Durant cette étude, nous aborderons le développement réseau qui permet de faire communiquer plusieurs machines distantes entre elles. Python fournit deux niveaux d'accès aux services réseau.

*À bas niveau, vous pouvez accéder au support des **sockets** de base du système d'exploitation sous-jacent, ce qui vous permet d'implémenter les clients et les serveurs pour les protocoles orientés connexion.*

*Python procure également des bibliothèques qui offrent un accès supérieur à des protocoles réseaux spécifiques au niveau des applications, telles que le **FTP**, le **HTTP**, **SMTP**, etc.*

Principe général d'une communication réseau – les sockets

Pour que nous puissions communiquer en réseau, il faut qu'une machine puisse rendre un ou plusieurs services. Dans ce cas de figure, cette machine est considérée comme un serveur. Bien entendu, sur le même réseau local, plusieurs machines peuvent être des serveurs. Voici quelques principes simples sur la notion de communication réseau :

- *Pour localiser le serveur qui vous intéresse, il faudra connaître son **adresse IP**. Ensuite, sur ce serveur, pour utiliser le bon service auquel vous souhaitez soumettre une question, vous devez l'identifier par son **numéro de port**.*
- *Toutes les autres machines qui souhaitent utiliser ce service sur ce serveur sont considérées comme des machines clientes.*
- *Pour que le client et le service puissent se comprendre et communiquer correctement, il faut qu'ils parlent le même langage, c'est-à-dire qu'ils utilisent le même **protocole**.*
- *Enfin, un serveur doit normalement répondre à tous les clients qui se connectent. Il doit donc prendre en compte une gestion multi-tâche (**multi-threading**).*

Le développement réseau implique la mise en œuvre d'un canal de communication bidirectionnel (dialogue dans les deux sens) entre deux machines dont les deux extrémités (chacune étant en contact avec une machine) représentent les sockets (points de connexion).

Il faudra donc mettre en place une socket côté client, mais aussi une socket côté serveur pour que le canal de communication puisse s'établir. Il va de soit que si le serveur veut répondre à plusieurs clients en même temps, il devra disposer d'autant de sockets qu'il y a de clients (chaque socket représente un client).

*Nous avons évoqué la notion d'adresse IP (**@IP**). Elle doit être unique dans le réseau local, ce qui permet d'identifier une machine (un système numérique : ordinateur, smartphone, raspberry, etc.), à l'image de notre adresse personnelle pour recevoir correctement notre courrier.*

Au dessus de l'adresse IP, il existe plusieurs modes de transport de l'information. Deux sont couramment utilisés :

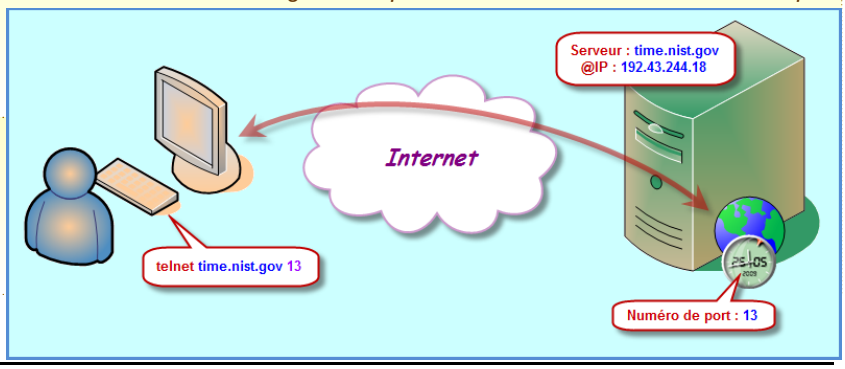
- ***TCP** : (**T**ransfert **C**ontrol **P**rotocol) qui est mode de transport sûr, qui contrôle par un système d'accusé-réception que l'information est bien parvenue à son destinataire. Ce principe est vraiment bien constitué, la contre partie c'est que ce protocole est très verbeux et prend pas mal de temps supplémentaires. Toutefois, c'est celui qui est le plus utilisé. Ce protocole **TCP** est dit « orienté connexion », c'est-à-dire que les applications sont connectées pour communiquer et que nous pouvons être sûr, quand nous envoyons une information au travers du réseau, qu'elle a bien été réceptionnée par l'autre application. Si la connexion est rompue pour une raison quelconque, les applications doivent rétablir la connexion pour communiquer de nouveau sur le réseau.*
- ***UDP** : (**U**ser **D**atagram **P**rotocol) Il existe des cas de figure, où le temps de communication est primordial, notamment pour les flux audio (téléphone) ou les flux vidéo. Ici, même si certains passages ne passe pas très bien tant-pis, il faut que le flux ne soit pas interrompu. Dans ce cas là, nous n'avons pas besoin de vérifier que chaque paquet envoyé (petite portion d'information) soit bien reçu par le destinataire. Une fois que le canal de communication est mis en place, on laisse passer le flux sans interruption quoi qu'il arrive, le transfert est alors beaucoup plus rapide. Ainsi, le protocole **UDP** envoie des informations au travers du réseau sans se soucier de savoir si elles seront bien réceptionnées par la cible. Ce protocole n'est pas connecté, une application envoie quelque chose au travers du réseau en spécifiant une cible. Ce type de protocole est utile si vous avez besoin de faire transiter beaucoup d'informations au travers du réseau mais qu'une petite perte occasionnelle d'informations n'est pas très handicapante. Nous trouvons ce type de protocole dans les flux importants de données. Cela fait beaucoup à transmettre mais ce n'est pas dramatique s'il y a une petite perte d'informations de temps à autre puisque, quelques millisecondes plus tard, le serveur renverra de nouveau les informations.*

Première application cliente - connaître la date et l'heure exacte avec le service ITS

Je vous propose de mettre en œuvre notre première application cliente qui interroge un service sur le réseau (ici le réseau Internet). Le service que nous allons interroger est le service de date standard **ITS** (**I**nternet **T**ime **S**ervice) qui utilise le port **« 13 »** et qui se trouve aux États-Unis avec l'URL suivante : **« time.nist.org » (@IP 192.43.244.18)**.

*La plupart des serveurs UNIX implémentent en permanence ce système de date. Le serveur auquel nous allons nous connecter se trouve au **NIST** à Boulder dans le Colorado, et fournit l'heure d'une horloge atomique au césium. Naturellement, le temps affiché n'est pas parfaitement précis à cause des délais de propagation des informations sur le réseau. Par convention, le service date est toujours rattaché au port **13**.*

*Le programme (le service) intégré au serveur fonctionne en permanence sur la machine distante, attendant un paquet du réseau qui tente de communiquer avec le port **« 13 »**. Lorsque le système d'exploitation de l'ordinateur distant reçoit le paquet contenant une requête de connexion sur*



*le port **« 13 »**, le processus d'écoute du serveur est activé et la connexion est alors établie. Cette connexion demeure jusqu'à ce qu'elle soit arrêtée par l'une des deux parties.*

Une fois que la connexion est établie, le service (le programme) de l'ordinateur distant renvoie un ensemble de données, ici une chaîne de caractères correspondant à la date et à l'heure « temps universel », puis il clôture la connexion, puisque le client a obtenu sa réponse. Les ressources utilisées (mémoire interne, tâche concurrente, etc.) sont libérées pour répondre le plus tôt possible à un nouveau client.

principal.py

```
from socket import *

adresse = 'time.nist.gov'
port = 13
localisation = (adresse, port)

service = socket(AF_INET, SOCK_STREAM)
service.connect(localisation)
octets = service.recv(100)
réponse = octets.decode('utf8')

print('Chaîne retournée par le service Date :', réponse)
date = réponse.split(' ')
print('Date = {}, Heure = {} TU'.format(date[1], date[2]))
service.close()
```

Shell Python (Résultat)

```
Chaîne retournée par le service Date :
57987 17-08-22 14:57:08 50 0 0 587.9 UTC(NIST) *

Date = 17-08-22, Heure = 14:57:08 TU
```

*La création d'une socket est très facile à mettre en œuvre puisque une fonction qui porte son nom existe dans le module **« socket() »**. Cette fonction renvoie un objet correspondant au type de communication inter-processus souhaité. Ici, il s'agit d'une communication par Internet avec donc une adresse **IP** de type **IPv4 -** constante **« AF_INET »** et utilisant le mode de transport **TCP -** constante **« SOCK_STREAM »**.*

C'est très souvent ce type de socket qui est utilisé, du coup vous n'êtes pas obligés de spécifier ces arguments puisqu'ils sont proposés par défaut. D'autres propositions peuvent être faites. Vous pouvez choisir les adresses IP de type IPv6 – constante « AF_INET6 » et prendre le protocole de transport UDP – constante « SOCK_DGRAM ». Il existe également la possibilité de communiquer sur un BUS CAN (Bus utilisé dans les voitures, notamment pour communiquer avec les différents capteurs et actionneurs) – constante « AF_CAN ».

*Une fois que l'objet de type **« socket »** est crée, vous pouvez dès lors vous connecter au service avec lequel vous souhaitez soumettre votre question au moyen de sa méthode interne **« connect() »**. Pour cela, bien entendu, il faut localiser le service grâce à son adresse **IP** (ou son nom **DNS** équivalent) et son numéro de **port** qui sont proposés sous forme de **« tuple »**.*

*Une fois que le canal de communication est mis en place, le client peut maintenant dialoguer avec le service mis en relation. Deux méthodes sont prévues pour cela **« send() »** pour poser votre question, et **« recv() »** qui permet de récupérer la réponse à la question.*

Attention, les informations qui sont transités sur le réseau sont systématiquement considérées comme un flux d'octets et pas spécialement une chaîne de caractères. Python utilise un type approprié pour une séquence d'octets – le type « bytes ». Les données de type « bytes » sont très similaires aux données de type « str », à la différence près que ce sont strictement des séquences d'octets, et non des séquences de caractères.

*Les caractères peuvent bien entendu être encodés en octets, et les octets décodés en caractères, mais pas de manière univoque, puisqu'il existe plusieurs normes d'encodage/décodage, la même chaîne **« str »** peut être convertie en plusieurs chaînes **« bytes »** différentes. Ceci dit, l'encodage qui rencontre un énorme succès actuellement est l'encodage **« Utf-8 »** (indifféremment **« Utf-8 »**, **« utf-8 »**, **« Utf8 »** ou **« utf8 »**).*

*Pour en revenir à notre projet, le service date n'attends pas de question particulière puisqu'une seule réponse est fournie et c'est toujours la même, c'est l'heure actuelle en temps universel sous forme de chaîne de caractères. Du coup, dans notre exemple, nous n'avons pas besoin d'utiliser la méthode **« send() »**.*

*Pour conclure, dès que vous avez obtenue votre réponse et que vous ne souhaitez plus soumettre d'autres questions, il est fortement recommandé de clôturer votre connexion, grâce à la méthode **« close() »**, afin de libérer les ressources réseau, et tout ce qui s'y rattachent. Enfin, vous devez spécifier la taille du **« buffer »** (zone mémoire tampon) de lecture, qui est généralement de **« 1024 »**, dans la méthode **« recv() »**.*

principal.py (version simplifiée)

```
from socket import *

service = socket()
service.connect(('time.nist.gov', 13))
réponse = service.recv(1024).decode('utf8')
date = réponse.split(' ')
print('Date = {}, Heure = {} TU'.format(date[1], date[2]))
```

```
service.close()
```

Shell Python (Résultat)

```
Date = 17-08-22, Heure = 14:57:08 TU
```

AFFICHAGE D'UNE HORLOGE 'TEMPS UNIVERSEL' DANS UNE FENÊTRE

Reprenons les principes du projet précédent afin de fabriquer une application qui affiche une horloge en temps réel en consultant périodiquement le service **« ITS »**.

principal.py

```
from socket import *
from tkinter import *

class Fenêtre(Tk):
   def __init__(self):
      super().__init__()
      self.title('Horloge TU')
      self.heure = StringVar()
      self.temps()
      label = Label(self, textvariable=self.heure)
      label.config(font=('Arial', 28), bg='yellow', fg='green')
      label.pack(ipadx=5, padx=5, ipady=3, pady=5)
      self.mainloop()

   def temps(self):
      service = socket()
      service.connect(('time.nist.gov', 13))
      réponse = service.recv(50).decode('utf8')
      self.heure.set(réponse.split(' ')[2]+' TU')
      self.after(2000, self.temps)

# Programme principal
if __name__ == '__main__' : Fenêtre()
```

*Pour réaliser cette application, nous utilisons un **« Label »** en changeant son texte périodiquement. Pour cela, il existe une méthode **« after() »** qui est intégrée à tous les widgets de **« tkinter »**, quels qu'ils soient. Cette méthode permet d'exécuter la fonction ou la méthode de votre choix en spécifiant le temps d'attente avant qu'effectivement elle se lance (d'où le terme **after**).*

Afin d'obtenir une périodicité, l'astuce consiste à utiliser la méthode « after() » dans la même méthode qu'elle même sollicite (récursivité).

La périodicité est réglée ici à 2 secondes (2000 ms). Pourquoi ne pas choisir 1 seconde puisque c'est ce qui conviendrait le mieux ? Faites le test ! Et vous remarquerez que cela ne fonctionne plus du tout correctement. C'est là que nous découvrons les limites du langage Python, par rapport notamment au C++ qui a un temps de réponse 100 fois supérieur (langage compilé). Effectivement, la communication réseau prend toujours pas mal de temps (mise en place de la connexion, obtention de la réponse, clôture de la connexion, etc.).

UTILISER L'HORLOGE INTERNE

Lorsque vous utilisez le service **« ITS »**, c'est juste pour l'utiliser une seule fois, un petit peu comme l'horloge parlante pour connaître l'heure exacte à un moment donné. Ce n'est pas très malin de prendre pas mal de ressources réseau pour fabriquer une horloge pour cela. Je vous propose plutôt de prendre le module **« time »** qui possède notamment la fonction **« strftime() »**. Voici, ci-dessous le résultat et le code correspondant :

principal.py

```
from tkinter import *
from time import *

class Fenêtre(Tk):
   def __init__(self):
      super().__init__()
      self.title('Horloge')
      self.heure = StringVar()
      self.temps()
      label = Label(self, textvariable=self.heure)
      label.config(font=('Arial', 28), bg='yellow', fg='green')
      label.pack(ipadx=5, padx=5, ipady=3, pady=5)
      self.mainloop()

   def temps(self):
      self.heure.set(strftime('%A %d %B %Y - %H:%M:%S'))
      self.after(200, self.temps)

# Programme principal
from locale import *
if __name__ == '__main__' :
   setlocale(LC_TIME, '') # Pays correspondant au système d'exploitation
```

```
   Fenêtre()
```

*La fonction **«strftime()»** donne l'heure exacte actuelle (celle du système d'exploitation) sous forme de chaîne de caractère en indiquant ce que vous désirez représenter grâce à un motif (chaîne) qui contient les formateurs concernés.*

Les formateurs sont préfixés par le symbole « % » suivi de la lettre correspondante, comme par exemple la lettre « A » qui correspond au nom du jour de la semaine. Dans ce projet, j'ai placé pratiquement tous les marqueurs importants. Il suffit de regarder le résultat pour savoir à quoi correspond chacun des formateurs : «%A » nom du jour de la semaine, « %d » jour du mois, « %B » nom du mois, «%Y » année, «%H » l'heure, «%M » les minutes, «%S » les secondes. Il existe aussi «%m » pour le numéro du mois, «%x » pour la représentation de la date complète et «%X » pour la représentation complète de l'heure (Mettez uniquement ce formateur dans le motif pour tester).

*Par contre, par défaut, les noms du jour de la semaine et du mois seront écrit malheureusement en Anglais. Il est nécessaire que le script prenne automatiquement en compte la langue utilisée par le système d'exploitation. C'est ce que nous faisons au moyen de la fonction **« setlocale() »** qui se trouve dans le module **« locale »** en désactivant la constante **« LC_TIME »**.*

Deuxième application qui permet de récupérer une page Web brute sans interprétation

Notre deuxième application nous permet de communiquer au travers du protocole **« HTTP »** et de consulter un site Web sans passer par un navigateur. Un service Web est standardisé, et il est toujours sur le port **« 80 »**.

*Nous verrons ainsi la réponse **« HTTP »** suivi de son contenu qui est une simple page Web, directement avec le balisage **« HTML »** sans interprétation. Le résultat de notre premier exemple se fait en mode **« console »**.*

principal.py

```
from socket import *

service = socket()
service.connect(('www.google.fr', 80))              # connexion à un site Web
requête = 'GET / HTTP/1.1\n\n'                      # requête correspondant au protocole HTTP
service.send(requête.encode('utf8'))
while True:                                         # Récupérer de tout le contenu quel que soit la taille
   réponse = service.recv(1024)
   self.document.insert(END, réponse.decode('utf8'))
   if len(réponse)<1024: break
service.close()
```

Shell Python (Résultat)

```
HTTP/1.1 302 Found        # En-tête de réponse lié au protocole HTTP
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Referrer-Policy: no-referrer
Location: http://www.google.fr/?gfe_rd=cr&dcr=0&ei=skegWY3oBovBaIHllLAD
Content-Length: 266
Date: Fri, 25 Aug 2017 15:52:18 GMT

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8"> # Contenu de la page Web
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.fr/?gfe_rd=cr&amp;dcr=0&amp;ei=skegWY3oBovBaIHllLAD">here</A>.
</BODY></HTML>
```

*Nous ne rentrerons pas dans les détails, mais le protocole **HTTP** se fait sous forme de texte. Il existe plusieurs requêtes **HTTP**, respectivement : **GET** (récupérer une ressource), **POST** (envoyer une ressource), **PUT** (modifier une ressource) et **DELETE** (supprimer une ressource).*

*Ici, nous demandons à consulter une page Web (la page d'accueil **« / »**), c'est donc plutôt la méthode **GET** qui est la plus adaptée. C'est ce qu'utilise un navigateur (c'est la requête par excellence).*

*Nous avons mis en place une boucle, ce qui nous permet de récupérer tout le contenu de la réponse, même si celle-ci dépasse la capacité du tampon (**1024** octets).*

Page Web brute dans une application fenêtrée – Les Threads

Nous allons nous servir de l'expérience précédente pour mettre en œuvre une application fenêtrée que nous allons étoffer au fur et à mesure, en faisant en sorte qu'elle soit la plus conviviale possible. Nous plaçons une zone de texte qui comportera les informations issues de la requêtes, avec la réponse **HTTP** et la page Web non interprétée. Au-dessous, nous plaçons une ligne d'adresse et un bouton pour soumettre notre requête. Pour l'instant, nous supposons que l'adresse est valide. Dans un autre chapitre, nous prendrons en compte une erreur éventuelle de saisie de la part de l'utilisateur.

*À ce stade, il faut bien comprendre que lorsque les échanges au niveau du réseau peuvent être relativement conséquent, il est préférable de séparer la communication elle-même de la partie interface utilisateur. Effectivement, cette dernière est gérée, ne l'oublions pas, par une boucle qui tourne constamment, au moyen de la méthode **« mainloop() »**, jusqu'à ce qu'une demande d'interruption de l'utilisateur soit sollicité lorsque nous cliquons sur le bouton de clôture de l'application.*

Cette boucle est très difficilement compatible avec une méthode qui prend beaucoup de temps pour réaliser son traitement, comme cela peut être le cas avec une communication réseau.

Il faut alors prévoir un processus multi-tâches, la tâche principale pour l'application fenêtrée, et la tâche annexe en simultanée pour la communication réseau. Heureusement, cela se fait très simplement en Python.

La mise en œuvre d'une tâche concurrente s'effectue en héritant de la classe « Thread » qui se trouve dans le module « threading ». Ensuite, le traitement que devra effectuer cette tâche doit être écrit impérativement dans la méthode redéfinie « run() ».

Lorsque tout le code est définitivement terminée dans cette méthode, alors la tâche se termine tout simplement. Par contre, pour démarrer cette tâche (qui travaille en parallèle avec la tâche principale), vous devez explicitement appelé la méthode « start() ».

Enfin, si la tâche doit être en relation avec les éléments constituant l'interface graphique, il suffit alors de proposer un constructeur avec tous les paramètres nécessaires dont les arguments consécutifs seront les widgets en relation.

réseau.py

```
from socket import *
from threading import Thread
from tkinter import END

class Réseau(Thread):
   def __init__(self, adresse, document):
      super().__init__()
      self.adresse = adresse
      self.document = document

   def run(self):
      service = socket()
      service.connect((self.adresse.get(), 80))
      requête = 'GET / HTTP/1.1\n\n'
      service.send(requête.encode('utf8'))
      while True:
         réponse = service.recv(1024)
         self.document.insert(END, réponse.decode('utf8'))
         if len(réponse)<1024: break
      service.close()
```

principal.py

```
from tkinter import *
from réseau import Réseau

class Fenêtre(Tk):
   def __init__(self):
      super().__init__()
      self.title('Éditeur HTML')
      self.geometry('640x480')
      self.placerWidgets()
      self.mainloop()

   def placerWidgets(self):
      self.document = Text(self)
      self.document.pack(fill=BOTH, expand=1, padx=5)
      Label(self, text='Adresse :').pack(side=LEFT, padx=5)
      self.adresse = Entry(self)
      self.adresse.pack(side=LEFT, fill=X, expand=TRUE, ipady=3, pady=7)
      Button(self, text='Consulter', command=self.consulter).pack(side=RIGHT, padx=5)

   def consulter(self):
      self.document.delete(0.0, END)
      Réseau(self.adresse, self.document).start()

if __name__ == '__main__' : Fenêtre()
```

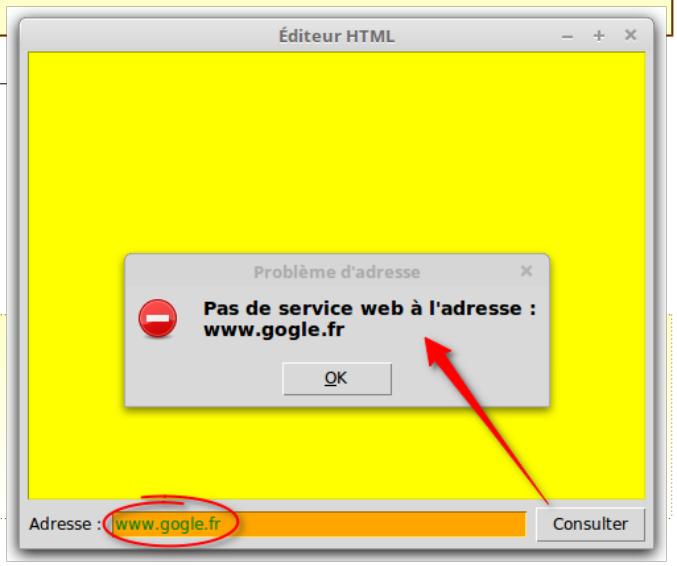
```
HTTP/1.1 302 Found
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Referrer-Policy: no-referrer
Location: http://www.google.fr/?gfe_rd=cr&dcr=0&ei=StuiWcrpG4iAaM26iRA
Content-Length: 265
Date: Sun, 27 Aug 2017 14:46:34 GMT

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.fr/?gfe_rd=cr&amp;dcr=0&amp;ei=StuiWcrpG4iAaM26iRA">here</A>
.
</BODY></HTML>
```

Adresse : www.google.fr Consulter

GESTION DES EXCEPTIONS – TRY … EXCEPT …

Cela vous est certainement arrivé plusieurs fois, lorsque vous lancer une application, quelque fois une erreur apparaît en ouvrant éventuellement le fichier (le script) qui pose problème, sachant que l'erreur vient surtout de notre propre code. Plutôt que d'avoir ces messages intempestifs, sans trop savoir quoi faire d'ailleurs, il est plus judicieux d'anticiper les erreurs possibles et de prévoir une alternative dans le cas d'une erreur exceptionnelle (qui n'arrive que très rarement).

Dans notre projet, par exemple, il peut nous arriver de proposer une adresse qui ne correspond pas à un service Web. Dans ce cas de figure, l'application s'arrête intempestivement sans que nous puissions continuer. C'est dommage ! il serait préférable d'avoir une boîte de dialogue qui s'ouvre automatiquement en cas de problème pour nous avertir. Il suffit ensuite de corriger notre erreur de saisie pour arriver à notre résultat définitif.



Heureusement, il existe une solution simple que propose en natif le langage Python, comme beaucoup d'autres langages d'ailleurs. Le principe est de tenter une suite d'instructions dans le bloc « try ».

Si tout se passe bien, tout ce qui est écrit dans ce bloc est exécuté normalement.

Si par contre, un problème apparaît à une ligne de code particulière, toutes les lignes écrites dans la suite du bloc « try » n'est plus exécutée (ce n'est plus la peine), automatiquement l'exécution passe directement dans le bloc « except » (exception) pour exécuter ce qui est prévu en cas de problème.

Enfin, Tout ce qui est écrit dans « except » n'est pas exécuté dans le cas où tout se passe correctement.

Grâce à ce système, les applications ne s'arrêtent plus de façon impromptue. Nous devons impérativement prévoir ce mécanisme pour la communication en réseau ou pour la gestion des fichiers. Par exemple, un câble réseau peut très bien être débranché ou un fichier peut tout simplement ne pas exister.

réseau.py

```
from socket import *
from threading import Thread
from tkinter import END
from tkinter.messagebox import showerror

class Réseau(Thread):
   def __init__(self, adresse, document):
      super().__init__()
      self.adresse = adresse
      self.document = document

   def run(self):
      try:
         service = socket()
         service.connect((self.adresse.get(), 80))
         requête = 'GET / HTTP/1.1\n\n'
         service.send(requête.encode('utf8'))
         while True:
            réponse = service.recv(1024)
            self.document.insert(END, réponse.decode('utf8'))
            if len(réponse)<1024: break
         service.close()
      except:
         showerror("Problème d'adresse", "Pas de service web à l'adresse : "+self.adresse.get())
```

principal.py

```
from tkinter import *
from réseau import Réseau

class Fenêtre(Tk):
   def __init__(self):
      super().__init__()
      self.title('Éditeur HTML')
      self.geometry('480x400')
      self.placerWidgets()
      self.mainloop()

   def placerWidgets(self):
      self.document = Text(self, bg='yellow', fg='brown')
      self.document.pack(fill=BOTH, expand=TRUE, padx=5)
      Label(self, text='Adresse :').pack(side=LEFT, padx=5)
      self.adresse = Entry(self, bg='orange', fg='green')
      self.adresse.pack(side=LEFT, fill=X, expand=TRUE, ipady=3, pady=7)
      Button(self, text='Consulter', command=self.consulter).pack(side=RIGHT, padx=5)

   def consulter(self):
      self.document.delete(0.0, END)
      Réseau(self.adresse, self.document).start()

if __name__ == '__main__' : Fenêtre()
```

AJOUT D'UN ASCENSEUR VERTICAL – SCROLLBAR ET FRAME

Si la page d'accueil du site Web que vous consultez est relativement conséquent, vous ne verrez pas la totalité du document **HTML** sur votre application, d'où l'idée de rajouter un ascenseur vertical. Nous pouvons aussi rajouter un ascenseur horizontal, mais la classe **« Text »** propose par défaut, le mode **« wrap »**, c'est-à-dire que si la ligne est trop grande, le reste du texte est automatiquement affiché en dessous sur une nouvelle ligne.

*Pour désactiver ce mode, il suffit de rajouter la valeur **« NONE »** au paramètre **« wrap »** du constructeur **« wrap=NONE »**.*

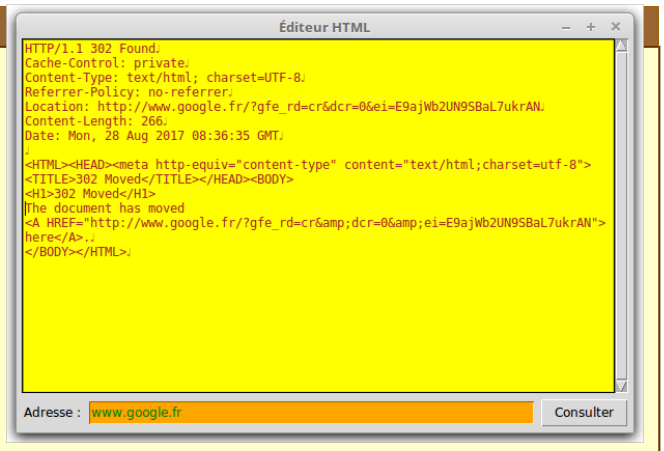
*La création d'un ascenseur est un peu particulier dans **« tkinter »**. Il doit être associé à un autre widget qui lui même doit connaître cet ascenseur. Ils sont intimement liés. Pour cela, vous avez besoin de deux composants, d'une par la classe **« Scrollbar »** qui représente l'ascenseur (vertical par défaut), et la classe **« Frame »** qui est un panneau intermédiaire dans*

lequel vous allez placer votre widget, ici ***« Text »*** *avec votre* ***« ascenseur »****. Grâce à ce panneau intermédiaire, c'est beaucoup plus facile de disposer vos deux composants pour qu'ils s'étendent le plus possible.*

Seule l'apparence change, donc aucune modification n'est apportée dans le script « réseau.py ».

principal.py

```
from tkinter import *
from réseau import Réseau

class Fenêtre(Tk):

   def __init__(self):
      super().__init__()
      self.title('Éditeur HTML')
      self.geometry('640x480')
      self.placerWidgets()
      self.mainloop()

   def placerWidgets(self):
      self.ascenseur()
      Label(self, text='Adresse :').pack(side=LEFT, padx=5)
      self.adresse = Entry(self, bg='orange', fg='green')
      self.adresse.pack(side=LEFT, fill=X, expand=TRUE, ipady=3, pady=7)
      Button(self, text='Consulter', command=self.consulter).pack(side=RIGHT, padx=5)

   def ascenseur(self):
      zone = Frame(self)
      zone.pack(fill=BOTH, expand=1, padx=5)
      vertical = Scrollbar(zone)
      self.document = Text(zone, bg='yellow', fg='brown', yscrollcommand=vertical.set)
      vertical.config(command=self.document.yview)
      self.document.pack(side=LEFT, fill=BOTH, expand=TRUE)
      vertical.pack(side=RIGHT, fill=Y)

   def consulter(self):
      self.document.delete(0.0, END)
      Réseau(self.adresse, self.document).start()

if __name__ == '__main__' : Fenêtre()
```



Pour mettre en relation les deux éléments, l'ascenseur et la zone de texte, vous devez d'abord créer votre ascenseur, ensuite vous réglez le paramètre ***« yscrollcommand »*** *lors de la création de la zone de texte avec la commande* ***« set() »*** *de votre ascenseur. Plus tard, vous configurez votre ascenseur pour qu'il soit capable de faire défiler votre document avec sa commande* ***« yview() »****.*

Pour définir un ascenseur horizontal, vous devez proposer la constante ***« HORIZONTAL »*** *avec le paramètre* ***« orient »*** *du constructeur. Par ailleurs, il faudra plutôt prendre le paramètre* ***« xscrollcommand »*** *et la commande* ***« xview() »****.*

PREMIER SERVICE – MONO-PASSE – MONO-UTILISATEUR

Avec Python nous pouvons également créer nos propres services, ce qui est bien sûr beaucoup plus intéressant. Comme nous restons dans le domaine de la connexion réseau, nous continuons à utiliser les sockets que nous connaissons déjà. Je dis des sockets puisqu'il en faut une pour le service lui-même, et ensuite autant de sockets qu'il y a de clients connectés. Afin de bien maîtriser tous les mécanismes en jeu, je vous propose de réaliser un service très rudimentaire qui renvoie simplement un texte soumis par le client en le mettant en majuscule.

L'objectif ici est de bien comprendre la mise en œuvre d'un service et de réduire au maximum la difficulté du traitement à réaliser. Pour générer un service, vous devez passer par un certain nombre de phase obligatoires :

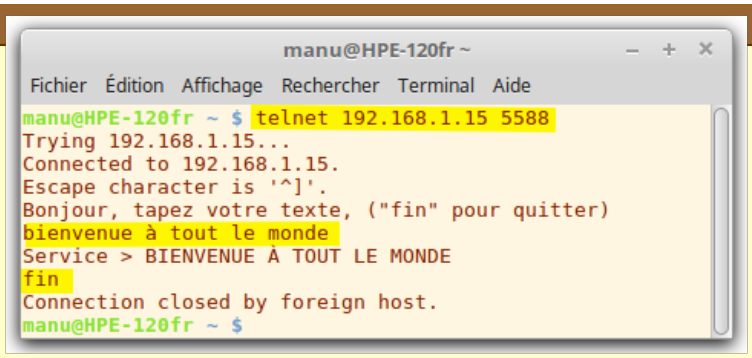
- ***Création du service*** *: cette première phase consiste à créer une* ***« socket »****, mais au lieu de demander une connexion (c'est un service), nous générons notre service en spécifiant l'adresse de l'ordinateur hôte ainsi que le numéro de service souhaité, tout ceci au travers de la méthode* ***« bind() »****. L'idéal est de mettre en place un bloc* ***« try...except »*** *pour contrôler que le numéro de port n'existe pas déjà, ou tout simplement pour vérifier que le service n'est pas déjà lancé.*
- ***Écoute et attente d'une demande de connexion par un nouveau client*** *: une fois que le service est lancé, vous devez tout de suite* ***« écouter »*** *toutes les communications qui transitent sur le réseau, notamment pour ce numéro de service, afin de vérifier qu'un nouveau client n'est pas en train de faire une demande de connexion. Cela se fait au moyen de la méthode* ***« listen() »****. Il est possible de spécifier un argument en donnant le nombre de clients possibles, en simultané, qui seront autorisés à utiliser ce service. Par défaut, c'est le système qui se débrouille tout seul. Sinon, c'est souvent la valeur* ***« 5 »*** *qui est proposée.*
- ***Acceptation et établissement de la connexion avec un client :*** *dès qu'un client demande à se connecter, une* ***« socket »*** *est mis en place automatiquement, et cette socket dons la suite du code représentera ce client en particulier. C'est la méthode* ***« accept() »*** *qui renvoie la socket ainsi que l'adresse du client (sous forme de tuple). La particularité de cette méthode, et c'est assez rare pour être souligné, c'est qu'elle est bloquante. Tant qu'aucun client ne se présente, le curseur d'exécution reste sur cette méthode et dès qu'un client apparaît, la suite du code est exécutée pour ce client en particulier.*

- ***Dialogue avec le client en suivant un protocole personnalisé :*** *Ensuite, nous communiquons avec le client avec la* ***« socket »*** *associée, comme nous l'avons déjà fait précédemment, en utilisant les méthodes* ***« send() »*** *et* ***« recv() »****. C'est à vous de décider quel protocole à mettre en œuvre, c'est-à-dire qui commence à parler (ici c'est le serveur) et quels sont les types d'informations à transmettre (ici des chaînes de caractères) et sous quel forme.*
- ***Clôturer la connexion avec le client lorsque les échanges sont terminés*** *: dès que la communication est terminée avec le client, il faut penser à clôturer la socket le concernant.*

Pour tester notre service, je vous invite à utiliser un client « telnet ».

service.py

```
from socket import *
from sys import exit

hôte = '192.168.1.15'
port = 5588
service = socket()

# Mise en place du service
try: service.bind((hôte, port))
except:
   print('Ce service est déjà en exécution')
   exit()

# Écoute du réseau et attente d'une demande de connexion d'un client pour ce service
service.listen()
print('Service en fonctionnement')

# Établissement de la connexion avec un client
client, adresse = service.accept()
print('Nouveau client : ', adresse)

# Dialogue avec le client
client.send('Bonjour, tapez votre texte, ("fin" pour quitter)\n'.encode('utf8'))
while True:
   try:
      message = client.recv(1024).decode('utf8').strip()
      if message.lower() == 'fin': break
      print('Client >', message)
      client.send(('Service > '+message.upper()+'\n').encode('utf8'))
   except:
      print('Impossible de décoder correctement le message')
      break

# Clôture des connexions
client.close()
print('Client déconnecté')
service.close()
print('Service interrompu')
```



Shell Python (Résultat)

```
Service en fonctionnement
Nouveau client :  ('192.168.1.15', 55796)
Client > bienvenue à tout le monde
Client déconnecté
Service interrompu
```

Vous remarquez deux particularités dans notre script. D'une part, nous avons un autre bloc ***« try...accept »*** *dans la partie où le service dialogue avec le client (protocole d'échange). Si le message reçu ne peut pas être décodé correctement, tout le protocole va être détruit, autant clôturer la session.*

Ensuite, vous remarquez la présence de la méthode ***« strip() »*** *qui, je le rappelle, permet de supprimer tous les espaces ou les caractères spécifiques en début ou en fin de phrase, notamment dans notre cas, le caractère de mise à la ligne qui peut poser problème.*

SERVICE MULTI-UTILISATEUR

Pour gérer la connexion de plusieurs clients simultanément, vous êtes obligé de faire une gestion multi-tâches, c'est-à-dire de mettre en place un **« Thread »** pour chaque client connecté, ce que nous avons appris à faire dans l'un des chapitres précédents. Le principe est que lorsqu'un client se connecte, le thread de traitement pour ce client est exécuté tout de suite et le service attend de nouveau qu'un autre client se connecte, et ceci indéfiniment.

Cette fois-ci le projet précédent est séparé en deux scripts, le premier ***« client.py »*** *qui gère le traitement d'un client, le second* ***« service.py »*** *étant la version modifiée de l'application précédente.*

client.py

```
from socket import *
from threading import Thread
```

```
class Client(Thread):
   def __init__(self, connexion, adresse):
      super().__init__()
      self.connexion = connexion
      self.adresse = adresse

   def run(self):
      print('Nouveau client : ', self.adresse)
      self.connexion.send('Bonjour, tapez votre texte, ("fin" pour quitter)\n'.encode('utf8'))
      while True:
         try:
            message = self.connexion.recv(1024).decode('utf8').strip()
            if message.lower() == 'fin': break
            print(self.adresse,'>', message)
            self.connexion.send(('Service > '+message.upper()+'\n').encode('utf8'))
         except:
            print(self.adresse, 'Impossible de décoder correctement le message')
            break
      self.connexion.close()
      print('Client', self.adresse, 'déconnecté')
```

service.py

```
from socket import *
from sys import exit
from client import Client

hôte = ''
port = 5588
service = socket()

# Mise en place du service
try: service.bind((hôte, port))
except:
   print('Ce service est déjà en exécution')
   exit()

# Écoute du réseau et attente d'une demande de connexion des clients pour ce service
service.listen()
print('Service en fonctionnement')

# Établissement des connexions et dialogues avec plusieurs clients
while True:
   connexion, adresse = service.accept()
   Client(connexion, adresse).start()
```

Shell Python (Résultat)

```
Service en fonctionnement
Nouveau client :  ('192.168.1.15', 58196)
('192.168.1.15', 58196) > bienvenue
Nouveau client :  ('192.168.1.21', 40512)
('192.168.1.21', 40512) > salut à tout le monde
('192.168.1.15', 58196) > bonjour, tu es là
('192.168.1.21', 40512) > eh oui!
Client ('192.168.1.21', 40512) déconnecté
Client ('192.168.1.15', 58196) déconnecté
Nouveau client :  ('192.168.1.15', 58418)
Client ('192.168.1.15', 58418) déconnecté # ce client ne possède pas le même port que précédemment
```

*Le service est maintenant beaucoup plus réduit puisqu'il ne dispose plus du traitement spécifique au protocole à suivre avec le client. Cette dernière partie est réalisée par la tâche **« Client »**.*

MISE EN ŒUVRE D'UN SERVICE « TCHAT »

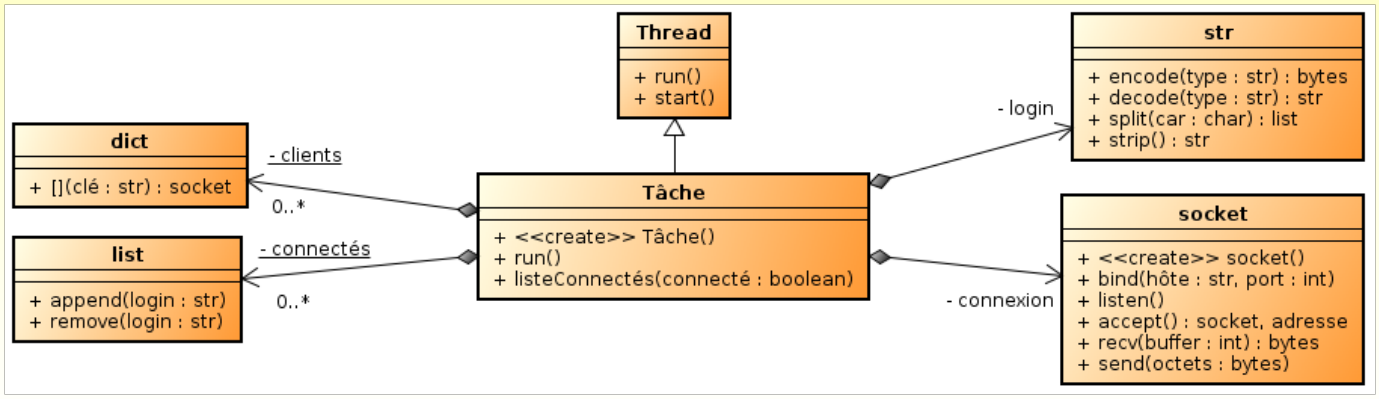
Afin de valider les différentes compétences que nous venons d'acquérir dans le domaine du réseau, je vous propose de fabriquer un service **« tchat »**. Je rappelle que ce service permet de mettre en relation plusieurs clients afin qu'ils puissent communiquer entre eux et s'envoyer des messages en **« temps réel »**.

Pour qu'un client puisse envoyer un message à un autre client, il faut que ces deux clients soient connectés au service, ce qui sous-entend que toutes les connexions sont délivrées par le service et que tous les messages transitent par le service.

*Un client reste connecté autant que nécessaire, nous devons donc prévoir un système multitâches. L'identification de chaque client se fait simplement au travers d'un **« login »** unique, qui doit être envoyé dès que le client se connecte.*

*Lorsque un client soumet un message, il doit être précédé du **« login »** du destinataire, avec un caractère de séparation **« @ »**.*

*Lorsque le client décide de se déconnecté du service, l'application devra envoyer le message **« stop »**, précédé de son nom de **« login »**, avant sa déconnexion effective.*



tâche.py

```
from socket import *
from threading import Thread

class Tâche(Thread):
   # Attributs de classe
   clients = {}      # Dictionnaire avec login et connexion de chaque client connecté
   connectés = []    # Liste de tous les clients connectés

   def __init__(self, client):
      super().__init__()
      self.connexion, adresse = client
      self.login = self.connexion.recv(30).decode('utf8').strip()
      Tâche.clients[self.login] = self.connexion
      Tâche.connectés.append(self.login)
      self.connexion.send(b'Bienvenue au service "tchat"\n')
      self.listeConnectés(True)

   def run(self):
      while True:
         réception = self.connexion.recv(1024).decode('utf8').strip().split('@')
         destinataire = réception[0]
         message = réception[1]
         if message == 'stop': break
         texte = self.login + ' > ' + message + '\n'
         Tâche.clients[destinataire].send(texte.encode('utf8'))
      self.connexion.send(b'stop\n') # Utile pour l'application cliente fenêtrée
      self.connexion.close()
      del Tâche.clients[self.login]
      Tâche.connectés.remove(self.login)
      self.listeConnectés(False)

   # Tous les clients sont au courant de ceux qui sont connectés
   def listeConnectés(self, connecté):
      message = 'Connectés : {}\n'.format(Tâche.connectés)
      if connecté: état = self.login + ' vient de se connecter\n'
      else:        état = self.login + ' vient de se déconnecter\n'
      for connexion in Tâche.clients.values():
         connexion.send(état.encode('utf8'))
         connexion.send(message.encode('utf8'))
```

service.py

```
from socket import *
from sys import exit
from tâche import Tâche

hôte = ''
port = 7777
service = socket()

# Mise en place du service
try: service.bind((hôte, port))
except:
   print('Ce service est déjà en exécution')
   exit()
# Écoute du réseau et attente d'une demande de connexion des clients pour ce service
service.listen()
print('Service en fonctionnement')
```

```
# Établissement des connexions et dialogues avec plusieurs clients
while True:
    client = service.accept()
    Tâche(client).start()
```

```
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
manu@manu-N550JV ~ $ telnet 172.16.20.31 7777
Trying 172.16.20.31...
Connected to 172.16.20.31.
Escape character is '^]'.
manu
Bienvenue au service "tchat"
manu vient de se connecter
Connectés : ['bruno', 'manu']
bruno@salut à toi
bruno > bonjour à toi aussi
bruno vient de se déconnecter
Connectés : ['manu']
manu@stop
Connection closed by foreign host.
manu@manu-N550JV ~ $
```

*Je vous invite à tester ce service en simulant deux clients, sur des postes différents, au travers de **« telnet »**, comme cela vous est montré ci-contre. Remarquez bien, la phase de connexion et de déconnexion, ainsi que la syntaxe à utiliser pour envoyer des messages à son interlocuteur.*

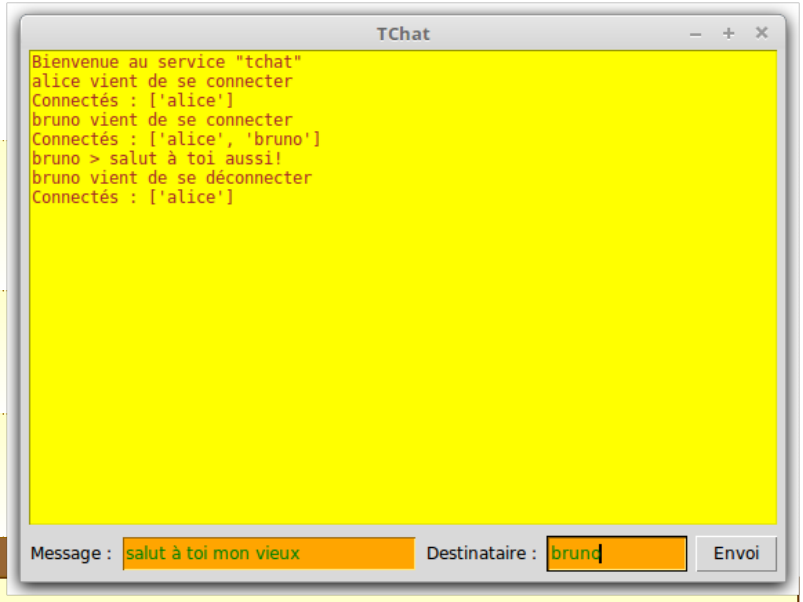
À chaque phase de connexion ou de déconnexion, tous les utilisateurs en connexion sont au courant de ceux qui restent.

*Pour l'instant, toutes les communications ne se font qu'en réseau local et ne peut pas se faire par **« Internet »**. Il faudrait pour cela utiliser d'autres technologies, notamment les **« websockets »**, se qui dépasse ici le cadre de notre étude .*

*Au niveau du code de la classe **« Tâche »**, vous remarquez la présence de chaînes particulières qui débute par la lettre **« b »**, comme : **b'Bienvenue au service "tchat"\n'** ou **b'stop\n'** : . Cette syntaxe nous permet de spécifier directement un message, non pas de type **« str »**, mais de type **« bytes »** (d'où la lettre **« b »**) en utilisant le format **« utf8 »**. Cela nous évite d'utiliser systématiquement la méthode **« encode('utf8') »**.*

CLIENT QUI COMMUNIQUE AVEC LE SERVICE « TCHAT »

Après avoir bien tester notre service au travers de clients **« telnet »**, je vous propose de réaliser notre propre application cliente, dédiée à la communication simultané entre plusieurs intervenants, en respectant le protocole que nous avons mis en œuvre dans le service.



*Notre application cliente sera décrite dans un seul script, nommé **« clientchat.py »** avec également un deuxième fichier de configuration **« config.ini »** qui nous permet de choisir l'adresse du service avec son numéro de port ainsi que l'identification du client.*

Chaque application cliente, sur un poste séparé, doit posséder son propre fichier de configuration, notamment pour l'identification de la personne qui travaille sur ce poste.

L'avantage d'avoir un fichier de configuration est qu'il est très facile de changer les valeurs à l'aide d'un simple éditeur de texte, dans le cas par exemple où le service doit être déplacé sur une autre machine.

config.ini

```
[chat]
adresse=192.168.1.15
port=7777
login=alice
```

clientchat.py

```
from tkinter import *
from socket import *
from threading import Thread
from configparser import ConfigParser

# Tâche qui attend tous les messages venant des autres clients par le service
class Réception(Thread):
    def __init__(self, connexion, réponse):
        super().__init__()
        self.connexion = connexion
        self.réponse = réponse

    def run(self):
        while True:
            message = self.connexion.recv(1024).decode('utf8').strip()
            if message == 'stop': break
            self.réponse.insert(END, message+'\n')
```

```
# Classe principale de l'application avec les différents "widgets"
class Fenêtre(Tk):
   def __init__(self):
      super().__init__()
      self.title('TChat')
      self.geometry('600x480')
      self.placerWidgets()
      self.seConnecter()
      self.mainloop()

   def placerWidgets(self):
      self.réponse = Text(self, bg='yellow', fg='brown')
      self.réponse.pack(fill=BOTH, expand=TRUE, padx=5)
      Label(self, text='Message :').pack(side=LEFT, padx=5)
      self.message = Entry(self, bg='orange', fg='green')
      self.message.pack(side=LEFT, fill=X, expand=TRUE, ipady=3, pady=7)
      Label(self, text='Destinataire :').pack(side=LEFT, padx=5)
      self.destinataire = Entry(self, bg='orange', fg='green', width=12)
      self.destinataire.pack(side=LEFT, ipady=3, pady=7)
      Button(self, text='Envoi', command=self.envoyer).pack(side=RIGHT, padx=5)

   def envoyer(self):
      protocole = self.destinataire.get()+'@'+self.message.get()
      self.service.send(protocole.encode('utf8'))

   def seConnecter(self):
      config = ConfigParser()
      config.read('config.ini')
      hôte = config.get('chat', 'adresse')
      port = config.getint('chat', 'port')
      self.login = config.get('chat', 'login')
      self.service = socket()
      self.service.connect((hôte, port))
      self.service.send(self.login.encode('utf8'))
      Réception(self.service, self.réponse).start()

   def destroy(self):
      protocole = self.login+'@stop'
      self.service.send(protocole.encode('utf8'))
      super().destroy()
if __name__ == '__main__' : Fenêtre()
```

Dès que les différents composants constituant la fenêtre sont placés à l'intérieur, l'application demande tout de suite à se connecter au service en tenant compte du fichier de configuration.

*Le fichier de configuration proposé est de type **« INI »** dont l'architecture est relativement simple. Il possède généralement un ensemble de sections dont le nom est écrit entre crochets **« [ ] »**, chaque section propose ensuite un ensemble de **« clé/valeur »** séparé par l'opérateur **« = »**.*

Il existe une classe spécifique « ConfigParser » qui permet de lire et de récupérer les différentes « valeurs » associées aux « clés », placées dans une rubrique spécifique nommée « section ». Plusieurs méthodes existent, mais deux méthodes sont particulièrement intéressantes pour la récupération de toutes ces informations de configuration :

- ***read()*** *: cette méthode permet de charger en mémoire le contenu du fichier de configuration proposé en argument.*
- ***get(), getint(), getfloat(), getboolean()*** *: méthodes qui nous permet de récupérer la valeur enregistrée dans une **« section »** pour une **« clé »** spécifique proposées en argument de la méthode. Le choix de la méthode permet de récupérer la valeur suivant le type désiré, respectivement : chaîne de caractères, entier, réel et booléen.*
- *Pour ces deux types de méthodes, vos arguments seront systématiquement écrit sous forme de chaînes de caractères, respectivement pour le **« nom du fichier »**, pour le **« nom de la section »** et enfin pour le **« nom de la clé »**.*

SERVICE DE STOCKAGE DE PHOTOS

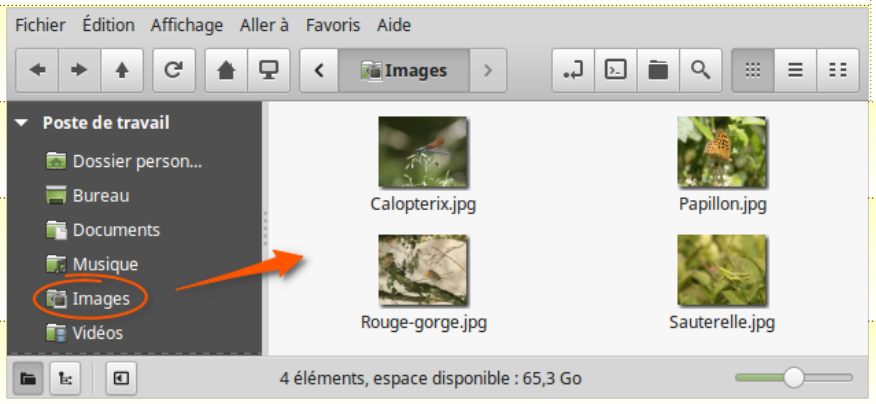
Nous concluons notre étude par un projet un peu plus conséquent qui consiste à archiver nos photos sur un serveur afin de pouvoir les récupérer par la suite. Nous nous intéressons ici uniquement à la partie service. Nous traiterons la partie cliente dans le prochain chapitre.

Le point important entre les deux systèmes, pour que les différentes applications puissent échanger des informations convenablement, il est nécessaire de définir un protocole de communication. Au préalable, nous devons prévoir toutes les possibilités offertes par le service afin de bien spécifier les échanges attendus par la suite.

*Si nous regardons d'un peu plus près, il s'agit en réalité d'un simple transfert de fichiers avec des tailles qui peuvent être relativement conséquentes. Au travers du réseau, pour de gros fichiers, il est plus judicieux et surtout plus efficace d'effectuer le transfert de chaque fichier par paquet de **1024** octets.*

Dans la conception du service, nous devons choisir un répertoire particulier sur la machine hôte qui va servir de stockage. Pourquoi pas d'ailleurs le répertoire ***« Images »*** *déjà prévu pour cela.*

Nous devons définir les différentes possibilités de gestion de ce service. Quatre fonctionnalités apparaissent :

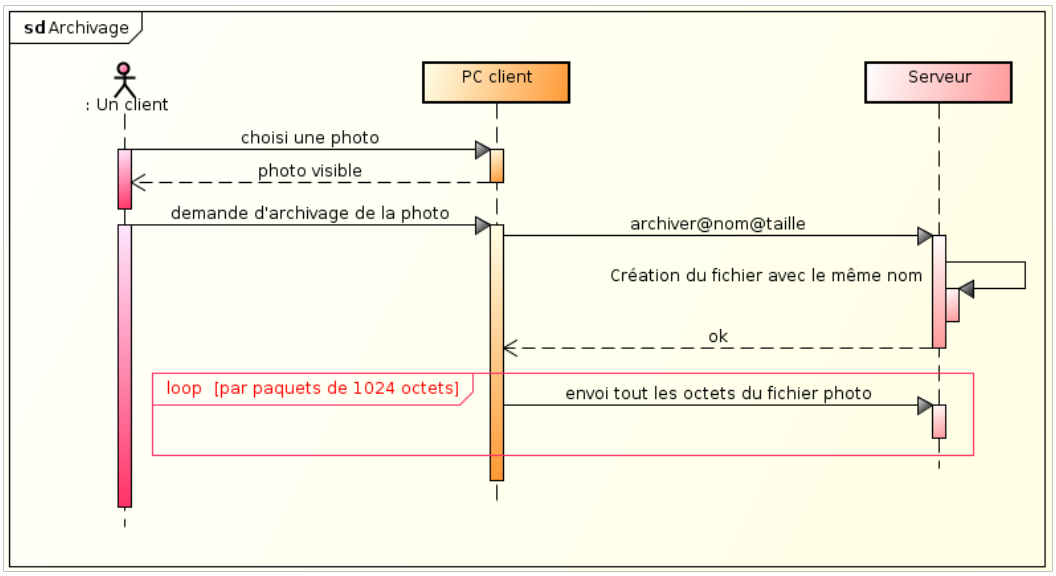


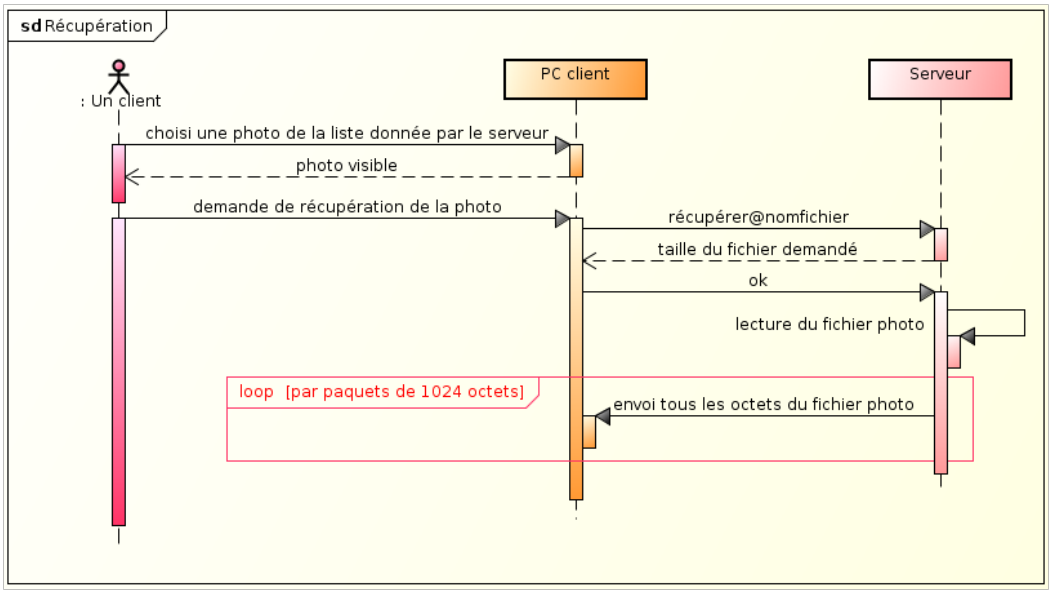
- ***l'archivage*** *qui consiste à stocker dans ce répertoire spécifique une photo envoyée par un client.*
- *La* ***récupération*** *qui effectue l'opération inverse de l'archivage, c'est-à-dire qui permet de restituer une photo stockée vers le client qui désire la récupérée.*
- *La* ***suppression*** *d'une photo dans le répertoire de stockage afin de libérer de l'espace pour d'autres photos.*
- *Enfin, obtenir la* ***liste*** *des photos actuellement archivées, afin de pouvoir choisir celle que nous devons restituer.*

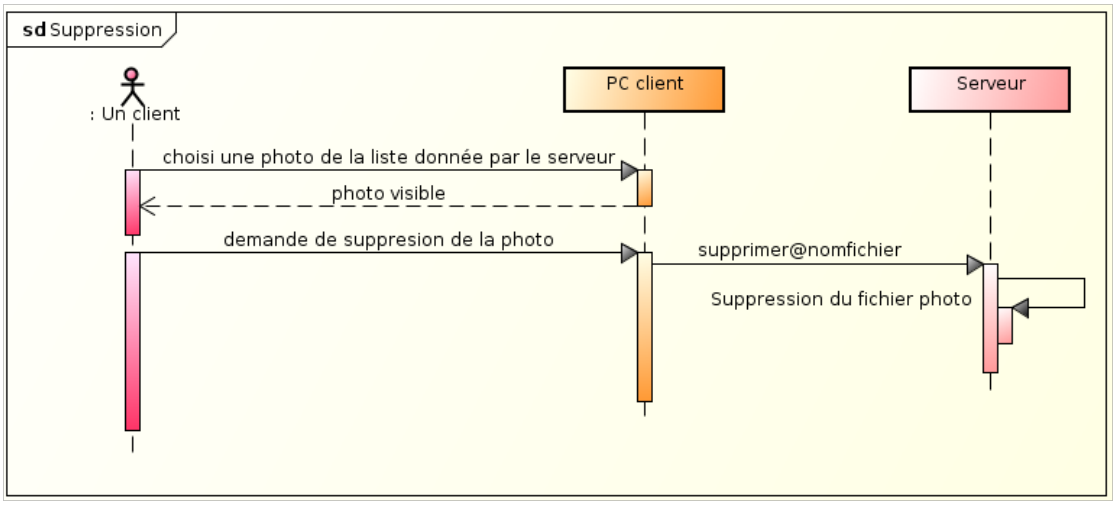
Vu que le service peut réaliser plusieurs tâches différentes, il faut imaginer un ***protocole*** *qui permet de savoir à tout moment ce que le client désire réellement. Nous allons rester simple afin que les codes respectifs soient le plus réduit possible.*

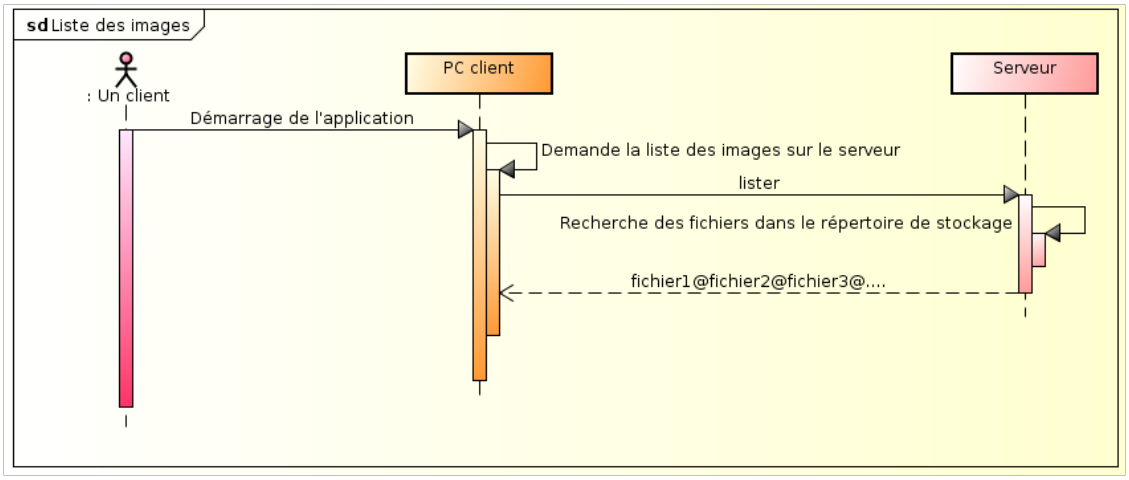
L'échange doit se faire dans un premier temps sous forme d'une chaîne de caractères dans laquelle nous allons préciser la commande souhaitée, suivi éventuellement du nom du fichier, et suivi éventuellement de la taille du fichier, chaque information séparée par le caractère « @ », comme suit : commande@nomfichier@taillefichier.

La suite du protocole dépend de la commande, parmi celle évoquée (« archiver », « récupérer », « supprimer » et « lister »). Vous avez ci-dessous une solution des différents protocoles imaginés pour répondre à ces quatre fonctionnalités spécifiques, suivi des scripts python correspondant.









service.py

```
from socket import *
from sys import exit
from threadservice import Tâche

hôte = ''
port = 7788
service = socket()

# Mise en place du service
try: service.bind((hôte, port))
except:
   print('Ce service est déjà en exécution')
   exit()

# Écoute du réseau et attente d'une demande de connexion des clients pour ce service
service.listen()
print('Service en fonctionnement')

# Établissement des connexions et dialogues avec plusieurs clients
while True:
   client = service.accept()
   Tâche(client).start()
```

Mise à part une importation, ce script est totalement similaire à ce que nous avons déjà construit au préalable.

threadservice.py

```
from socket import *
from threading import Thread
import os
répertoire = '/home/manu/Images'

class Tâche(Thread):
   def __init__(self, client):
      super().__init__()
      self.connexion, adresse = client
      os.chdir(répertoire)
```

```
    def run(self):
        réception = self.connexion.recv(1024).decode('utf8').strip().split('@')
        if réception[0]=='archiver': self.archiver(réception[1], int(réception[2]))
        elif réception[0]=='récupérer': self.récupérer(réception[1])
        elif réception[0]=='supprimer': os.remove(réception[1])
        else:
            liste = '@'.join(os.listdir())
            self.connexion.send(liste.encode('utf8'))
        self.connexion.close()

    def archiver(self, nomfichier, taille):
        self.connexion.send(b'ok')
        with open(nomfichier, 'wb') as fichier:
            while taille>0:
                octets = self.connexion.recv(1024)
                fichier.write(octets)
                taille-=1024
            fichier.close()

    def récupérer(self, nomfichier):
        taille = os.path.getsize(nomfichier)
        self.connexion.send('{}'.format(taille).encode('utf8'))
        réponse = self.connexion.recv(10).decode('utf8')
        if réponse=='ok':
            with open(nomfichier, 'rb') as fichier:
                while taille>1024:
                    octets = fichier.read(1024)
                    self.connexion.send(octets)
                if taille>0:
                    octets = fichier.read(taille)
                    self.connexion.send(octets)
                fichier.close()
```

Comme pour les projets antérieurs, nous prévoyons un thread pour chaque client connecté. La durée de la connexion cette fois-ci est relativement courte puisque la tâche prend en compte qu'une seule requête et le service se déconnecte automatiquement dès que la réponse est fournie au client. Cela permet d'utiliser le minimum de ressource réseau. Cela ne sert à rien de rester connecter si plus rien ne se passe par ailleurs.

*La variable **« répertoire »** est une variable globale qui correspond, on l'imagine, au répertoire de stockage de l'ensemble des photos à archiver. Dans le constructeur, nous demandons à ce que cela devienne le répertoire par défaut.*

*Nous trouvons dans la méthode **« run() »** (appelée automatiquement au démarrage de la tâche) l'algorithme qui permet de récupérer la commande souhaitée par le client suivant le protocole que nous venons de définir.*

*Pour les commandes qui réclament pas mal de traitement, nous avons conçus des méthodes spécifiques pour cela. C'est notamment le cas pour les méthodes **« archiver() »** et **« récupérer() »**, ce qui est logique puisque c'est au travers de celles-ci que nous transférons, d'un côté ou de l'autre, l'ensemble des octets correspondant à des fichiers **« Photo »**.*

globalement, nous avons très peu de lignes de code pour traiter toute cette problématique.

APPLICATION CLIENTE EN RELATION AVEC CE SERVICE DE STOCKAGE DE PHOTOS

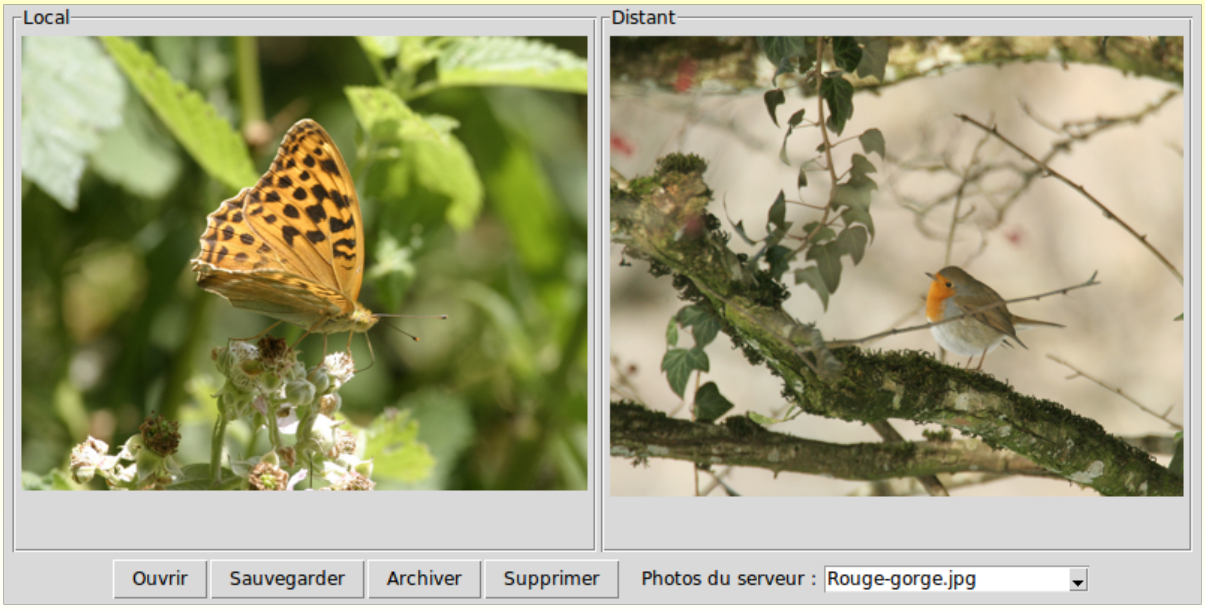
La partie cliente est beaucoup plus étoffée. Elle doit certes s'occuper de la communication avec le service en respectant les différents protocoles que nous venons de définir pour que les échanges puissent se faire correctement, mais elle doit aussi prévoir l'affichage des photos qui se trouvent sur le poste local, avec en plus les photos distantes que le service possède actuellement. Tout ceci complique relativement la tâche du développeur.

*Le projet client est scindé en trois parties. Le premier fichier **« config.ini »** est le fichier de configuration. Le deuxième fichier **« client.py »** s'occupe de la fenêtre principale avec les différents composants graphiques la constituant. Le dernier script **« vignette.py »** correspond à la gestion de l'affichage d'une photo. C'est un script sur lequel nous avons déjà travaillé, mais que nous devons compléter pour répondre à cette application là.*

La fenêtre est relativement sobre au niveau de sa constitution. Deux zones principales pour afficher les deux types de photo, locales ou distantes. Sur la partie basse, nous avons une barre de boutons qui vont nous permettre de choisir les actions à mener à la fois en local, mais aussi avec le service distant.

*Dans la barre de boutons, nous avons une **« Combobox »** qui recense l'ensemble des photos actuellement disponibles sur le serveur. Cette information, nous l'avons vu lors de la description des protocoles, est automatiquement connue dès le démarrage de l'application.*

Cette « Combobox » n'existe pas en natif de « tkinter ». Pour l'obtenir, vous devez faire l'importation du module suivant : « tkinter.ttk »



Au niveau de la conception de l'interface, je me suis servi d'un composant ***« PanedWindow »*** *afin de répartir les deux vignettes (locale et distante) équitablement. J'ai également, par soucis de clarté, décoré les vignettes avec un composant de type* ***« LabelFrame »*** *afin de proposer une étiquette adaptée. Le reste du code est relativement bien documenté.*

Les deux premiers boutons ***« Ouvrir »*** *et* ***« Sauvegarder »*** *concernent la lecture d'une photo présente sur le disque dur local (de l'hôte) et le stockage d'une photo récupérée du serveur pour l'enregistrer également sur le disque dur local de l'hôte.*

Pour afficher une photo distante, il suffit de la choisir à l'aide de la ***« Combobox ».*** *Elle alors automatiquement récupérée du service distant et affichée dans sa zone prévue à cet effet.*

config.ini

```
[archivage]
adresse=172.16.20.31
port=7788
```

client.py

```
from tkinter import *
from tkinter.ttk import Combobox
from tkinter.messagebox import *
from configparser import ConfigParser
from vignette import Vignette
from socket import *

# Fenêtre principale de l'application
class Fenêtre(Tk):
   def __init__(self, titre):
      super().__init__()
      self.title(titre)
      self.geometry('800x400')
      self.localisation()
      self.disposition()
      self.lister()                                  # Récupère la liste des photos présentes dans le serveur
      self.resizable(width=False, height=False)      # Empêche le redimensionnement de la fenêtre
      self.mainloop()

   # Localisation du service avec son adresse IP et son numéro de port
   def localisation(self):
      config = ConfigParser()
      config.read('config.ini')
      self.service = (config.get('archivage', 'adresse'), config.getint('archivage', 'port'))

   # Placement des différents composants constituant l'IHM
   def disposition(self):
      panneaux = PanedWindow(self)                   # Partage l'intérieur de la fenêtre en deux parties
      panneaux.pack(fill=BOTH, expand=YES, padx=5)
      panneaulocal = LabelFrame(self, text='Local')   # Création de panneaux avec étiquettes
      panneaudistant = LabelFrame(self, text='Distant')
      self.local = Vignette(panneaulocal, self.service)
      self.distant = Vignette(panneaudistant, self.service)
      panneaux.add(panneaulocal)
      panneaux.add(panneaudistant)
      self.panneauBoutons()
```

```
   # Mise en place des différents boutons pour proposer des actions spécifiques
   def panneauBoutons(self):
      boutons = Frame(self)
      boutons.pack(padx=5, pady=3)
      Button(boutons, text='Ouvrir', command=self.local.ouvrir).pack(side=LEFT)
      Button(boutons, text='Sauvegarder', command=self.distant.sauvegarder).pack(side=LEFT)
      Button(boutons, text='Archiver', command=self.archiver).pack(side=LEFT)
      Button(boutons, text='Supprimer', command=self.supprimer).pack(side=LEFT)
      Label(boutons, text='   Photos du serveur :').pack(side=LEFT)
      self.photos = Combobox(boutons)
      self.photos.bind('<<ComboboxSelected>>', self.récupérer)
      self.photos.pack(side=LEFT, padx=3)

   # Quatre méthodes qui respectent le protocole du service soit directement, soit par les vignettes
   def lister(self):
      connexion = socket()
      connexion.connect(self.service)
      connexion.send(b'lister')
      réponse = connexion.recv(1024).decode('utf8').split('@')
      self.photos['values'] = réponse

   def archiver(self):
      self.local.archiver()
      self.lister()

   def supprimer(self):
      self.distant.supprimer(self.photos.get())
      self.lister()

   def récupérer(self, event): self.distant.récupérer(self.photos.get())

# Programme principal
if __name__ == '__main__' :
   programme = Fenêtre('Visionneuse')
```

vignette.py

```
from tkinter import *
from tkinter.filedialog import *
from PIL import Image, ImageTk
from socket import *
import os
import io


# Création d'un vignette à partir d'un Canvas
class Vignette(Canvas):
   def __init__(self, fenêtre, service):
      super().__init__(fenêtre)
      self.service = service
      self.pack(fill=BOTH, expand=1, padx=3, pady=3)
      self.fichier=''

   def ouvrir(self):
      self.fichier = askopenfilename()
      if not len(self.fichier) == 0:
         image = Image.open(self.fichier)
         image.thumbnail((self.winfo_width(), self.winfo_height()))
         self.photo = ImageTk.PhotoImage(image)
         self.create_image(0, 0, image=self.photo, anchor='nw')

   def sauvegarder(self):
      nomfichier = asksaveasfilename(filetypes=[('Photos', '.jpg')], defaultextension='.jpg')
      with open(nomfichier, 'wb') as fichier:
         fichier.write(self.sauvegarde)
         fichier.close()

   def archiver(self):
      if not len(self.fichier) == 0:
         répertoire, nomfichier = os.path.split(self.fichier)
         taille = os.path.getsize(self.fichier)
         requête = 'archiver@{}@{}'.format(nomfichier, taille)
         connexion = socket()
         connexion.connect(self.service)
         connexion.send(requête.encode('utf8'))
         réponse = connexion.recv(10).decode('utf8')
```

```
        if réponse=='ok':
            with open(self.fichier, 'rb') as fichier:
                while taille>1024:
                    octets = fichier.read(1024)
                    connexion.send(octets)
                    taille-=1024

                if taille>0:
                    octets = fichier.read(taille)
                    connexion.send(octets)
                fichier.close()

    def récupérer(self, nomfichier):
        self.fichier = nomfichier
        requête = 'récupérer@'+nomfichier
        connexion = socket()
        connexion.connect(self.service)
        connexion.send(requête.encode('utf8'))
        taille = int(connexion.recv(10).decode('utf8'))
        connexion.send(b'ok')
        octets = io.BytesIO()
        while taille>0:
            octets.write(connexion.recv(1024))
            taille-=1024
        self.sauvegarde = octets.getvalue()
        image = Image.open(octets)
        image.thumbnail((self.winfo_width(), self.winfo_height()))
        self.photo = ImageTk.PhotoImage(image)
        self.create_image(0, 0, image=self.photo, anchor='nw')
        connexion.close()

    def supprimer(self, nomfichier):
        requête = 'supprimer@'+nomfichier
        connexion = socket()
        connexion.connect(self.service)
        connexion.send(requête.encode('utf8'))
```

Dans cette classe, nous retrouvons les méthodes qui gèrent l'ensemble des actions proposées par l'IHM, pour le poste local, et pour les commandes associées au service tout en respectant le protocole défini précédemment.