoi, au format défini par le RFC2616. Il va devoir être généré dynamiquement avec l'heure courante à chaque réponse.

`Content-Type:` Le type de contenu sert au navigateur à savoir *comment* afficher une ressource, et est donc très important pour que notre serveur retourne les résultats attendus. Ce champ est généralement déterminé par l'extension de la ressource demandée.

La fonction `get_http_content_type(extension: str)` du fichier `http.py` fournit, pour la plupart des extensions de fichiers connus, le `Content-Type` qui lui est associé.

`Content-Length:` La taille en octets du contenu qui suit l'en-tête. Elle est elle aussi importante, car les clients peuvent s'en servir pour déterminer quand arrêter de lire notre *socket*.

Elle devra donc elle aussi être déterminée à chaque envoi, en fonction de la taille du contenu que nous souhaitons renvoyer.

La fonction `prepare_reply` La préparation de l'en-tête, étant donné un code de statut et un éventuel contenu à envoyer, est faite par `prepare_reply(content: bytes, content_type: str, code: int)`. La fonction prend trois paramètres : le contenu `content` que l'on souhaite envoyer (c'est un tableau d'octets `bytes`), le type de contenu `content_type`, à mettre dans le champ HTTP `Content-Type` (une `str`), et le code de statut `code` (un entier `int`). La fonction s'utilise de deux manières différentes :

Code 200 Quand on souhaite renvoyer un code 200 au client, la fonction `prepare_reply` a besoin d'un contenu `content` et d'un type de contenu `content_type` bien renseignés :

```
# Génération d'un message `reply` avec contenu et code 200
reply, code = prepare_reply(content, content_type, 200)}
```

Autre code Dans ce cas, c'est la fonction `prepare_reply` qui ira elle-même chercher la page d'erreur à transmettre au client, en faisant appel à la fonction `get_http_code(code: int)`.

On lui passe donc un `content` et un `content_type` vides (resp. `b""` et `""`) :

```
# Génération d'un message `reply` sans contenu avec code différent de 200
reply, code = prepare_reply(b"", "", 501)
```

Notez que la fonction `prepare_reply` renvoie un *tuple*, une paire de variables : la réponse à mettre dans la *socket* `reply`, et le code de statut `code`, qui sera ultérieurement utilisé pour *logguer* les requêtes.

Petit rappel sur les *f-strings* Python fournit une façon fort pratique de générer dynamiquement des chaînes de caracètres, appelées *f-strings*[15]. Couplées aux définitions de chaînes de caractères ou d'octets *multilignes* (utilisant 3 guillemets ouvrants, et fermants), c'est tout ce dont on a besoin :

```
header = f"""HTTP/1.1 {???}
Content-Type: {???}
Date: {now_rfc2616()}
Content-Length: {???}
```

15. Ce guide fournit quelques exemples qui pourraient vous aider à mieux comprendre les *f-strings*.

```
Server: RegardeMamanJeFaisUnServeurWeb/0.1

"""
```

Le code précédent fournit un exemple de la façon dont vous pourriez construire votre en-tête. On y voit une *f-string* multiligne : à l'intérieur, on peut utiliser des crochets `{...}` pour y mettre des variables, des calculs, ou des appels de fonctions—leur valeur sera ajoutée à la chaîne. Pour générer la réponse totale, la fonction `prepare_reply` n'aura plus qu'à convertir la variable `header` du type `str` au type `bytes` en appelant la méthode `str.encode()`, et à la concaténer au contenu `content` avec l'opérateur `+`.

Exercice 3.11 : Implémentation d'une réponse statique

Éditez le fichier `server.py` pour faire appel à, et implémenter, la fonction `prepare_reply`. Appuyez-vous sur la documentation fournie dans le script pour vous aider à comprendre plus précisément le rôle de chaque fonction.

1. Commencez par modifier `handle_client` pour faire appel à la fonction `prepare_reply`, dans les 3 cas évoqués précédemment :
   (a) Si la requête n'a pas pour « verbe » `GET`, appelez `prepare_reply` avec le code `501`.
   (b) Sinon, si la requête demande la ressource `/`, appelez `prepare_reply` avec le contenu "You are beautiful today !", le type de contenu adapté, et le code `200`.
   (c) Sinon, appelez et renvoyez `prepare_reply` avec le code `404`.
2. En utilisant les indications précédentes, et la documentation de la fonction, implémentez `prepare_reply`. Vous ferez notamment appel aux fonctions `get_http_code(code: int)` (définie dans `http.py`) et `now_rfc2616()` (définie dans `date.py`).
   Attention : si le code n'est pas égal à 200, c'est à `prepare_reply` de générer du contenu correspondant à l'erreur, et de renseigner le `Content-Type` approprié.
3. Réexécutez votre module, puis pointez à nouveau votre navigateur vers l'URL servie, puis vers une URL qui n'existe pas.
   Le texte affiché est-il celui attendu ? Les codes de réponse retournés sont-ils corrects ?
   Notez vos observations dans le fichier README, et sauvegardez.

Nous savons désormais générer une réponse HTTP en fonction du code et/ou du contenu souhaité. Néanmoins notre code n'est pas « propre » : la fonction `handle_client` gère à la fois les erreurs et le contenu valide. Il est temps de la scinder : en réalisant une fonction dédiée au traitement de requêtes valides.

La fonction `prepare_resource` La fonction `prepare_resource(root: str, req: dict)` a pour but de générer les ressources demandées par le client quand sa requête est valide. Elle fera bientôt appel aux fonctions du fichier `file.py`, en charge d'intéragir avec notre système de fichiers, pour aller chercher les ressources demandées « sur disque » [16]. Pour le moment néanmoins, commençons par implémenter une version sans fichiers.

Cette fonction va générer un contenu (e.g. un texte, ou le contenu d'un fichier), assorti d'informations telles que son `Content-Type` et un code de réponse, avant d'appeler la fonction `prepare_reply` pour enrober la réponse correctement. `prepare_resource` renvoie dans tous les cas le résultat de `prepare_reply` : un couple `(data: str, code: int)`.

16. Le stockage persistent est souvent assuré par la technologie Solid-State Drive (SSD), de nos jours. Ces derniers ne sont pas vraiment constitués de disques, mais l'abus de langage persiste.

Exercice 3.12 : Génération de texte sans passer par le disque

1. Commençons par modifier la fonction `handle_client` du fichier `server.py` afin qu'elle délègue la préparation de la ressource demandée à `prepare_resource`. On utilisera la fonction `prepare_reply` pour générer chaque réponse : c'est maintenant notre empaqueteur officiel de réponses HTTP.
   (a) Dans `handle_client`, modifiez le bloc `if/elif/else`, de sorte qu'on y vérifie seulement la validité du « verbe » HTTP.
   — Si c'est bien un `GET`, appelez `prepare_resource` en lui fournissant les paramètres demandés.
   — Sinon, préparez une `501` en appelant `prepare_reply`.
   (b) Dans `prepare_resource`, vérifiez avec un bloc `if/else` que la ressource demandée est `/`, auquel cas renvoyez "You are beautiful today !" ; sinon, renvoyez une `404`.
2. Testez votre serveur. Fonctionne-il toujours comme avant ? Bien, vous avez refactorisé votre code perte de fonctionnalité.
3. Ajoutons une nouvelle page `/meteo` :
   (a) Augmentez le bloc `if/else` de la fonction `prepare_resource`, pour gérer un troisième cas où la ressource demandée est `/meteo`.
   (b) La météo, ça va, ça vient : renvoyez aléatoirement l'une des deux chaînes de caractères possibles en utilisant `random.choice(SEQ)` : "It's a good day !" ou "It's a rainy day !".
4. Exécutez le module, et pointez votre navigateur web sur http://localhost:8000/, puis http://localhost:8000/meteo.
   Voyez-vous les messages attendu ? Rafraîchissez la page pour voir si le texte change.
   Ouvrez les outils de développement pour vérifier les codes de réponse.
5. Essayez maintenant d'accéder à une page autre que la racine, par exemple : http://localhost:8000/page-fantome. Que voyez-vous ?
6. Prenez note de vos observations dans le fichier README, et sauvegardez.

À ce moment, vous devriez avoir les resultats `pytest` suivants :

```
tests/test_file.py FFFF                                  [ 12%]
tests/test_http_request.py ..............                [ 54%]
tests/test_log.py FFFFFFF                                [ 75%]
tests/test_server.py .F......                            [100%]
```

Pourquoi l'un des tests du fichier `server` ne passe-t-il pas ? En regardant de plus près, on observe que le test vérifie la récupération du contenu d'un fichier sur disque. Lançons-nous dans cette périlleuse aventure.

3.6 Récupération des fichiers sur disque

Nous présenterons d'abord les éléments théoriques à comprendre pour ce faire, avant de plonger dans la pratique.

3.6.1 En théorie

La tâche se décompose en plusieurs étapes : (i) convertir l'adresse de la ressource demandée (son URN) en un chemin local de fichier, (ii) vérifier l'existence et la capacité à accéder au fichier, et (iii) récupérer le contenu du fichier.

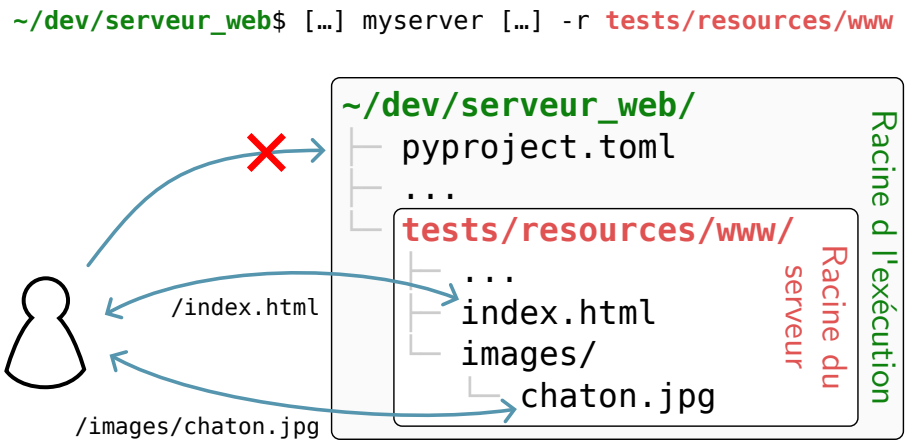


FIGURE 11 – Correspondance entre les ressources demandées par le client, et leur localisation sur le système de fichier du serveur. On voit que seuls sont accessibles les ressources appartenant à la racine du serveur.

Convertir le chemin Comme illustré dans la figure 11, quand un utilisateur effectue une requête par exemple vers la ressource `/index.html` (plus précisément son URN), il appartient à notre code de localiser le fichier, relativement à la *racine du serveur* passée en paramètre `--root/-r` à son lancement. On parle de *racine virtuelle*, car le serveur doit fournir des fichiers comme si le chemin du paramètre *root* était la racine de son système de fichiers. Le serveur doit notamment interdire l'accès à tout contenu dans des répertoires parents à sa racine.

Si, comme on l'a proposé jusqu'ici, le paramètre *root* est un chemin relatif comme `tests/resources/www`, alors les fichiers seront recherchés dans ce dossier, relativement à la *racine de l'exécution*, l'endroit où l'on exécute le serveur. Par exemple :

— `/index.html` deviendra `tests/resources/www/index.html` ;

— `/images/chaton.jpg` deviendra `tests/resources/www/images/chaton.jpg`.

Si l'on renseigne un chemin absolu, comme `/home/adrien/dev/site_perso`, alors :

— `/index.html` deviendra `/home/adrien/dev/site_perso/index.html` ;

— `/images/chaton.jpg` deviendra `/home/adrien/dev/site_perso/images/chaton.jpg`.

Techniquement, c'est la librairie `os.path` [17] qui va nous permettre de concaténer ces chemins, et notamment sa fonction `os.path.join`.

Vérifier l'existence du fichier Une fois calculé le chemin local, nous assurer de son existence permet de renvoyer une erreur `404` si le fichier concerné n'existe pas. Pour ce faire, nous disposons ici de la fonction `os.path.isfile` : elle vérifie à la fois que le chemin *existe*, et qu'il représente un fichier.

Récupérer le contenu Lire le contenu d'un fichier est assez trivial en Python. Il n'y a qu'une subtilité, commune à tout langage de programmation : le mode d'ouverture. Un fichier peut être ouvert en mode *texte*, ou en mode *binaire* [18] Ce mode est à préciser lors de l'ouverture du fichier. L'ouverture en mode binaire permet d'obtenir le contenu *brut*, tandis que le mode texte facilite le traitement de documents textuels. Ne souhaitant pas nous embêter, nous ouvrirons tous les fichiers en mode binaire, pour obtenir dans tous les cas un tableau de `bytes` représentant son contenu brut. Voilà le code Python pour ce faire :

```
with open(chemin_du_fichier, 'rb') as f:
    content = f.read()
```

17. On choisit encore d'utiliser une librairie bas-niveau, plutôt que par exemple pathlib, plus puissante, mais qui a l'inconvénient de cacher les opérations bas niveau.

18. Impliquant des différences de traitement du fichier similaires à la différence entre les types `str` et `bytes`.

Le bloc `with EXPRESSION as VARIABLE:` affecte le résultat de l'`EXPRESSION` à la `VARIABLE`, tant qu'on ne quitte pas le bloc. La `VARIABLE` est automatiquement à la sortie du bloc.

La fonction standard `open(CHEMIN, MODE)` ouvre le fichier à l'adresse `CHEMIN` selon le `MODE` passé en paramètre. Ici, nous utilisons le mode `'rb'`, signifiant *read* (mode lecture) et *binary* (mode binaire). Parmi les autres modes les plus fréquents, il existe `'w'` (*write*, mode écriture—efface entièrement le fichier avant d'y ajouter du contenu), et `'a'` (*append*, mode ajout—écrit à la fin du fichier).

On dispose donc d'une variable `f` au sein du bloc `with`, contenant le *descripteur de fichier* à l'adresse `chemin_du_fichier`. Pour retourner tout son contenu, on utilise la méthode `f.read()`, et on affecte le résultat à une variable `content`.

⚠ Attention : Que répondre quand le client demande un dossier ?

En règle générale, quand le client fait une requête à un chemin terminant par `/` (un dossier, donc), le serveur web vérifie l'existence d'un fichier `index.html` dans ledit dossier, auquel cas il le renvoie. Si aucun index n'est trouvé, certains serveurs web (comme Apache) génèrent par défaut une liste HTML du contenu du dossier. Cette deuxième solution est bien pratique, mais elle diminue la sécurité [a] : vos usager·es peuvent naviguer l'arborescence de votre serveur.

Dans notre cas, quand nous recevrons une requête vers un dossier (terminant par `/`), nous rechercherons un fichier `index.html` dans ledit dossier. Nous renverrons une `404` si ce n'est pas le cas.

a. Dans le jargon de la sécurité informatique, on dit que cette fonctionnalité « augmente la surface d'attaque ».

Conversion, vérification du chemin, récupération du contenu. Nous avons tout : place à la pratique.

3.6.2 En pratique

Architecture du fichier `file.py` Le fichier `file.py` doit contenir tout le code en lien avec le système de fichiers. Pour l'instant nous avons trois fonctions vides, documentées, et des tests unitaires qui en vérifient le fonctionnement. L'*architecture* du module vous est donc déjà fournie, et vous n'avez « plus qu'à » combler les trous [19].

La fonction `resolve_path` La conversion des chemins et la vérification de l'existence des fichiers doivent être assurées par la fonction `resolve_path(res:str, root: str)`. Cette fonction interne au module `file.py` sera appelée par `resolve_location`, qui sera, elle, appelée par la fonction `prepare_resource` du module `server.py`. Deux fonctions de test en valident le fonctionnement : `test_resolve_path` et `test_resolve_path_weirdos`, qui nous permettent respectivement de valider le comportement de la fonction avec des entrées correctes, et avec des entrées non conventionnelles (*weirdos*).

✎ Exercice 3.13 : Implémentation de la fonction `resolve_path`

Implémentez la fonction `resolve_path` pour qu'elle réponde au cas nominal, c'est-à-dire que le fichier existe et est accessible.

1. Vérifiez d'abord la chaîne `res` :
   (a) Si c'est un dossier, ajoutez `index.html` en boût de chaîne pour rechercher plutôt un fichier d'index au sein du dossier ;
   (b) Supprimez tous les `/` éventuels en début de chaîne.

19. On peut débattre longtemps de l'*élégance* d'une architecture logicielle. Notre solution est largement perfectible. Si vous en avez le temps, nous vous suggérons de repenser cette architecture pour la rendre plus élégante et mieux optimisée.

2. Utilisez `os.path.join` pour concatener `root` (la racine du serveur) et `res` (l'URN de la ressource) : vous obtenez le chemin d'accès au fichier sur le système de fichiers du serveur.
3. Vérifiez l'existence du fichier avec `os.path.isfile`.

   À la lecture de la documentation de la fonction, vous saurez le résultat à renvoyer si le fichier n'existe pas.
4. Exécutez les tests avec `pytest -v tests/test_file.py`. Les deux tests sus-mentionnés devraient passer.
5. Sauvegardez votre travail.

La fonction `get_resource` La fonction `get_resource(res_path: str)` est en charge de retourner le contenu d'un fichier, référencé par son chemin sur le système de fichiers du serveur : `res_path`.

Exercice 3.14 : Implémentation de la fonction `get_resource`

1. Éditez `file.py` pour implémenter la fonction `get_resource` en suivant les indications de la page 58.
2. Exécutez les tests avec `pytest -v tests/test_file.py`. Les deux tests `test_get_resource` et `test_get_resource_image` doivent passer.
3. Sauvegardez votre travail.

La fonction `resolve_location` La fonction `resolve_location(res:str, root: str)` doit appeler `resolve_path` pour obtenir le chemin local de la ressource demandée. Son rôle est de renvoyer ledit chemin local, ainsi que l'extension du fichier requêté. C'est la fonction qui sera appelée par la fonction `prepare_resource` du fichier `server.py` pour récupérer le chemin d'accès au fichier.

Exercice 3.15 : Implémentation de la fonction `resolve_location`

Éditez `file.py` pour implémenter la dernière fonction, `resolve_location` :

1. Votre fonction fera appel à `resolve_path` pour récupérer `res_path`, le chemin local de la ressource.
2. Vous utiliserez `os.path.splitext` pour récupérer l'extension dans une nouvelle variable.
3. Vous devez supprimez l'éventuel `.` de l'extension, avant de le renvoyer à l'appelant.
4. Cette fonction n'étant pas testée, vous pouvez sauvegarder votre travail et passer à la suite.

3.6.3 Câblage final

Nous avons désormais tous les outils dans `file.py` pour que notre serveur puisse envoyer des ressources depuis son disque, il ne reste qu'à les appeler dans la fonction `prepare_resource` du fichier `server.py`.

Le but de `prepare_resource` étant de vérifier l'existence d'un fichier, et de préparer soit un code 404, soit un code 200 et le contenu du fichier—son implémentation se fait en deux temps :

1. On commence par vérifier l'existence du fichier avec un appel à `resolve_location` ;
2. En fonction des cas :
   — Si le fichier n'est pas trouvé, on renvoie une 404 ;
   — S'il est trouvé, on récupère la ressource avec `get_resource`, et son *Content-Type* avec `get_http_content_type` ;
3. On retourne le résultat de `prepare_reply` dans tous les cas.

Exercice 3.16 : Implémentation de `prepare_resource` et test du serveur

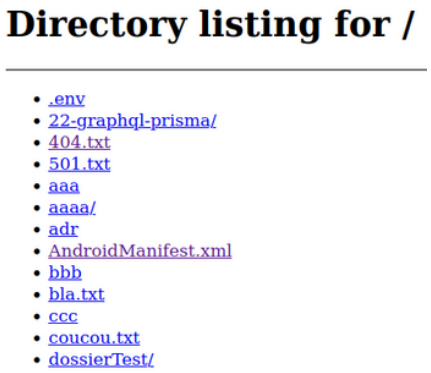
1. Implémentez `prepare_resource` comme expliqué ci-dessus.
2. Exécutez les tests : ils devraient tous passer sauf ceux de `test_log.py`.
3. Testez votre serveur en utilisant le dossier `tests/resources/www` comme racine (*root*, le paramètre `-r`).
4. Relancez votre serveur en indiquant comme racine le chemin d'un site HTML fonctionnel (comme votre projet de TW1).
5. Sauvegardez votre travail.

Félicitations ! Vous avez réalisé un serveur web capable de servir des fichiers d'un dossier de votre choix, en utilisant HTTP. Nous ne l'avons testé qu'en local (avec le client et le serveur sur la même machine), mais il fonctionnerait tout aussi bien depuis l'autre boût d'Internet. (Vous pouvez d'ailleurs essayer de vous connecter au serveur web de votre voisin, en indiquant dans votre navigateur son adresse IP et le port de son serveur.) Ne nous quittons néanmoins pas sans quelques exercices bonus, et sans avoir implémenté une journalisation des requêtes digne de ce nom.

Exercice Bonus B3.1 : Liste des fichiers d'un répertoire

Ainsi que décrit dans l'encart « Que répondre quand le client demande un dossier ? » en page 59, lorsque la ressource demandée est un répertoire et qu'il n'existe pas de fichier `index.html`, certains serveurs proposent *une liste des fichiers* contenus dans le répertoire.

Si vous en avez le temps, nous vous proposons d'adapter la fonction `resolve_path` pour qu'elle retourne une telle liste. Vous pouvez vous inspirer de la capture d'écran suivante pour cela, ou ajouter vos propres informations (par exemple, la date de dernière modification).



Notez que la liste propose *des liens* vers les fichiers et répertoires. En cliquant dessus, on peut naviguer dans l'arborescence et visualiser les fichiers. Assurez-vous que cela fonctionne dans votre cas. Prenez une capture d'écran, et ajoutez-la à votre prochain commit.

3.7 Gestion des logs

Un serveur web correct doit être en mesure de fournir un certain nombre d'informations dans les logs, pour plusieurs raisons :

— Permettre le debug en cas de problème (d'accès, de configuration, etc.).
— Obtenir des statistiques d'utilisation du service.
— Pour des raisons légales : en cas d'utilisation illégale du serveur, l'hébergeur est sommé de fournir les logs d'accès.

Appuyez-vous sur la documentation fournie dans le script et dans le répertoire `docs/` pour vous aider à comprendre ce que chaque fonction fait précisément.

Exercice 3.17 : Implémentation des fonctions de gestion de fichier

1. Éditez `log.py` de sorte à faire passer les tests du fichier `test_log.py`.
2. Appelez la fonction `log_reply()` dans la fonction `handle_client()`, afin que le serveur affiche dans sa sortie santard l'heure de chaque requête, ses caractéristiques, et le code de retour obtenu.

3.8 Bonus : Ajouter des articles de blogs depuis un formulaire

Vous êtes très en avance ? Nous aurons toujours des idées pour vous faire progresser. Votre serveur pourrait permettre de poster (verbe HTTP POST) de nouveaux articles depuis une page contenant un formulaire HTML, et créer de nouveaux fichiers avec le contenu envoyé (titre, article, image...). Doublé à l'exercice bonus précédent proposant de lister le contenu d'un dossier (les articles d'un blogs), vous disposeriez là d'un vrai système de blog à la Wordpress [20]. Il s'agit déjà là de plus qu'un serveur de fichiers, c'est un véritable Content Managing Service (CMS) Une véritable prouesse.

20. moyennant une ou deux fonctionnalités additionnelles.

Glossaire

API Application Programming Interface. 33, 35, 36

beta Une version *beta* d'un logiciel ou d'un protocole est une version non finalisée, publiée pour démonstration. Une version *alpha* est encore moins finalisée, et pas forcément publiée.. 41

bit Plus petite unité d'information numérique, ne pouvant prendre que deux valeurs : zéro ou un.. 23

CI Continuous Integration. 4, 12, 19, 20, 63

CMS Content Managing Service. 62

CSS Cascading Style Sheets. 3, 41, 44

DNS Domain Name System. 42

FAI Fournisseur d'Accès à Internet. 63

forge Par analogie avec la métallurgie, la forge est l'endroit où les développeur·euses se retrouvent pour développer des outils. Elle est notamment constituée d'un système de gestion de version logiciel (e.g. git), d'un système de discussion, de suivi des tickets, d'une CI, etc. 3, 4, 25, 29, 33, 47, 48

FTP File Transfer Protocol. 42

HTML HyperText Markup Language. 3, 18, 41

HTTP HyperText Transfer Protocol. 3, 16, 24, 25, 41, 42, 50

HTTPS HyperText Transfer Protocol Secure. 42

IANA Internet Assigned Numbers Authority. 43

Internet Ensemble des réseaux IP interconnectés mondialement par de nombreux routeurs, câbles de cuivre et fibres optiques. Internet (ou simplement « le net ») est une infrastructure de communication numérique aujourd'hui majeure, qui est née dans les années 1970. 22, 63

IP Internet Protocol. 23, 42, 63

JS Javascript. 41, 44

MAC Media Access Control. 23

NAT Network Address Translation. 24

OS Operating System. 5, 25

P2P Peer-to-peer. 22

protocole Ensemble de règles qui permettent à divers acteurs de se comprendre, de communiquer et d'effectuer diverses tâches collectivement. Ces règles sont spécifiées dans des documents appelés standards ou spécifications. Voir par exemple la spécification de HTTP. 23, 24

RFC L'organisme de standardisation *Internet Engineering Task Force* (IETF) publie les spécifications d'Internet sous le nom de Request For Comments (RFC). l'IETF faisant autorité dans le milieu, « RFC » en devient presque synonyme de « spécification ».. 41

routeur Ordinateur spécialisé dans la transmission de paquets IP. On connaît bien celui qui relie notre domicile à Internet, moins ceux qui relient les domiciles d'un Fournisseur d'Accès à Internet (FAI) entre eux, et les FAI entre eux. Par bien des aspects, ce sont les routeurs qui permettent à Internet de fonctionner, en routant inlassablement les messages que l'on s'y envoie. 22, 63

socket Prise ou connecteur. Ce terme est très utilisé en réseau (e.g. socket TCP) pour désigner, une fois une connection établie, le connecteur utilisé pour lire et recevoir des données. 33, 66

SSD Solid-State Drive. 56

SSH Secure SHell. 10

TCP Transmission Control Protocol. 24, 25, 41

UDP User Datagram Protocol. 24, 25

URI Uniform Resource Identifier. 41, 42

URL Uniform Resource Locator. 42, 43, 56

URN Uniform Resource Name. 42, 43, 47, 57, 58, 60

A Où trouver de l'aide

En fonction des cas, on trouve des réponses à nos questions soit dans le manuel Linux, soit dans la documentation du langage de développement qu'on utilise, soit sur des forums spécialisés.

A.1 Aide syntaxique des commandes

La plupart des commandes Linux disposent d'une petite aide syntaxique, qu'on peut afficher en ajoutant le paramètre `--help` après la commande.

```
$ ls --help
Utilisation : ls [OPTION]... [FICHIER]...
Afficher des renseignements sur les FICHIERs (du répertoire actuel par défaut).
Trier les entrées alphabétiquement si aucune des options -cftuvSUX ou --sort
ne sont utilisées.

Les arguments obligatoires pour les options longues le sont aussi pour les
options courtes.
  -a, --all                  ne pas ignorer les entrées débutant par .
  [...]
```

A.2 Manuel Linux pour les commandes

La plupart des commandes Linux disposent d'un manuel très fourni et didactique. Si vous cherchez par exemple comment fonctionne la commande `cp`, vous trouverez tout ce qu'il vous faut en tapant `man cp` dans le terminal :

```
$ man cp
CP(1)                          Commandes de l'utilisateur                          CP(1)

NOM
       cp - Copier des fichiers et des répertoires

SYNOPSIS
       cp [OPTION]... [-T] SOURCE CIBLE
       cp [OPTION]... SOURCE... RÉPERTOIRE
       cp [OPTION]... -t RÉPERTOIRE SOURCE...

DESCRIPTION
       Copier la SOURCE vers la CIBLE, ou plusieurs SOURCEs vers le RÉPERTOIRE.

       Les  paramètres  obligatoires  pour  les options de forme longue le sont aussi
       pour les options de forme courte.

       -a, --archive
              identique à -dR --preserve=all
       [...]
```

Pour vous déplacer dans le fichier, vous pouvez utiliser ↑, ↓, PageUp et PageDown.

Pour chercher une chaîne de caractères dans la doc, tapez / suivi de la chaîne, puis faites ↵. Pour vous déplacer à l'occurrence suivante/précédente de votre recherche, faites respectivement n/N.

A.3 Documentation pour le développement

Tout comme vous, les développeurs d'un langage ou d'une bibliothèque documentent systématiquement leurs travail. Il est indispensable de lire la documentation pour bien utiliser un langage et ses bibliothèques. La documentation Python est d'ailleurs réputée pour sa grande qualité ; exemples :

— Chapitre Structures de données : tout ce qu'il faut savoir sur les `list`, `dict`, etc.
— Guide pratique : programmation avec les sockets : un tutoriel sur l'utilisation des sockets TCP.
— Documentation du module `socket` de la bibliothèque standard.

La documentation Python est aussi disponible directement depuis l'interpréteur :

```
$ python3
>>> help(list)
Help on class list in module builtins:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 | [...]
```

Les mêmes raccourcis clavier que pour le manuel y sont utilisables (cf. plus haut).

A.4 Et pour le reste, Internet et ses forums

On s'adresse directement à son moteur de recherche favori lorsque notre question est plus large ou vague. Quand on est devant une erreur dont on ne comprend pas le sens, c'est typiquement le premier réflexe à avoir.

Par exemple, si je ne comprends pas l'erreur que j'obtiens en utilisant `socket.connect`, je peux aller chercher sur Internet :

```
>>> import socket
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> s.connect("www.python.org")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: connect(): AF_INET address must be tuple, not str
```

Copier-coller la ligne du `TypeError` m'amènera vite sur un fil de discussion StackOverflow, où on m'expliquera en anglais que la méthode `socket.connect` attend un *tuple* contenant deux éléments : `socket.connect((adresse:str, port:int))`.