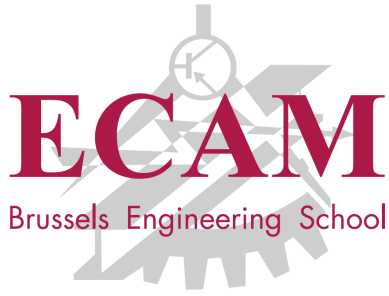


PI2T Développement informatique

Séance 2

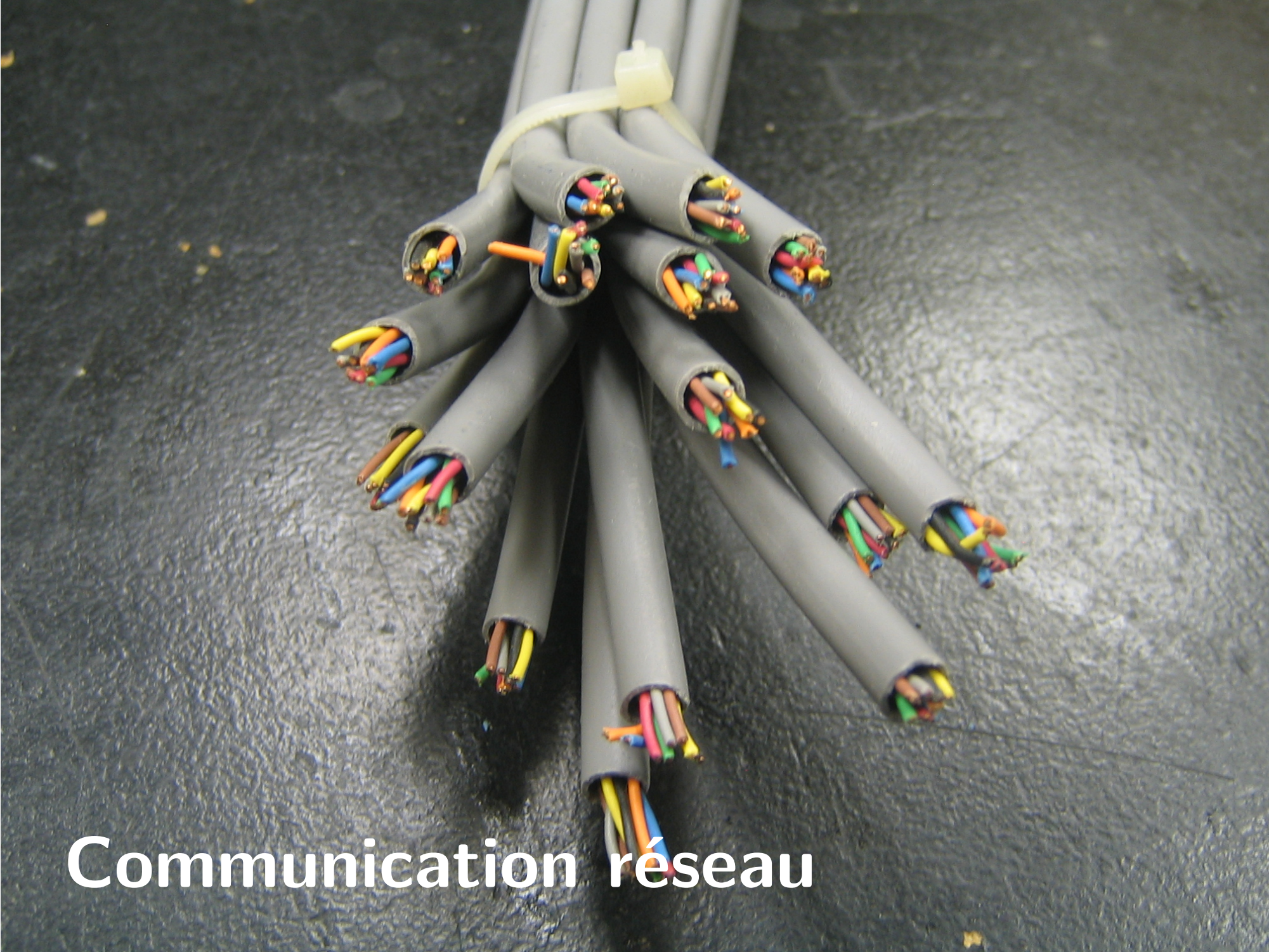
Programmation réseau

Sébastien Combéfis, Quentin Lurkin

16 février 2016

Objectifs

- Comprendre les principes de la communication par le réseau
  - Modes et caractéristiques des communications
  - Architecture peer-to-peer et client/serveur
  - Protocoles TCP et UDP
- Programmation réseau en Python
  - Le module `socket`
  - Entrée/sortie sur l'interface réseau
  - Protocole de communication



Communication réseau

Réseau

- Communication entre différentes machines

  À travers un réseau informatique

- Communication entre différents programmes

  Qui peuvent être sur la même machine ou sur des différentes

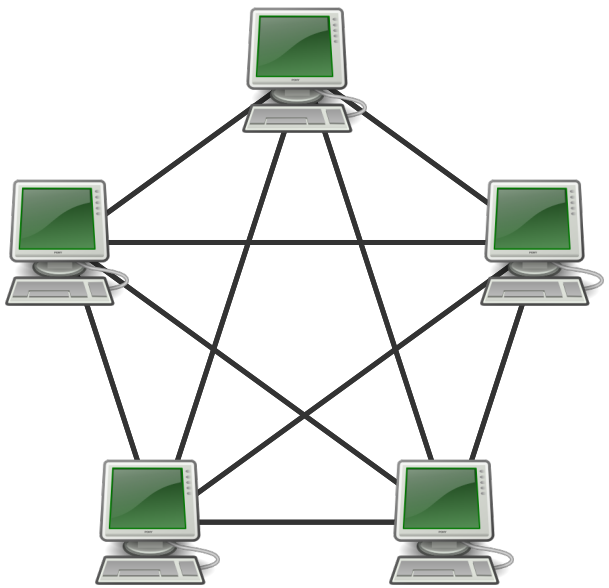
- Différents objectifs possibles
  - Commande ou log à distance
  - Fournir ou récupérer des données
  - Offrir des services (HTTP, SMTP...)

Architecture peer-to-peer

- Le même programme est exécuté sur plusieurs machines

  Napster, Usenet, Gnutella, Bittorrent, Spotify, Bitcoin...

- Toutes les machines peuvent être interconnectées

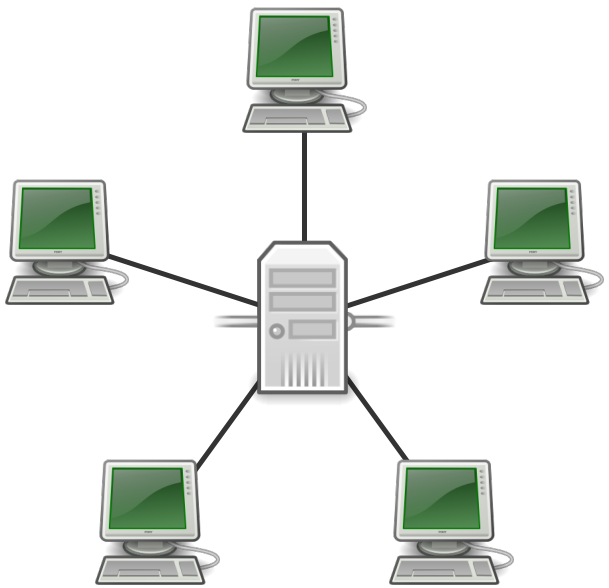


Architecture client/serveur

- Un client envoie des requêtes à un serveur

  HTTP, FTP, serveur d'impression...

- Toutes les machines clientes se connectent au serveur



Identification

- Chaque machine possède une adresse IP

  Identification des machines sur le réseau

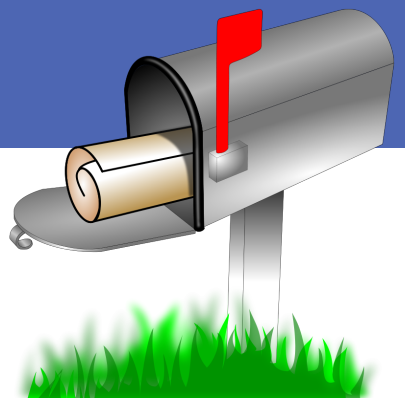
- Chaque connexion est associée à un numéro de port

  Identification des programmes sur la machine

IP 1.2.3.4
Port 2536

IP 5.6.7.8
Port 80

Protocole UDP



- Communication par échange de datagrammes

  Paquets discrets de données

- Caractéristiques du User Datagram Protocol
  - Transfert non fiable (réception et ordre non garantis)
  - Sans connexion (protocole léger)
  - Adapté à l'architecture peer-to-peer
- Identification avec une adresse UDP $\langle IP, Port \rangle$

Protocole TCP

- Communication par flux (stream)

  Flux continu de données

- Caractéristiques du Transmission Control Protocol
  - Transfert fiable (réception et ordre garantis)
  - Avec connexion (protocole lourd)
  - Adapté à l'architecture client/serveur
- Identification avec une adresse TCP $\langle IP, Port \rangle$

Protocole de communication

- Spécification des messages échangés entre les machines

  Ensemble des messages valides, avec leur structure

- Accord sur le protocole de communication à utiliser

  Une machine peut couper la communication en cas d'erreur

- Deux modes de communication possibles
  - Blocs binaires (module `pickle`)
  - Séquences de caractères (encodage/décodage de caractères)


Socket

Socket réseau

- Bout d'une communication inter-processus à travers un réseau

  Socket Internet basés sur le protocole IP

- Plusieurs types de sockets

  Par exemple, datagramme (datagram) et flux (stream)

- Adresse d'un socket composée d'une adresse IP et d'un port

- Utilisation du module `socket` en Python

Nom d'hôte

- Un nom d'hôte permet d'identifier une machine

  Obtenable par des fonctions du module `socket`

```
1  print(socket.getfqdn('www.google.be'))          # Fully Qualified Domain Name
2  print(socket.gethostname())                     # Nom d'hôte de la machine
3
4  print(socket.gethostbyname('www.google.be'))    # Hôte à partir du nom
5  print(socket.gethostbyaddr('213.186.33.2'))     # Hôte à partir de l'adresse
```

```
wa-in-f94.1e100.net
MacBook-Pro-de-Sebastien-3.local

64.233.184.94
('cluster002.ovh.net', ['2.33.186.213.in-addr.arpa'],
['213.186.33.2'])
```

Création d'un `socket`

- Socket représenté par un objet de la classe `socket.socket`

  Même classe qui gère tous les types de socket

- Deux paramètres essentiels
  - Famille des adresses (`family`) (`AF_INET` par défaut)
  - Type de socket (`type`) (`SOCK_STREAM` par défaut)

```
1  s = socket.socket()
2  print(s.getsockname())
3
4  t = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
5  print(t.getsockname())
```

```
('0.0.0.0', 0)
('::', 0, 0, 0)
```

Connexion

- Connexion à une machine avec la méthode `connect`

  Le format de l'adresse de connexion dépend de la famille

- Adresse socket IPv4 formée du nom de l'hôte et du port

  Adresse représentée par un tuple à deux éléments

- Adresse du socket de la forme ⟨`socket.gethostname()`, XXX⟩

```
1 s.connect(('www.python.org', 80))
2 print(s.getsockname())
```

```
('192.168.1.3', 61774)
```

Binding

- Attacher le socket avec la méthode `bind`

  Associer un socket à une adresse spécifiée

- Adresse socket IPv4 formée du nom de l'hôte et du port

  Adresse représentée par un tuple à deux éléments

```
1 s.bind((socket.gethostname(), 6000))
2 print(s.getsockname())
```

```
('192.168.1.3', 6000)
```

Fermeture de la connexion

- Une fois la communication terminée, il faut fermer le socket

  Permet de libérer les ressources allouées par le système d'exploitation

- Utilisation de la méthode `close`

  Doit être fait pour chaque socket ouvert

```
1 s.close()
```


Application chat

Envoi de données (UDP)

- Envoi de données via un socket avec la méthode `sendto`

  Permet d'envoyer des octets à un destinataire

- Les données envoyées doivent être au format binaire

  Conversion d'une chaine de caractères avec `encode`

- Il faut vérifier que l'envoi est bien complet

  La valeur de retour est le nombre d'octets effectivement envoyés

```
1  s = socket.socket(type=socket.SOCK_DGRAM)
2
3  data = "Hello World!".encode()
4  sent = s.sendto(data, ('localhost', 5000))
5  if sent == len(data):
6      print("Envoi complet")
```

Boucle d'envoi (UDP)

- Envoi des données par paquets à l'aide d'une boucle

  Utilisation de la taille totale du message

```
1  s = socket.socket(type=socket.SOCK_DGRAM)
2
3  address = ('localhost', 5000)
4  message = "Hello World!".encode()
5
6  totalsent = 0
7  while totalsent < len(message):
8      sent = s.sendto(message[totalsent:], address)
9      totalsent += sent
```

Réception de données (UDP)

- Réception de données via un socket avec la méthode `recv`

  Permet de recevoir des octets envoyés par l'autre bout du socket

- Il faut spécifier le nombre maximal d'octets à recevoir

  Préférable d'utiliser une petite valeur puissance de 2 comme 4096

- Liste vide renvoyée s'il n'y a rien à lire sur le socket

```
1  s = socket.socket(type=socket.SOCK_DGRAM)
2  s.bind((socket.gethostname(), 5000))
3
4  data = s.recvfrom(512).decode()
5  print('Reçu', len(data), 'octets :')
6  print(data)
```

Création d'une classe Chat

- Initialisation du socket dans le constructeur

  Mise en place d'un timeout pour que `recvfrom` soit non bloquant

- Plusieurs variables d'instance
  - `__s` : le socket
  - `__running` : booléen indiquant si le programme tourne
  - `__address` : adresse du destinataire

```
1  class Chat():
2      def __init__(self, host=socket.gethostname(), port=5000):
3          s = socket.socket(type=socket.SOCK_DGRAM)
4          s.settimeout(0.5)
5          s.bind((host, port))
6          self.__s = s
```

Méthodes pour les commandes

- Une méthode pour chaque commande disponible

```
1  def _exit(self):
2      self.__running = False
3      self.__address = None
4      self.__s.close()
5
6  def _quit(self):
7      self.__address = None
8
9  def _join(self, param):
10     tokens = param.split(' ')
11     if len(tokens) == 2:
12         self.__address = (socket.gethostbyaddr(tokens[0])[0], int(tokens[1]))
13
14 def _send(self, param):
15     if self.__address is not None:
16         message = param.encode()
17         totalsent = 0
18         while totalsent < len(message):
19             sent = self.__s.sendto(message[totalsent:], self.__address)
20             totalsent += sent
```

Méthode pour recevoir des messages

- La méthode `_receive` est une boucle qui attend des messages

  Réception active uniquement lorsque le programme tourne

- Exception `socket.timeout` pour éviter l'appel bloquant

  Permet d'arrêter la réception lorsque le programme est quitté

```
1  def _receive(self):
2      while self.__running:
3          try:
4              data, address = self.__s.recvfrom(1024)
5              print(data.decode())
6          except socket.timeout:
7              pass
```

Démarrage de l'écoute

- Dictionnaire des commandes disponibles

  Associe le nom de la commande avec la méthode qui la gère

- Démarrage de la réception de données

  Utilisation d'un thread pour avoir une exécution parallèle

```
1  def run(self):
2      handlers = {
3          '/exit': self._exit,
4          '/quit': self._quit,
5          '/join': self._join,
6          '/send': self._send
7      }
8      self.__running = True
9      self.__address = None
10     threading.Thread(target=self._receive).start()
11     # ...
```

Traitement des commandes

- Extraction de la commande et de ses paramètres

  En coupant le string lu au clavier par rapport au premier espace

- Si la commande est valide, appel de la méthode associée

  Deux cas à gérer selon qu'il y a un paramètre ou non

```
1  def run(self):
2      # ...
3      while self.__running:
4          line = sys.stdin.readline().rstrip() + ' '
5          command = line[:line.index(' ')]
6          param = line[line.index(' ')+1:].rstrip()
7          if command in handlers:
8              handlers[command]() if param == '' else handlers[command](param)
9          else:
10             print('Unknown command:', command)
```

Application client/serveur

Création du serveur et écoute

- Le socket serveur doit écouter sur un port

  Il s'agit du port sur lequel les clients vont pouvoir se connecter

- Attente d'un client avec la méthode `accept`

  Renvoie un tuple avec un socket client et l'adresse de ce dernier

```
1  s = socket.socket()
2  s.bind((socket.gethostname(), 6000))
3
4  s.listen()
5  client, addr = s.accept()
```

Création du client et connexion

- Le socket client doit se connecter au serveur

  Connection à une adresse TCP (IP et port)

- Le socket est ensuite utilisé pour communiquer

  Utilisation des méthodes `send` et `recv`

```
1  s = socket.socket()
2
3  s.connect((socket.gethostname(), 6000))
```

Envoi de données (TCP)

- Envoi de données via un socket avec la méthode `send`

  Permet d'envoyer des octets à l'autre bout du socket

- Les données envoyées doivent être au format binaire

  Conversion d'une chaine de caractères avec `encode`

- Il faut vérifier que l'envoi est bien complet

  La valeur de retour est le nombre d'octets effectivement envoyés

```
1  s = socket.socket()
2  s.connect(('www.google.be', 80))
3
4  data = "Hello World!".encode()
5  sent = s.send(data)
6  if sent == len(data):
7      print("Envoi complet")
```

Boucle d'envoi (TCP)

- Envoi des données par paquets à l'aide d'une boucle

  Utilisation de la taille totale du message

```
1  s = socket.socket()
2  s.connect(('www.google.be', 80))
3
4  msg = "Hello World!".encode()
5  totalsent = 0
6  while totalsent < len(msg):
7      sent = s.send(msg[totalsent:])
8      totalsent += sent
```

Réception de données (TCP)

- Réception de données via un socket avec la méthode `recv`

  Permet de recevoir des octets envoyés par l'autre bout du socket

- Il faut spécifier le nombre maximal d'octets à recevoir

  Préférable d'utiliser une petite valeur puissance de 2 comme 4096

- Liste vide renvoyée s'il n'y a rien à lire sur le socket

```
1  s = socket.socket()
2  s.connect(('www.google.be', 80))
3  s.send(b'GET / HTTP/1.0\n\n')
4
5  data = s.recv(512).decode()
6  print('Reçu', len(data), 'octets :')
7  print(data)
```

Boucle de réception (TCP)

- Réception des données par paquets à l'aide d'une boucle

  Jusqu'à recevoir un message vide

```
1  s = socket.socket()
2  s.connect(('www.google.be', 80))
3  s.send(b'GET / HTTP/1.0\n\n')
4
5  chunks = []
6  finished = False
7  while not finished:
8      data = client.recv(1024)
9      chunks.append(data)
10     finished = data == b''
11 print(b''.join(chunks).decode())
```

Serveur echo

- Boucle d'acceptation de clients

  On accepte un client à la fois

- Mise en attente des demandes tant qu'un client est traité

  Taille file d'attente modifiable avec un paramètre de `listen`

```
1  class EchoServer():
2      # ...
3
4      def run(self):
5          self.__s.listen()
6          while True:
7              client, addr = self.__s.accept()
8              print(self._receive(client).decode())
9              client.close()
10
11     # ...
```

Client echo

- Connexion au serveur echo

  Utilisation de l'adresse TCP du serveur (IP, port)

- Envoi du message texte au serveur

```
1  class EchoClient():
2      # ...
3
4      def run(self):
5          self.__s.connect(SERVERADDRESS)
6          self._send()
7          self.__s.close()
8
9      # ...
```

Protocole de communication

Gestion des erreurs

- Erreur de type `OSError` pour les opérations sur les sockets

  Ou des erreurs spécialisées :

  `socket.herror`, `socket.gaierror` et `socket.timeout`

- Aussi levée par `recv` si la connexion se coupe

  Dans le cas d'une connexion TCP

```
1  s = socket.socket()
2  try:
3      s.connect(("www.gloogle.be", 82))
4  except OSError:
5      print('Serveur introuvable, connexion impossible.')
```

Protocole de communication

- Spécification des messages échangés entre les programmes

  Comprendre les requêtes et réponses

- Plusieurs aspects
  - Mode texte ou binaire
  - Liste des messages valides
  - Structure des messages et types des données
  - Gestion des versions et options

- Gestion des erreurs et gestionnaires

  Identification des erreurs et des comportements

Message binaire

- Sérialisation des données pour les envoyer sur un socket
  - Chaines de caractères avec l'encodage (`encode/decode`)
  - Objets Python avec `pickle` (`dumps/loads`)
  - Données primitive avec `struct` (`pack/unpack`)

```
1 data = 12
2 print(str(data).encode())
3 print(pickle.dumps(data))
4 print(struct.pack('I', data))
```

```
b'12'
b'\x80\x03K\x0c.'
b'\x0c\x00\x00\x00'
```

Additionneur d'entiers

- Serveur de calcul de la somme de nombres entiers

  Le serveur reçoit une liste d'entiers et calcule la somme

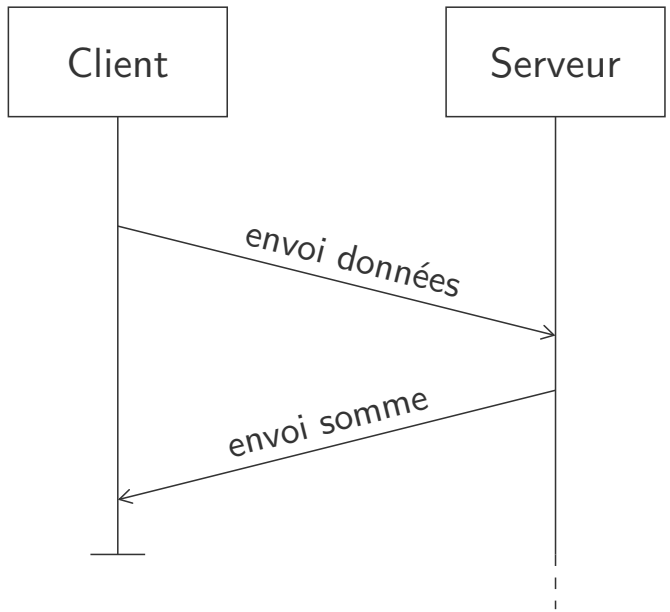
- Définition du protocole de communication

  Échanges des données au format binaire

Diagramme de séquence

- Enchainement des messages échangés au court du temps

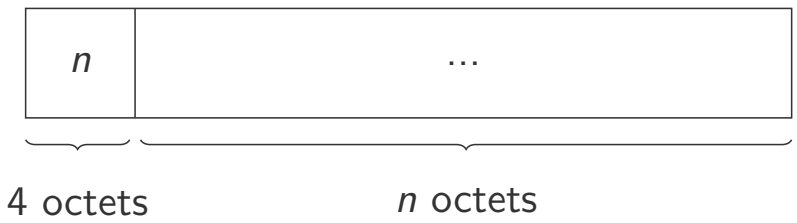
  Messages échangés entre les acteurs



Format des messages

- Envoi de la liste de nombres entiers au serveur

  `pickle` pour envoyer la liste et `struct` pour la taille



- Envoi de la somme au client (avec `struct`)

```
1  def _compute(self):
2      totalsent = 0
3      msg = pickle.dumps(self.__data)
4      self.__s.send(struct.pack('I', len(msg)))
5      while totalsent < len(msg):
6          sent = self.__s.send(msg[totalsent:])
7          totalsent += sent
8      return struct.unpack('I', self.__s.recv(4))[0]
```

Crédits

- https://www.flickr.com/photos/pascalcharest/308357541
- https://openclipart.org/detail/180746/tango-computer-green
- https://openclipart.org/detail/36565/tango-network-server
- https://en.wikipedia.org/wiki/File:Google_Chrome_for_Android_Icon_2016.svg
- https://en.wikipedia.org/wiki/File:ASF-logo.svg
- https://openclipart.org/detail/9368/mailbox
- https://openclipart.org/detail/26431/phone
- https://www.flickr.com/photos/mwichary/2503896186
- https://www.flickr.com/photos/domainededrogant/10932900653
- https://www.flickr.com/photos/chrisgold/6383016703
- https://www.flickr.com/photos/p_d_gibson/2028929267