Programmation réseau

Éric Jacoboni

7 février 2022

Université Jean Jaurès, Toulouse

Sommaire

Sommaire

Rappels

Rappels

- TCP/IP est une pile de protocoles utilisés pour le transfert des données sur l'Internet.
- Cette pile peut être décomposée en 4 couches (contre 7 dans le modèle OSI) :

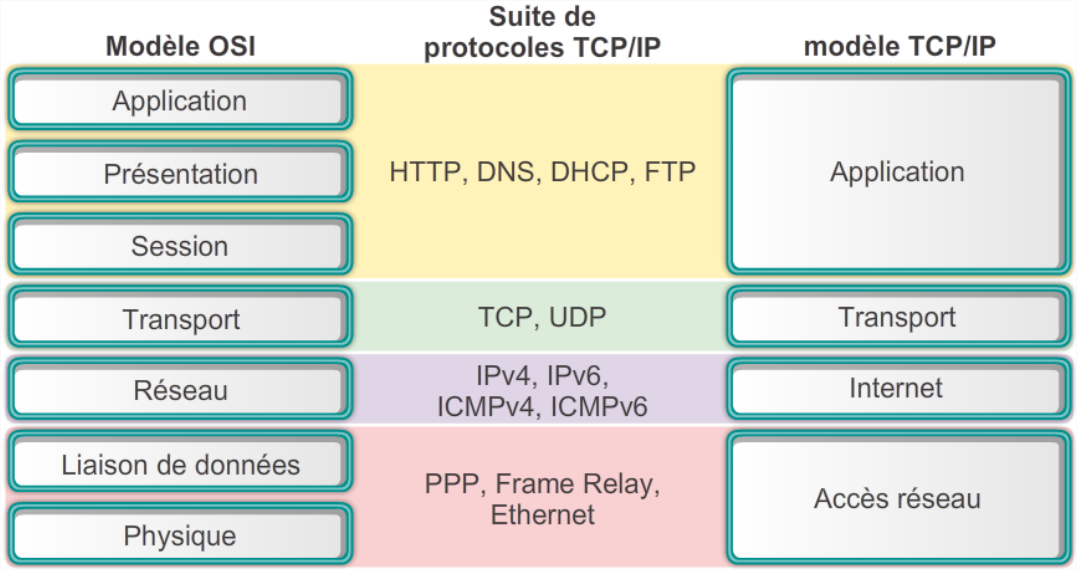


Image extraite de la page https ://qebusen.fmgm2018.com

Remarque

Pour plus de rappels sur l'architecture de TCP/IP, voir ce lien.

Rappels

- UDP et TCP sont des *protocoles de transport* qui reposent sur le protocole IP (*Internet Protocol*) de la *couche réseau*.
- UDP (*User Datagram Protocol*) est un protocole de transmission de données entre deux hôtes reliés par réseau. Il est *non connecté* et *non fiable*.
- TCP (*Transmission Control Protocol*) est un protocole *connecté* et *fiable*.

Remarque

Voir ce lien pour une description détaillée de TCP et ce lien pour UDP.

Rappels

- *Non connecté* signifie qu'il n'y pas d'établissement préalable d'une connexion avant la transmission (comme pour le courrier terrestre).
- *Connecté* signifie qu'il faut d'abord que les deux hôtes établissent une connexion avant de commencer leur transmission (comme pour le téléphone).
- *Fiable/Non Fiable* est lié au contrôle des erreurs. Un protocole fiable réemet les données en cas d'erreur.

Paramètres d'une communication réseau

Avec TCP et UDP, une communication est entièrement définie par *cinq* paramètres :

- Le type du protocole utilisé (TCP ou UDP).
- L'adresse IP de la machine source.
- Le numéro de port associé au protocole sur la machine source.
- L'adresse IP de la machine destination.
- Le numéro de port associé au protocole sur la machine destination.

Paramètres d'une communication réseau

- Les communications réseau impliquent souvent un *client* et un *serveur* : c'est le client qui prend l'initiative de la communication pour demander un service au serveur (qui se borne à attendre les requêtes des clients et à y répondre).
- **Avec TCP**, le client et le serveur ne peuvent communiquer qu'*après avoir établi une connexion* : chacun d'eux connaît donc l'adresse IP de l'autre, ainsi que les numéros de port utilisés. La communication des données se réduit donc à envoyer ou recevoir des données dans un « canal » établi au préalable.
- **Avec UDP**, ils fonctionnent *sans connexion*. C'est au moment de la réception que le serveur peut identifier l'adresse IP et le port de son client et qu'il peut donc utiliser ces informations pour lui répondre.

Paramètres d'une communication réseau : côté client

Comme, dans les deux cas, le protocole est commun aux deux extrémités, une communication se traduit donc localement par *trois valeurs* du point de vue d'un client :

- Le type du protocole utilisé (TCP ou UDP).
- L'adresse IP de la machine serveur.
- Le numéro de port associé au protocole sur le serveur.

Les deux autres paramètres sont déduits par le système du client :

- Le client connaît son adresse IP.
- Le port utilisé par le client est choisi dynamiquement par le système.

Paramètres d'une communication réseau : côté serveur

Du point de vue du serveur, par contre, deux paramètres suffisent :

- Le type du protocole utilisé (TCP ou UDP).
- Le numéro de port sur lequel on attend les communications.

Les trois autres paramètres sont déduites par le système du serveur :

- Le serveur connaît son adresse IP.
- Le serveur connaît l'adresse IP du client et le numéro de port utilisé par ce dernier (soit par la connexion TCP, soit par le contenu de la requête UDP).

Les sockets

Les sockets

- Les sockets (*prises bi-directionnelles*) permettent aux applications de communiquer entre elles via le réseau. Elles ont été introduites par Unix BSD.
- Pour communiquer, les applications créent chacune une socket qui, du point de vue du système, est ensuite considérée comme un simple *descripteur de fichier* (un entier) : on peut donc y lire et y écrire en appelant les appels classiques *read* et *write*, ou utiliser des appels spécifiques aux sockets.
- Les sockets sont donc des *mécanismes de communication* qui permettent à des programmes d'échanger des données.
- Elles sont des points d'accès à la couche transport (*userland*) ou à la couche réseau (*noyau*).

Les sockets

On distingue deux types de sockets, aux fonctionnements très semblables :

- Les sockets de domaine Unix (*AF_UNIX*), permettent les échanges entre deux programmes *sur la même machine* en faisant référence à un nom de fichier.
- Les sockets de domaine Internet (*AF_INET*) permettent les échanges via le réseau. Ce sont celles qui sont présentées ici.
- Ces dernières sont essentiellement de trois types :
  - *SOCK_STREAM* pour les communications TCP.
  - *SOCK_DGRAM* pour les communications UDP.
  - *SOCK_RAW* pour les communications avec la couche réseau.

Adresses des sockets

Pour communiquer, il faut créer une *socket locale* et la brancher sur une *socket distante* (les notions de *locale* et *distante* sont relatives à l'application). Chaque socket est décrite par une adresse de socket dont la représentation varie en fonction du niveau d'abstraction :

- Elle est représentée par une structure *struct sockaddr_in* en C (pas simple à gérer...).
- Dans des langages de plus haut niveau (Perl, Python, Ruby), elle est représentée directement comme un tableau ou un tuple contenant ses différentes composantes.
- En Java, cette adresse est décrite par la classe *InetSocketAddress* qui se construit à partir de deux informations : l'adresse IP et le port.
- En .NET, elle est décrite par la classe *IPEndPoint* qui se construit à partir d'une *IPAddress* et d'un numéro de port.
- En Go et en Rust, elle est décrite par une chaîne de caractères de la forme *"adresse_ip:port"*

L'API de bas niveau

- Une socket est décrite par une *struct sockaddr_in*, dont l'initialisation est la partie la plus pénible de la manipulation des sockets avec l'API C (on verra plus loin des exemples de cette structure).
- Le reste est assez simple : on fait les appels dans le bon ordre et *on n'oublie pas de fermer la socket*.
- Le plus difficile étant de manipuler la structure *struct sockaddr_in*, nous utiliserons un langage plus évolué – *Python*, *Go*, *Rust* – afin de nous concentrer sur les concepts plutôt que sur la gestion des pointeurs...

L'API de bas niveau

- *socket(2)* : Création d'une socket locale, qui sera décrite par un descripteur de fichier, comme *open(2)* et *pipe(2)*.
- *connect(2)* : Avec TCP, *connecte* la socket locale à une socket distante (qui doit être en écoute) ; avec UDP, sert uniquement à créer une *association* avec une socket distante (la socket locale ne pourra plus lire et écrire que sur cette socket distante).
- *read(2)*, *recv(2)*, *recvfrom(2)* : Lecture sur la socket locale. Avec une socket UDP, seule *recvfrom(2)* est possible (elle permet de récupérer les informations sur l'expéditeur). Ces appels sont *bloquants* (comme avec les tubes) et sont tamponnés (ne pas oublier d'envoyer des `\r\n` en fin d'émission).
- *write(2), send(2), sendto(2)* : Écriture sur la socket. Avec une socket UDP, seule *sendto(2)* est possible (elle permet de d'indiquer la socket distante).

L'API de bas niveau

- *close(2), shutdown(2)* : Fermeture de la socket. *shutdown(2)* permet de ne fermer qu'une ou l'autre des extrémités, ou les deux. L'oubli de fermeture d'une socket peut être clause de blocage ! (comme pour les tubes).
- *bind(2)* : Attachement d'une socket locale à un port local. Généralement, uniquement pour les applications serveurs (les clients utilisent le plus souvent des ports alloués dynamiquement par le système).
- *listen(2)* : fixe la taille de la file d'attente des connexions pour une application serveur (uniquement pour TCP).
- *accept(2)* : l'application serveur TCP se bloque tant qu'il n'y a pas de connexion dans la file d'attente. Sinon, cet appel extrait la première demande de la file, crée une nouvelle socket et renvoie son descripteur. Celui-ci permet alors de communiquer avec celui qui a demandé cette connexion.

Schéma de communication UDP

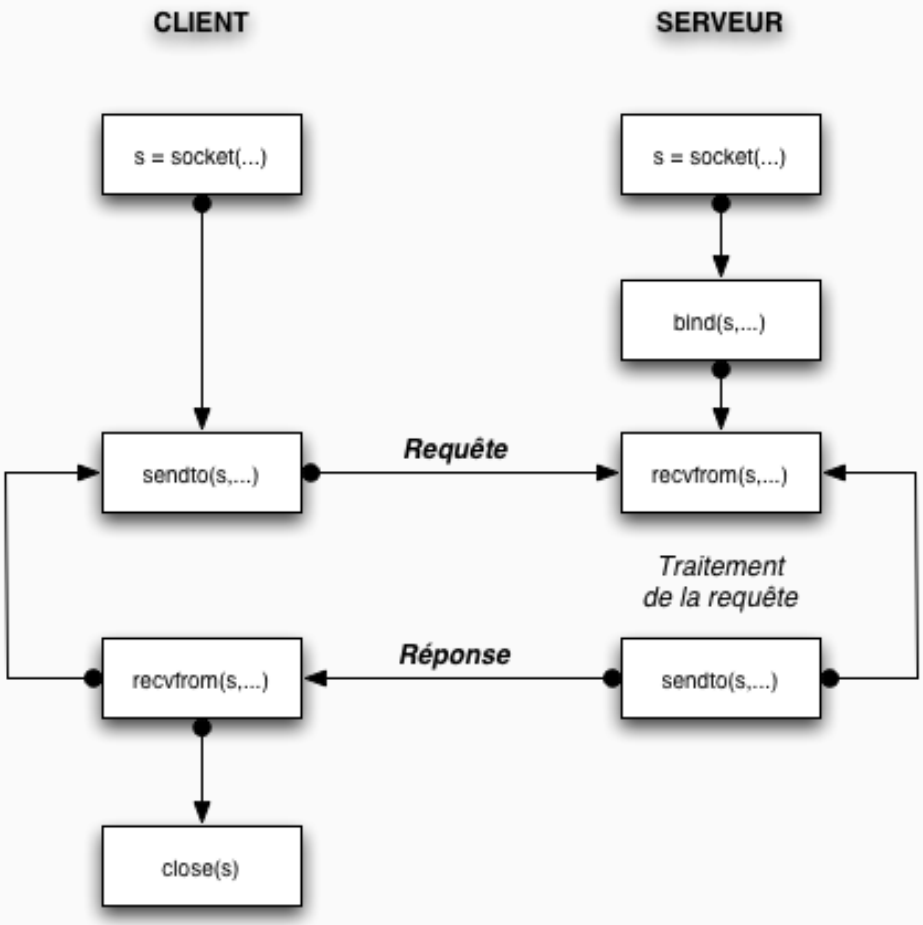
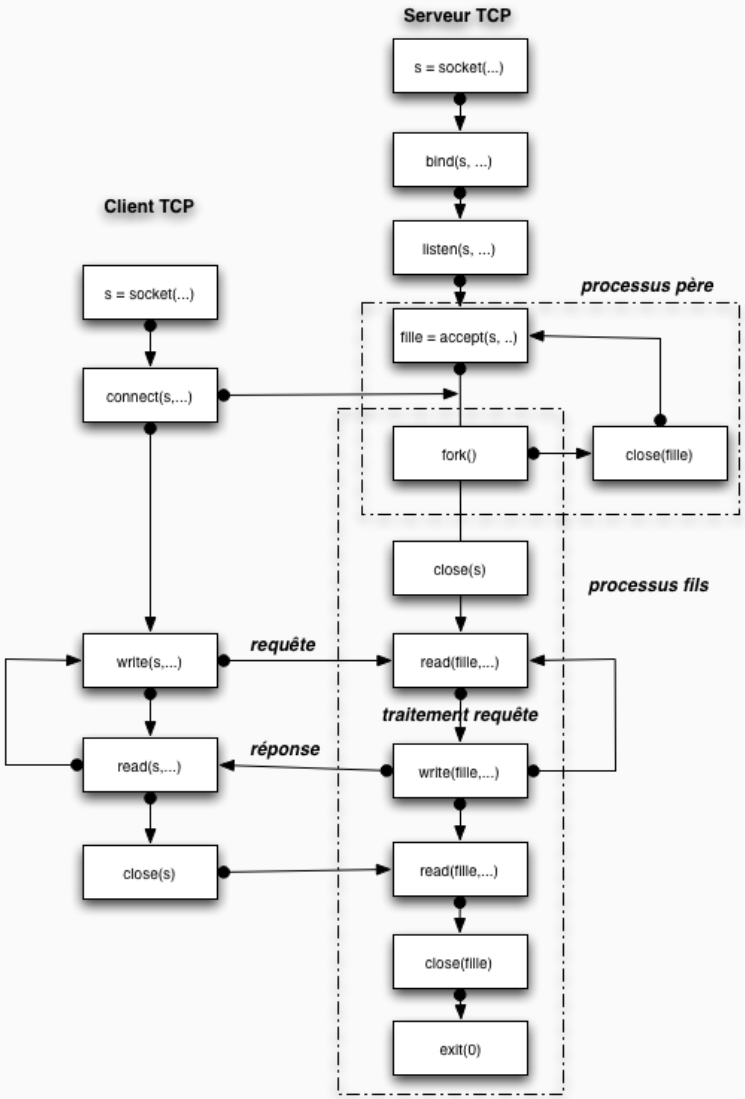


Schéma de communication TCP



Remarques

- Ce schéma part du principe que c'est le client qui met fin à la communication (cas du protocole TELNET, par exemple).
- Avec certains protocoles (HTTP, ECHO, etc.), c'est le serveur qui met fin à la connexion après avoir envoyé sa réponse.
- Dans ce dernier cas, si le client tente de continuer à lire sans tester la fin de fichier (*read* renvoie 0), il est bloqué.

Les sockets en Python

Création d'une socket en Python

- Le module *socket* de Python fournit toutes les opérations nécessaires à l'utilisation d'une socket.
- On crée une socket UDP ou TCP en créant un objet de la classe *socket* définie dans ce module.
- On utilise ensuite les méthodes sur cette socket (ces méthodes portent les mêmes noms que les appels systèmes que nous venons de présenter).
- On ferme la socket quand on veut mettre fin à la communication (important !).

Exemple

Utilisation classique

```
>> import socket
>> conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)    # ou conn = socket()
>> conn.connect(("www.univ-tlse2.fr", 80))                     # connection au serveur web de la fac
>>  # on cause en HTTP au serveur en passant par conn et en envoyant des *bytes*
>> conn.sendall(b'HEAD / HTTP/1.1\r\n')
>> conn.sendall(b"Host: localhost\r\n\r\n")
>> text_io = conn.makefile()                                    # pour pouvoir lire ligne/ligne...
>> for lig in text_io:
>>    print(lig, end='')
>> conn.close()
```

Utilisation d'une clause with

```
>> import socket
>> with socket.socket() as conn:
>>    conn.connect(("www.univ-tlse2.fr", 80))
>>    conn.sendall(b'HEAD / HTTP/1.1\r\n')
>>    conn.sendall(b"Host: localhost\r\n\r\n")
>>    text_io = conn.makefile()
>>    for lig in text_io:
>>       print(lig, end='')
```

Envoi de données

- **Écriture dans une socket TCP** : on utilise les méthodes *send()* ou *sendall()* en leur passant des *bytes*.
- **Écriture dans une socket UDP** : on utilise la méthode *sendto()* avec en paramètre la chaîne de bytes et un tuple contenant le nom (ou l'adresse) de l'hôte destinataire et le port destinataire. On peut également passer des options.

Remarques :

- *On peut également utiliser la méthode connect() sur une socket UDP afin de préciser l'hôte destinataire et son port, auquel cas on pourra ensuite utiliser les méthodes send() et sendall().*
- *Si le message à envoyer est stocké dans une chaîne, il faut l'encoder avec sa méthode encode() (message.encode()) avant de passer le résultat aux méthodes send**

Réception de données

- **Lecture dans une socket TCP** : avec la méthode *recv()* associée à une taille de tampon. Cette méthode renvoie des bytes (qu'on pourra ensuite reconvertir en chaîne avec *str()*). Elle renvoie *b''* (un bytes vide) s'il n'y a plus rien à lire.
- **Lecture dans une socket UDP** : avec la méthode *recvfrom()* associée à une taille de tampon. Cet appel renvoie un tuple *(bytes, adresse)*. La partie adresse est elle-même un tuple décrivant la socket émettrice.

Remarques :

- **Conversion en chaîne** *: les méthodes recv* renvoient un objet bytes. Si on a besoin d'une chaîne, il faut décoder le message reçu avec sa méthode decode : message.decode().*
- **Lecture de lignes** *: si l'on sait que l'on va recevoir des lignes de texte (cas de HTTP/POP3/SMTP/etc.), le plus simple est sans doute d'encapsuler la socket dans un ≪ IOWrapper ≫ afin de pouvoir ensuite la lire comme un fichier texte. On peut ensuite utiliser les méthodes readline() ou readlines() du wrapper, ou itérer directement sur le wrapper. Voir l'exemple précédent.*

Fermeture d'une socket

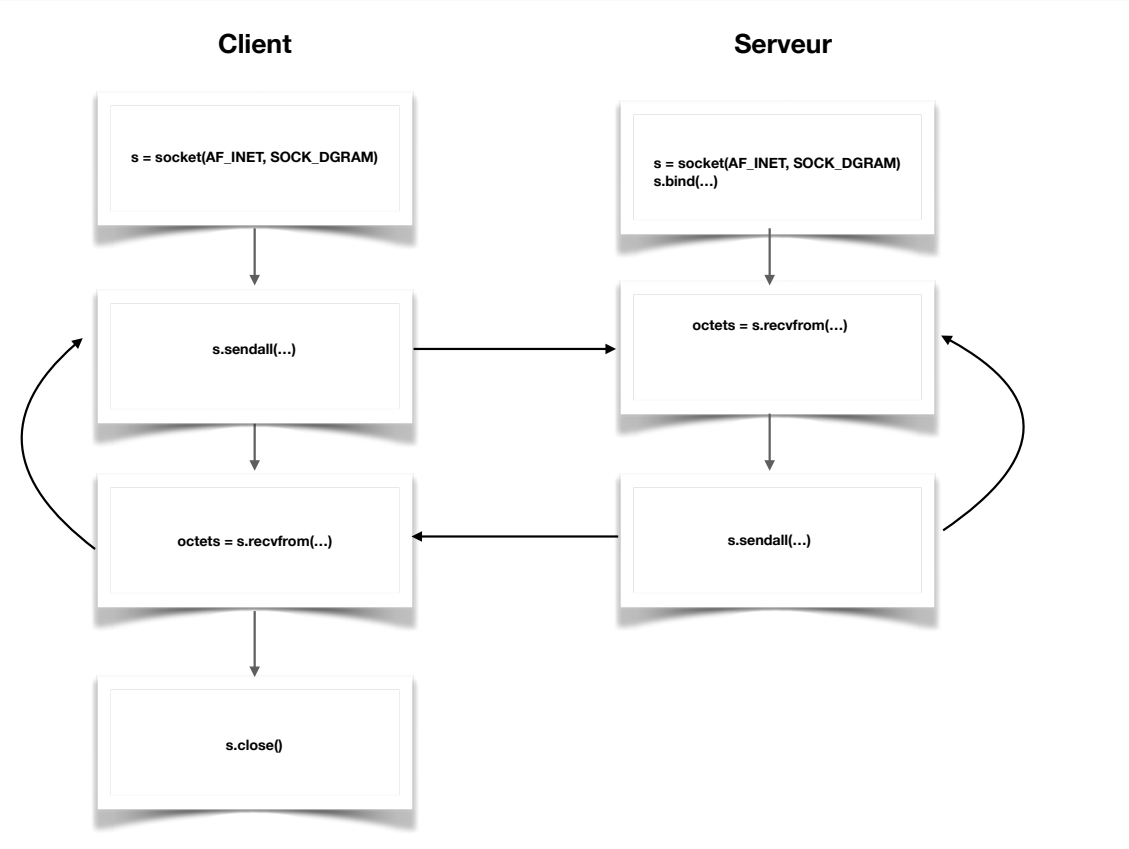
- On ferme une socket par un appel à ses méthodes *close()* ou *shutdown()*.
- *shutdown()* permet de choisir l'extrémité que l'on ferme (extrémité de lecture, d'écriture, ou les deux).

Remarques :

- *Comme on l'a vu, si une socket est ouverte avec une close with, elle est automatiquement fermée à ses deux extrémités à la fin du bloc.*
- *La fermeture des sockets est aussi importante que celle des tubes : si une socket émettrice reste ouverte alors qu'elle n'a plus rien à émettre et si un client attend de lire sur cette socket, il sera bloqué tant qu'elle n'est pas fermée...*

Communication UDP en Python

Schéma de communication UDP en Python



Algorithme d'un client ou d'un serveur UDP

1. Création de la socket.
2. Liaison de la socket à une IP et un port d'écoute avec *bind()* (uniquement pour le serveur).
3. Boucle de réception/traitement/réponse utilisant *sendto()* et *recvfrom()*
4. Fermeture de la socket à l'arrêt du client ou du serveur.

Serveur d'écho UDP

Fichier echo_server.py : initialisations

```
import socket
import sys

if len(sys.argv) != 2:
    print(f"Usage: {sys.argv[0]} <port>", file=sys.stderr)
    sys.exit(1)

TAILLE_TAMPON = 256

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Liaison de la socket à toutes les IP possibles de la machine
sock.bind(('', int(sys.argv[1])))
print("Serveur en attente sur le port " + sys.argv[1], file=sys.stderr)
```

Serveur d'écho UDP (suite)

Fichier echo_server.py : boucle d'attente du serveur

```
while True:
    try:
        # Récupération de la requête du client
        requete = sock.recvfrom(TAILLE_TAMPON)

        # Extraction du message et de l'adresse sur le client
        (mess, adr_client) = requete
        ip_client, port_client = adr_client

        print(f"Requête provenant de {ip_client}. Longueur = {len(mess)}",
                  file=sys.stderr)

        # Construction de la réponse
        reponse = mess.decode().upper()

       # Envoi de la réponse au client
       sock.sendto(reponse.encode(), adr_client)

    except KeyboardInterrupt: break

sock.close()
print("Arrêt du serveur", file=sys.stderr)
sys.exit(0)
```

La même chose en Go...

Fichier serveur.go : programme principal

```
package main

import (...)

func main() {
  nom_prog := os.Args[0][strings.LastIndex(os.Args[0], "/")+1:]

  if len(os.Args) != 2 {
    fmt.Fprintf(os.Stderr, "Usage: %s <port>\n", nom_prog)
    os.Exit(1)
  }

  sockAddr, _ := net.ResolveUDPAddr("udp", ":"+os.Args[1]) //  ":" = toutes les interfaces
  socketLocale, _ := net.ListenUDP("udp", sockAddr)        // Bind...

  setupCloseHandler() // Mise en place de la gestion de Ctrl-C pour arrêter le serveur
  traitementClient(socketLocale)
}
```

Remarques

- On crée une socket UDP sur le port passé au programme avec *net.ResolveUDPAddr(...)* et on la met en écoute avec *net.ListenUDP(...)* (qui effectue un appel à *bind()*...)
- La pseudo-adresse IP *" :"* passée à *net.ResolveUDPAddr(...)* désigne *toutes* les adresses IP de la machine locale.

La même chose en Go (suite)...

Fichier serveur.go (suite)

```
// setupCloseHandler gère les signaux SIGINT (Ctrl-C) et SIGTERM
func setupCloseHandler() {
  c := make(chan os.Signal)
  signal.Notify(c, os.Interrupt, syscall.SIGTERM)
  go func() {
    <-c
    fmt.Println("Bye...")
    os.Exit(0)
  }()
}

// traitementClient gère le dialogue entre le serveur et les clients
func traitementClient(sock *net.UDPConn) {
  buf := make([]byte, 128)
  for {
    nb_octets, sock_source, _ := sock.ReadFromUDP(buf)
    sock.WriteToUDP(bytes.ToUpper(buf[:nb_octets]), sock_source)
  }
}
```

Remarques

- Pour gérer l'arrêt du serveur par Ctrl-C, on met en place un ≪ canal ≫ de signaux : *make(chan os.Signal)*.
- On demande le transfert vers ce canal de tous les signaux d'interruption (Ctrl-C) et SIGTERM : *signal.Notify(...)*.
- On lance une goroutine qui attend qu'un signal arrive dans le canal et qui affiche alors un message avant de mettre fin au programme.

La même chose en Rust

Fichier serveur.rs : programme principal

```
extern crate ctrlc;
use std::{net, env, process, path};

fn main()  {
    let args: Vec<String> = env::args().collect();
    let nom_prog = args[0].split(path::MAIN_SEPARATOR).last().unwrap();

    if args.len() != 2 {
        eprintln!("Usage: {nom_prog} <port>");
        process::exit(1);
    }

    let sock_addr = format!("0.0.0.0:{}", &args[1]);
    let socket_locale = net::UdpSocket::bind(sock_addr).expect("Pb bind");

    ctrlc::set_handler(|| {
        println!("Bye !");
        process::exit(0);
    }).unwrap();

    traitement_client(socket_locale);
}

fn traitement_client(sock: net::UdpSocket) {
    let mut buf = [0;128];
    while let Ok((nb_octets, sock_source)) = sock.recv_from(&mut buf) {
        sock.send_to(&buf[..nb_octets].to_ascii_uppercase(), sock_source).unwrap();
    }
}
```

Encodages des messages

Remarques

- Que ce soit avec Python, Go, Rust et tous les langages disposant d'un type chaîne ≪ évolué ≫, il **faut** convertir en octets les messages envoyés et, inversement, reconvertir en chaîne les suites d'octets récupérées...
- En Python, cela passe par l'appel des fonctions *encode()/decode()*, en Go et en Rust, par l'utilisation de tranches d'octets...

Pour info : mise en place d'une socket UDP en C

```
int main(int argc, char *argv[]) {
  struct sockaddr_in adresse_socket_serveur;
  size_t taille_adresse_socket_serveur;
  int numero_port_serveur, sock;

  ...
  numero_port_serveur = atoi(argv[1]);

  if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1) ...   // Erreur de création de la socket

  adresse_socket_serveur.sin_family      = AF_INET;
  adresse_socket_serveur.sin_addr.s_addr = INADDR_ANY;
  adresse_socket_serveur.sin_port        = htons(numero_port_serveur);

  taille_adresse_socket_serveur = sizeof(adresse_socket_serveur);

  if (bind(sock, (struct sockaddr *)&adresse_socket_serveur,
                   taille_adresse_socket_serveur) == -1) ...   // Erreur de bind

  /* À partir de là, la socket est créé et attend les connexions...
     On lit avec revfrom() et on envoie avec sendto */
```

Client UDP en Python

Fichier echo_client.py

```
from socket import *
import sys

if len(sys.argv) != 3:
    print(f"Usage: {sys.argv[0]} <ip> <port>", file=sys.stderr)
    sys.exit(1)

TAILLE_TAMPON = 256

with socket(AF_INET, SOCK_DGRAM) as sock:
    # Remarque : pas besoin de bind car le port local est choisi par le système

    mess = input("Entrez votre message : ")

    # Envoi de la requête au serveur (ip, port) après encodage de str en bytes
    sock.sendto(mess.encode(), (sys.argv[1], int(sys.argv[2])))

    # Réception de la réponse du serveur et décodage de bytes en str
    reponse, _ = sock.recvfrom(TAILLE_TAMPON)
    print("Réponse = " + reponse.decode())
```

La même chose en Java...

Fichier ClientEcho.java

```
import ...

public class ClientEcho {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("Usage: java ClientEcho <ip_serveur> <port>");
            System.exit(1);
        }


  try (var sock = new DatagramSocket()) {
        byte[] requete;
      byte[] reponse = new byte[1000];
        Console clavier = System.console();

      // Construction de l'adresse de socket du serveur
      InetAddress adr_serveur = InetAddress.getByName(args[0]);

      requete = clavier.readLine("Entrez votre message : ").getBytes();

      // Création du datagramme à envoyer
      DatagramPacket envoi = new DatagramPacket(requete,
                                requete.length, adr_serveur, Integer.parseInt(args[1]));

      // Création d'un datagramme "vide" pour recevoir la réponse
      DatagramPacket recept = new DatagramPacket(reponse,
                                        reponse.length);

      // Envoi du datagramme
      sock.send(envoi);
```

La même chose en Java (suite)...

Fichier ClientEcho.java (suite et fin...)

```
        // Réception de la réponse
        sock.receive(recept);

        // Affichage de la réponse
        String mess = new String(reponse, 0, recept.getLength());
        System.out.println(mess);
    } catch (Exception e) {
          System.err.println(e.getMessage());
          System.exit(2);
      }
    } // main
} // class
```

La même chose en Rust

Fichier client.rs : programme principal

```
use std::io::{stdin, stdout, Write};
use std::net::UdpSocket;
use std::path;

fn main() {
    let args: Vec<String> = std::env::args().collect();
    let program_name = args[0].split(path::MAIN_SEPARATOR).last().unwrap();

    if args.len() != 2 {
        eprintln!("Usage: {program_name} <ip:port>");
        std::process::exit(1);
    }

    // Port = 0 => port choisi par l'OS...
    let socket_locale = UdpSocket::bind("127.0.0.1:0").expect("Pb bind...");
    socket_locale.connect(&args[1]).expect("Pb connect...");

    loop {
        let mess = input("Entrez un message (quit pour quitter) : ");
        if mess.to_lowercase() == "quit" {
            println!("Bye...");
            break;
        }
        socket_locale.send(mess.as_bytes()).expect("Pb send...");
        let mut buf= [0; 16];
        if let Ok(size) = socket_locale.recv(&mut buf) {
            println!("reçu : {}", String::from_utf8_lossy(&buf[..size]));

        }
    }
}
```

La même chose en Rust (suite...)

Fichier client.rs : suite...

```
fn input(mess: &str) -> String {
  print!("{mess}");
  stdout().flush().unwrap();
  let mut got = String::new();
  stdin().read_line(&mut got).unwrap();
  got.pop().unwrap();   // suppression du \n
  got
}
```

Remarques

- Ici, il faut lier explicitement une socket locale en précisant qu'on laisse le système décider du port (appel à *UdpSocket*::*bind()* avec un numéro de port égal à 0)
- Puis, on la ≪ connecte ≫ au serveur en utilisant la fonction *connect()*, ce qui permet d'utiliser ensuite les fonctions *send* et *recv*, comme pour une connexion TCP...
- Pour mimer l'opération de saisie de Python, on a écrit une fonction *input*.
- Comme en Python et en Go, on convertit en octets avant d'envoyer (appel à *as_bytes()*) et on effectue l'opération inverse à la réception (appel à *String*::*from_utf8_lossy()*).

Communication TCP en Python

Algorithme d'un serveur TCP

1. Création de la socket.
2. Liaison de la socket à un port d'écoute avec *bind()*.
3. Fixer la taille de la file d'attente avec *listen()*.
4. Se mettre en attente d'une connexion avec *accept()*.
5. Créer un thread pour traiter cette requête pendant que le père continue d'attendre une autre connexion.
6. Fermeture de la socket principale à l'arrêt du serveur

Mise en œuvre en Python

- Pour créer des connexions filles, on utilisera des instances de la classe *threading.Thread*.
- Même remarque que précédemment : parfois, c'est le serveur qui met fin à la connexion, parfois c'est le client...

Exemple : un serveur echo...

Fichier serveur.py : programme principal

```
import ...

if len(sys.argv) != 2:
print(f"Usage: {sys.argv[0]} <port>", file=sys.stderr)
sys.exit(1)

sock_locale = socket.socket()
sock_locale.bind(("", int(sys.argv[1])))
sock_locale.listen(4)

while True:
    try:
        sock_client, adr_client = sock_locale.accept()
        threading.Thread(target=traiter_client, args=(sock_client,)).start()
    except KeyboardInterrupt:
        break

sock_locale.shutdown(socket.SHUT_RDWR)
print("Bye")

for t in threading.enumerate():
    if t != threading.main_thread(): t.join

sys.exit(0)
```

Exemple : un serveur echo (suite)...

Fichier serveur.py : traitement d'une connexion

```
def traiter_client(sock_fille):
  while True:
      mess = sock_fille.recv(256)
      if mess.decode() == "":
          break
      sock_fille.sendall(mess.upper())
```

Remarques

- Comme d'habitude, ce qui est reçu par *recv* est un tableau d'octets. Il faudrait donc appeler la méthode *decode()* pour pouvoir le traiter comme une chaîne de caractères.
- Inversement, il faut appeler la méthode *encode()* sur une chaîne de caractères avant de l'envoyer avec *sendall*.
- Ici, on peut directement appeler *upper()* sur un tableau d'octets : c'est la raison pour laquelle on n'a pas besoin d'effectuer ces conversions.
- Avant de terminer le programme principal, on attend que toutes les connexions aient été fermées (c'est un choix, pas une obligation...)

La même chose en Rust...

Fichier serveur.rs : le programme principal

```
use std::{env, thread, process, path};
use std::io::{Read, Write};
use std::net::{TcpListener, TcpStream};

fn main() {
    let args: Vec<String> = env::args().collect();
    let program_name = args[0].split(path::MAIN_SEPARATOR).last().unwrap();

    if args.len() != 2 {
        eprintln!("Usage: {program_name} <port>");
        process::exit(1);
    }

    let sock_addr = format!("0.0.0.0:{}", args[1]);
    let socket_locale = TcpListener::bind(sock_addr).expect("Pb bind...");

    ctrlc::set_handler(|| {
        println!("Bye !");
        process::exit(0);
    }).unwrap();

    // Lancement d'un thread par connexion
    for stream in socket_locale.incoming().flatten() {
        thread::spawn(|| {
            traitement_client(stream)
        });
    }
}
```

La même chose en Rust (suite...)

Fichier serveur.rs : suite...

```
fn traitement_client(mut stream: TcpStream) {
  let mut buf = [0; 128];
  while let Ok(size) = stream.read(&mut buf) {
      stream.write_all(&buf[..size].to_ascii_uppercase()).expect("Pb write...");
  }
}
```

Remarques

- Comme précédemment, une adresse IP égale à *0.0.0.0* signifie ≪ toutes les adresses de l'hôte ≫.
- L'appel à *socket_locale.incoming()* renvoie un itérateur sur toutes les demandes de connexion des clients (cela revient à faire une boucle sur *accept*). Cet itérateur renvoie les sockets des clients.
- On lance un thread pour la connexion par un appel à *Thread::spawn()*.
- On termine la lecture des données lorsque l'appel à *read()* renvoie une erreur.

Algorithme d'un client TCP

1. Création de la socket.
2. Appel de sa méthode *connect()* en lui fournissant l'IP et le port du serveur dans un tuple.
3. Utilisation des méthodes *sendall()* et *recv* pour envoyer les données (ou utilisation d'un « IO Wrapper ») pour lire les réponses ligne par ligne – voir exemple plus haut).
4. Fermeture de la socket principale à l'arrêt du serveur

Client TCP en Python

Fichier client.py

```
import ...

if len(sys.argv) != 2:
  print(f"Usage: python {sys.argv[0]} <hote:port>\n", file=sys.stderr)
  sys.exit(1)

adresse, port = sys.argv[1].split(":")
with socket.socket() as sock_locale:
  sock_locale.connect((adresse, int(port)))
  while True:
      commande = input("Entrez une commande (quit pour quitter) : ")
      if commande.upper() == "QUIT":
          break
      sock_locale.send(commande.encode())
      reponse = sock_locale.recv(256)
      print(reponse.decode())
```

La même chose en Java

Fichier Client.java

```
import ...

public class Client {
    public static void main(String[] args) {
      if (args.length != 2) {
          System.err.println("Usage: java Client <ip_serveur> <port>");
          System.exit(1);
      }

      Console c = System.console();
      String commande, reponse;

      try (var socketLocale = new Socket(args[0], Integer.parseInt(args[1]))) {
          var output = new PrintWriter(socketLocale.getOutputStream(), true);
          var input = new BufferedReader(new InputStreamReader(socketLocale.getInputStream()));

          while (true) {
            System.out.print("Entrez une commande (quit pour quitter) : ");
            commande = c.readLine();
            if (commande.toUpperCase().equals("QUIT")) {
                break;
            }
            output.println(commande);
            reponse = input.readLine();
            System.out.println(reponse);
          }
      } catch(java.io.IOException e) {
          System.err.println(e);
          System.exit(2);
      }
    }
}
```

La même chose en Go

Fichier client.go

```
package main

import ...

func input(r *bufio.Reader, mess string) string { ... }

func main() {
  nom_prog := os.Args[0][strings.LastIndex(os.Args[0], "/")+1:]
  if len(os.Args) != 2 {
    fmt.Fprintf(os.Stderr, "Usage: %s <ip:port>\n", nom_prog)
    os.Exit(1)
  }

  socketLocale, _ := net.ResolveTCPAddr("tcp", os.Args[1])
  connexion, _ := net.DialTCP("tcp", nil, socketLocale)
  defer connexion.Close()
  reader := bufio.NewReader(os.Stdin)

  for {
    mess := input(reader, "Entrez un message (quit pour quitter) : ")
    if strings.ToLower(mess) == "quit" {
      fmt.Println("Bye...")
      break
    }
    connexion.Write([]byte(mess))
    buf := make([]byte, 16)
    nb_octets, _ := connexion.Read(buf)

    fmt.Printf("reçu : %s\n", string(buf[:nb_octets]))
  }
}
```

La même chose en Rust

Fichier client.rs

```
use ...

fn input(mess: &str) -> String { ... }

fn main() {
    let args: Vec<String> = env::args().collect();
    let program_name = args[0].split(path::MAIN_SEPARATOR).last().unwrap();

    if args.len() != 2 {
        eprintln!("Usage: {program_name} <ip:port>");
        process::exit(1);
    }

    let mut socket_locale = TcpStream::connect(&args[1]).expect("Pb connect...");

    loop {
        let mess = input("Entrez un message (quit pour quitter) : ");
        if mess.to_lowercase() == "quit" {
            println!("Bye...");
            break;
        }
        socket_locale.write_all(mess.as_bytes()).expect("Pb write...");
        let mut buf= [0; 16];
        if let Ok(size) = socket_locale.read(&mut buf) {
            println!("reçu : {}", String::from_utf8_lossy(&buf[..size]));
        };
    }
    drop(socket_locale);
}
```