

# Programmation client-serveur

Arnaud Giersch

novembre 2022

## Résumé

L'objectif de ce document est de synthétiser les points essentiels à connaître et à comprendre pour pouvoir faire les TPs de programmation client-serveur en S3. On commence par exposer les principes généraux en section 1. Un exemple est ensuite développé en section 2. Les éléments essentiels de programmation sont donnés en section 3, puis utilisés en section 4 pour écrire les programmes de l'exemple. L'exécution des programmes exemples est exposée en section 5. Enfin, en section 6 deux solutions sont décrites pour compléter le programme serveur de manière à servir plusieurs clients. Quelques conseils pratiques sont finalement donnés en section 7.

## 1 Généralités

En programmation client-serveur, on a généralement à écrire deux programmes : un programme appelé *serveur* et un programme appelé *client*. Ces deux programmes sont amenés à établir un dialogue à travers une *connexion réseau*. Cela signifie que les deux programmes peuvent être exécutés sur des systèmes (machines) différents, et que le réseau va être utilisé pour communiquer.

Dans le cadre des TPs de réseau, les communications se feront normalement en utilisant le protocole TCP/IP qui est un protocole *connecté* et *fiable*. Ce protocole est dit fiable car les données envoyées sont reçues dans l'ordre, sans perte, ni duplication. C'est seulement pour le dernier TP que le protocole non connecté UDP sera abordé.

Pour pouvoir communiquer en TCP, il faut commencer par établir une connexion. Pour cela, le programme qui a le rôle de *serveur* se met état d'attendre une demande de connexion. Le programme *client* va alors demander à se connecter au serveur. Lorsque ce dernier accepte la demande de connexion, la connexion est établie et le dialogue peut commencer<sup>1</sup>.

Le client et le serveur dialoguent ensuite suivant un *protocole* qui définit la nature et l'ordre des messages échangés. Le protocole dépend du problème à résoudre, et est à définir avant d'écrire les programmes. Bien souvent les protocoles simples auront la forme suivante :

1. la connexion est établie ;
2. le client envoie une requête (il pose une question), et attend la réponse du serveur ;
3. le serveur reçoit la requête du client, construit sa réponse et envoie la réponse au client ;
4. le client reçoit la réponse du serveur ;
5. la connexion est fermée (terminée).

Les principales caractéristiques des programmes client et serveur sont ainsi :

**serveur** : le programme serveur est celui qui attend les demandes de connexion. Normalement, il n'y a qu'un seul processus serveur qui est exécuté. Le serveur est généralement écrit pour pouvoir répondre à un nombre infini de clients. On peut distinguer les serveurs *itératifs* qui répondent aux clients les uns après les autres, et les serveurs *parallèles*, plus évolués, qui sont capables de répondre à plusieurs clients en même temps.

**client** : le programme client est celui qui fait la demande de connexion au serveur. On peut exécuter plusieurs processus clients pour un seul serveur.

---

1. On peut remarquer que ce fonctionnement est analogue à une conversation téléphonique. Avant de pouvoir dialoguer, il faut établir une connexion où l'un des interlocuteurs (le client) appelle l'autre (le serveur). Lorsque le serveur répond, la connexion est établie et elle reste active jusqu'à ce qu'un des interlocuteurs raccroche.

**Numéro de port.** Les connexions TCP se font entre les systèmes en utilisant les adresses IP et des numéros de port. Un numéro de port est un nombre entier sur 16 bits (1–65535). Une connexion TCP est alors identifiée de manière unique par un quadruplet (adresse source, port source, adresse destination, port destination).

## 2 Exemple

Pour la suite, nous allons développer un exemple simple : un serveur de calcul qui doit permettre de calculer des additions.

Le serveur commence par se mettre en attente de connexions sur le port n° 4242. Lorsque le client se connecte au serveur, il lui envoie deux nombres entiers. Le serveur reçoit ces deux valeurs, en calcule la somme puis retourne le résultat au client. Les valeurs envoyées par le programme client seront demandées à l'utilisateur, et le résultat sera affiché.

Le déroulement de l'exécution est représenté par les figures 1 à 6 (pages 3 et suivante). Au départ, on a deux machines interconnectées par un réseau, et d'adresses IP respectives 172.20.178.54 et 172.20.178.69 (cf. figure 1). Les adresses sont bien sûr fictives et servent uniquement pour l'exemple. Le programme serveur est alors démarré sur la machine 172.20.178.54. Il se met en attente de connexions sur le port n° 4242 (figure 2).

Sur la figure 3, le programme client est démarré sur l'autre machine. Il prend ses valeurs depuis l'utilisateur, puis effectue une connexion au serveur en utilisant l'adresse IP et le numéro de port du serveur (172.20.178.54:4242). Le client envoie ensuite les valeurs au serveur (figure 4). Le serveur reçoit les valeurs envoyées.

Ensuite, figure 5, le serveur calcule sa réponse et la retourne au client en utilisant la même connexion. Le client reçoit la réponse du serveur, et termine en l'affichant sur sa console (figure 6).

La section suivante présente les fonctions de bibliothèque utilisées dans les extraits de code qui apparaissent sur les figures.

## 3 Mise en œuvre

L'objet système associé à une connexion réseau est appelé *socket*. Les sockets correspondent à des descripteurs de fichiers bas-niveau et sont donc représentées par des nombres entiers. Elles peuvent être utilisées comme des descripteurs de fichiers bas-niveau standards.

La plupart des exercices de TP sont à programmer en utilisant la bibliothèque client-serveur fournie. Les fonctions de la bibliothèque sont données ci-dessous. L'utilisation directe des fonctions standard pour ouvrir des connexions fera l'objet d'un sujet de TP ultérieur.

Lorsqu'il est disponible, le paramètre `debug` peut prendre une valeur entre 0 et 2 pour activer des messages de débogage : plus la valeur est grande, plus détaillés seront les messages.

```
#include "client_serveur.h"
```

Fichier d'en-tête à inclure pour utiliser la bibliothèque client-serveur.

```
int creer_serveur_tcp(int port, int debug)
```

Cette fonction permet de créer un serveur TCP en attente de connexion sur le numéro de port donné. Les paramètres sont :

**port** numéro de port sur lequel se mettre en écoute.

**debug** active des messages de débogage si différent de 0.

La valeur retournée par la fonction est la socket d'écoute, ou `< 0` en cas d'erreur. C'est la socket serveur à passer à la fonction `attendre_client_tcp`.

```
int attendre_client_tcp(int socket_serveur, int debug)
```

Cette fonction, à utiliser par le serveur, permet d'attendre la connexion d'un client. L'exécution est bloquée jusqu'à ce qu'un client se connecte. Les paramètres sont :

**socket\_serveur** socket serveur à utiliser. Cette socket a été retournée par la fonction `creer_serveur_tcp`.

**debug** active des messages de débogage si différent de 0.

La valeur retournée par la fonction est la socket pour la connexion avec le client, ou `< 0` en cas d'erreur. C'est cette socket qui sera utilisée pour communiquer avec le client.

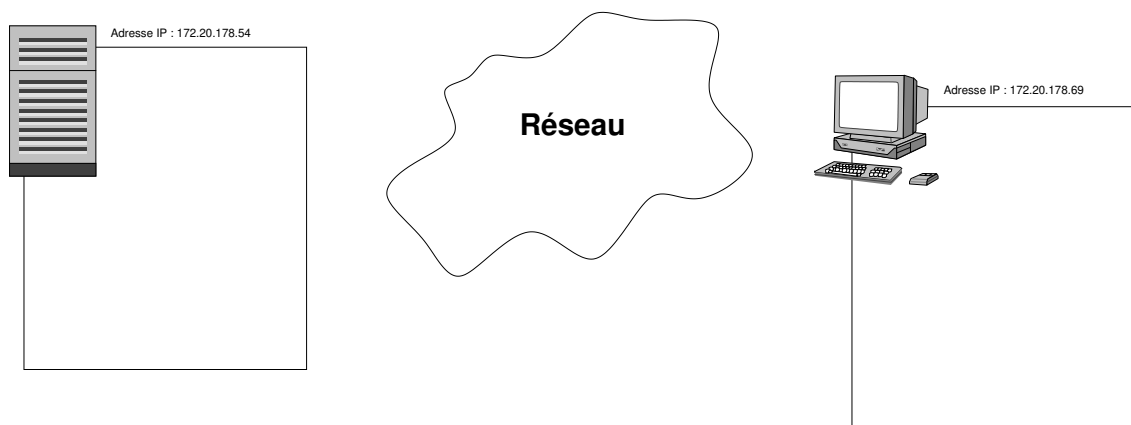


FIGURE 1 – Deux machines quelconques, connectées au réseau

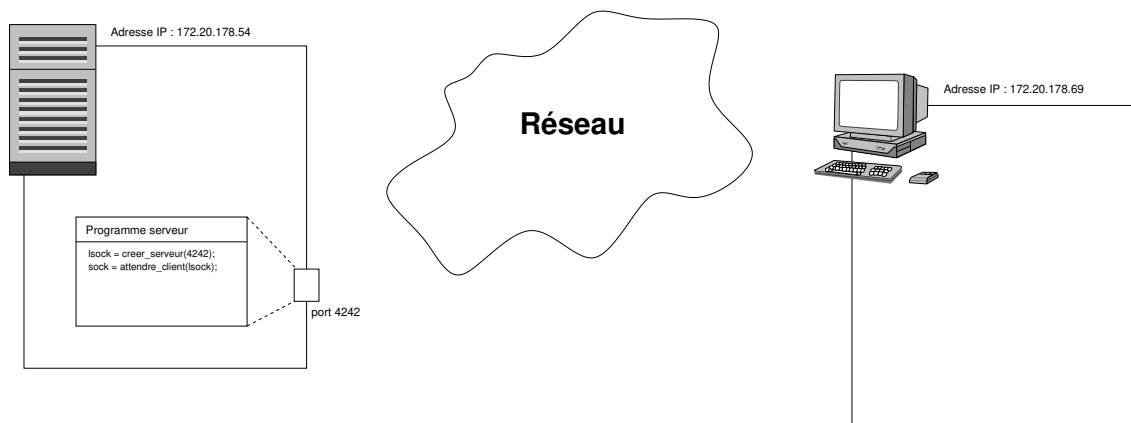


FIGURE 2 – Le programme serveur est démarré sur la machine 172.20.178.54. Il se met en attente de connexions sur le port n° 4242.

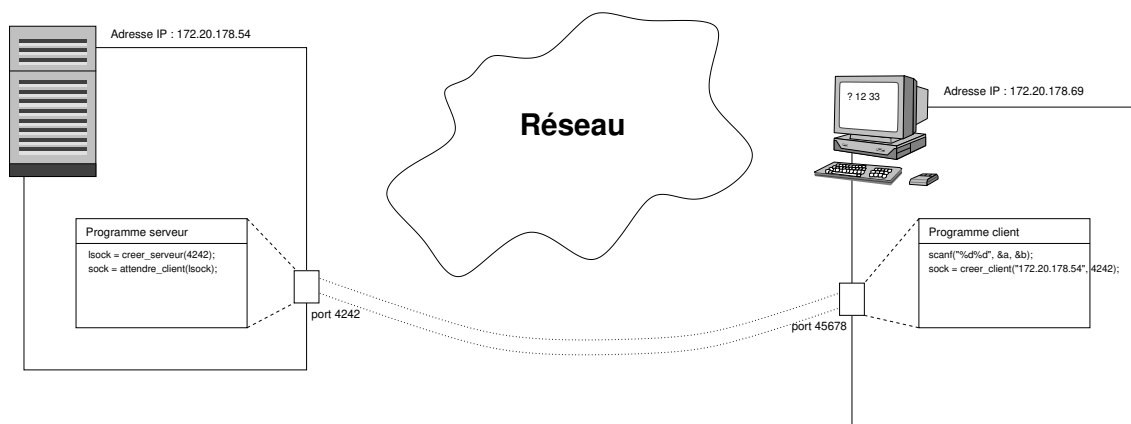


FIGURE 3 – Le programme client est démarré sur la machine 172.20.178.69. Il initie une connexion vers le serveur (adresse 172.20.178.54, port 4242).

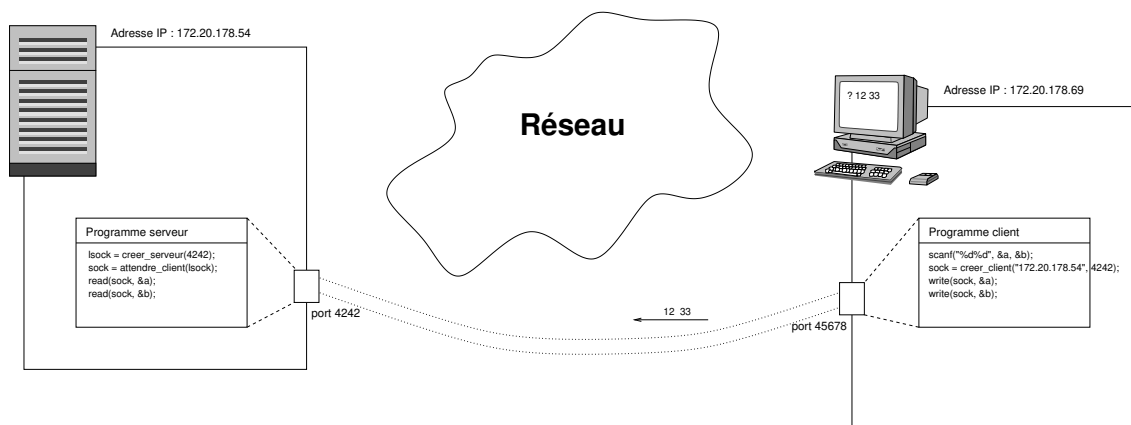


FIGURE 4 – Le programme client envoie sa requête au serveur.

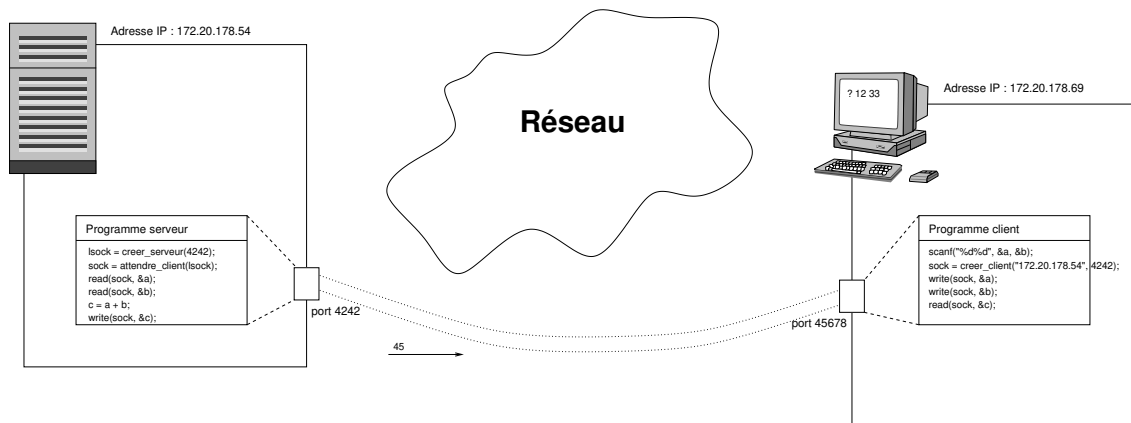


FIGURE 5 – Le programme serveur calcule et retourne sa réponse au client.

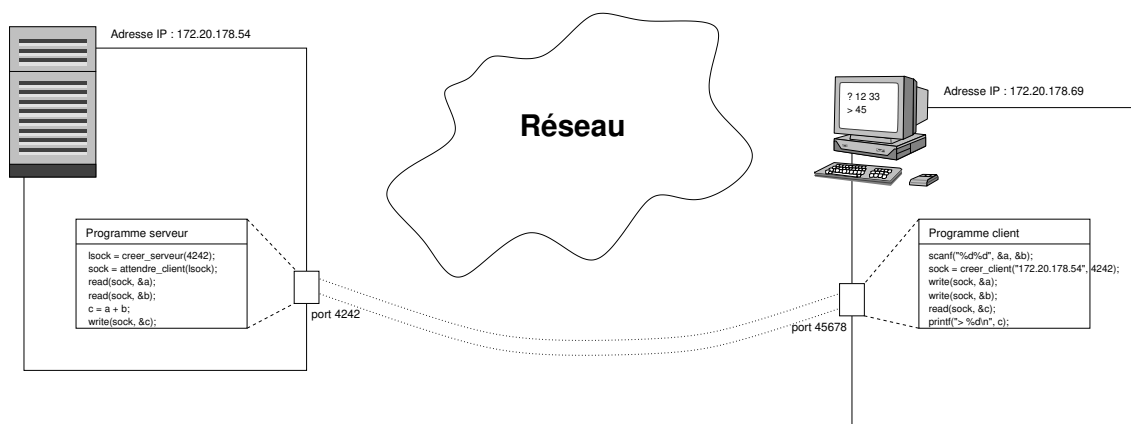


FIGURE 6 – Le programme client affiche la réponse reçue.

```
int creer_client_tcp(const char *nom, int port, int debug)
```

Cette fonction, à utiliser par le client, permet d'effectuer une connexion TCP vers un serveur donné. Les paramètres sont :

**nom** nom ou adresse IP du serveur sous forme de chaîne de caractères (ex. : "example.net" ou "172.20.178.54").

**port** numéro de port du serveur.

**debug** active des messages de débogage si différent de 0.

La valeur retournée par la fonction est la socket pour la connexion avec le serveur, ou  $< 0$  en cas d'erreur. C'est cette socket qui sera utilisée pour communiquer avec le serveur.

**Communications.** Les sockets retournées par les fonctions permettent de communiquer en utilisant la connexion qui a été établie. En particulier, il est possible d'effectuer des lectures/écritures en utilisant les primitives `read(2)` et `write(2)`. Une opération `read` correspond alors à la réception de données, et une opération `write` à l'envoi de données.

**Fermeture.** La connexion est terminée en fermant la socket associée avec la fonction `close(2)`.

## 4 Écriture des programmes

Comme nous l'avons vu dans l'exemple, il y a deux programmes à écrire : un programme client et un programme serveur. Normalement, les deux programmes doivent être terminés avant de pouvoir les tester. On écrira donc souvent les deux programmes simultanément.

Cependant, si le protocole de communication est complètement et correctement spécifié, il est possible d'écrire les deux programmes de manière indépendante. Dans ce cas, un exercice intéressant est de se mettre en binôme, l'un écrivant le client et l'autre le serveur, puis de faire communiquer les deux programmes. On peut ensuite inverser les rôles, et écrire l'autre partie (celui qui a commencé par programmer le client écrit le serveur, et réciproquement).

**Programme serveur.** Le programme 1 (page 6) donne une implémentation possible pour le programme serveur de l'exemple. On y retrouve, aux lignes 24, 26, 29, 31, 34 et 36, les instructions essentielles qui étaient présentes dans les figures. Dans ce programme, ligne 35, un message de diagnostic supplémentaire est affiché sur la console du serveur pour indiquer la requête reçue.

Pour le reste, on trouve ligne 1 l'inclusion des fichiers d'en-tête. La constante `DEBUG` (ligne 6) permet de définir facilement si on souhaite ou non les messages de débogage de la bibliothèque. La macro `VERIFIER` (ligne 8) est utilisée pour vérifier rapidement les codes de retour des fonctions, et interrompre le programme lorsqu'une erreur est détectée.

Le numéro de port utilisé par le serveur doit être donné sur la ligne de commande à l'exécution (cf. lignes 16 à 20). Enfin, les sockets sont correctement fermées avant de terminer le programme (instructions `close`, lignes 39 et 41).

**Programme client.** Une implémentation possible pour le programme client de l'exemple est donnée par le programme 2 (page 7). Comme pour le programme serveur, on y retrouve, aux lignes 26, 28, 31, 33, 36 et 40, les instructions essentielles qui étaient présentes dans les figures.

Le fichiers d'en-tête, la constante `DEBUG` et la macro `VERIFIER` sont les mêmes que dans le programme serveur.

L'adresse du serveur et son numéro de port doivent ici aussi être donnés sur la ligne de commande à l'exécution (cf. lignes 16 à 21). De même, la socket est correctement fermée à la fin de la communication (instruction `close`, ligne 39).

## 5 Compilation et exécution

Les programmes sont compilés en suivant les instructions de compilation de la page des TPs sur <https://cours-info.iut-bm.univ-fcomte.fr> :

```
$ gcc -g -Og -Wall -Wextra exemple_serveur.c client_serveur.a -o exemple_serveur
$ gcc -g -Og -Wall -Wextra exemple_client.c client_serveur.a -o exemple_client
```

```

1 #include "client_serveur.h"
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 #define DEBUG 0
7
8 #define VERIFIER(expr) \
9     if (!(expr)) { \
10         fprintf(stderr, "%s:%d: erreur: %s\n", __FILE__, __LINE__, #expr); \
11         exit(2); \
12     }
13
14 int main(int argc, char *argv[])
15 {
16     if (argc != 2) {
17         fprintf(stderr, "Usage: %s port\n", argv[0]);
18         return 1;
19     }
20     int port = atoi(argv[1]);
21     int a, b, c;
22     int ret;
23
24     int lsock = creer_serveur_tcp(port, DEBUG);
25     VERIFIER(lsock != -1);
26     int sock = attendre_client_tcp(lsock, DEBUG);
27     VERIFIER(sock != -1);
28
29     ret = read(sock, &a, sizeof(int));
30     VERIFIER(ret == sizeof(int));
31     ret = read(sock, &b, sizeof(int));
32     VERIFIER(ret == sizeof(int));
33
34     c = a + b;
35     printf("# calcul de %d + %d -> %d\n", a, b, c);
36     ret = write(sock, &c, sizeof(int));
37     VERIFIER(ret == sizeof(int));
38
39     ret = close(sock);
40     VERIFIER(ret == 0);
41     close(lsock);
42
43     return 0;
44 }

```

Programme 1 – exemple\_serveur.c

```

1 #include "client_serveur.h"
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 #define DEBUG 0
7
8 #define VERIFIER(expr) \
9     if (!(expr)) { \
10         fprintf(stderr, "%s:%d: erreur: %s\n", __FILE__, __LINE__, #expr); \
11         exit(2); \
12     }
13
14 int main(int argc, char *argv[])
15 {
16     if (argc != 3) {
17         fprintf(stderr, "Usage: %s serveur port\n", argv[0]);
18         return 1;
19     }
20     char *serveur = argv[1];
21     int port = atoi(argv[2]);
22     int a, b, c;
23     int ret;
24
25     printf("? ");
26     ret = scanf("%d%d", &a, &b);
27     VERIFIER(ret == 2);
28     int sock = creer_client_tcp(serveur, port, DEBUG);
29     VERIFIER(sock != -1);
30
31     ret = write(sock, &a, sizeof(int));
32     VERIFIER(ret == sizeof(int));
33     ret = write(sock, &b, sizeof(int));
34     VERIFIER(ret == sizeof(int));
35
36     ret = read(sock, &c, sizeof(int));
37     VERIFIER(ret == sizeof(int));
38
39     close(sock);
40     printf("> %d\n", c);
41
42     return 0;
43 }

```

Programme 2 – exemple\_client.c

Bien que le résultat ne soit pas très visuel, une capture d'écran présentant une exécution est donnée par la figure 7 (page 9). On peut quand même remarquer que la requête du client est correctement reçue par le serveur. Dans chaque terminal, la première commande affiche l'adresse IP de la machine.

Dans le terminal du haut, on peut voir la compilation et l'exécution du programme serveur sur la machine 172.20.178.54. Le numéro de port 4242 est donné sur la ligne de commande.

Dans le terminal du bas, on peut voir la compilation et l'exécution du programme client sur une autre machine (ici 172.20.178.69). L'adresse IP et le numéro de port du serveur sont donnés sur la ligne de commande.

## 6 Servir plusieurs clients

Le programme serveur développé dans l'exemple ne sait servir qu'un seul client. Généralement on souhaitera un programme capable de servir plusieurs clients. On présente ici deux manières de procéder. D'abord en écrivant un serveur itératif, servant les différents clients les uns après les autres. Ensuite en écrivant un serveur parallèle, capable de servir plusieurs clients en même temps.

### Serveur itératif

Une première solution est donc d'écrire un *serveur itératif* servant les différents clients les uns après les autres. Pour cela, il suffit de compléter le programme avec une boucle pour recommencer l'attente d'un client. La nouvelle structure du programme est présentée par le programme 3 ci dessous.

```
23  ...
    int lsock = creer_serveur_tcp(port, 16, DEBUG);
    while (1) {
        int sock = attendre_client_tcp(lsock, DEBUG);
        // *** dialogue avec le client, en utilisant la socket 'sock'
        close(sock);
    }
    close(lsock);
```

Programme 3 – Structure d'un serveur itératif.

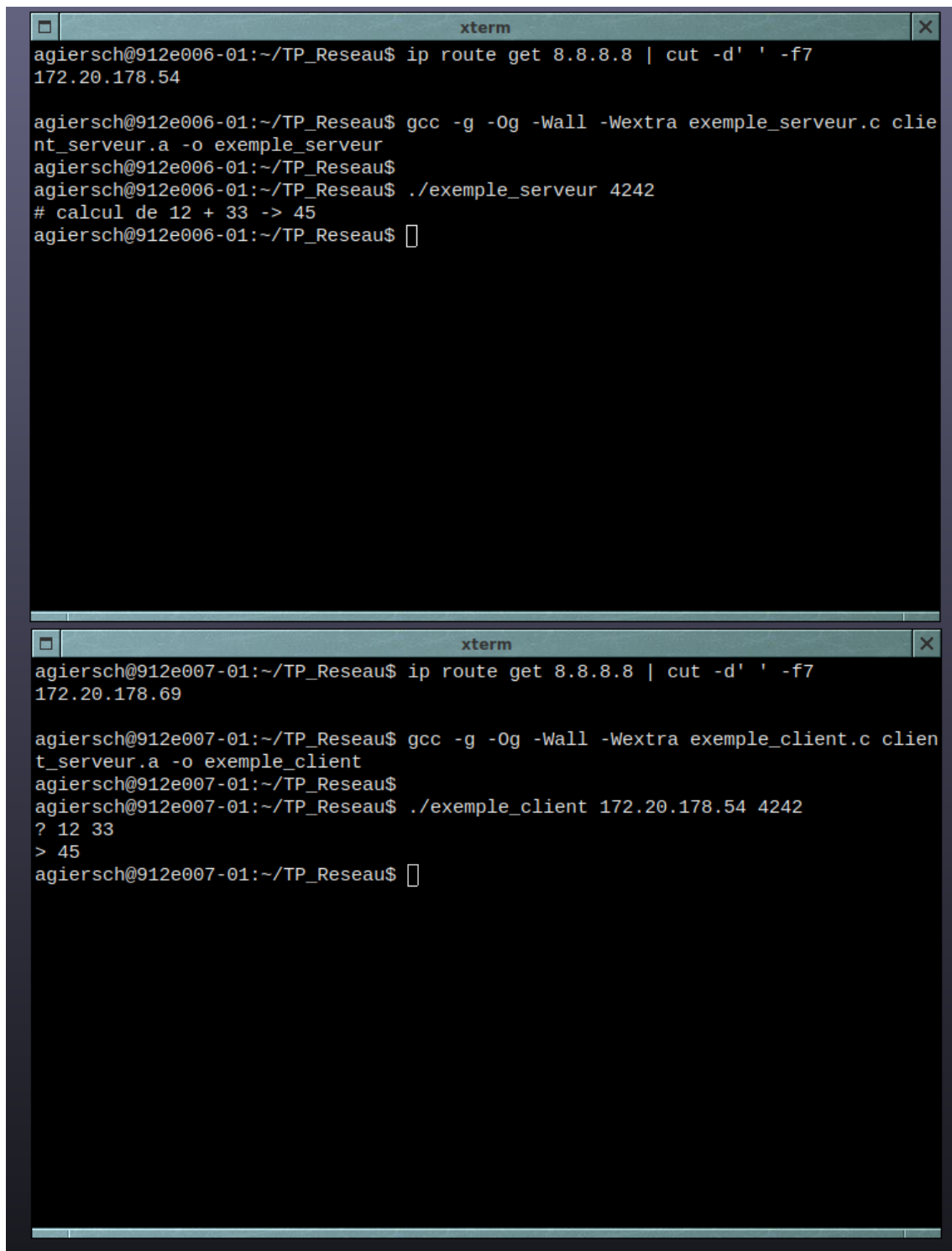
### Serveur parallèle

Une autre solution, plus sophistiquée, est d'écrire un *serveur parallèle* capable de servir plusieurs clients en même temps. Pour cela, dès que la connexion avec un client est acceptée, un processus fils est créé avec `fork()` pour gérer la connexion. Cette solution est ébauchée par le programme 4 ci dessous.

```
23  ...
    int lsock = creer_serveur_tcp(port, 16, DEBUG);
    while (1) {
        int sock = attendre_client_tcp(lsock, DEBUG);
        if (fork() == 0) {
            // processus fils
            close(lsock);
            // *** dialogue avec le client, en utilisant la socket 'sock'
            exit(0);
        }
        close(sock);
    }
    close(lsock);
```

Programme 4 – Structure d'un serveur parallèle.





The image displays two terminal windows, each titled 'xterm'. The top window shows the execution of a server program. The user runs a command to get the IP route to 8.8.8.8, which returns 172.20.178.54. Then, the user compiles a C program named 'exemple\_serveur.c' with the linker script 'client\_serveur.a' into an executable named 'exemple\_serveur'. Finally, the user runs the executable with the argument '4242', which outputs the result of the calculation 12 + 33, which is 45. The bottom window shows the execution of a client program. The user runs the same IP route command, which returns 172.20.178.69. Then, the user compiles a C program named 'exemple\_client.c' with the linker script 'client\_serveur.a' into an executable named 'exemple\_client'. Finally, the user runs the executable with the arguments '172.20.178.54' and '4242'. The program outputs a question mark, followed by the numbers '12 33', and then the result '> 45'.

```
agiersch@912e006-01:~/TP_Reseau$ ip route get 8.8.8.8 | cut -d' ' -f7
172.20.178.54

agiersch@912e006-01:~/TP_Reseau$ gcc -g -Og -Wall -Wextra exemple_serveur.c client_serveur.a -o exemple_serveur
agiersch@912e006-01:~/TP_Reseau$ ./exemple_serveur 4242
# calcul de 12 + 33 -> 45
agiersch@912e006-01:~/TP_Reseau$
```

```
agiersch@912e007-01:~/TP_Reseau$ ip route get 8.8.8.8 | cut -d' ' -f7
172.20.178.69

agiersch@912e007-01:~/TP_Reseau$ gcc -g -Og -Wall -Wextra exemple_client.c client_serveur.a -o exemple_client
agiersch@912e007-01:~/TP_Reseau$ ./exemple_client 172.20.178.54 4242
? 12 33
> 45
agiersch@912e007-01:~/TP_Reseau$
```

FIGURE 7 – Exemple d’exécution des programmes serveur (en haut) et client (en bas).

## 7 Conseils pratiques

**Exécution.** Même si ça peut être plus intéressant, il n'est pas nécessaire d'utiliser deux machines distinctes pour exécuter les programmes. Il est tout à fait possible d'exécuter le client et le serveur dans deux terminaux sur un même système. On utilisera alors l'adresse IP de bouclage 127.0.0.1.

**Socket utilisée.** Parfois l'exécution du serveur peut échouer avec une erreur du type :

```
# serveur> ! erreur bind(): Address already in use (errno = 98)
```

Cela peut signifier que le numéro de port demandé est déjà utilisé ; il faut choisir un autre numéro. Ça peut également arriver lorsque le programme serveur est redémarré trop rapidement. En effet, le système peut interdire de réutiliser tout de suite un numéro de port. Dans ce cas, la solution est soit d'utiliser un autre numéro de port, soit d'attendre quelques minutes.

**Tester les programmes.** Pour tester les programmes des exercices de TP, il peut parfois être intéressant d'utiliser un programme de confiance. On pourra par exemple utiliser l'outil `nc(1)` (*netcat*) pour construire un prototype de client ou de serveur<sup>2</sup>.

Pour certains exercices, une *implémentation de référence* des programmes client et serveur est également fournie et peut être utilisée.

**Envoi/réception des données.** Une communication en TCP est un flux d'octets sans aucune structure et ce, quelque soit le nombre de `write` utilisés pour envoyer les données. Le nombre d'appels à `read` pour recevoir les données est indépendant du nombre de `write`. Dans certains cas, il peut être nécessaire d'effectuer plusieurs `read` pour lire toutes les données, même si un buffer *a priori* suffisamment grand est fourni.

**Données de taille variable.** Dans certains problèmes, les programmes doivent s'échanger des données dont la taille est variable. À cause du point précédent, il faut mettre en place un mécanisme pour que le récepteur sache s'il a reçu l'intégralité des données. Il y a principalement trois solutions possibles :

- l'émetteur ferme la connexion lorsqu'il a terminé son envoi. Cette solution n'est envisageable que si c'est le dernier message échangé.
- l'émetteur envoie une valeur spéciale pour annoncer la fin des données. Cette valeur spéciale ne doit évidemment pas faire partie des valeurs valides pour les données qui précèdent.
- l'émetteur envoie d'abord un premier message qui annonce la longueur de ce qui suit. Cette solution implique de connaître à l'avance la longueur des données à envoyer.

**Systèmes hétérogènes.** Les programmes client et serveur s'exécutent potentiellement sur des systèmes et des architectures différentes. Il faut donc faire particulièrement attention au codage des données pour que les programmes se comprennent correctement. Par exemple un entier de type *int* n'est pas toujours codé par 32 bits en *little-endian*...

---

2. Notez qu'il existe au moins deux implémentations de netcat avec un fonctionnement légèrement différent. Consulter le manuel pour les détails.