Architecture micro-service

TP A - Éléments de réseau

Philippe Roussille

3iL INGENIEURS
</les sciences informatiques>

1 Introduction

Années 1990. Dans un coin tranquille de l'usine CanaDuck, **Ginette et Roger**, ingénieurs curieux (et un peu plus si affinités), passent leurs soirées à tester les premiers câbles réseau entre deux PC. Leur idée ? Permettre aux ouvriers d'échanger sans bouger. Avant de créer un système complet, ils expérimentent : un message envoyé, un message reçu… un serveur qui répète, une messagerie simplifiée… Le futur IRC de CanaDuck se prépare doucement.

Ce TP se fait **en binôme**. À chaque étape, testez vos programmes **entre votre machine et celle de votre camarade** (gauche ou droite).

PS : sous Linux, pour connaître votre IP, la commande est `ip a` ou `ifconfig`.

Si vous êtes tout seul (ce qui arrive même aux meilleurs), vous pouvez utiliser l'adresse `localhost` ou `127.0.0.1` qui correspond à votre machine locale. Petit rappel de réseau : pour écouter sur toutes les interfaces, on utilisera l'adresse `0.0.0.0`.

2 Actions et étapes

2.1 Étape 1 - Connexion minimale

2.1.1 Détails

But : établir un premier échange "Bonjour client Bonjour serveur".

- Fournir `server.py` et `client.py` de base.
- Un seul échange, puis fermeture de la connexion.

2.1.2 Ça peut vous être utile...

```
# Création d'une socket côté serveur
serveur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serveur.bind(("", 63000))
serveur.listen(1)
conn, addr = serveur.accept()
```

```python
# Création d'une socket côté client
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(("localhost", 63000))

# ???

client.send("Bonjour serveur !".encode())
message = serveur.recv(1024).decode()
```

2.1.3 Testez !

L'un joue le serveur, l'autre le client. Inversez les rôles ensuite.

2.1.4 Questions

— À quel moment la socket côté serveur est-elle bloquante ?
— Que se passe-t-il si le client se connecte avant que le serveur ne soit prêt ?
— Quelle est la différence entre `bind()` et `listen()` ?

2.2 Étape 2 - Serveur Echo

2.2.1 Détails

But : permettre un échange continu de messages côté client, avec réponse miroir côté serveur.

— Le client envoie un message à chaque `input()`.
— Le serveur le reçoit, le renvoie immédiatement.
— L'échange continue jusqu'à ce que le message "fin" soit envoyé.

2.2.2 Ça peut vous être utile...

```python
# Lecture et renvoi en boucle
while True:
    msg = conn.recv(1024).decode()
    if msg == "fin":
        break
    conn.send(msg.encode())
```

2.2.3 Testez !

Envoyez-vous chacun trois messages différents. Essayez d'utiliser des accents, des emojis, etc.

2.2.4 Questions

— Pourquoi faut-il une boucle dans le serveur ?
— Que se passe-t-il si on oublie de tester `msg == "fin"` ?
— Est-ce que le serveur peut envoyer plusieurs réponses d'affilée ?

2.3 Étape 3 - Serveur multi-clients simple

2.3.1 Détails

But : adapter le serveur Echo pour qu'il traite plusieurs connexions successives (un client à la fois).

- Le serveur écoute en permanence.
- Dès qu'un client se connecte, il échange avec lui comme un Echo.
- Quand ce client envoie "fin", on ferme sa socket, et on recommence avec un autre.

2.3.2 Ça peut vous être utile...

```
while True:
    conn, addr = serveur.accept()
    print("Connexion de", addr)
    while True:
        msg = conn.recv(1024).decode()
        if msg == "fin":
            break
        conn.send(msg.encode())
    conn.close()
```

2.3.3 Testez !

Connectez-vous l'un après l'autre. Le serveur ne doit pas s'arrêter après le premier échange.

2.3.4 Questions

- Le serveur peut-il rester actif après une déconnexion client ?
- Que faut-il modifier pour accepter plusieurs clients à la suite ?
- Peut-on imaginer accepter des clients en parallèle ?

2.4 Étape 4 - Messagerie interactive 1 :1

2.4.1 Détails

But : permettre un dialogue à tour de rôle entre client et serveur, chacun écrivant dans `input()`.

- Le client saisit un message avec `input()` et l'envoie.
- Le serveur affiche, répond avec `input()` et renvoie la réponse.
- L'échange continue jusqu'à ce que quelqu'un écrive "fin".

2.4.2 Ça peut vous être utile...

```
# Côté serveur
message = conn.recv(1024).decode()
print("Client dit:", message)
reponse = input("Répondre > ")
conn.send(reponse.encode())
```

2.4.3 Testez !

Faites une vraie petite discussion, comme un chat (min. 5 échanges).

2.4.4 Questions

— Comment s'assurer que les deux côtés ne parlent pas en même temps ?
— Peut-on rendre cet échange non bloquant ? Comment ?
— Quelle est la meilleure façon de quitter proprement la communication ?

2.5 Étape 5 - Calculatrice en réseau

2.5.1 Détails

But : exécuter un calcul envoyé par le client et retourner le résultat.

— Le client envoie une expression (ex : `3*5+1`).
— Le serveur l'évalue et renvoie le résultat.
— Bonus : détection d'erreur de calcul ou de syntaxe.

2.5.2 Ça peut vous être utile...

```
# Réception d'une expression à calculer
expression = conn.recv(1024).decode()
print("Expression reçue:", expression)

# Évaluation sécurisée
try:
    result = eval(expression)
    conn.send(str(result).encode())
except Exception as e:
    conn.send(f"Erreur: {e}".encode())
```

2.5.3 Testez !

Testez mutuellement au moins 3 calculs chacun, avec un piège dans le lot (ex : division par zéro).

2.5.4 Questions

— Quels sont les risques d'utiliser `eval()` ? (souvenirs de FONDADEV)
— Comment renvoyer une erreur sans faire planter le serveur ?

2.6 Étape 6 - Vers un mini-protocole

2.6.1 Détails

But : structurer les échanges avec des commandes textuelles préfixées (`/commande`).

— Exemples :
  — `/all <texte>` : message simple à stocker.
  — `/me <action>` : message stylisé.
  — `/bye` : fermeture de session.

— `/help` : commande d'aide.
— Côté serveur, parser les commandes et adapter le comportement.

2.6.2 Ça peut vous être utile...

```
# Découper la commande en parties
parts = message.split(" ", 1)
commande = parts[0]
contenu = parts[1] if len(parts) > 1 else ""

if commande == "/me":
    conn.send(f"* {pseudo} {contenu}".encode())
elif commande == "/all":
    conn.send(f"[{pseudo}] {contenu}".encode())
```

2.6.3 Exemple d'échange attendu...

```
Client > /me applaudit
Serveur > * Ginette applaudit

Client > /all Salut tout le monde !
Serveur > [Ginette] Salut tout le monde !

Client > /bye
Serveur > À bientôt Ginette !
```

2.6.4 Testez !

Testez chacun 3 commandes différentes, puis échangez les rôles.

2.6.5 Questions

— Pourquoi structurer les messages avec `/commande` ?
— Comment distinguer facilement les types de messages côté serveur ?

2.7 Étape 7 - (Bonus) Plusieurs clients en parallèle

2.7.1 Détails

But : permettre à plusieurs clients de se connecter **en même temps**. Cette étape est optionnelle et un peu plus avancée.

— Le serveur accepte toujours des connexions dans une boucle.
— Chaque client est traité dans un **thread séparé**.
— Le serveur reste donc disponible pour les autres clients pendant qu'il discute avec un.

2.7.2 Ça peut vous être utile...

```
import threading

# Fonction appelée pour gérer un client individuel (dans un
 ↪  thread séparé)
```

```python
def gerer_client(conn, addr):
    while True:
        # On lit le message envoyé par le client
        msg = conn.recv(1024).decode()
        if msg == "fin":
            break  # Si le message est "fin", on ferme la
             ↪  connexion
        # On renvoie le même message (echo)
        conn.send(msg.encode())
    # Une fois terminé, on ferme la connexion avec ce client
    conn.close()

# Boucle principale du serveur : accepte les connexions
while True:
    conn, addr = serveur.accept()
    # Pour chaque nouveau client, on crée un thread dédié
    threading.Thread(target=gerer_client, args=(conn,
 ↪  addr)).start()
```

2.7.3 Testez !

Connectez deux (ou plus) clients en parallèle et testez que chacun peut parler au serveur sans bloquer les autres.

2.7.4 Questions

— Que se passe-t-il si deux clients envoient des messages en même temps ?
— Peut-on garder un état partagé entre clients ? Est-ce souhaitable ?
— Que faut-il pour aller plus loin vers une vraie messagerie ?

2.8 Étape 8 - (Bonus) Accès concurrent : protéger les zones critiques

2.8.1 Détails

But : empêcher que plusieurs threads n'écrivent en même temps sur une ressource partagée (ex : un fichier ou une liste).

— Lorsqu'un client écrit sur une ressource commune, il doit **verrouiller** cette ressource.
— Les autres clients doivent attendre que le verrou soit relâché.

Implémentez un serveur qui écrit les messages des clients dans une liste, à la suite.

2.8.2 Ça peut vous être utile...

```python
import threading
lock = threading.Lock()

# Exemple d'utilisation
with lock:
```

```
    messages.append(nouveau_msg)
    print("Ajout sécurisé")
```

2.8.3 Testez !

Créez une liste partagée `messages = []`, et assurez-vous que plusieurs clients peuvent y ajouter un message sans provoquer de mélange ou d'erreur.

2.8.4 Questions

— Pourquoi faut-il protéger certaines sections du code ?
— Que risque-t-on si deux clients modifient une même ressource simultanément ?

3 Pour aller plus loin...

En Python, il existe une classe très pratique appelée `socketserver.TCPServer`. Elle permet de créer un serveur TCP avec beaucoup moins de code que ce que nous avons écrit ici.

Cette classe prend en charge :

— la création de la socket,
— l'écoute des connexions,
— l'appel automatique d'un gestionnaire (`RequestHandler`) à chaque client.

C'est une excellente porte d'entrée vers l'écriture de **serveurs plus robustes**, sans avoir à tout écrire soi-même. Elle ne remplace pas les fondamentaux que vous avez expérimentés ici, mais elle **les encapsule intelligemment**.

À explorer si vous êtes curieux :

```
from socketserver import TCPServer, StreamRequestHandler

# On définit une classe qui gère une connexion client.
# Elle hérite de StreamRequestHandler, qui fournit wfile (sortie)
 ↪  et rfile (entrée).
class MonHandler(StreamRequestHandler):
    def handle(self):
        # Ici on peut lire avec self.rfile.readline() ou écrire
         ↪  avec self.wfile.write()
        self.wfile.write(b"Bonjour client !")

# Création du serveur, lié à l'adresse et au port (ici toutes
 ↪  interfaces, port 63000).
# À chaque nouvelle connexion, une instance de MonHandler sera
 ↪  créée.
server = TCPServer(("", 63000), MonHandler)

# Lancement du serveur : boucle infinie qui accepte les
 ↪  connexions
server.serve_forever()
```

Vous avez désormais toutes les bases pour aborder des architectures client/serveur plus riches. Ginette et Roger sont fiers de vous !

À noter : lors du **prochain TP**, nous utiliserons justement `socketserver.TCPServer` comme fondation de notre serveur. Plus besoin de réinventer la roue : cette classe fera le gros du travail pour nous (création de socket, gestion des connexions…).