LIFASR5, TP noté, 2018/2019, sujet 2

Description du programme. Votre programme doit implémenter un serveur réseau TCP qui utilise un protocole déterminé. Le programme prend un seul argument, qui est le numéro de port TCP sur lequel il doit se mettre en écoute.

Le serveur doit accepter les clients qui demandent à s'y connecter, et doit être capable d'en prendre en charge plusieurs à la fois. À chaque fois que le serveur accepte une nouvelle connexion :

- il affiche sur sa sortie standard le message
  `## Nouveau client connecté <adresse> <port>`
- il envoie au client la réponse (sans rien de plus)
  `Bienvenue\r\n`
- il crée un fils qui sera chargé du dialogue avec le client, selon le protocole qui sera décrit plus loin,
- il revient se mettre en attente de nouvelles demandes de connexion.

C'est le client qui décide quand la connexion se termine, en fermant lui-même la socket de dialogue. Lorsqu'un client ferme la connexion, le serveur doit afficher sur sa sortie standard :
`## Client déconnecté`

Le protocole. Il peut être décrit de la façon suivante :

- Le client envoie une commande et des données.
- La commande est une ligne de texte terminée par un `\r\n`.
- Les données sont
  — soit des chaînes de caractères terminées par un `\r\n`,
  — soit des données binaires dont la taille `t` en octets est fournie par la commande concernée.
- Les commandes comportent deux parties :
  — la commande de base, soit `RECOP`, soit `AFFIC`;
  — La description de la donnée : `CHN` pour une chaîne de caractères, `BIN t` pour des données binaires de taille `t`.

Les commandes prises en charge par le serveur sont :

- `RECOP` signifie que le serveur doit répéter la donnée sur la socket,
- `AFFIC` signifie que le serveur doit afficher la donnée précédée de `##` sur sa sortie standard. Dans le cas d'un `AFFIC`, après avoir reçu les données, le serveur envoie au client un acquittement sous la forme du texte `DONE\r\n`.

Consignes et script de test. L'archive que vous allez utiliser contient tout ce dont vous avez besoin pour ce TP noté :

- une version de `socklib` qui va bien pour ce TP;
- le code source `server.cpp` à compléter;
- un `Makefile`, pour fabriquer l'exécutable `server`;
- le script de test `test.py`, que vous utiliserez pour tester votre serveur avec `./test.py server`

Le script `test.py` sera utilisé également pour vous noter : **vous devez suivre scrupuleusement les consignes sur l'affichage et l'envoi de message qui vous sont données pour pouvoir obtenir des points**. Si on vous demande d'afficher `## toto` sur la sortie d'erreur standard, mais que vous l'affichez sur la sortie standard, vous n'aurez pas les points correspondants...

`test.py` effectue quatre séries de tests (vérification des arguments, mise à l'écoute du serveur, vérification des connexions, vérification du protocole). Les explications sur chaque test sont précédées de `***` (et s'affichent normalement en jaune). Quand vous lancez `./test.py server`, les tests réussis s'affichent précédés de `OK` (sur fond vert), et les problèmes sont précédés de `Not OK` (sur fond rouge). Pour comprendre d'où vient le problème, remontez à l'explication précédant le `Not OK`.

Rendu. Vous déposerez votre fichier `server.cpp` dans la case `tpnote` de l'UE dans votre suivi TOMUSS. **Pour obtenir une note non nulle**, il est nécessaire (mais pas suffisant) :

- que vous déposiez bien le fichier `server.cpp`, et celui-ci seulement,
- que votre programme puisse être compilé avec le `Makefile` fourni,
- que le le script de test propose au moins un test `OK`.

À vous de jouer! Notez que la question `i` ($1 \leq i \leq 11$) correspond au `TODOi` repéré par un commentaire dans le code source.

1) Votre programme ne doit prendre comme argument que le port sur lequel le serveur doit se mettre à l'écoute. Il doit vérifier le nombre d'arguments qui lui sont passés, et si ce nombre n'est pas bon, afficher
`## usage <commande> PORT`
sur la sortie d'erreur standard (`<commande>` doit être le nom effectif de la commande). Complétez le code au niveau du `TODO1` de façon à respecter ces consignes.

Vous devez, comme pour toutes les questions suivantes, tester votre programme : utilisez le script fourni avec la commande `./test.py server`.

2) Au niveau du `TODO2`, complétez le code en définissant une variable `port` de type `const char *` initialisée pour que, dans la suite du programme, elle pointe vers la chaîne de caractères qui contient le port passé en argument du programme. Le

programme doit ensuite afficher sur la sortie standard (<port> est remplacé par sa valeur) :
`## le port utilisé est <port>`

À ce stade, la première série de tests doit être parfaite.

3) TODO3 : en utilisant la fonction `create_server_socket()` de la `socklib`, créer une socket d'écoute `s` sur le port passé en argument du programme. En cas d'erreur (si le port demandé ne peut pas être ouvert), votre programme doit afficher sur la sortie d'erreur standard :
`## erreur à la création du serveur sur le port <port>`
(en remplaçant <port> par sa valeur) et se terminer ne retournant `EXIT_FAILURE`.

4) Dans la boucle d'attente des clients, complétez le code au niveau du TODO4, de façon à ce que le serveur se mette en attente d'une demande de connexion sur la socket d'écoute `s`. Quand un client vient se connecter avec succès, on appelle `sd` la socket de dialogue obtenue. En cas d'échec, le serveur affiche
`## erreur lors de la connexion d'un client`
puis se termine en retournant `EXIT_FAILURE`.

À ce stade, la deuxième série de tests doit être parfaite.

5) TODO5 : le serveur envoie le message `Bienvenue` au client. Veillez à ce que le message envoyé se termine bien par `\r\n`.

6) TODO6 : modifiez le code de façon à ce que le processus principal crée un fils. Le code exécuté spécifiquement par le fils est compris entre l'accolade ouvrante et l'accolade fermante marquées par le commentaire `processus fils`. Le père quant à lui doit revenir se mettre en attente du client suivant.

7) Jusqu'à présent, on a ignoré la terminaison des fils quand un client ferme la connexion. Complétez les deux parties du code où se trouve le commentaire TODO7, de façon à ce que le père prenne en compte la mort de ses fils. Spécifiez l'option `SA_RESTART` dans la structure `struct sigaction` que vous utiliserez pour gérer correctement le signal `SIGCHLD`.

À ce stade, la troisième série de tests doit être parfaite.

8) Dans la boucle de traitement des commandes (voir les commentaires dans le code), le processus fils créé s'occupe de traiter chacune des commandes envoyées par le client. Lorsque le client ferme la connexion, on sort de la boucle : le fils ferme alors la socket de dialogue, et se termine. Le code pour la réception d'une commande `com` vous est fourni à titre d'exemple (voir le README1). Ensuite, en fonction de la commande envoyée, le fils doit réagir différemment. Commencez par traiter le cas d'une commande `RECOP CHN`, au niveau du TODO8. Dans ce cas, le fils doit :
— recevoir une ligne de texte avec la fonction `recv_line()`,
— traiter la valeur de retour (utilisez la variable `res`) de `recv_line()` : le client peut décider de fermer la connexion, ou bien il peut y avoir une erreur ; dans les deux cas, il faut sortir de la boucle de traitement des commandes.
— enlever les `\r` ou `\n` intempestifs par lesquels peut se finir cette ligne avec la fonction `clean_line()`,
— renvoyer la ligne au client en la terminant par `\r\n` avec la fonction `send_str()`,
puis il vient se remettre en attente de la commande suivante.

9) TODO9 : traitez le cas d'une commande `AFFIC CHN`. Le fils doit :
— recevoir une ligne de texte avec `recv_line()`,
— traiter la valeur de retour (encore `res`) de `recv_line()`,
— envoyer au client l'acquittement constitué du texte `DONE\r\n`,
— enlever les `\r` ou `\n` intempestifs par lesquels peut se finir la ligne reçue,
— l'afficher sur sa sortie standard en la faisant précéder de `##` , et en ajoutant un retour à la fin (utilisez `endl` ou `'\n'`).
Le fils vient ensuite se mettre en attente de la commande suivante.

10) TODO10 : traitez le cas d'une commande `RECOP BIN t`. Notez que le nombre d'octets à recevoir vous est fourni dans le source : utilisez juste la variable `t`. Le fils doit :
— recevoir `t` octets depuis la socket de dialogue avec la fonction `recv_all()`,
— traiter la valeur de retour (toujours `res`) de `recv_all()`, comme dans le cas de `recv_line()`,
— envoyer ces `t` octets sur la socket de dialogue avec la fonction `send_all()`,
puis se remettre en attente de la commande suivante.

11) TODO11 : traitez le cas d'une commande `AFFIC BIN t`. Le nombre d'octets `t` à recevoir vous est à nouveau fourni dans le code. Le fils doit :
— recevoir `t` octets depuis la socket de dialogue,
— traiter la valeur de retour (définitivement `res`) de `recv_all()`,
— envoyer au client l'acquittement `DONE\r\n`,
— envoyer les `t` octets reçus sur la sortie standard,
puis se remettre en attente de la commande suivante.

À ce stade, la quatrième série de tests doit être parfaite. Vous devez maintenant avoir le maximum de points.