TDLOG séance 10 - Programmation réseau et web

Xavier Clerc – xavier.clerc@enpc.fr

5 janvier 2026

Table des matières

1 Introduction

Lors de cette séance, nous allons nous intéresser à la programmation réseau et web, et présenter un certain nombre de concepts et outils utiles dans ce cadre.

Ainsi, nous allons commencer par présenter les notions de bases de la programmation réseau

puis la programmation par *sockets* (sections 2 et 3). Nous présenterons ensuite quelques éléments de base de la programmation web, illustrés par la librairie `web.py` (section 4).

2 Notions de base sur les réseaux

Une connaissance approfondie des réseaux dépasse de loin le périmètre de ce cours. Nous avons donc choisi volontairement de ne présenter que les notions qui semblent minimales à la compréhension des problématiques réseaux, ainsi qu'aux impacts en termes d'architecture logicielle.

2.1 Connexions

Le principe de base des réseaux est de pouvoir connecter des machines afin qu'elles puissent échanger des données. Pour ce faire, il faut que les machines concernées disposent non seulement de l'infrastructure physique nécessaire à la connexion (câble, antenne, *etc*), mais également que des logiciels soient prêts sur les différentes machines à échanger effectivement des données.

Ainsi, outre la connexion physique des machines, il est nécessaire que s'établisse un lien entre programmes. Pour cela, il faut avant tout que les machines puissent s'identifier sur le réseau. La figure Figure 2.1 montre deux machines ayant établi ce lien logique. Il repose communément sur deux éléments : une adresse qui identifie une machine et un port qui identifie un service logiciel. Comme le montre la figure, il existe plusieurs adresses pour une machine :

— une adresse physique, dite MAC (Media Access Control address), qui identifie de manière universellement unique un équipement de communication (une machine peut donc avoir plusieurs adresses MAC si elle dispose par exemple de plusieurs accès au réseau, *p. ex.* par wifi et ethernet) ;
— une adresse logique, dite IP (Internet Protocol address), qui doit être unique sur le réseau considéré, mais peut être réutilisée sur un autre réseau, cette adresse peut être assignée manuellement ou distribuée automatiquement par une machine désignée ;
— un nom symbolique (optionnel) qui permet de désigner une machine par un nom plutôt que par une suite de chiffres.

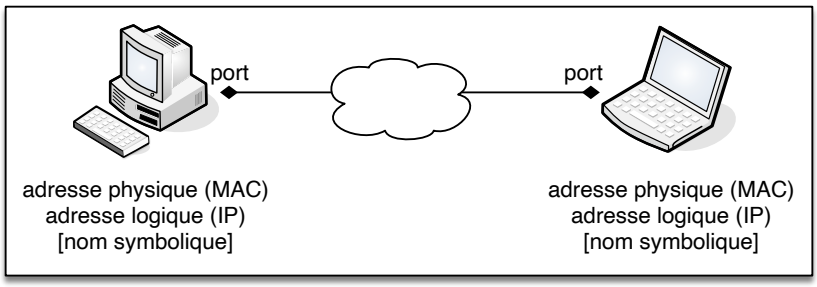

FIGURE 2.1 – Connexion entre deux machines.

Les adresses MAC sont de la forme `ff:13:07:19:79:ad`, soit un sextuplet d'octets en notation hexadécimale. Les adresses IP prennent deux formes selon que l'on utilise IP v4 ou IP v6. IP v4 est la forme originelle et s'écrit par un quadruplet d'octets en notation décimale séparés par des points (*p. ex.* `192.168.1.17`). Cette norme permet un maximum théorique de 2^{32} adresses, mais en pratique beaucoup moins d'adresses sont disponibles (certaines étant réservées). En outre, la multiplication des éléments ayant accès au réseau Internet (ordinateurs, tablettes, téléphones, interphones, objets connectés, *etc*) fait que même la valeur de 2^{32} serait assez rapidement atteinte. Pour cette raison, la norme IP v6 a été définie ; les adresses IP v6 sont de la

forme `13ab:0c7d:e1f9:7900:13ab:0c7d:e1f9:7900`, soit un octuplet de couples d'octets en notation hexadécimale. Cela permet un maximum théorique de 2^{128} adresses.

2.2 Types de réseaux

Sans chercher à présenter un panorama des types de réseaux, nous nous contentons de présenter à la figure Figure 2.2 les deux extrêmes du spectre : (a) un réseau local et (b) le réseau Internet.

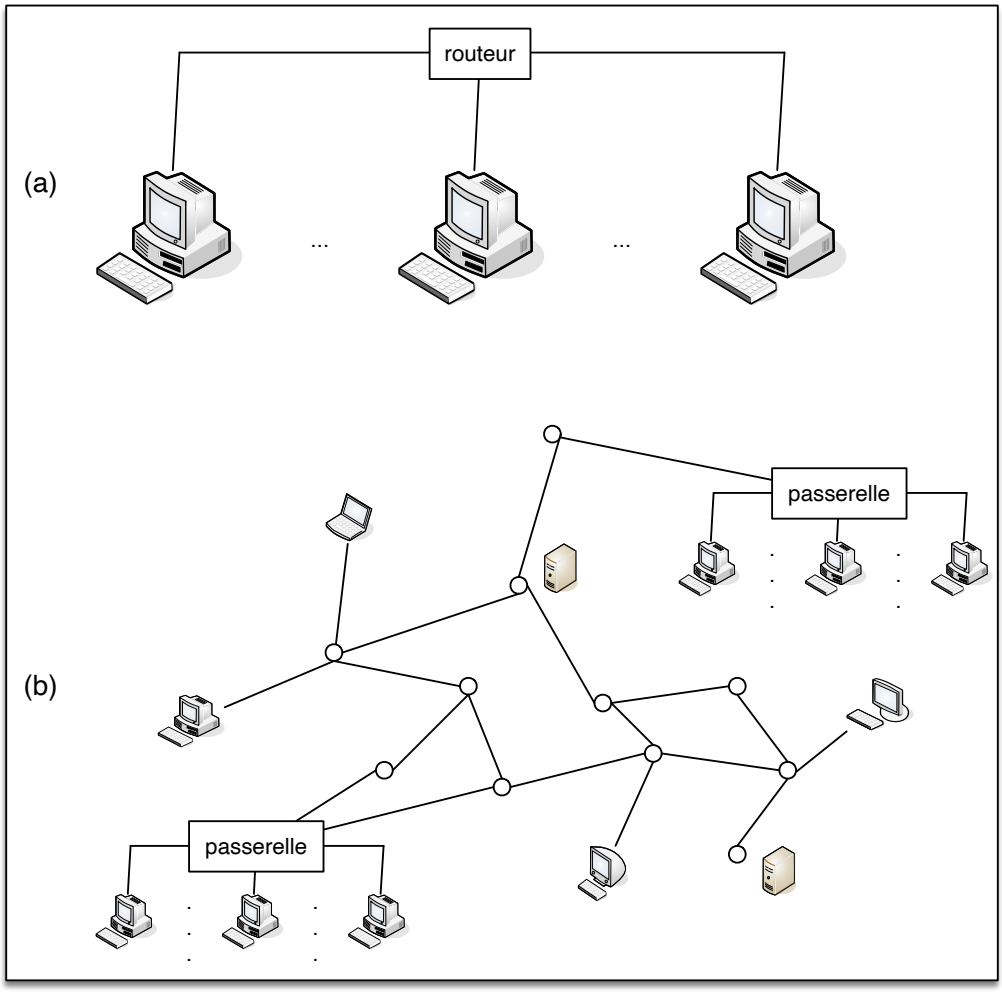

FIGURE 2.2 – Réseau local (a), réseau Internet (b).

Sur un réseau local, les machines sont typiquement reliées physiquement par des câbles à un élément matériel appelé routeur (*router*, *hub*, ou *switch*). Le rôle de cet équipement est de "router" (*i. e.* trouver une route ou chemin) les paquets de telle manière qu'ils soient envoyés à leur destinataire.

Sur le réseau Internet, les connexions entre nœuds du réseau sont irrégulières au sens où chaque élément n'est pas connecté à un élément central comme dans le cas du réseau local. C'est du reste une bonne propriété d'Internet puisque la défaillance d'un nœud est naturellement palliée par le calcul d'un autre chemin entre deux machines.

Comme le montre la figure Figure 2.2, des réseaux locaux peuvent évidemment se connecter à Internet (c'est même l'origine de son nom : INTERconnection of NETworks). Dans ce cas, les équipements permettant de passer du réseau local au réseau Internet sont généralement désignés

sous le nom de *passerelles*. Les rôles de passerelle et de routeur sont communément assurés par le même équipement.

Comme utilisateur du web, vous êtes habitués à utiliser une URL (uniform resource locator) pour accéder à un site. Une URL est typiquement de la forme : `http://www.nom.fr/page`, où on distingue les éléments suivants :

- `http` qui est le protocole réseau ;
- `nom.fr` (nom de domaine) qui est le nom symbolique de la machine ;
- `www` et `page` (respectivement nom de sous-domaine et chemin) qui permettent à la machine recevant une demande de savoir quelle *page* doit être envoyée au programme ayant effectué la requête.

Sur Internet, la question du nom symbolique déjà évoquée est réglée par un système d'annuaires distribués et répliqués connu sous le nom de DNS (Domain Name system). Cet annuaire contient "simplement" une liste d'associations ⟨ nom symbolique, adresse IP ⟩.

2.3 Architectures d'applications réseaux classiques

La figure Figure 2.3 montre quelques architectures logicielles. La première (a) montre en fait le modèle classique, dit aussi *desktop*, dans lequel les calculs et les données sont faits localement sur les machines, en l'absence de réseau.

La deuxième (b) montre le modèle client-serveur qui est celui du web : des clients envoient des requêtes à un serveur qui leur répond. C'est le serveur qui détient les données, les calculs pouvant être répartis entre le client et le serveur.

La troisième (c) montre le modèle pair-à-pair (*peer-to-peer* ou *p2p*) : toutes les machines agissent à la fois comme source et destination d'informations et celles-ci sont répliquées (entièrement ou en partiellement) sur les différentes machines. Contrairement au modèle précédent, il n'y a plus de *single point of failure* (la défaillance d'une seule machine, en l'occurrence le serveur, ne rend pas le service indisponible) ; en revanche, il est plus difficile de savoir quelle(s) machine(s) possède(nt) les données *correctes* ou *qui font foi*.

La figure Figure 2.4 montre l'architecture dite *cloud computing*. Bien que récemment très médiatisée, il s'agit essentiellement d'une variation du modèle client-serveur dont l'argument essentiel est la souplesse. La souplesse revendiquée provient de la capacité à mobiliser plus ou moins de machines et d'unités de stockage en fonction des besoins et cela de manière transparente pour le client.

Il est utile de distinguer les aspects matériels des aspects logiciels ; en effet, il est tout à fait possible d'utiliser un logiciel client-serveur sur le réseau internet (c'est même l'essence du web) ou un logiciel pair-à-pair sur un réseau local. Cela alors même que, comme le montrent les schémas, une architecture client-serveur est semblable à l'organisation d'un réseau local et qu'une architecture pair-à-pair est semblable à l'organisation d'Internet.

2.4 Ports

Outre la connexion physique, pour faire communiquer deux programmes, il est nécessaire d'opérer une connexion logique. Nous avons vu qu'un équipement réseau (carte pour réseau câblé ou doté d'une antenne) dispose d'une adresse dite *physique* qui identifie le matériel de manière mondialement unique. Il est d'usage de définir une adresse logique (par exemple, adresse

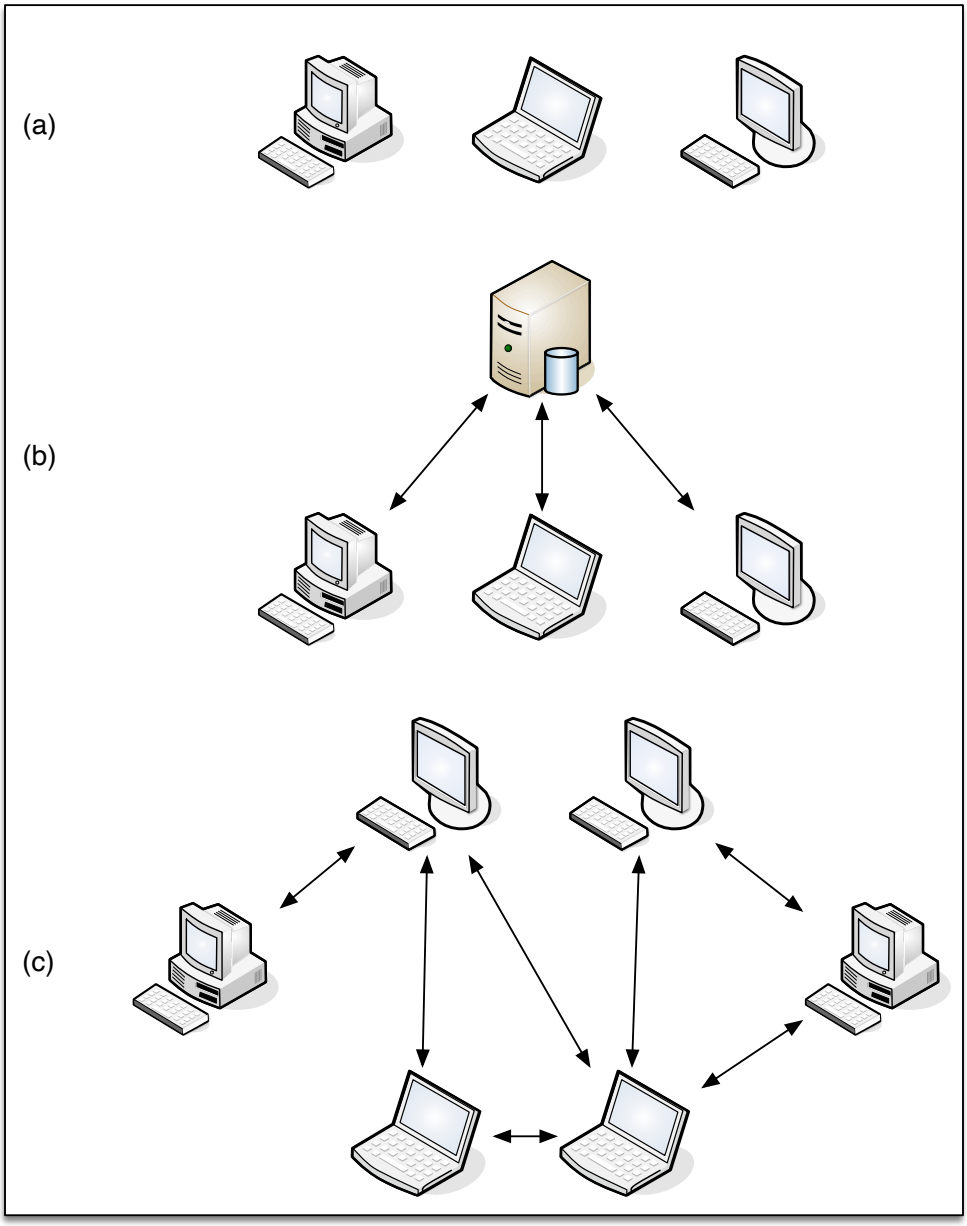

FIGURE 2.3 – Architectures réseau.

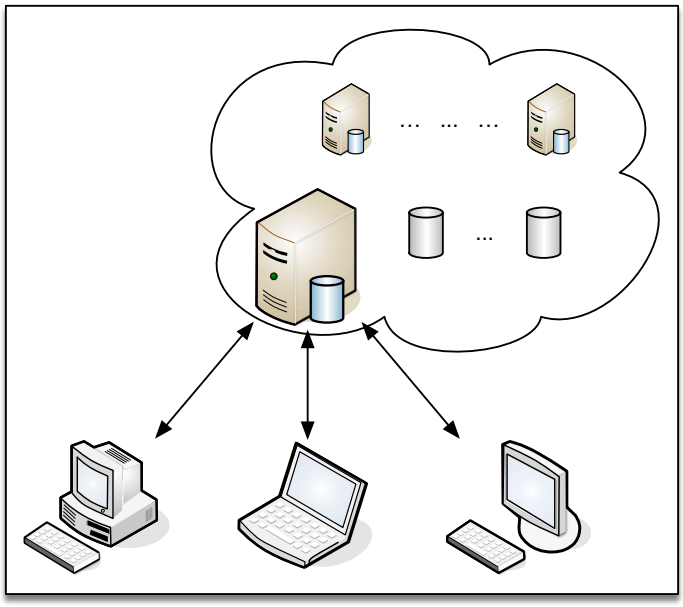

FIGURE 2.4 – Architecture *cloud*.

IP) qui doit être unique au sein des machines connectées entre elles (*i. e.* plusieurs réseaux privés non reliés peuvent utiliser les mêmes adresses logiques). Enfin, pour simplifier la vie des utilisateurs, on peut doter les machines (toutes, ou seulement les plus importantes) de noms symboliques. Il est alors nécessaire de disposer d'un service qui permette de faire la conversion des noms symboliques en adresses (c'est le rôle qui échoie au système DNS pour le réseau Internet, chargé de convertir un nom de domaine en adresse IP).

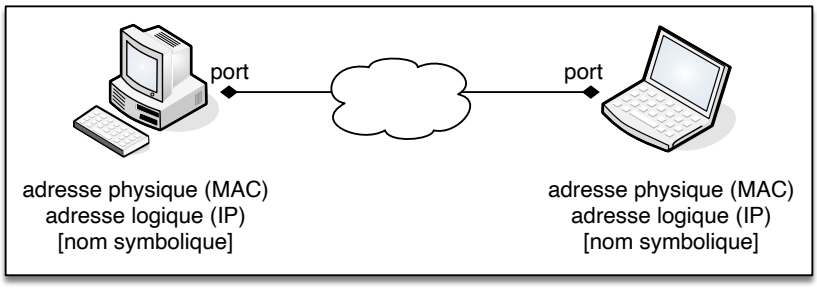

FIGURE 2.5 – Connexion et adresses de deux machines.

Comme le montre la figure Figure 2.5, les adresses permettent de lier logiquement deux machines. Un problème subsiste cependant si ces deux machines sont susceptibles d'exécuter simultanément plusieurs programmes accédant au réseau. Pour résoudre ce problème, on a recours à la notion de *port*. Il s'agit d'un entier positif, associé à une connexion réseau. Ainsi, lorsqu'un programme désire établir une connexion vers un programme s'exécutant sur une autre machine, il doit donner à la fois l'adresse ou le nom de la machine et également le *port* sur lequel le programme distant attend une connexion. À chaque service réseau ou programme est donc associé un numéro de port par convention. La table 2.1 liste les ports de quelques services réseau ; les numéros de port supérieurs à 1023 sont considérés comme libres et peuvent être utilisés par d'autres programmes.

port	service	nature
21	FTP	échange de fichiers
22	SSH	accès distant sécurisé
23	telnet	accès distant non-sécurisé
25	SMTP	envoi de courrier électronique
53	DNS	annuaire de noms de domaine
67 et 68	DHCP	distribution automatique d'adresses IP
80	HTTP	navigation web
110	POP	accès email
143	IMAP	accès email

TABLE 2.1 – Quelques ports standards.

2.5 Pile réseau

La notion de pile réseau est une modélisation en couches des différents services que doit proposer le réseau. Il existe différentes modélisations comprenant plus ou moins de couches (en général de quatre à sept). Nous présentons ici la pile dite IP qui est celle au cœur d'Internet.

La figure Figure 2.6 montre les quatre couches de la pile IP. Il faut voir chacune des deux colonnes comme représentant une machine avec un programme s'exécutant. Les flèches montrent le cheminement des données partant du programme sur la machine de gauche et allant jusqu'au programme sur la machine de droite.

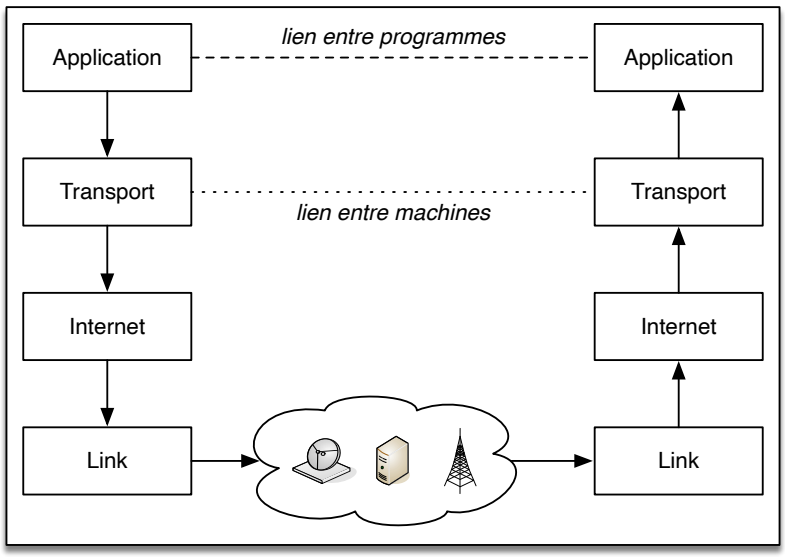

FIGURE 2.6 – Pile IP.

De haut en bas, les quatre couches sont donc : *application* (*i. e.* le programme en cours d'exécution), *transport* (*i. e.* les routines du système d'exploitation chargées de faire le lien entre les machines), *internet* (*i. e.* la gestion des adresses IP) et *link* (*i. e.* le lien entre les éléments matériels constitutifs du réseau).

Comme on le voit, les données issues d'un programme doivent traverser toutes ces couches vers le bas avant d'être effectivement émises sur le réseau sous forme de *paquets*. Lorsque la machine destinataire reçoit un paquet, les couches de la pile sont parcourues en sens inverse jusqu'à remonter au programme destinataire.

Il est important de noter deux notions complémentaires : l'*encapsulation* et l'*enveloppe*. A chaque fois que les données descendent d'une couche, elles sont encapsulées par l'ajout d'une enveloppe. L'enveloppe contient les informations nécessaires pour que la couche homologue de la machine destinataire et/ou des équipements intermédiaires puissent effectuer les contrôles nécessaires (est-elle bien destinataire ? les données ont-elles été corrompues ? *etc*). Elle permet également d'effectuer le routage d'un paquet de sa source à sa destination. Le routage consiste à déterminer le chemin à emprunter à travers le réseau.

Bien entendu, cette accumulation de données de contrôle a pour conséquence que les paquets envoyés sur le réseau sont sensiblement plus volumineux que les données qui intéressent les applications.

2.6 Modes connectés et déconnectés

Pour ces raisons de surcoût (notamment dues aux enveloppes), la plupart des piles réseaux proposent deux modes de liaisons entre applications : le mode connecté (parfois également appelé *flux*) et le mode déconnecté (parfois également appelé *datagramme*). Au sein de la pile IP, on peut assimiler le premier mode au protocole TCP et le seconde au protocole UDP.

Dans un mode connecté, de nombreux contrôles sont effectués et il est garanti par le protocole que :

- les données arriveront dans l'ordre d'émission ;
- s'il est détecté qu'un paquet a été "perdu" ou "corrompu", le destinataire demandera automatiquement à l'émetteur sa ré-émission ;
- si l'un des deux programmes décide de couper la connexion, cette information sera *ultimement* (*i. e.* sans garantie de délai maximal) notifiée à l'autre programme.

Ces garanties permettent au développeur d'un programme réseau de disposer de bases relativement solides. Il n'a pas à s'inquiéter de la perte de paquets ou de leur réception dans le désordre. En revanche, toutes ces garanties ont un coût en termes de place (nécessité de placer plus d'informations dans l'enveloppe) et en temps (en cas de détection d'un paquet perdu ou corrompu, celui-ci sera réenvoyé jusqu'à ce qu'il soit correctement reçu).

Si ces garanties induisent un surcoût inacceptable pour l'application, il est possible d'avoir recours à des modes déconnectés. Ceux-ci ne fournissent aucune des garanties évoquées ci-dessus, mais permettent de minimiser le surcoût dû à l'enveloppe et de ne pas attendre que les paquets arrivent tous et dans l'ordre. Les modes déconnectés sont ainsi particulièrement utiles si l'on souhaite envoyer énormément de données et que la perte de quelques paquets n'est pas problématique, typiquement dans des systèmes de *streaming* audio/vidéo ou de jeux vidéos.

3 Utilisation de *sockets* en Python

Dans la plupart des langages de programmation et sur les différents systèmes d'exploitation, on trouve des *sockets* qui permettent de faire communiquer des programmes entre eux. Ces programmes peuvent se trouver sur une même machine, ou sur des machines reliées par un réseau. Les *sockets* sont un mécanisme de bas niveau qui permet d'établir des connexions et d'envoyer et recevoir des données sous la forme de suites d'octets. Elles établissent un canal de communication bidirectionnel entre deux programmes.

On distingue deux modes de fonctionnement des *sockets* selon qu'elles sont utilisées du côté client ou du côté serveur. Du côté du client, la *socket* établit une connexion vers le serveur en

indiquant l'adresse et le port du programme serveur. Du côté du serveur, la *socket* se place en écoute sur un port et attend que des clients sollicitent une connexion.

3.1 Serveur en mode connecté

En Python, on a recours au module `socket`. La création d'une *socket* se fait par la classe de même nom qui prend typiquement deux paramètres : le type d'adresse (`AF_INET` pour IP v4 ou `AF_INET6` pour IP v6) ainsi que le type de socket (`SOCK_STREAM` pour TCP soit le mode connecté ou `SOCK_DGRAM` pour UDP soit le mode déconnecté). L'appel de `socket` renvoie une instance de la classe `_socketobject`.

Ensuite, si l'on souhaite utiliser la *socket* créée en mode serveur, il faut indiquer l'adresse sur laquelle on souhaite attendre la connexion par la méthode `bind`, puis lancer l'écoute par la méthode `listen`. Une fois cela fait, un appel à `accept` est bloquant (*i. e.* attend jusqu'à ce qu'une connexion soit établie) et renvoie un couple contenant une instance de *socket* pour communiquer avec le client, ainsi que son adresse. Le programme Code 3.1 montre le code d'un serveur qui attend une connexion sur le port `9999` (l'adresse `127.0.0.1` sert à désigner la machine locale, *i. e.* sur laquelle s'exécute le programme). À chaque connexion, il attend un message du client, convertit celui-ci en majuscule, l'envoie, et ferme la connexion afin d'attendre la connexion d'un autre client.

Si l'on souhaite, comme c'est généralement le cas, que le programme serveur puisse gérer plusieurs connexions simultanément, il faut après qu'une connexion a été acceptée lancer un traitement dédié à la gestion de la connexion. Cela peut se faire en lançant un nouveau processus, ou bien un nouveau thread (cf. séance sur la concurrence). Dans ce dernier cas, il faut être vigilant au fait que les accès aux données partagées par les différents threads doivent être synchronisés.

Code 3.1 – Exemple de *socket* en mode serveur.

```
1  import socket
2
3  adresse_serveur = "127.0.0.1", 9999
4  taille_max_donnees = 2048
5  serveur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6  serveur.bind(adresse_serveur)
7  serveur.listen(1)
8  while True:
9    connexion_client, adresse_client = serveur.accept()
10   donnees = connexion_client.recv(taille_max_donnees)
11   if donnees: connexion_client.send(str(donnees).upper())
12   connexion_client.close()
```

3.2 Client en mode connecté

L'utilisation d'une *socket* en mode client débute de la même manière qu'en mode serveur, par un appel au constructeur `socket`. Ensuite, la connexion est établie en appelant la méthode `connect` de l'instance obtenue, et en lui passant les adresse et port du serveur. Une fois cela fait, la *socket* peut être utilisée comme côté serveur, par le biais des méthodes `send` et `recv`. Le programme Code 3.2 montre le code d'un client se connectant au serveur programmé ci-dessus pour lui

envoyer une chaîne de caractères puis récupérer les données afin de les afficher. Ici, comme dans le cas du serveur, il est important de noter que les méthodes `recv` sont bloquantes (on attend jusqu'à ce que des données soient disponibles en lecture).

Code 3.2 – Exemple de *socket* en mode client.

```
adresse_serveur = "127.0.0.1", 9999
taille_max_donnees = 2048
connexion = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connexion.connect(adresse_serveur)
connexion.send(bytearray("Message a envoyer"))
donnees = connexion.recv(taille_max_donnees)
print(str(donnees))
```

3.3 Utilisation en mode déconnecté

Pour utiliser les *sockets* en mode déconnecté, il faut commencer par remplacer `SOCK_STREAM` par `SOCK_DGRAM` lors de la création. Ensuite, les changements portent sur les méthodes appelées pour les envois et réceptions de données. En effet, à partir du moment où l'on n'opère plus en mode connecté, il faut préciser l'adresse et le port de destination lors de chaque envoi. Et, symétriquement, à la réception on dispose non seulement des données reçues, mais également de l'adresse de l'expéditeur. On utilise alors par exemple les méthodes `sendto` et `recvfrom` en lieu et place des méthodes `send` et `recv`.

Ainsi, pour préciser l'adresse à l'envoi, on écrira `sendto(donnees, adresse)`, tandis qu'à la réception on recevra un couple plutôt qu'une seule valeur : `donnees, adresse = recvfrom(taille)`. Dans les deux cas, l'adresse est un couple ⟨ hôte, port ⟩, comme la variable `adresse_serveur` dans les extraits de code.

4 Programmation web

4.1 Protocoles

Les *sockets* ne fournissent qu'une abstraction de très bas niveau sur les opérations nécessaires à la communication via le réseau. Elles ne permettent en effet que d'échanger des suites d'octets entre machines. Afin de construire des services plus avancés, notamment à travers le réseau Internet, on a recours à la notion de protocole réseau.

Un protocole est une spécification (idéalement formelle) des règles régissant une communication. Un protocole décrit les différents états d'un service et de ses interlocuteurs, ainsi que les types de messages qui peuvent être échangés. Ces types de messages dépendent des états des machines parties prenantes, et peuvent en retour modifier ces états. On définit un protocole pour chaque service, tels que par exemple ceux listés à la table 2.1.

Le protocole au cœur du web est le protocole HTTP (pour HYPERTEXT TRANSFER PROTOCOL) ou HTTPS (où S signifie secure). HTTP(S) fonctionne au-dessus de TCP qui fournit les primitives de gestion de la connexion. Les serveurs web (*p. ex.* Apache [1]) répondent à des requêtes le plus souvent

1. https://httpd.apache.org

émises par des navigateurs web (*p. ex.* Firefox [2]) par le biais d'URLs.

Le protocole HTTP définit comment un client demande les données relatives à une URL, une URL simple ayant la forme :

```
http://www.nom.fr/page.html
```

Afin de pouvoir écrire des pages dont le contenu dépend de paramètres, on peut avoir recours à des URLs plus complexes, telles que la suivante :

```
http://www.nom.fr/page.html?c1=v1&...&cn=vn
```

où les couples $\langle c_i, v_i \rangle$ représentent les clefs et valeurs d'un dictionnaire contenant l'ensemble des paramètres passés à la page.

Pour demander à un serveur le contenu d'une page, il suffit d'envoyer la requête :

```
GET /page.html
```

sur une *socket* après s'être connecté à la machine de nom `www.nom.fr` sur le port 80 (qui est le port désigné pour HTTP). Il est possible d'exprimer d'autres types de demandes par ce que l'on appelle en HTTP des méthodes :

- `GET` télécharge le contenu de la page ;
- `POST` fait de même, en utilisant un autre encodage des paramètres (qui n'apparaissent alors plus dans l'URL).
- `PUT` permet d'envoyer une page vers le serveur ;
- `DELETE` permet de supprimer une page sur le serveur ;
- `HEAD` permet d'obtenir des métadonnées associées à la page (*p. ex.* date de dernière modification) ;
- *etc*

En réponse à une telle requête, le serveur envoie au client un code de statut, et éventuellement des données additionnelles (typiquement le contenu de la page si elle existe). Le code de statut permet par exemple de savoir qu'une page n'existe pas (code 404, largement connu des internautes), qu'une page a été déplacée, *etc* Le protocole est décrit dans son ensemble par le document disponible à l'adrese suivante :

```
http://tools.ietf.org/html/rfc2616
```

Les protocoles étant le plus souvent complexes à implémenter, Python propose de nombreux modules permettant d'accéder facilement aux différents services Internet (web, mails, shells distants, *etc*). Ces modules sont regroupés dans la documentation en ligne à l'adresse suivante :

```
http://docs.python.org/3/library/internet.html
```

2. http://www.mozilla.org/fr/firefox

4.2 Rendu de pages

Les différents protocoles Internet permettent de définir comment sont échangées des données entre machines, mais ne disent pratiquement rien sur le format de ces données. Afin de définir comment doit effectivement s'afficher une page web ou un mail dans un logiciel client, on a donc recours à des formats de données, le plus utilisé sur Internet étant le langage HTML pour HyperText Markup Language.

En marge de HTML, qui sert essentiellement à décrire le contenu statique d'une part, la programmation d'une page complexe peut nécessiter l'utilisation d'autres langages tels que JavaScript (pour les aspects dynamiques) ou CSS (pour la gestion du style). Un important point commun de tous ces langages de programmations et formats de données est qu'ils utilisent et sont transmis comme du texte simple.

On distinguera au sein de ces langages deux catégories :

- les véritables langages de programmation (comme JavaScript), qui permettent de faire des traitements complexes (utilisation de variables, de tests, de boucles, *etc*) ;
- les langages de rendu (comme HTML ou CSS) qui servent simplement à décrire des données et comment les présenter (position, police de caractères, *etc*).

4.3 Langage HTML

Le langage de présentation HTML est fondé sur du texte simple et repose sur des balises, de la même manière que le langage XML. Une page HTML simple prend la forme montrée par le code Code 4.1.

Code 4.1 – Exemple de page HTML.

```
<html>
  <head><title>Titre de la page</title></head>
  <body>
    texte a afficher<br>
    suite
  </body>
</html>
```

Les tags `html` (ouverture par `<html>` et fermeture par `</html>`) entourent l'ensemble des données de la page, et les tags `head` et `body` délimitent respectivement les métadonnées de la page et son contenu effectif. Le tag `title` sert à indiquer le titre de la page, tandis que `br` permet de demander un retour à la ligne. De nombreux tags permettent de définir la manière dont le texte s'affiche, par exemple :

- `b` pour l'utilisation d'une police grasse ;
- `i` pour l'utilisation d'une police italique ;
- `h1`, `h2`, *etc* pour des titres de niveau 1, 2, *etc*

Bien entendu, ces tags sont combinables, de telle sorte qu'un titre de premier niveau en italique sera obtenu par le texte : `<h1><i>Texte du titre</i></h1>`.

Éléments constitutifs du web, les liens entre pages sont exprimés à l'aide du tag `a`, en donnant à la fois le texte du lien et l'adresse de destination ; par exemple :

```
<a href="url-de-destination">texte du lien</a>
```

De très bons tutoriels et documents de référence pour HTML (et d'autres technologies liées au web) sont disponibles sur le site http://www.w3schools.com.

Pour permettre la saisie et l'envoi de paramètres depuis une page vers une autre, on utilise des formulaires (tag `form`) pour lesquels on doit indiquer la page de destination et la méthode de passage des paramètres. Le contenu du formulaire est donné par des tags `input`, comme le montre le code Code 4.2.

Code 4.2 – Exemple de formulaire HTML.

```
<form action="url-de-destination" method="get">
  <input type="type-element" name="nom-element" value="valeur-element">
    texte affiché
  </input>
</form>
```

Parmi les types d'éléments disponibles, on dispose notamment de :

- `checkbox` pour des cases à cocher ;
- `radio` pour des cases de sélection ;
- `text` pour des champs de texte ;
- `submit` pour le bouton de validation ;
- *etc* [3]

La page présentée par le code Code 4.3, qui contient un formulaire, sera affichée par un navigateur web comme le montre la figure Figure 4.1.

Code 4.3 – Exemple de page HTML contenant un formulaire.

```
<html>
  <head><title>Titre de la page</title></head>
  <body>
    <h1>Quels equipements electroniques possedez-vous ?</h1>
    <form action="dest.html" method="get">
      <input type="checkbox" name="telephone" value="1">telephone␣
↪portable</input> <br>
      <input type="checkbox" name="tablette" value="1">tablette</input>
↪ <br>
      <input type="checkbox" name="ordinateur" value="1">ordinateur</
↪input> <br>
      <br>
      Possedez-vous une connexion Internet ?
      <input type="radio" name="connexion" value="1">oui</input>
      <input type="radio" name="connexion" value="0">non</input> <br>
      <input type="submit" name="Envoyer">
    </form>
```

(suite sur la page suivante)

3. La page http://www.w3schools.com/html/html_forms.asp présente d'autres types d'éléments, ainsi que de nombreux exemples.

(suite de la page précédente)

```
15    </body>
16  </html>
```

FIGURE 4.1 – Rendu d'une page HTML comportant un formulaire.

Si l'utilisateur coche les cases comme montré sur la figure Figure 4.1, l'appui sur le bouton "Envoyer" va provoquer la requête par le navigateur web de la page se trouvant à l'URL suivante :

```
/dest.html?telephone=1&ordinateur=1&connexion=1&Envoyer=Envoyer
```

On y voit comment la sélection de l'utilisateur se trouve encodée par les paramètres passés à la page, à partir des noms et valeurs donnés dans le formulaire.

4.4 webs 1.0 et 2.0

Même si les webs 1.0 et 2.0 se distinguent surtout au niveau des usages (le second mettant l'accent sur les fonctionnalités sociales), ils diffèrent également en termes de développement. Les figures Figure 4.2 et Figure 4.3 montrent sous forme de chronogrammes les interactions typiques entre un client et le serveur.

Dans le cas du web classique, chaque requête émanant de l'utilisateur consiste à suivre un lien hypertexte et en conséquence à demander l'affichage d'une nouvelle page HTML. Le serveur envoie alors au client l'ensemble du texte de la page (le navigateur du client, après parcours de la page, sera éventuellement amené à demander l'envoi d'autres ressources comme des images).

Dans le cas du web 2.0, il est fréquent que l'utilisateur ne change pas de page, mais demande l'actualisation des données qui s'y trouvent. Le cas typique est celui de l'utilisation d'un *webmail* : par un clic, l'utilisateur demande l'affichage d'un message donné. Ici, le serveur n'envoie pas les données de l'ensemble de la page, mais seulement celles qui sont nécessaires. Techniquement, cela se fait le plus souvent par le biais d'AJAX, pour Asynchonous JAVascript and xml. Comme son nom l'indique, cette méthode de programmation implique d'utiliser d'autres langages en plus d'HTML. Il est cependant possible de développer des sites fondés sur AJAX sans écrire explicitement de code JavaScript (celui-ci est alors généré automatiquement) par le biais de *frameworks* tels que Django [4] pour Python.

4. Disponible à l'adresse http://djangoproject.com

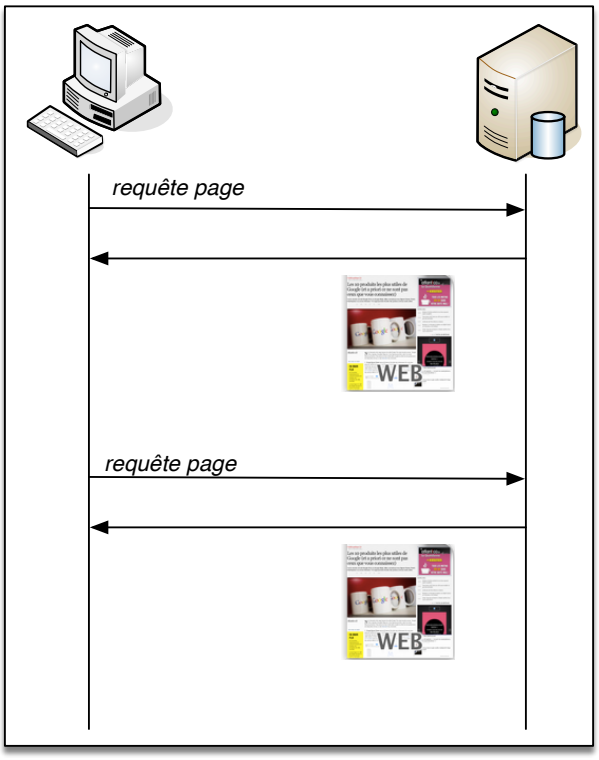

FIGURE 4.2 – Échange de données typique en web 1.0.

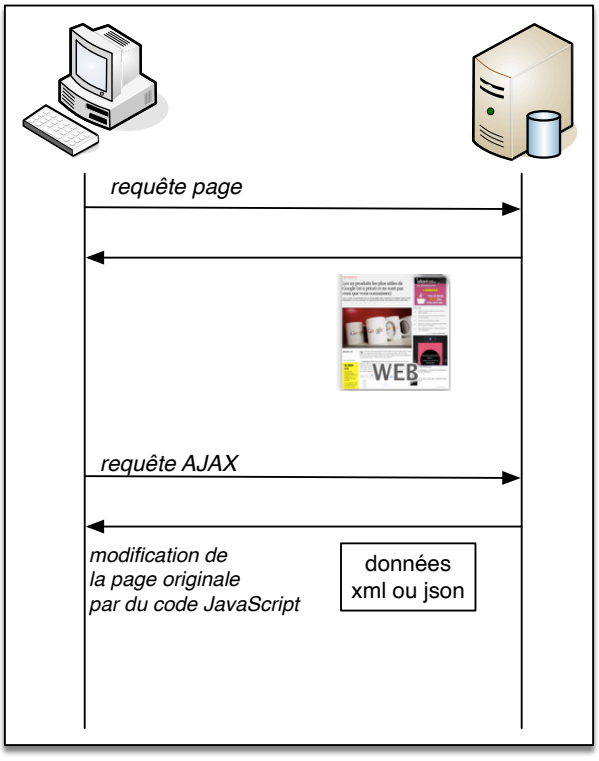

FIGURE 4.3 – Échange de données typique en web 2.0.

4.5 Librairie `web.py`

La librairie `web.py`, disponible à l'adresse http://webpy.org, permet d'écrire facilement des sites web (de type 1.0) en utilisant le langage Python pour construire le contenu des pages. La librairie est très facile d'emploi, et permet de s'abstraire complètement des opérations d'envoi de données. Basiquement, il suffit d'avoir un programme comme celui présenté par le code Code 4.4.

Code 4.4 – Canevas de site web.

```
import web

if __name__ == '__main__':
  urls = (
    url_1, nom_classe_1,
    ...
    url_n, nom_classe_n
  )
  webapp = web.application(urls, globals())
  webapp.run()
```

Les couples $\langle url_i, nom\_classe_i \rangle$ servent à indiquer que c'est la classe nom_classe_i qui est responsable du calcul du contenu de la page se trouvant à l'URL url_i. Par exemple :

```
urls = (
  "/index.html", "index"
)
```

Dès lors, un serveur web complet est lancé lorsque l'on exécute le programme Python, et on peut accéder à la page `index.html` en ouvrant un navigateur web et en utilisant l'adresse `http://localhost:8080/index.html`. Dans cette adresse, 8080 est le numéro de port utilisé par le serveur web, et peut être modifié en passant une autre valeur sur la ligne de commande lorsque l'on appelle l'interpréteur Python, par exemple : `python monserveur.py 9999`.

Il ne reste plus alors qu'à programmer les classes relatives aux pages à générer. Ces classes doivent implémenter une méthode (au sens Python) dont le nom correspond à la méthode (au sens HTTP) utilisée pour les requérir. Typiquement, il s'agira de la méthode `GET`, ce qui conduit à des classes de la forme montrée par le code Code 4.5. À l'intérieur d'une méthode `GET`, l'accès aux paramètres passés à l'URL (sous la forme `?p1=v1&...&pn=vn` dans l'URL) se fait par un dictionnaire obtenu par l'appel `web.input()`.

Code 4.5 – Classe implémentant une page.

```
class index:
  def GET(self):
    ... calcul de la page
    return chaine de caracteres contenant le code HTML de la page
```

Dans de nombreux cas, le code HTML d'une page ne dépend qu'en partie des paramètres reçus. Pour alléger la construction du code HTML, on a alors recours à un moteur de *templating*. Dans le cas de `web.py`, on va créer un tel moteur en appelant `render = web.template.`

`render("templates/")` où `templates` désigne le nom d'un sous-répertoire dans lequel on va placer des modèles pour les pages que l'on souhaite générer. On modifie alors le code Code 4.5 comme le montre le code Code 4.6.

Code 4.6 – Classe implémentant un page via un moteur de rendu.

```
class index:
  def GET(self):
    ... calculs divers
    return render.index(nom, liste)
```

Le moteur de rendu va alors chercher un fichier de nom `templates/index.html` et interpréter son contenu comme une fonction calculant le code HTML, en lui passant les paramètres fournis. Le code Code 4.7 donne un exemple de template. Un template comporte du code HTML enrichi d'expressions spécifiques introduites par le caractère `$`. Dans le code Code 4.7, `$nom` est remplacé par la valeur du paramètre `nom` passé pour le rendu de la page. La construction `for` permet de parcourir les éléments d'une liste. Il existe d'autres constructions reconnues par le moteur de rendu ; la documentation peut être consultée à l'adresse http://webpy.org/templetor.

Code 4.7 – Exemple de template de page.

```
$def with (nom, liste)
<!DOCTYPE html>
<html>
  <head>
    <title>titre de la page</title>
  </head>
  <body>
    L'utilisateur $nom déclare:
    <ul>
      $for elem in liste:
        <li>$elem</li>
    </ul>
  </body>
</html>
```