Les sockets

Table des matières

1. Introduction....2
2. UDP....2
   Étape 0 : bibliothèques à inclure....3
   Étape 1 : création des descripteurs de la communication....4
   Étape 2 : réservation du port (coté serveur seul)....4
   Étape 3 : Le client envoie un datagramme au serveur....5
   Étape 3 bis : Le serveur reçoit le datagramme du client....6
   Fin de la communication....7
3. TCP....8
   Étape 1 : création des descripteurs de la communication....9
   Étape 2 : Enregistrement auprès du système d'exploitation (Serveur)....9
   Étape 3 : Déclaration du nombre maximal de connexions acceptées (Serveur)....10
   Étape 4 : Attente de connexions (Serveur)....10
   Étape 4 bis : Connexion du client (Client)....11
   Étape 5 : send / recv....11
   Étape 6 : fermeture de la connexion....12

Apparu dans les systèmes UNIX, un socket est un élément logiciel qui est aujourd’hui répandu dans la plupart des systèmes d’exploitation. Il s’agit d’une interface logicielle avec les services du système d’exploitation, grâce à laquelle un développeur exploitera facilement et de manière uniforme les services d’un protocole réseau.

Il lui sera ainsi par exemple aisé d’établir une session TCP, puis de recevoir et d’expédier des données grâce à elle. Cela simplifie sa tâche car cette couche logicielle, de laquelle il requiert des services en appelant des fonctions, masque le travail nécessaire de gestion du réseau, pris en charge par le système. Le terme socket désigne en pratique chaque variable employée dans un programme afin de gérer l’une des sessions.

1. Introduction

Ce document vous propose de découvrir l'interface de programmation socket qui permet à une application de communiquer en passant directement par la couche transport.

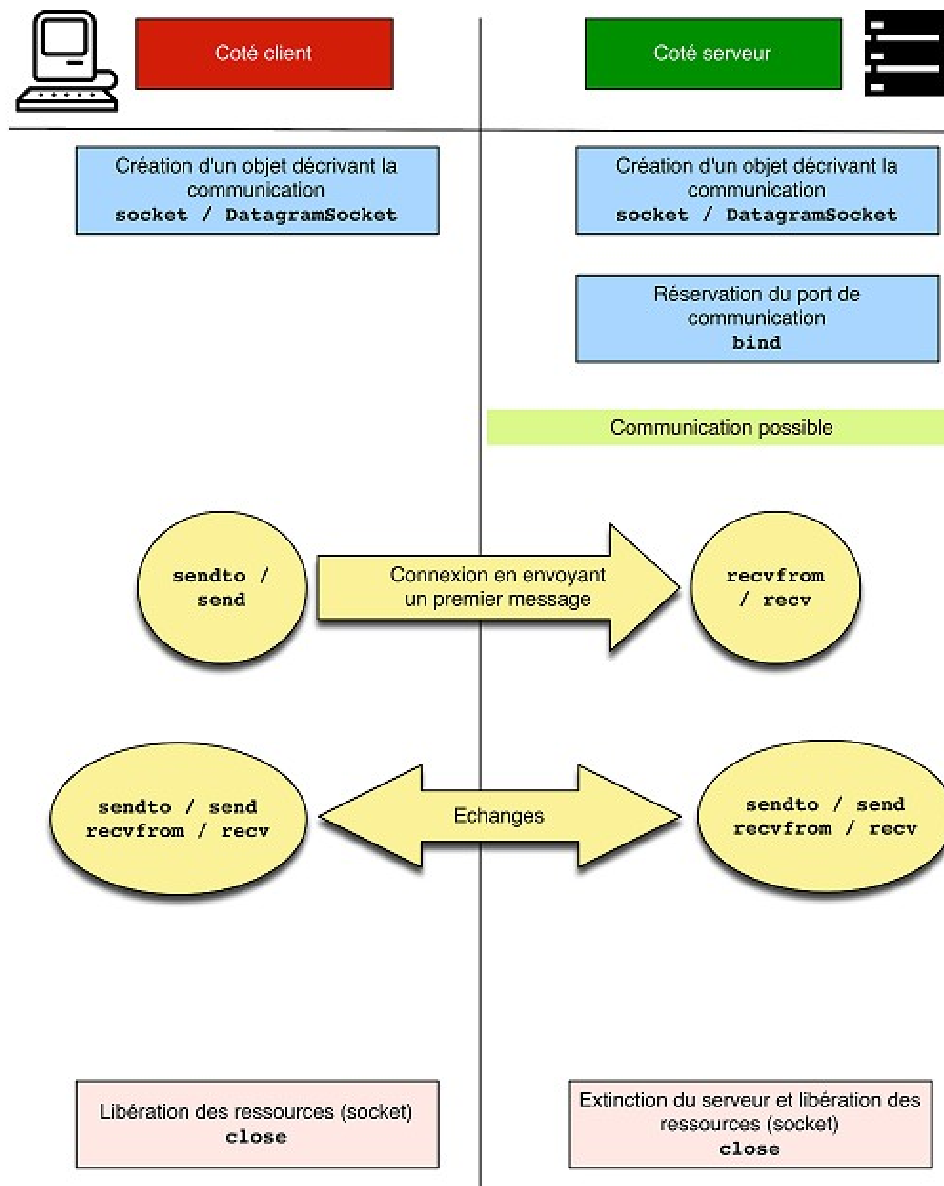
Il s'agit d'une interface bas niveau : elle ne permet que de communiquer des données sérialisées et non d'échanger des structures abstraites telles que des objets. Plusieurs interfaces de plus haut niveau permettent de définir une application distribuée de manière plus directe, en transmettant par exemple des objets complets comme les RMI de Java ou les bus à objets.

Cependant, comprendre le fonctionnement l'interface socket permet de comprendre comment un programme communique au travers du réseau et permet aussi d'imaginer comment fonctionnent des interfaces de plus haut niveau. En outre, le fonctionnement de cette interface est très proche de celui de la couche transpor : il y a une fonction pour l'ouverture et la fermeture de session, il est possible de sélectionner le mode de communication (datagramme ou connecté), etc.

Les documents présentés ici décrivent l'interface de programmation socket en faisant le lien avec les séances sur la couche transport que vous venez de suivre. Les exemples sont proposés en langages C, Java et Python, mais cette interface de programmation existe pour la quasi-totalité des langages modernes.

2. UDP

Nous allons tout d'abord voir, dans ce document, comment mettre en œuvre un dialogue entre un client et un serveur en **mode datagramme**. Le schéma ci-dessous illustre les différentes étapes que doivent accomplir un serveur (à droite) et un client (à gauche) pour pouvoir échanger des données en utilisant le protocole UDP.



Étape 0 : bibliothèques à inclure

Pour utiliser les fonctions de la bibliothèque *socket*, il est nécessaire d'importer une ou plusieurs bibliothèques de fonctions qui contiennent les primitives que nous utiliserons ci-dessous.

En Langage C	`#include <sys/socket.h>`
	`#include <netdb.h>`
	`#include <unistd.h>`

En Langage Java	`import java.net.*;` `import java.io.*;`
En Langage Python	`import socket`

Vous pourrez avoir besoin d'inclure d'autres bibliothèques, notamment stdio.h et string.h en langage C, pour afficher et manipuler les résultats.

Étape 1 : création des descripteurs de la communication

La première étape à réaliser, aussi bien par le serveur que par le client, est de créer une structure ou un objet qui va identifier la communication pour y faire référence par la suite. Un programme peut ouvrir de multiples connexions simultanément et il est, par conséquent, nécessaire d'être capable de les distinguer.

En langage C et en Python, nous devrons faire appel à la fonction **socket** qui renvoie un nombre entier qui servira d'identifiant de la communication. Cette fonction prend plusieurs arguments : le premier est le type de socket (PF_INET signifie que l'on souhaite créer un canal de communication TCP/IP et SOCK_DGRAM que l'on souhaite une communication en mode datagramme). En Java, on crée un objet appartenant à la classe **DatagramSocket** qui est directement configuré de la bonne manière.

En Langage C	`int s = socket (PF_INET, SOCK_DGRAM, 0);`
En Langage Java	`DatagramSocket s = new DatagramSocket();`
En Langage Python	`s = socket.socket (socket.AF_INET, socket.SOCK_DGRAM)`

Une fois cette étape réalisée, la variable *s* nous permettra de faire référence au canal de communication que l'on aura créé. C'est cette variable *s* que l'on va passer aux différentes fonctions pour mettre en œuvre les étapes de notre communication.

Étape 2 : réservation du port (coté serveur seul)

Le serveur, avant de pouvoir accepter des communications, devra demander au système d'exploitation de lui transmettre tous les datagrammes qui lui parviendront sur le port de communication choisi. Pour cela, il faudra réserver ce port auprès du système. Il est aussi possible de limiter l'attente à une adresse IP lorsque l'on est sur une machine en possédant plusieurs. Cette étape, nécessaire, passe par une fonction généralement appelée **bind** et spécifique au serveur. Les paramètres de cette fonction, selon les langages, peuvent lui être passés directement (Python) ou par l'intermédiaire d'une structure (C) ou d'un objet (Java).

En Langage C

```
// Structure contenant l'adresse et le port sur lesquels
écouter
//     Type d'adresse ; AF_INET = IPv4
//     Adresse du récepteur - INADDR_ANY = n'importe quelle
interface
//     Port sur lequel écouter
struct sockaddr_in myAddress;
myAddress.sin_family      = AF_INET;
myAddress.sin_addr.s_addr = htonl(INADDR_ANY);
myAddress.sin_port        = htons(12345);

// Enregistrement au niveau de l'OS
//     Paramètre 1 : descripteur de connexion
//     Paramètre 2 & 3 : adresse et taille de l'adresse
bind(s, (struct sockaddr *)&myAddress, sizeof(myAddress));
```

En Langage Java	`// Objet représentant l'adresse et le numéro de port sur lesquels écouter`
	`//     Paramètre 1 : Adresse IP sur laquelle écouter (null pour toutes)`
	`//     Paramètre 2 : port sur lequel écouter`
	`InetSocketAddress myAddress = new InetSocketAddress((InetAddress)null, 12345);`
	
	`// Enregistrement au niveau de l'OS`
	`s.bind(myAddress);`

En Langage Python	`myAddress = ''          # Écouter sur toutes les interfaces réseau`
	`myPort    = 12345          # Port sur lequel écouter`
	`s.bind((myAddress, myPort))`

Si l'appel à bind réussit, le serveur est maintenant prêt à recevoir les datagrammes entrants.

Étape 3 : Le client envoie un datagramme au serveur

Dès que le serveur est prêt à recevoir les datagrammes, le client peut lui envoyer un message en appelant directement la fonction d'envoi : **send** ou **sendto** selon les langages. L'appel à cette fonction et notamment ses paramètres est assez différent selon les langages. Cependant, elle a besoin au minimum qu'on lui spécifie le message à envoyer (une chaîne de caractères contenue dans la variable *message* dans nos exemples) et l'identité de l'application réceptrice. Cette identité, comme nous l'avons vu dans les vidéos, est composée de l'adresse de la machine hébergeant l'application serveur, ainsi que du port qui a été réservé par cette application.

Dans les exemples ci-dessous, on commence par récupérer l'adresse IP du serveur situé sur la machine locale (`localhost`), on prépare ensuite (si nécessaire) une structure contenant l'identification du serveur (adresse IP et numéro de port notamment). On prépare le message à envoyer, puis on l'envoie au moyen de la fonction idoine.

En Langage C	<pre>// Interrogation du DNS pour obtenir l'adresse IP de la destination
	struct hostent *destination = gethostbyname("localhost");
	in_addr_t destIPAddr = *((in_addr_t *)(destination->h_addr));
	
	// structure représentant l'adresse (+ numéro de port) de destination
	struct sockaddr_in destAddress;
	destAddress.sin_family = AF_INET;
	destAddress.sin_addr.s_addr = destIPAddr;
	destAddress.sin_port = htons(12345);
	
	// Préparation du message à envoyer
	char message[] = "Hello World\n";
	
	// Envoi effectif du message
	// Paramètre 1 : identificateur de socket
	// Paramètres 2 & 3 : message & longueur du message
	// Paramètre 4 : drapeaux pour transmissions particulières
	// Paramètres 5 & 6 : adresse destination et taille de la structure
	sendto(s, message ,strlen(message), 0, (struct sockaddr *)&destAddress, sizeof(destAddress));</pre>

En Langage Java	<pre>// Interrogation du DNS pour obtenir l'adresse IP de la destination
	InetAddress destination = InetAddress.getByName("localhost");
	
	// Objet représentant l'adresse
	// Paramètre. 1 : adresse IP
	// Paramètre 2 : numéro de port
	InetSocketAddress destIPAddr=new InetSocketAddress(destination, 12345);
	
	// Préparation du message à envoyer
	String message = "Hello World\n";
	byte[] payload = message.getBytes();
	DatagramPacket packet = new DatagramPacket(payload, payload.length, destIPAddr);
	
	// Envoi effectif du message
	s.send(packet);</pre>

En Langage Python	<pre>destIPAddr = "localhost"
	destPort = 12345
	message = "Hello, World\n"
	
	s.sendto(message, (destIPAddr, destPort))</pre>

Étape 3 bis : Le serveur reçoit le datagramme du client

Du coté du serveur, on se préparera à recevoir des données provenant du client en allouant un peu de mémoire pour le tampon de réception, puis en appelant la fonction **recvfrom**, qui est bloquante.

Cela veut dire que l'exécution du programme serveur s'arrêtera à cette ligne en attendant qu'un message parvienne à notre application.

Une fois que le serveur aura reçu un datagramme, il pourra utiliser le contenu du message et il disposera aussi de l'adresse de l'émetteur qui sera stockée dans une structure qu'il aura préalablement allouée si nécessaire. Cela permettra alors au serveur d'envoyer une réponse au client de la même manière.

En Langage C

```c
// Tampon de réception
char message[1024];
int nbCars;

// Descripteur pour stocker l'adresse de l'émetteur
struct sockaddr_in sourceAddr;
socklen_t length = sizeof(sourceAddr);

// Réception effective (bloquante) du message
//     Paramètre 1 : socket
//   Paramètres 2 & 3 : tampon de réception et taille de ce tampon
//   Paramètre 4 : drapeaux (inutilisé ici)


//   Paramètres 5 & 6 : adresse source & taille de l'adresse
nbCars = recvfrom (s, message, 1024, 0, (struct sockaddr *)&sourceAddr, &length);
```

En Langage Java

```java
//Tampon de réception

byte[] message = new byte[1024];

// Réception effective (bloquante) du message
DatagramPacket packet = new DatagramPacket(message, message.length);
s.receive(packet);

// Récupération de l'adresse de l'émetteur
InetSocketAddress sourceAddr =(InetSocketAddress)packet.getSocketAddress();
```

En Langage Python

```python
# Réception, taille du tampon = 1024 octets
message, sourceAddr = s.recvfrom(1024)
```

Fin de la communication

UDP fonctionnant en mode datagramme, la communication s'arrête dès que les deux correspondants ne transmettent aucune donnée. Il n'y a donc pas de fonction réelle pour mettre fin à la communication, mais il est toutefois nécessaire d'appeler une fonction pour libérer les ressources (mémoire etc.) qui ont été réservées par le système lorsque l'on n'a plus l'usage du canal de communication. Cette opération est réalisée au moyen d'une fonction nommée close.

En Langage C

```c
close(s);
```

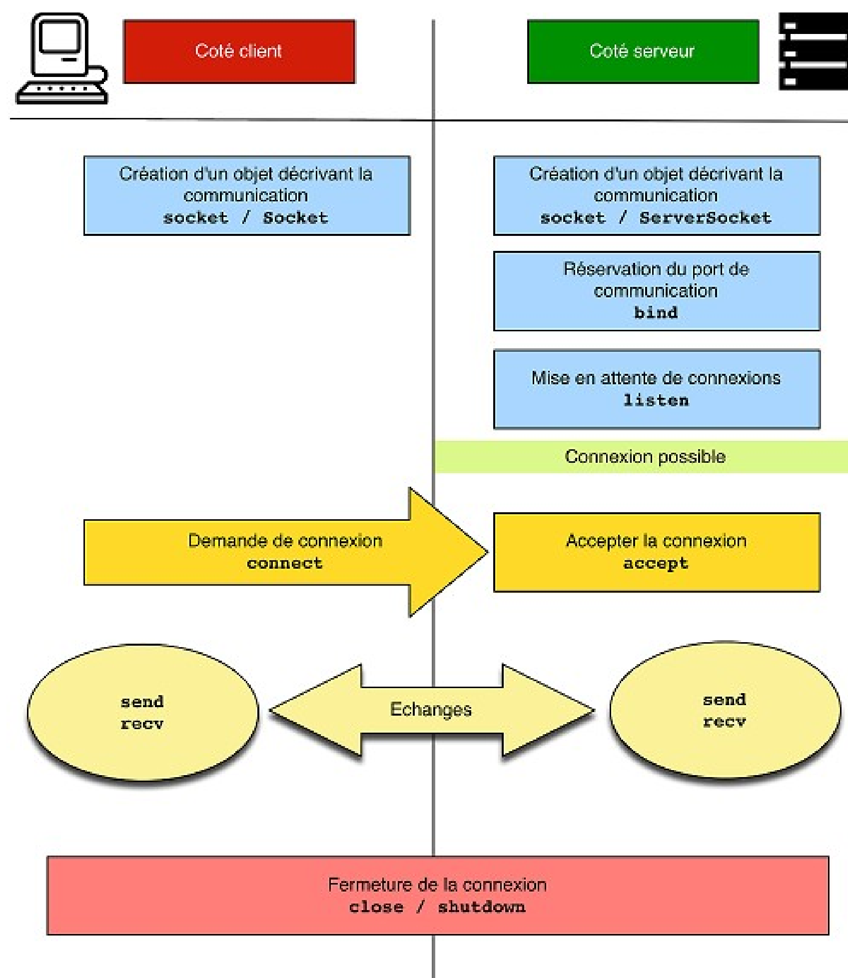
En Langage Java	`s.close();`
En Langage Python	`s.close()`

3. TCP

Nous allons maintenant voir comment mettre en œuvre un dialogue entre un client et un serveur en mode connecté. Le schéma ci-dessous illustre les différentes étapes à accomplir pour un serveur (à droite) et pour un client (à gauche).

On peut immédiatement remarquer que cet échange nécessite plus d'étapes qu'en mode datagramme. En effet, le mode connecté diffère du mode datagramme par deux aspects essentiels :

1. Il est nécessaire d'ouvrir une connexion (dialogue SYN-SYN ACK et ACK), et de la fermer (FIN - FIN ACK). On ne peut donc pas commencer directement par envoyer un segment.
2. Un serveur peut vouloir limiter le nombre de clients avec qui il interagit simultanément pour contrôler l'utilisation des ressources (mémoire, CPU, ...). En mode datagramme, la notion de connexion n'existant pas, la couche transport ne sait pas si une communication est terminée ou non. Elle ne peut pas gérer elle-même le nombre de clients connectés. En revanche, le mode connecté offre cette possibilité : au delà d'un nombre prédéterminé de clients, TCP refusera les nouvelles demandes de connexion.



Étape 1 : création des descripteurs de la communication

La première étape à réaliser est la même que dans le cas du mode datagramme : il s'agit de créer un objet capable de décrire la connexion. Cependant, à l'inverse du mode datagramme, le rôle du serveur et du client sont assez différents ici, ne serait-ce qu'à cause de l'échange d'ouverture de connexion qui n'est pas symétrique. Aussi en langage Java, le type de descripteur est différent sur le client et sur le serveur. Pour les langages C et Python, on utilisera une structure identique.

On peut noter dans les exemples ci-dessous que le type de *socket* créé est différent. Il s'agit d'une *socket* de type SOCK_STREAM, indiquant qu'on a cette fois un flux de données plutôt que des datagrammes isolés.

En Langage C	`int s = socket(PF_INET, SOCK_STREAM, 0);`
En Langage Java(coté serveur)	`// Cote serveur` `ServerSocket s = new ServerSocket(12345);`
En Langage Java (coté client)	`// Cote client` `Socket s = new Socket();`
En Langage Python	`s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`

Étape 2 : Enregistrement auprès du système d'exploitation (Serveur)

La deuxième étape réalisée par le serveur seul est, elle aussi, similaire à celle d'une communication datagramme. Il s'agit de demander au système d'exploitation de réserver des ressources et en particulier le port de communication. Elle s'effectue en appelant la même primitive : **bind**. Notez qu'en Java, cet appel est réalisé directement lors de l'appel au constructeur de la classe *ServerSocket*.

En Langage C

```
struct sockaddr_in myAddress;
myAddress.sin_family      = AF_INET;
myAddress.sin_addr.s_addr = htonl(INADDR_ANY);
myAddress.sin_port        = htons(12345);


// Enregistrement au niveau de l'OS
bind(s, (struct sockaddr *)&myAddress, sizeof(myAddress));
```

En Langage Java

(n'existe pas, réalisé au moment de l'appel au constructeur de la classe ServerSocket)

En Langage Python

```
myAddress = ''
myPort = 12345

s.bind((myAddress, myPort))
```

Étape 3 : Déclaration du nombre maximal de connexions acceptées (Serveur)

Une fois que le serveur a réservé les ressources au niveau du système d'exploitation, il va se mettre en attente de connexions. Pour cela, il fera appel à une fonction spécifique au mode connecté : **listen**. Cette fonction, qui n'est pas nécessaire avec les ServerSocket de Java, permet de spécifier le nombre de connexions maximal que TCP pourra tolérer sur ce port, c'est-à-dire de limiter le nombre de clients que l'application voudra gérer simultanément.

En Langage C	`// Démarrage de l'écoute active (non bloquante) des demandes de connexion`
	`//     Parametre 1 : socket`
	`//     Parametre 2 : nombre maximum de connexions`
	
	`listen(s, 5);`

En Langage Java	(n'existe pas, réalisé au moment de l'appel au constructeur de la classe ServerSocket)
En Langage Python	`s.listen(5)`

Étape 4 : Attente de connexions (Serveur)

Le serveur peut donc accepter plusieurs connexions en parallèle. Pour cela, il va créer dynamiquement un objet socket à chaque nouvelle connexion, laissant à l'objet principal l'unique tâche de recevoir les nouvelles demandes de connexion. Cette création s'effectue automatiquement au moyen de la fonction **accept** qui est appelée après l'appel à **listen** (inutile d'attendre qu'une connexion arrive effectivement). Dans les exemples ci-dessous, le nouvel objet socket est appelé *sData* et l'adresse du correspondant, qu'il est possible de récupérer de manière analogue au mode datagramme, est appelée *senderAddress*.

En Langage C	`struct sockaddr_in senderAddress;`
	`socklen_t length = sizeof(senderAddress);`
	
	`// Création d'une nouvelle socket dédiée à la communication`
	`int sData = accept (s, (struct sockaddr *)&senderAddress, &length);`

En Langage Java	`// Création d'une nouvelle socket dédiée à la communication`
	
	`Socket sData = s.accept();`
	
	`InetSocketAddress senderAddress=(InetSocketAddress)sData.getRemoteSocketAddress();`

En Langage Python	`sData, senderAddress = s.accept()`

Étape 4 bis : Connexion du client (Client)

Le client, contrairement au mode datagramme, ne peut se contenter d'envoyer un message pour démarrer la connexion. En effet, il est nécessaire de passer par le dialogue d'ouverture de connexion. Le client devra donc explicitement faire appel à une fonction appelée **connect** avant de pouvoir transférer des données.

En Langage C

```
// Recuperaton de l'adresse IP de la destination
struct hostent *destination = gethostbyname("localhost");
in_addr_t destIPAddr = *((in_addr_t *)(destination->h_addr));

// Creation de la structure identifiant l'application destination
struct sockaddr_in destAddress;
destAddress.sin_family      = AF_INET;
destAddress.sin_addr.s_addr = destIPAddr;
destAddress.sin_port        = htons(12345);

// Ouverture de la connexion
connect(s, (struct sockaddr *)&destAddress, sizeof(destAddress));
```

En Langage Java

```
// Interrogation du DNS pour obtenir l'adresse IP de la destination
InetAddress destination = InetAddress.getByName("localhost");

// Objet représentant l'adresse
//     Paramètre 1 : adresse IP
//     Paramètre 2 : numéro de port
InetSocketAddress destIPAddr=new InetSocketAddress(destination, 12345);

// Ouverture de la connexion

s.connect(destIPAddr);
```

En Langage Python

```
destIPAddr = "localhost"
destIPPort = 12345

s.connect((destIPAddr, destIPPort))
```

Étape 5 : send / recv

Une fois que les appels à **accept** et **connect** ont abouti, les deux correspondants peuvent échanger des données au moyen des fonctions **send** et **recv**.

En Langage C	`// Emission de donnees`
	`char messageEmis[] = "Hello World\n";`
	`send(sData, messageEmis ,strlen(messageEmis), 0);`
	
	`// Reception de donnees`
	`char messageRecu[1024];`
	`int nbCars;`
	`nbCars = recv (sData, messageRecu, 1024, 0);`

En Langage Java	`// Emission de donnees`
	`String messageEmis = "Hello World\n";`
	`PrintWriter bufferEmission = new PrintWriter(new BufferedWriter(new OutputStreamWriter(sData.getOutputStream())), true);`
	`bufferEmission.println(messageEmis);`
	
	`// Reception de donnees`
	`BufferedReader bufferReception = new BufferedReader(new InputStreamReader(sData.getInputStream()));`
	`String messageRecu = bufferReception.readLine();`

En Langage Python	`# Emission de donnees`
	`sData.send("Hello World\n")`
	
	`# Reception de donnees`
	`messageRecu = sData.recv(1024)`

Étape 6 : fermeture de la connexion

Enfin, lorsque l'intégralité des données a été transmise, il est nécessaire de terminer explicitement la connexion en appelant la fonction **close**. Celle-ci va déclencher la procédure de fermeture de connexion (envoi des messages FIN et FIN ACK). Notez qu'il est possible de ne fermer qu'une direction de la communication au moyen de la fonction **shutdown**, mais cette possibilité est, en général, peu utilisée.

En Langage C	`close(sData);`
	
	`close(s);`

En Langage Java	`sData.close();`
	
	`s.close();`

En Langage Python	`sData.close()`
	
	`s.close()`