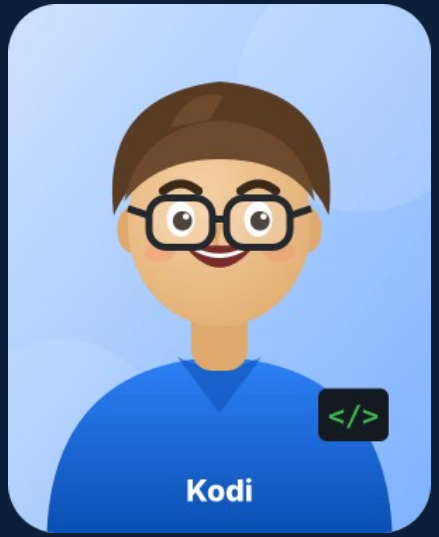
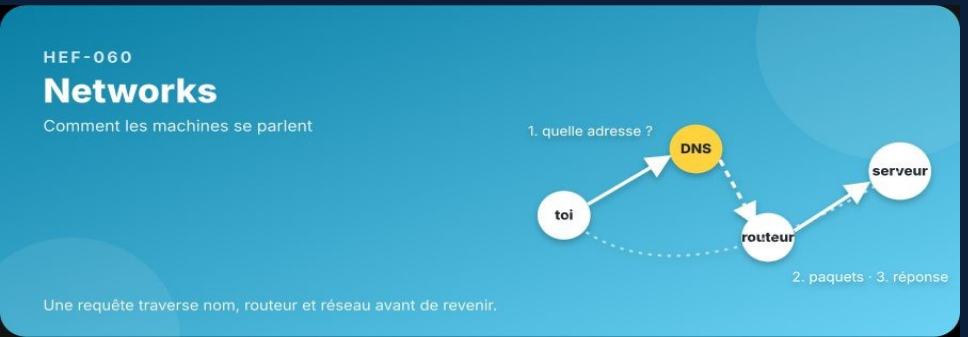
HEF-060 · MISSION 05 / 10

TCP et UDP : deux façons de parler

TCP garantit que tout arrive dans l'ordre. UDP envoie vite et ne promet rien. Quand choisir lequel ?





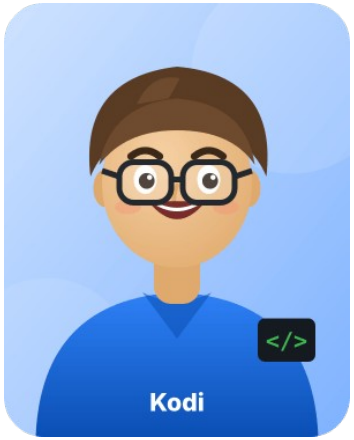
Badge : Maître des sockets

85 min · intermédiaire

L'histoire

L'histoire

HashBox ajoute deux fonctions : un chat entre apprenants et des cours en visioconférence. Kodi veut utiliser TCP pour les deux. Sam hésite.



> *« Pour le chat, oui : on ne veut perdre aucun message. Pour la vidéo, un paquet perdu vaut mieux qu'une image en retard. Il faut deux outils. »*

Tu comprends les deux protocoles de transport, et tu écris tes premiers serveurs et clients avec des sockets.

Ta mission

À la fin, je serai capable de…

1. expliquer le rôle des ports et des sockets
2. décrire la poignée de main en trois temps de TCP
3. expliquer ce que TCP garantit et ce que UDP ne garantit pas
4. écrire un serveur et un client TCP avec `socket`
5. expliquer le problème des frontières de messages dans TCP

Défi d'échauffement

Défi d'échauffement

Deux façons de transmettre un message : (A) un **appel téléphonique** : on compose, l'autre décroche, on se parle, on raccroche ; si on n'entend pas, on répète. (B) une **carte postale** : on l'envoie, aucune confirmation, elle peut se perdre ou arriver dans le désordre. Quels usages conviennent à chacune ? Que ferais-tu pour envoyer un livre en 10 cartes postales ?

Le concept

Les ports et les sockets

Une adresse IP désigne une **machine**. Un **port** (nombre de 0 à 65 535) désigne un **programme** sur cette machine. Un **socket** est l'extrémité d'une communication, identifiée par `(adresse IP, port)`.

Port	Service
22	SSH
25	SMTP (envoi de mail)
53	DNS
80	HTTP
443	HTTPS

Le concept

Les ports et les sockets (suite)

Une connexion est identifiée par **quatre** valeurs : IP source, port source, IP destination, port destination.

UDP : le message isolé

- **Pas de connexion** : on envoie des **datagrammes** indépendants.
- **Pas de garantie** : perte, doublon, désordre possibles.
- Léger et rapide.
- Usages : DNS, voix et vidéo en direct, jeux, découverte réseau.

TCP : le flux fiable

- **Connexion** établie avant l'échange.
- **Flux d'octets** ordonné et sans perte.

Le concept

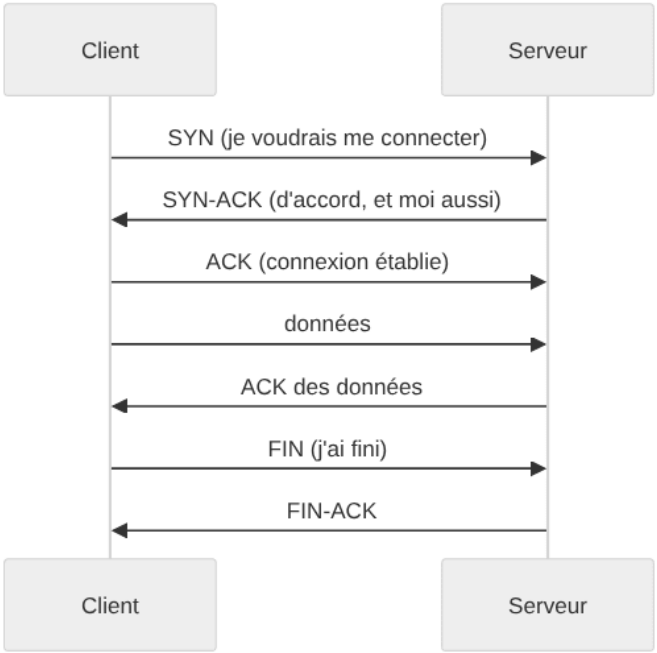
TCP : le flux fiable (suite)

- **Accusés de réception**, retransmission en cas de perte, ordre garanti.
- **Contrôle de flux** (ne pas submerger le récepteur) et de **congestion** (ne pas submerger le réseau).
- Usages : web, e-mail, transferts de fichiers, bases de données.

La poignée de main en trois temps

Le concept

La poignée de main en trois temps (suite)



Le concept

La poignée de main en trois temps (suite)

Description du schéma : le client envoie SYN, le serveur répond SYN-ACK, le client confirme avec ACK : la connexion est établie. Ils échangent des données avec accusés de réception, puis se quittent avec FIN.
Le coût : **un aller-retour** avant le premier octet de données.

TCP est un flux, pas des messages

TCP ne conserve **pas** les frontières des messages. Si tu envoies « bonjour » puis « monde », le récepteur peut lire « bonjourmonde » en une fois, ou « bonj » puis « ourmonde ». Il faut donc **définir un protocole de découpage** : un séparateur (ligne `\n`) ou une longueur annoncée.

Un serveur TCP et un client en Python

Le concept

Un serveur TCP et un client en Python (suite)

```
import socket, threading

def serveur(ecouteur):
    connexion, _ = ecouteur.accept()
    with connexion:
        donnees = connexion.recv(1024)
        connexion.sendall(b"echo: " + donnees)

ecouteur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ecouteur.bind(("127.0.0.1", 0))              # port 0 : le système choisit un por
   t libre
ecouteur.listen()
port = ecouteur.getsockname()[1]
threading.Thread(target=serveur, args=(ecouteur,), daemon=True).start()

with socket.create_connection(("127.0.0.1", port), timeout=2) as client:
    client.sendall(b"bonjour")
    print(client.recv(1024))                 # b'echo: bonjour'
ecouteur.close()
```

Le concept

UDP en Python

```
import socket

recepteur = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
recepteur.bind(("127.0.0.1", 0))
port = recepteur.getsockname()[1]

emetteur = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
emetteur.sendto(b"salut", ("127.0.0.1", port))      # pas de connexion
print(recepteur.recvfrom(1024))                     # (b'salut', (...))
emetteur.close(); recepteur.close()
```

🧠 ***Astuce*** *: choisis TCP quand l'exactitude prime, UDP quand la fraîcheur prime.*

Exemple pas à pas

Problème : écrire un petit serveur d'écho **ligne par ligne** qui gère correctement les frontières de messages avec TCP.

```
import socket, threading

def gerer(connexion):
    with connexion, connexion.makefile("rwb") as fichier:
        for ligne in fichier:                          # lit jusqu'au \n
            fichier.write(b"echo: " + ligne)
            fichier.flush()

def serveur(ecouteur):
    while True:
        try:
            connexion, _ = ecouteur.accept()
        except OSError:
            return
        threading.Thread(target=gerer, args=(connexion,), daemon=True).start()
```

Exemple pas à pas (suite)

```
ecouteur = socket.socket()
ecouteur.bind(("127.0.0.1", 0))
ecouteur.listen()
port = ecouteur.getsockname()[1]
threading.Thread(target=serveur, args=(ecouteur,), daemon=True).start()

with socket.create_connection(("127.0.0.1", port), timeout=2) as c:
    f = c.makefile("rwb")
    for message in (b"un\n", b"deux\n", b"trois\n"):
        f.write(message)
        f.flush()
        print(f.readline())
ecouteur.close()
```

Lecture : on a défini un protocole simple : **un message = une ligne terminée par `\n`**. Grâce à lui, le récepteur sait où chaque message s'arrête, quelle que soit la façon dont TCP a découpé les octets. Un protocole peut aussi annoncer la longueur avant les données (`struct.pack("!I", n)`).

Exemple pas à pas (suite)

UDP : mesurer la perte (sur la boucle locale il y en a rarement, mais le principe demeure) :

```
import socket

r = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
r.bind(("127.0.0.1", 0)); r.settimeout(0.5)
port = r.getsockname()[1]
e = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
for i in range(5):
    e.sendto(f"msg{i}".encode(), ("127.0.0.1", port))
recus = []
try:
    while True:
        recus.append(r.recvfrom(100)[0].decode())
except socket.timeout:
    pass
print(recus, "→", 5 - len(recus), "perdu(s)")
```

À toi, avec aide

À toi, avec aide

Complète le tableau, puis justifie un choix pour HashBox :

Cas d'usage	TCP ou UDP ?	Pourquoi ?
Chat entre apprenants	?	?
Visioconférence	?	?
Téléchargement d'un support de cours	?	?
Requête DNS	?	?
Jeu de quiz en temps réel (score)	?	?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Pour le téléchargement, un octet manquant rend le fichier inutilisable.

Indice 2

Pour la voix, un retard rend le son inutilisable.

Indice 3

DNS : une petite question, une petite réponse, on réessaie si perdu.

À toi, avec aide · réponse

Réponse

Cas	Choix	Pourquoi
Chat	**TCP**	aucun message ne doit se perdre ni arriver dans le désordre
Visioconférence	**UDP**	une image tardive est inutile ; mieux vaut en perdre une que ralentir
Téléchargement	**TCP**	le fichier doit être complet et exact
Requête DNS	**UDP** (TCP en repli)	question courte, réponse courte, simple de réessayer
Quiz en temps réel (score)	**TCP** pour les actions importantes, UDP éventuellement pour des positions fréquentes	dépend de ce qui peut se perdre sans conséquence

À toi, en solo

Protocole à longueur annoncée. Écris un petit protocole où chaque message commence par 4 octets annonçant sa longueur (`struct.pack("!I", n)`). Implémente :

- `envoyer_message(sock, octets)` ;
- `recevoir_message(sock)` (lit exactement le bon nombre d'octets, même si TCP les livre en plusieurs morceaux).

Teste avec un serveur d'écho et des messages de 5, 5 000 et 100 000 octets.

À toi, en solo · réponse

Pistes de correction

```
import socket, struct, threading

def recevoir_exactement(sock, n):
    morceaux = b""
    while len(morceaux) < n:
        bloc = sock.recv(n - len(morceaux))
        if not bloc:
            raise ConnectionError("connexion fermée avant la fin du message")
        morceaux += bloc
    return morceaux

def envoyer_message(sock, octets):
    sock.sendall(struct.pack("!I", len(octets)) + octets)

def recevoir_message(sock):
    (n,) = struct.unpack("!I", recevoir_exactement(sock, 4))
    return recevoir_exactement(sock, n)

def echo(ecouteur):
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    c, _ = ecouteur.accept()
    with c:
        try:
            while True:
                envoyer_message(c, recevoir_message(c))
        except ConnectionError:
            pass

ecouteur = socket.socket()
ecouteur.bind(("127.0.0.1", 0)); ecouteur.listen()
port = ecouteur.getsockname()[1]
threading.Thread(target=echo, args=(ecouteur,), daemon=True).start()

with socket.create_connection(("127.0.0.1", port), timeout=5) as c:
    for taille in (5, 5000, 100_000):
        msg = bytes(range(256)) * (taille // 256 + 1)
        msg = msg[:taille]
        envoyer_message(c, msg)
        assert recevoir_message(c) == msg
```

Pistes de correction (suite)

```
    print("ok")
ecouteur.close()
```

La fonction `recevoir_exactement` est indispensable : `recv(n)` peut rendre moins de `n` octets.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Supposer 1 `send` = 1 `recv`	Messages coupés ou fusionnés	Définir un protocole de découpage
Oublier les délais d'attente	Programme figé si l'autre disparaît	`settimeout` partout
TCP pour tout	Latence inutile pour le temps réel	UDP pour ce qui supporte la perte
Fermer sans lire la fin	Données perdues	Fermer proprement (`shutdown`) après avoir tout lu

Mini-boss

Mini-boss

Mini serveur de chat TCP. Écris un serveur qui accepte plusieurs clients simultanés (un thread par client). Protocole : une ligne par message ; `/pseudo nom` définit le nom ; tout autre texte est diffusé aux autres clients sous la forme `nom: texte`.

1. Teste avec 3 clients simulés dans le même programme.
2. Gère la déconnexion d'un client sans planter le serveur.
3. Ajoute un délai d'inactivité de 2 secondes (déconnexion).
4. Explique quelles propriétés de TCP rendent ce chat simple à écrire.

Mini-boss · réponse

Pistes de correction

```
import socket, threading, time

clients, verrou = {}, threading.Lock()

def diffuser(source, texte):
    with verrou:
        for c, nom in list(clients.items()):
            if c is not source:
                try:
                    c.sendall(texte.encode() + b"\n")
                except OSError:
                    clients.pop(c, None)

def gerer(c):
    c.settimeout(2)
    with verrou:
        clients[c] = "anonyme"
    try:
        f = c.makefile("rb")
```

Mini-boss · réponse

Pistes de correction (suite)

```
        for ligne in f:
            texte = ligne.decode().strip()
            if texte.startswith("/pseudo "):
                clients[c] = texte[8:]
            else:
                diffuser(c, f"{clients[c]}: {texte}")
    except (OSError, socket.timeout):
        pass
    finally:
        with verrou:
            clients.pop(c, None)
        c.close()

def serveur(ecouteur):
    while True:
        try:
            c, _ = ecouteur.accept()
        except OSError:
            return
```

Mini-boss · réponse

Pistes de correction (suite)

```
        threading.Thread(target=gerer, args=(c,), daemon=True).start()

ecouteur = socket.socket(); ecouteur.bind(("127.0.0.1", 0)); ecouteur.listen()
port = ecouteur.getsockname()[1]
threading.Thread(target=serveur, args=(ecouteur,), daemon=True).start()

def client(nom):
    s = socket.create_connection(("127.0.0.1", port), timeout=2)
    s.sendall(f"/pseudo {nom}\n".encode())
    return s

a, b, c = client("ada"), client("bob"), client("cleo")
time.sleep(0.2)
a.sendall(b"salut tout le monde\n")
time.sleep(0.2)
print(b.recv(100), c.recv(100))
a.close()                                  # un client part, le serveur continue
b.sendall(b"ada est partie\n")
time.sleep(0.2)
```

Mini-boss · réponse

Pistes de correction (suite)

```
print(c.recv(100))
ecouteur.close()
```

TCP fournit un flux **fiable et ordonné** : le chat n'a pas à gérer la perte ni le réordonnancement des messages.

Quiz éclair

Quiz éclair

1. TCP garantit :

A. la vitesse maximale
B. la confidentialité
C. l'ordre et l'absence de perte des octets

2. La poignée de main TCP se déroule en :

A. trois temps (SYN, SYN-ACK, ACK)
B. un seul message
C. cinq échanges

3. UDP est adapté à :

A. les transferts de fichiers

Quiz éclair

Quiz éclair (suite)

B. la voix et la vidéo en direct
C. les paiements

4. 🔁 *Révision mission 04* : Le routeur choisit la route avec :

A. le plus court préfixe
B. le plus petit numéro
C. le plus long préfixe

5. 🔁 *Révision mission 04* : Le TTL sert à :

A. empêcher un paquet de tourner indéfiniment
B. chiffrer le paquet
C. compresser le paquet

Quiz éclair · réponse

Réponses

1. **C**.
2. **A**.
3. **B**.
4. **C**.
5. **A**.

Je réfléchis

1. Dans ta vie, quand préfères-tu un échange « avec accusé de réception » plutôt qu'un message sans garantie ?
2. Pourquoi un protocole doit-il définir où s'arrête un message ?

À retenir

1. Port = programme ; socket = (IP, port).
2. TCP : connexion, ordre, fiabilité ; coût d'un aller-retour.
3. UDP : datagrammes sans garantie, rapide.
4. TCP est un flux : il faut définir les frontières de messages.
5. Toujours des délais d'attente.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Maître des sockets

Prochaine mission : DNS : l'annuaire d'internet

