Compte-rendu du TP1

1 Compiler et passer des paramètres

Télécharger les fichiers fournis depuis Moodle. Un squelette de base est fourni pour la mise en oeuvre des programmes de TP. Vous y trouverez :

1. un squelette pour un programme client
2. un squelette pour un programme serveur
3. un Makefile pour compiler

Vous êtes encouragés pour la suite des TP à adopter une structuration semblable à celle du répertoire fourni, en particulier pour des applications impliquant plusieurs programmes / fichiers sources.
Avant de lire le contenu des fichiers

1. compiler avec la commande `make`
2. lancer le programme serveur et le relancer si nécessaire pour aller plus loin dans l'exécution.
3. lancer le programme client et le relancer si nécessaire pour aller plus loin dans l'exécution.

À ce stade, vous devez comprendre que la première étape qui est réalisée lorsqu'un code est fourni (c'est exactement la même chose pour un correcteur lorsqu'il évalue un travail pratique), est de compiler et d'exécuter les programmes. Tous les éléments permettant de lancer correctement un programme doivent être fournis. Ici, ces éléments sont les paramètres demandés.
Vous êtes maintenant invités à lire le contenu des fichiers fournis.

2 Votre premier programme client

À présent, un serveur est en cours d'exécution sur une machine de la FdS. Il attend des messages en provenance de clients. La sortie standard du serveur est projetée tout le long de cet exercice et permettra de notifier une réception d'un message de votre client et de relever d'éventuelles erreurs / incohérences dans les échanges.
L'objectif est alors d'écrire un programme client. Ce dernier demande en paramètre l'adresse (IP + numéro de port) de la socket du serveur (fournie par l'intervenant(e)) et un numéro de port pour la socket de votre client. Les étapes à réaliser dans votre programme sont données en commentaire dans le squelette :

1. Après avoir désigner la socket du serveur, demander à l'utilisateur de saisir une chaîne de caractères (de taille max 100 caractères). Pour la saisie, vous pouvez utiliser la fonction `scanf(...)` ou `fgets(...)` (qui permet de saisir des chaînes avec des espaces). Vous pouvez exceptionnellement utiliser les moyens d'entrée / sortie du C++ (`cin` et `cout`).
2. Envoyer la chaîne saisie à l'aide de la fonction `sendto(...)`. Traiter les valeurs de retour possibles de cette fonction (voir le cours ou, encore mieux, la documentation via la commande `man`). Tester votre programme.
3. Recevoir un entier à l'aide de la fonction `recvfrom(...)` et l'afficher. Traiter les valeurs de retour possibles de cette fonction (voir le cours ou la documentation). Tester votre programme.
4. Afficher l'adresse de la socket de l'expéditeur du message reçu à partir de l'adresse fournie par la fonction `recvfrom(...)` et s'assurer que cette adresse correspond bien à celle passée en paramètres. Rappel : l'adresse d'une socket inclut l'IP et le numéro de port.

À ce stade du TP, vous devez avoir compris les bases pour un échange de messages. Remarque : lors d'un échange, n'envoyer que les données utiles.

1. À propos des entrées / sorties en C : il existe trois fonctions : `scanf(...)`, `fgets(...)` qui toutes les deux prennent une chaîne de caractères de taille statique et `getline(...)` qui prend une chaîne de caractères dynamiques. Exemple :

```c
#include <stdio.h>

int main() {
    // 1 - Chaîne de taille statique
    char message[100];
    fgets(message, sizeof(message) /* == 100 */, stdin);
    scanf("%[^\n]%*c", message);

    // 2 - Chaîne de taille dynamique
    char *message; size_t mem;
    getline(&message, &mem, stdin);
}
```

   L'utilisation de `fgets(...)` ou `scanf(...)` est suffisante dans cette première partie du TP, comme la taille du message est donnée. Cependant, durant l'amélioration personnelle du TP, j'ai décidé de passer à `getline(...)` pour avoir une chaîne de taille arbitraire.

2. `sendto(...)` retourne -1 en cas d'erreur, et la taille du message envoyé sinon. Cependant, il faut faire attention. Si par exemple le client envoie une chaîne de taille 256 et que la taille maximum reçu par le serveur est 200, `sendto(...)` renverra tout de même 256. Le client n'a aucun moyen de savoir que

son message n'est pas entièrement envoyé. D'où la définition d'un protocole réseau propre au client / serveur pour pouvoir avoir ces informations. Un exemple est donné dans la question suivante.

3. Le passage d'un entier en chaîne de caractère en C est laborieux. Il y a une solution utilisée : `sprintf(...)`. Cette fonction permet de formatter une chaîne de caractère.

   Le but ici est d'envoyer un message au serveur plus grand que le maximum attendu. Comme ça, on peut remarquer que le résultat de `sendto(...)` et de l'entier reçu par `recvfrom(...)` ne correspond pas, et peut faire penser aux protocoles réseaux à implémenter. Par exemple, le client envoie un message, le serveur lui retourne la taille lue, puis le client envoie la suite du message si la taille lue ne correspond pas à la taille de la chaîne, et ainsi de suite.

4. Pour afficher l'adresse, il ne faut pas oublier de convertir la structure de donnée actualisée par `recvfrom(...)` en chaîne grâce aux fonctions `inet_itoa(...)` pour l'IP et `ntohs(...)` pour le port.

3 Programmer le serveur

À vous maintenant de programmer sur le serveur communiquant avec votre client. Ce serveur doit :

— Attendre un message, sous forme de chaîne de caractères, affiche ce message et l'adresse de la socket du client. Tester votre programme en exécutant le serveur et le client.

— Répondre au client en lui envoyant la taille (en nombre d'octets) du message qu'il vient de recevoir. Cette taille est un entier. Tester.

Dans un premier temps, vous pouvez lancer vos programmes sur une même machine. Ensuite, vous pouvez vous habituer à utiliser des machines distantes à l'aide de la commande ssh. À la FdS, vous pouvez vous connecter, via ssh aux machines suivantes : prodpeda-x2go-focal1, prodpeda-x2go-focal2, ..., prodpeda-x2go-focal6.

C'est ici que j'ai mis en place mon propre protocole réseau. Comme je voulais pouvoir passer une chaîne de caractères de taille arbitraire dans `sendto(...)` et la recevoir complètement dans `recvfrom(...)`, mon émission et ma réception se font en 2 étapes : l'envoie (et la réception) de la taille du message, puis du message en lui-même.

```c
// Protocole d'envoie des messages
int sendMessage(int ds, const char *message, const struct sockaddr_in sockServ) {
    const struct sockaddr *serv = (const struct sockaddr *)&sockServ;
    socklen_t addrLen = sizeof(struct sockaddr_in);

    char size[100];
    sprintf(size, "%zu", strlen(message));

    if (sendto(ds, size, strlen(size) + 1, 0, serv, addrLen) == ERROR
        || sendto(ds, message, strlen(message) + 1, 0, serv, addrLen) == ERROR) {
        return ERROR;
    }
    return 0;
}

// Protocole de réception des messages.
char *receiveMessage(int ds, struct sockaddr_in *sockClient, socklen_t *lgAdr) {
    // Jusqu'à un message de taille 10^100 octets.
    char bytes[100];
```

```
    ssize_t res = recvfrom(ds, bytes, sizeof(bytes), 0, (struct sockaddr*)sockClient, lgAdr);
    if (res == ERROR)
        return NULL;
    int msgSize = atoi(bytes);
    char *message = malloc(msgSize);

    res = recvfrom(ds, message, msgSize, 0, (struct sockaddr*)&sockClient, lgAdr);
    if (res == ERROR) {
        free(message);
        return NULL;
    }

    return message;
}
```

4 Tests avancés

Les questions suivantes sont indépendantes (sauf indication contraire) :

- Modifier votre programme client pour supprimer le nommage de sa socket. Exécuter votre application. Le nommage supprimé était-il nécessaire ? Pourquoi ?
- Modifier le client pour qu'il envoie un tableau d'entiers (que vous pouvez initialiser à votre convenance) à la place de la chaîne de caractères saisie. Que faut-il modifier côté serveur pour que votre application s'exécute correctement ? Le serveur répond toujours en envoyant le nombre d'octets effectivement reçus.
- Faire pareil, mais cette fois, il est question d'envoyer la structure `sockaddr_in` qui désigne la socket du serveur

Remarque : vous pouvez tout à fait garder les échanges existants et en ajouter, au lieu de remplacer l'existant.

- Le nommage du client n'est pas *nécessaire*, car il n'est pas utilisé dans le programme client. Cependant, si le client veut être lancé sur le port donné en argument du programme, il faut la nommer. Dans le cas contraire, c'est un port aléatoire qui lui est assigné.
- Côté serveur, il suffit de modifier le `char*` en `int*` de `recvfrom(...)` pour récupérer le tableau d'entiers.
- Pareil, étant donné que le type de données envoyé est un `void*`, peu importe le type tant que c'est un pointeur, et tant que le nombre d'octets qui doivent être lus est bon.

— Que se passe-t-il si le client est lancé avant le serveur ?
— Que se passe-t-il si le client envoie un message juste avant le nommage coté serveur ? Pour réaliser ce test, modifier le serveur pour ajouter une saisie juste avant le nommage, puis lancer le serveur qui sera suspendu en attente de saisie, lancer le client qui doit dépasser l'étape d'envoi d'un message, puis reprendre l'exécution du serveur en faisant une saisie.
— Que se passe-t-il si le client envoie un message après le nommage coté serveur mais avant l'appel à `recvfrom(...)` par ce serveur ? Pour réaliser ce test, suivre le même principe que la question précédente.
— Que se passe-t-il si le serveur attend un message de taille plus petite que celle du message envoyé par le client ? Pour tester, il est question de modifier le paramètre taille de la fonction `recvfrom(...)`.
— Modifier vos programmes pour que le client envoie un tableau de doubles. La taille de ce tableau (en nombre d'octets) doit dépasser les 70Ko. Tester. Le serveur répond toujours en envoyant le nombre d'octets effectivement reçus.

— Si le client est lancé avant le serveur, le message est quand même envoyé.
— Le serveur ne reçoit jamais le message.
— De même, le serveur ne reçoit jamais le message.
— Le client renvoie une erreur : `Message too long`.

— Consulter la documentation de la fonction `getsockname(...)`
— Modifier le programme serveur pour que le nommage soit fait automatiquement par la couche transport.
— Utiliser la fonction `getsockname(...)` pour obtenir le numéro de port choisi par la couche transport.
— Exécutez votre application (attention aux paramètres à passer).

— `getsockname(...)` nomme automatiquement une structure `sockaddr_in`, il suffit de donner la taille de cette structure en argument à la fonction. Attention, il faut tout de même appeler `bind(...)` avant `getsockname(...)`.
— Pour que le nommage soit fait automatiquement, il faut passer le port 0 dans `sin_port` de la structure.
— Le numéro de port est un numéro aléatoire > 1024.
— J'ai choisi de garder deux implémentation : la désignation du port automatique et la désignation du port manuelle via argument de la ligne de commande. J'ai fait la même chose pour le client.

— Que se passe-t-il si vous lancez un serveur et deux clients ? Il n'est pas question de programmer un nouveau client mais uniquement de créer deux processus à partir du même programme.
— Que se passe-t-il si vous lancez deux serveurs avec le même numéro de port ? Cette question nécessite un choix explicite du numéro de port du serveur (à passer en paramètre).
— Modifier le programme serveur pour qu'il puisse dialoguer avec plusieurs clients ? Tester.

— Les deux clients peuvent envoyer un message au serveur. Cependant, si un message est reçu, le serveur s'arrête.
— Le serveur renvoie une erreur : `Address already in use`.
— Il suffit de mettre une boucle `while(1)` autour de la réception du message par le serveur.