Pseudos-codes des versions V1 et V2 de Clément Lacau, 3MIC-C

(Binôme avec: Alnet Paul)

Indications pour les pseudos-codes des versions V1 et V2:
Dans ces pseudo-codes nous considérons que nous n'envoyons qu'un seul message de taille fixe construit et affiché avec onstruire_message() et afficher_message(). Pour en envoyer plusieurs (nombre donné en paramètre -n ou 10 par défaut), il suffira de rajouter une boucle autour de la construction du message et l'envoi côté source, ou la réception du message et l'affichage côté puit.
De plus, le numéro de port indiqué est 9000 (à titre d'exemple) et le nom logique non défini dans ces pseudo-codes mais ils seront à récupérer via les entrées de l'application.
Le choix du type de communication se fera par le paramètre -u (UDP, par défaut TCP).

Les étapes sont marquées par des X.
Les étapes entre () sont facultatives.

X Ajouter le traitement du paramètre -u selon la méthode employée dans tsock_v0.c
    Si -u est donné en paramètre, on communiquera en UDP (V1), sinon en TCP (V2)

Pseudo-code V1 (UDP) :

Si on est en UDP et dans la source (émetteur du message):

X Créer un socket UDP local
    Appel à socket() | AF_INET
                     | SOCK_DGRAM
                     | 0 ↔ UDP

(X Adresser le socket local ↔ non nécessaire car cela sera réalisé automatiquement à l'envoi du premier message UDP via le socket considéré.)

X Construire l'adresse du socket distant (/du destinataire)
    | adr_dest ← numéro de port = 9000 (exemple, port donné en paramètre)
    | adr_dest ← adresse IP de la machine destinataire (avec gethostbyname(« nom_logique »), « nom_logique » donné en paramètre aussi).

X Construire le message à envoyer
    Appel à construire_message() | adresse de stockage du message
                                 | caractère
                                 | longueur du message

X Envoyer le message à l'adresse du socket distant
    Appel à sendto() | socket local (représentation interne)
                     | message stocké
                     | longueur du message
                     | 0
                     | adresse du socket distant
                     | longueur de l'adresse du socket distant

Si on est en UDP et dans le <u>puit (récepteur du message)</u> :

X Créer un socket UDP local
Appel à socket() | AF_INET
| SOCK_DGRAM
| 0 ↔ UDP

X Construire une adresse externe pour le socket local
| adr_local ← numéro de port = 9000 (exemple, port donné en paramètre)
| adr_local ← adresse IP = INADDR_ANY (n'importe quelle IP de la machine)

X Association de l'adresse externe à la représentation interne du socket local
Appel à bind() | socket local (représentation interne)
| adresse externe du socket local = (sockaddr *) &adr_local (conversion de type, *sockaddr_in → *sock_addr)
| taille de l'adresse externe du socket local = sizeof(adr_local)

X Recevoir le message du socket distant
Appel à recvfrom() | socket local (représentation interne)
| adresse de stockage du message
| longueur maximale du message
| 0
| stockage de l'adresse externe du socket distant émetteur en sortie de fonction
| taille de l'adresse externe du socket distant en sortie de fonction

X Afficher le message reçu
Appel à afficher_message() | message stocké
| longueur du message

<u>Pseudo-code V2 (TCP) :</u>

Si on est en TCP et dans la <u>source</u> (client de la connexion):

X Créer un socket TCP local
Appel à socket() | AF_INET
| SOCK_STREAM
| 0 ↔ TCP

(X Adresser le socket local ↔ non nécessaire car cela sera réalisé automatiquement à l'envoi d'une demande de connexion TCP via le socket considéré.)

X Construire l'adresse du socket distant serveur
| adr_dest ← numéro de port = 9000 (exemple, port donné en paramètre)
| adr_dest ← adresse IP de la machine destinataire (avec gethostbyname(« nom_logique »)
« nom_logique » donné en paramètre aussi).

X Demander une connexion du socket local client au socket distant serveur, en boucle tant que la demande n'est pas acceptée

Appel de connect() | socket local (représentation interne)
| adresse distante du socket serveur
| longueur de l'adresse du socket serveur
retour → 0 si succès, -1 si échec

X Construire le message à envoyer

Appel à construire_message() | adresse de stockage du message
| caractère
| longueur du message

X Envoyer le message à l'adresse du socket distant serveur

Appel à write() | socket local (représentation interne)
| message stocké
| longueur du message

(X Fermeture de la connexion dans les deux sens une fois celle-ci terminée.

Appel à shutdown() | socket local (représentation interne)
| sens = 2 (le socket local ne veut être ni recevoir ni émettre)

Facultatif car on ne compte pas réutilise les sockets après fermeture de la connexion. De plus, il serait important de gérer la suppression des sockets en cas de fork côté puit avec close(), partie bonus)

Si on est en TCP et dans le <u>puit</u> (serveur de la connexion):

X Créer un socket TCP local

Appel à socket() | AF_INET
| SOCK_STREAM
| 0 ↔ TCP

X Construire une adresse externe pour le socket local

| adr_local ← numéro de port = 9000 (exemple, port donné en paramètre)
| adr_local ← adresse IP = INADDR_ANY (n'importe quelle IP de la machine)

X Association de l'adresse externe à la représentation interne du socket local serveur

Appel à bind() | socket local (représentation interne)
| adresse externe socket local = (sockaddr *) &adr_local (conversion de type, *sockaddr_in → *sock_addr)
| taille de l'adresse externe du socket local = sizeof(adr_local)

(X Dimensionnement de la file d'attente des demandes de connexion

Appel à listen() | socket local (représentation interne)
| n ↔ la file d'attente est de n demandes de connexions en attente au maximum

Facultatif car pour l'instant on ne permet qu'une seule connexion à la fois, voir partie bonus)

X Mise en état d'acceptation d'une demande de connexion

Appel à accept() | socket local (représentation interne)
| stockage de l'adresse du socket distant client en sortie de fonction

| taille de l'adresse du socket client en sortie de fonction

(accept() retourne la représentation interne d'un nouveau socket local que l'on stockera dans sock_bis et qui est le nouveau socket consacré à la connexion établie).

X Recevoir le message du socket distant client sur le socket sock_bis

Appel à read() | socket local sock_bis (représentation interne)
| adresse de stockage du message
| longueur maximale du message

X Afficher le message reçu

Appel à afficher_message() | message stocké
| longueur du message

Version V3 (ajouts à V2):

-Formatage des messages émis → Modifier la fonction construction_message()

-Affichage des messages émis et reçus → Une fonction d'affichage des messages pour l'émission et une autre pour la réception.

-Traitement du nombre de messages -n :

Rajout des boucles autour des constructions/envois ou réception/affichage de messages et transmission de la valeur récupérée de l'option -n pour les boucles.

-Traitement de la longueur des message -l :

Transmission de la valeur récupéré de l'option -l vers les variables de stockage des messages, les fonctions de construction / affichage de message et les fonctions d'envoi / de réception de messages.

Version V4 (ajouts à V3):

-le serveur peut répondre à plusieurs demandes de clients émis simultanément

Dans la partie TCP, le thread main accepte toutes les connexions. La fonction listen() est utilisée pour dimensionner la file d'attente de demandes de connexions. Sans sous-traitance, ces demandes sont traitées l'une après l'autre par le main.

-le serveur ne s'arrête jamais (seuls ses fils ferment la connexion une fois les messages envoyés au client) .

Le main utilise fork() pour déléguer (sous-traiter) la gestion de chaque connexion à un thread fils créé par chaque accept() et continue d'accepter constamment des demandes de connexions. Les fermetures des sockets inutiles sont faites.

-le choix du rôle est paramétrable via des options appropriées (par ex à la place des options - -s et -p : -c -e pour client de la connexion / émetteur des données, -c -r pour client de la connexion / récepteur des données, etc.).

la source est forcément l’émettrice de la demande connexion TCP car le puit ne peut pas contacter la source. Une fois ces informations échangées, l’échange d’informations débute dans le sens choisi. On distingue donc la phase de connexion et la phase d’échange de messages en choisissant si on va émettre ou recevoir selon les paramètres une fois connecté. Traitement des paramètres pour permettre les exécutions de code appropriées.