

DÉVELOPPEMENT D'APPLICATIONS WEB

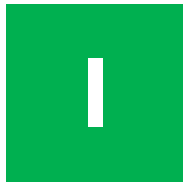
FRONTEND

BACKEND

BASES DE DONNÉES

JAVA

Services web



Serveurs Web

Et protocole HTTP

Applications Client/Serveur



Le poste client exécute une application locale : le **client**

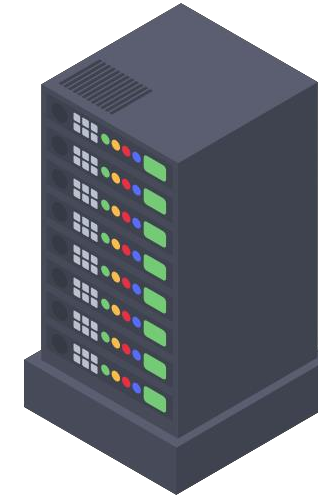
1. Le client envoie des **requêtes** au service



2. Le service envoie des **réponses** au client

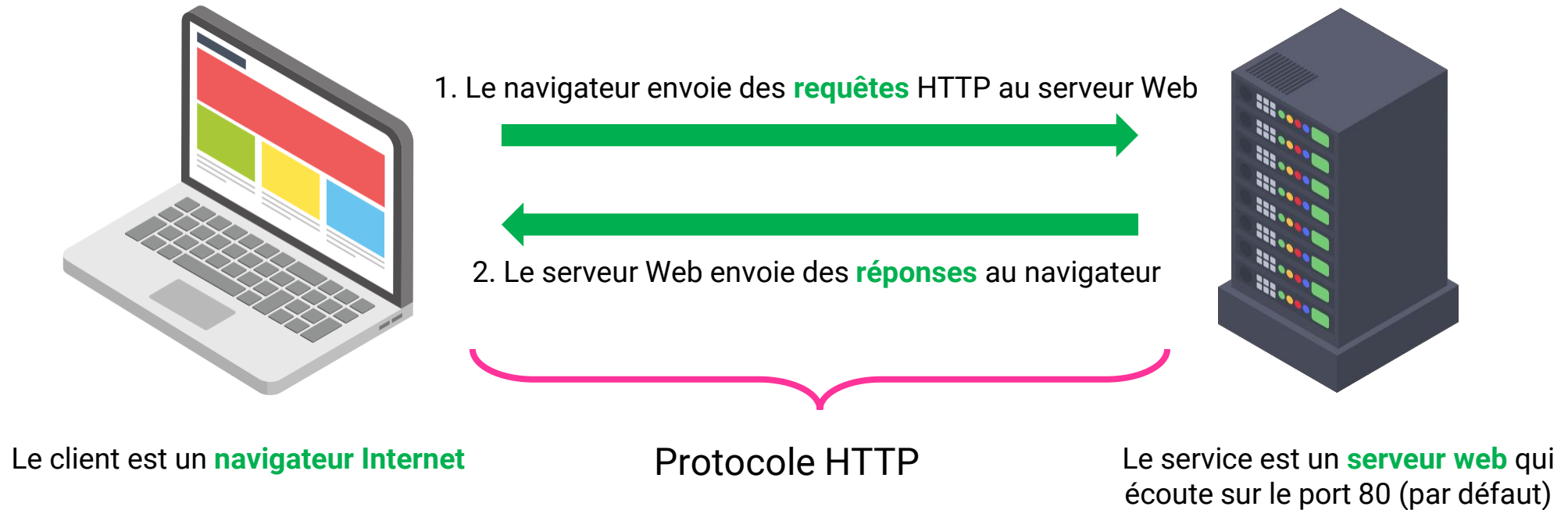


Protocole de communication



Le serveur exécute une application locale : un **service** qui écoute un port donné et attend que des requêtes y soient transmises.

Applications Web



Protocole HTTP



1. Requête du client : **GET / HTTP/1.0**



Traitement de la requête
par le serveur

2. Réponse du serveur : **HTTP/1.0 200 OK**



Requête HTTP



Format de la requête

Commande HTTP
Entêtes de la requête (headers)
Corps de la requête (body)

Exemples de requêtes

```
GET /index.html HTTP/1.0
```

```
POST /auth HTTP/1.0  
Content-Type: application/json  
Content-Length: 40  
  
{"login":"cmeunier","password":"toto21"}
```

La commande HTTP est composée de :

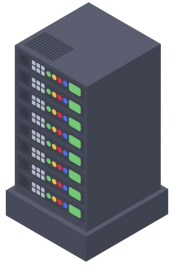
- la méthode HTTP (GET, POST, PUT, DELETE, ...)
- l'url à laquelle on souhaite accéder
- la version du protocole utilisé

La section des entêtes contient des informations sur les données transmises, celles acceptées en retour, le client utilisé pour se connecter, ...

Le corps de la requête (les données transmises) est séparé des entêtes par une ligne vide.

Chaque ligne de la requête (en dehors du body) se termine par un retour à la ligne (\n).

Requête HTTP



Format de la réponse

Statut de la réponse HTTP
Entêtes de la réponse (headers)
Corps de la réponse (body)

Exemples de requêtes

```
HTTP/1.0 404 Not found
```

```
HTTP/1.0 200 OK  
Content-Type: application/json  
Content-Length: 32  
  
{"token": "GFDJKGH4GSH6QSD64GDS"}
```

Le statut de la réponse HTTP est composé de :

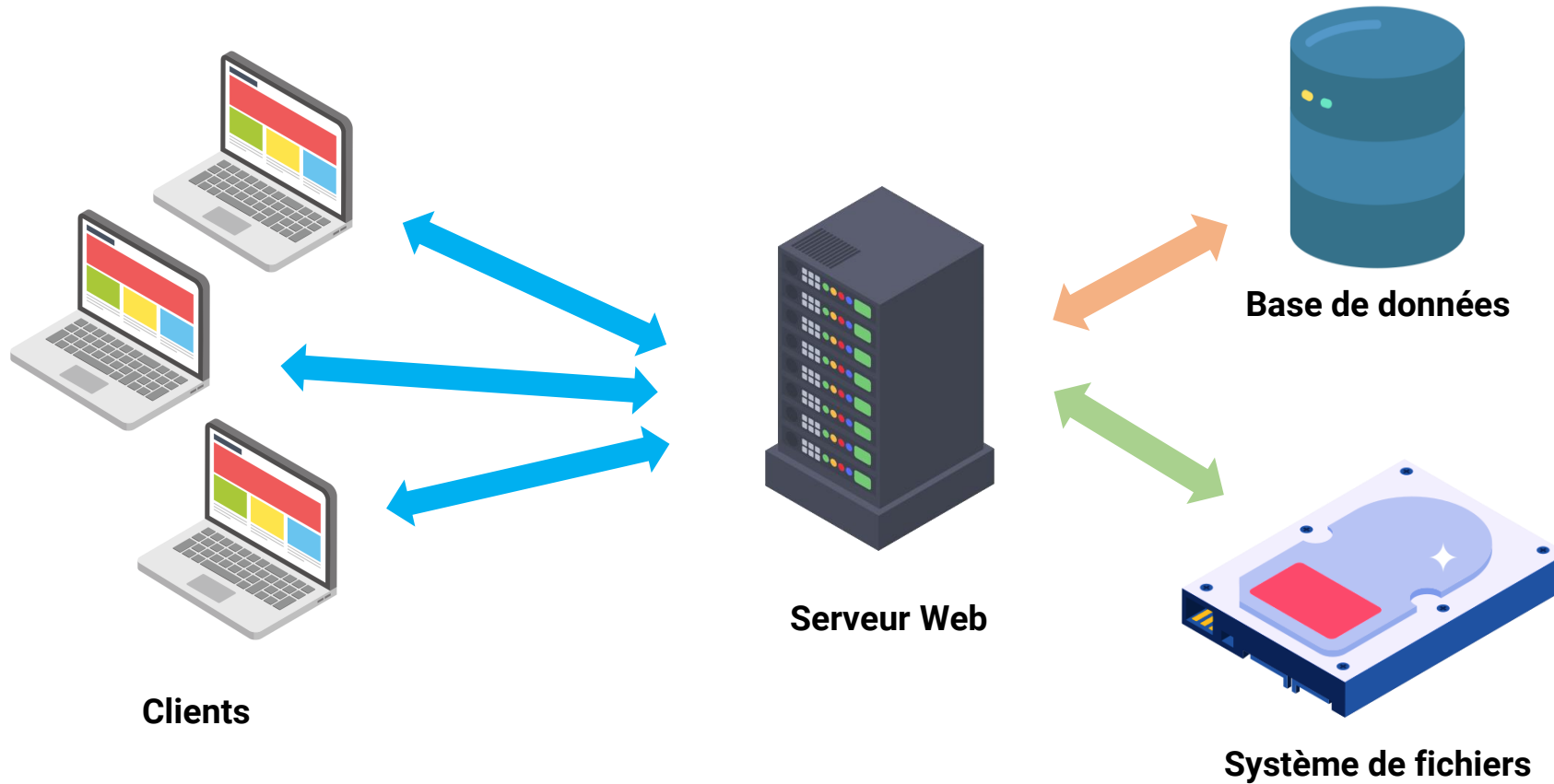
- la version du protocole utilisé
- le code de réponse (200, 404, 500, ...)
- le message associé au code de réponse

La section des entêtes contient des informations sur les données transmises, le type de serveur, ...

Le corps de la réponse (les données transmises) est séparé des entêtes par une ligne vide.

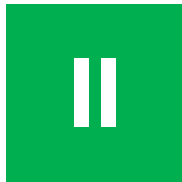
Chaque ligne de la réponse (en dehors du body) se termine par un retour à la ligne (\n).

Implémenter un serveur Web



Fonctionnalités nécessaires

- Gérer les connexions réseaux
- Accéder au système de fichiers
- Interagir avec une base de données
- Traiter plusieurs requêtes simultanément



Sockets

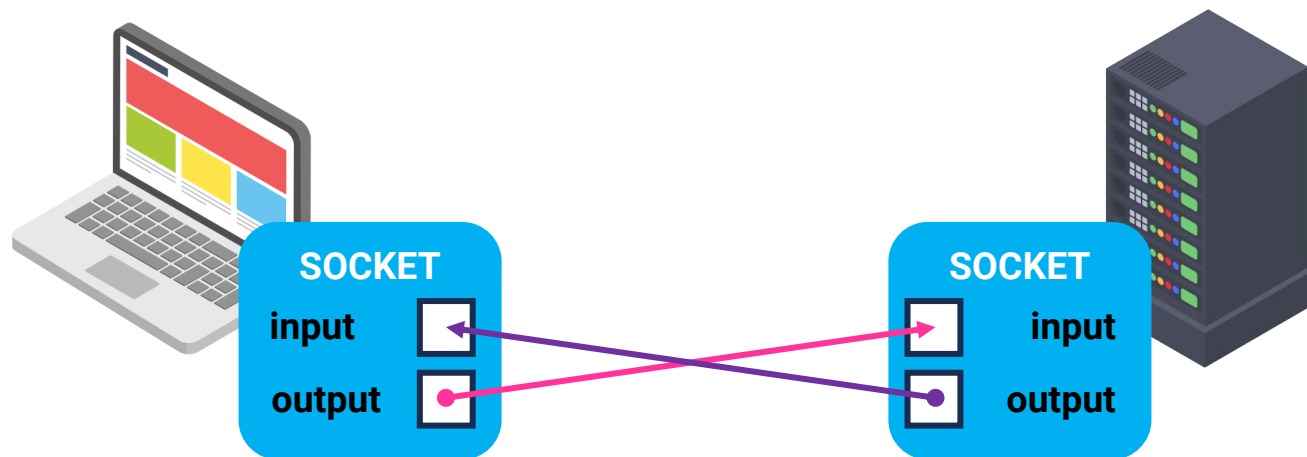
Gérer les connexions réseaux

Socket ?

Dans le cadre d'une application Client/Serveur, le client doit initier la connexion avec le service ciblé.

La communication est bidirectionnelle, c'est à dire que le client peut envoyer des données au serveur et en recevoir en retour.

La classe **Socket** est utilisée pour les communications bidirectionnelle utilisant le protocole TCP. HTTP est un protocole de communication situé au dessus de TCP.



Le client et le service dispose chacun d'une socket (ou équivalent) pour gérer la connexion qui les relie tous les deux. Une connexion HTTP ne relie qu'un seul client à un seul service.

Utilisation de la classe Socket

Pour ouvrir une connexion vers le serveur, le client doit initier la connexion en créant une socket indiquant **l'adresse du serveur** et **le port** sur lequel doit se faire la communication.

```
try {  
    Socket socketClient = new Socket("192.168.0.12", 80); ①  
  
    OutputStream output = socketClient.getOutputStream(); ②  
    InputStream input = socketClient.getInputStream();  
  
    // Echange d'information avec le serveur  
  
    socketClient.close(); ③  
}  
catch(Exception e)  
{  
    System.err.println(e.getMessage()); ④  
}
```

- ① Ouverture de la connexion avec le service
- ② La classe Socket fournit deux objets permettant de manipuler les flux d'entrée et de sortie.
- ③ Une fois l'échange avec le service terminé, les flux d'entrée/sortie et la connexion doivent être fermés.
- ④ L'ouverture de la connexion peut lever différents types d'exceptions (UnknownHostException, IOException) qu'il est nécessaire de capturer.

ServerSocket

Pour qu'un client puisse initier une connexion avec un service, il faut que ce dernier écoute un port en particulier (80 par défaut pour HTTP) et soit prêt à accepter la connexion du client.

Il est donc nécessaire de créer un point de communication réseau auquel les clients pourront se connecter. C'est le rôle de la classe **ServerSocket**.

```
ServerSocket serverSocket = new ServerSocket(80); 1
while(true)
{
    Socket clientSocket = serverSocket.accept(); 2
    // Traitement de la requête du client
    clientSocket.close() 3
}
```

1

Un point de communication est créé sur le port 80

2

Le service attend la connexion d'un client. Lorsqu'un client se connecte au service, un objet Socket représentant la connexion du service avec ce client est créé et retourné par la fonction **accept**.

Note : la fonction **accept** est bloquante.

3

Une fois le traitement terminé et la réponse envoyée, le service ferme la connexion avec le client et attend une nouvelle connexion.

Transmission d'information

Comme vu précédemment, un objet Socket fournit un flux de données en entrée et un flux de données en sortie :

```
Socket socket = new Socket("192.168.0.15", 80);

if(socket.isConnected())
{
    //Flux de sortie pour envoyer des informations
    OutputStream output = socket.getOutputStream();

    //Flux d'entrée pour recevoir des informations
    InputStream input = socket.getInputStream();
}
```

Les flux **OutputStream** et **InputStream** manipulent des octets bruts : **byte[]**. Utiles pour transmettre des données binaires (ex: une image) mais plus complexes lorsque l'on ne souhaite transférer que du texte.

Transmission d'information texte

Il est possible d'encapsuler les flux bruts pour gérer plus facilement des informations au format texte :

```
// OutputStreamWriter convertit un flux de caractères en un flux  
// d'octets selon un charset qui peut être spécifié  
// BufferedWriter permet de simplifier l'écriture de texte dans  
// le flux de caractères  
BufferedWriter output = new BufferedWriter(new OutputStreamWriter(s.getOutputStream()));  
  
//write permet d'envoyer une chaîne de caractères dans le flux de sortie  
output.write("GET /toto.php HTTP/1.0\n");  
  
//flush force l'envoi des données présentes dans le buffer vers le flux de sortie  
output.flush();
```

```
// InputStreamReader convertit un flux d'octets en un flux de  
// caractères selon un charset qui peut être spécifié  
// BufferedReader permet de simplifier la lecture de lignes ou  
// de blocs de texte depuis un flux de caractères  
BufferedReader input = new BufferedReader(new InputStreamReader(s.getInputStream()));  
  
// readLine permet de lire une ligne délimitée par un retour à la ligne (\n) ou la fin du flux de  
// données.  
String response = "";  
while((response = input.readLine()) != null && !response.isEmpty())  
    System.out.println(response);
```



Fichiers

Accès aux ressources du systèmes de fichiers

File

Java fournit une classe File pour manipuler les chemins de fichiers et obtenir des informations sur un fichier :

```
File fichier = new File("mon_fichier.txt");

//Si le fichier existe
if(fichier.exists())
{
    //Affiche la taille du fichier
    System.out.println("Taille du fichier : " + fichier.length());
}
else
{
    //Sinon crée le fichier
    fichier.createNewFile();
}
```


Fichiers texte

Pour lire simplement un fichier texte, il est possible d'utiliser la classe Scanner (qui permet également de saisir des données en provenance du clavier) :

```
File file = new File("mon_fichier.txt");

//Ouverture du fichier en lecture avec un
//objet Scanner
Scanner reader = new Scanner(file);

//Lecture du contenu du fichier texte
while(reader.hasNextLine())
{
    System.out.println(reader.nextLine());
}
```

Pour écrire dans un fichier texte, on peut utiliser à nouveau BufferedWriter (vu avec les sockets) et le flux créé par la classe FileWriter :

```
//Création du flux de caractères à partir du fichier
FileWriter writer = new FileWriter("mon_fichier.txt");

//Création du buffer d'écriture à partir du flux de
//caractères
BufferedWriter output = new BufferedWriter(writer);

//Ajout des informations dans le buffer avant envoi
//dans le fichier
output.write("Hello World");
output.newLine();
output.write("Fin du fichier");

//Force l'écriture dans le fichier du contenu du buffer
output.flush();

//Bien penser à fermer le fichier ouvert en écriture
output.close();
```

Données binaires

Dans le cas de fichiers binaires (comme des images), les classes `FileInputStream` et `FileOutputStream` seront utilisées :

```
//Ouverture du fichier en lecture et création d'un flux d'octets
FileInputStream input = new FileInputStream("source.png");

//Ouverture d'un fichier destination en écriture
FileOutputStream output = new FileOutputStream("target.png");

//Création d'un tableau d'octets permettant de lire le fichier par morceaux de 4096 octets
byte[] bytes = new byte[4096];

//Variable qui indiquera le nombre d'octets lus dans le fichier
int bytesRead = 0;

do {
    //Lecture d'au plus 4096 octets dans le fichier
    bytesRead = input.read(bytes);

    //Ecriture des données lues dans le fichier destination (on écrit juste le nombre d'octets lus)
    output.write(bytes, 0, bytesRead);
} while(bytesRead == 4096);
//Tant qu'il est possible de lire 4096 octets dans le fichier, c'est qu'il reste des choses à lire
```



Multithreading

Pour minimiser les temps de réponse

Retour aux sockets

Lors de la description de la classe `ServerSocket`, le code suivant avait été donné :

```
ServerSocket serverSocket = new ServerSocket(80);  
while(true)  
{  
    Socket clientSocket = serverSocket.accept();  
    // Traitement de la requête du client  
    clientSocket.close()  
}
```



Que se passe-t-il si le traitement de la requête du client prend beaucoup de temps ?

Le serveur ne traitant qu'une requête à la fois, les autres clients doivent attendre que la requête en cours soit traitée avant de pouvoir se connecter.
Pas ouf !

Les problèmes

```
ServerSocket serverSocket = new ServerSocket(80);  
while(true)  
{  
    Socket clientSocket = serverSocket.accept();  
    // Traitement de la requête du client  
    clientSocket.close()  
}
```



Au sein d'un processus, toutes les instructions du programme sont exécutées les unes à la suite des autres.

Ainsi, même en déplaçant le traitement de la requête dans une fonction, il faudrait attendre la fin de son exécution pour reprendre la suite de la boucle while.

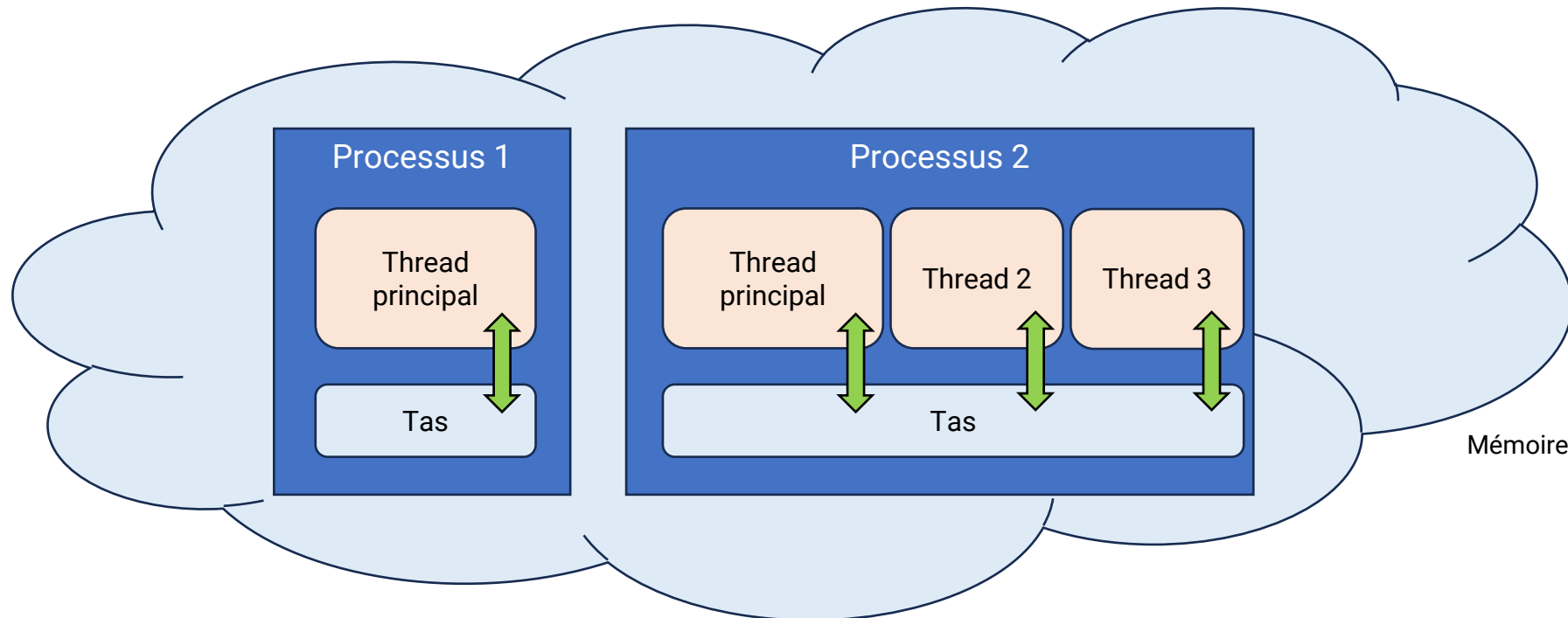
La solution : les Threads

Processus vs Thread

Pour exécuter un programme, la JVM crée un processus. Chaque processus possède son propre espace mémoire qui n'est pas partagé avec les autres processus.

Au sein de son espace mémoire, un processus peut contenir plusieurs threads. Ce sont des sortes de sous-processus qui vont s'exécuter parallèlement. Chaque thread à sa propre pile mais partage le tas avec les autres threads du processus.

Au démarrage de l'application, la JVM crée un thread principal chargé d'exécuter la fonction main.



Interface Runnable

Il est donc possible, en Java, de déplacer un traitement dans un nouveau thread, différent du thread principal. Ainsi ce dernier poursuit son exécution, tandis que le nouveau thread réalise son traitement, quelque soit le temps nécessaire.

Pour cela, le traitement va être déplacé dans une classe qui implémente l'interface Runnable :

```
class RequestProcessing implements Runnable
{
    private Socket clientSocket;

    RequestProcessing(Socket clientSocket)
    {
        this.clientSocket = clientSocket;
    }

    @Override
    public void run()
    {
        //Traitement à effectuer
    }
}
```

Et le thread principal créera un nouveau thread pour traiter chaque requête

```
:
ServerSocket serverSocket = new ServerSocket(80);
while(true)
{
    Socket clientSocket = serverSocket.accept();

    RequestProcessing request = new RequestProcessing(clientSocket);
    Thread thread = new Thread(request);
    thread.start();
}
```



Créer un serveur Web

Etes-vous prêt à créer votre propre serveur ?