Pablo Rauzy
pablo.rauzy@univ-paris8.fr
https://pablo.rauzy.name/teaching/rmpp

UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS

Réseaux : modèles, protocoles, programmation

Université Paris 8 – Vincennes à Saint-Denis
UFR MITSIC / L2 informatique

Séance 7 (TP) : ncat

N'oubliez pas :
- Les TPs doivent être rendus par courriel au plus tard le lendemain du jour où ils ont lieu avec "[rmpp]" suivi du numéro de la séance et de votre nom dans le sujet du mail, par exemple "[rmpp] TP7 Rauzy".
- Quand un exercice demande des réponses qui ne sont pas du code, vous les mettrez dans un fichier texte `reponses.txt` à rendre avec le code.
- Le TP doit être rendu dans une archive, par exemple un tar gzippé obtenu avec la commande `tar czvf NOM.tgz NOM`, où `NOM` est le nom du répertoire dans lequel il y a votre code (idéalement, votre nom de famille et le numéro de la séance, par exemple "rauzy-tp7").
- Si l'archive est lourde (> 1 Mo), merci d'utiliser `https://bigfiles.univ-paris8.fr/`.
- Les fichiers temporaires (si il y en a) doivent être supprimés avant de créer l'archive.
- Le code doit être proprement indenté et les variables, fonctions, constantes, etc. correctement nommées, en respectant des conventions cohérentes.
- Le code est de préférence en anglais, les commentaires (si besoin) en français ou anglais, en restant cohérent.
- **N'hésitez jamais à chercher de la documentation par vous-même sur le net!**

Dans ce TP :
- Programmation TCP.
- Utilisation de `poll` pour le multiplexage des entrées/sorties.

Exercice 0.

Bien démarrer.

1. Pensez à organiser correctement votre espace de travail, par exemple tout ce qui se passe dans ce TP pourrait être dans `~/rmpp/s7-tp/`.
2. Pensez à nommer le dossier que vous rendrez avec votre nom. Si vous ne le faites pas tout de suite, pensez à le faire avant de rendre votre TP.
3. Ce TP va consister en la réalisation d'un petit utilitaire réseaux de type `netcat`.
   Vous êtes encouragé à tester systématiquement votre logiciel après chaque modification de code, cela vous aidera à repérer les erreurs au plus tôt et donc le plus précisément possible.
   Comme d'habitude, n'hésitez pas à consulter les pages de `man` ou le web pour vous documenter.
   Ce guide est assez bon par exemple : `https://beej.us/guide/bgnet/`.
4. Pensez à regarder les pages de manuel des fonctions que vous utiliserez, pour ne pas oublier de `#include` nécessaires.
5. Résultat attendu :
   - Un programme `ncat.c` qui compile sans warning.
   - Quand on l'appelle avec un argument, on suppose que celui-ci est un numéro de port, et il se met alors en mode serveur (qui écoute sur ce port).
   - Quand on l'appelle avec deux arguments, on suppose que le premier est un nom d'hôte et le second un numéro de port, et il se met alors en mode client (qui se connecte à cet hôte sur ce port).
   - Quand on l'appelle avec un autre nombre d'argument, il doit afficher le message suivant :
     `Usage: %s [host] port`, où `%s` est le nom de l'exécutable.
   - Dans tous les cas on veut que la communication puisse avoir lieu dans les deux sens une fois la communication établie : les données reçues sur la socket doivent être écrites sur la sortie standard, et les données reçues sur l'entrée standard doivent être écrites sur la socket.
   - On veut aussi que cet outil soit utilisable pour tester des protocoles réseaux, on souhaite donc qu'il remplace les `\n` en fin de ligne par `\r\n` si c'est nécessaire.

Exercice 1.

Ce qu'on sait déjà faire.

1. → Créez le fichier `ncat.c` et dedans un `main` avec les déclarations de variable dont on aura besoin :
   - de quoi stocker une adresse,
   - de quoi récupérer l'IP de l'hôte passé en paramètre (pour le mode client),
   - de quoi avoir deux sockets (pour accepter le client en mode serveur).

2. → Vérifiez tout de suite que le nombre d'arguments passés au programme est valide, sinon afficher le message d'erreur.
3. → Ouvrez une socket TCP (de type `SOCK_STREAM`), en pensant bien à gérer les erreurs.
4. → Si le programme est appelé en mode serveur, le faire écouter et attendre une connexion sur le port passé en argument (pensez à gérer les erreurs). Une fois cette connexion accepté, fermer la socket initiale.
5. → Si le programme est appelé en mode client, le faire se connecter à l'hôte passé en argument sur le port spécifié (pensez à gérer les erreurs). Une fois connecté, assigné la valeur du descripteur de fichier de la socket connectée à la même variable que celle qui contient le descripteur de fichier du client dans le cas du mode serveur, ce sera plus simple pour la suite.
6. → Pensez à fermer la socket avant la fin du programme.
7. → Pour pouvoir tester votre code, ajoutez temporairement un appel à `read` sur la socket connectée et écrivez sur la sortie standard ce qui est lu sur la socket, puis un appel à `read` (ou autre) pour lire sur l'entrée standard et écrire sur la socket.
8. → Testez votre code en faisant appel à `nc` (ou `netcat` si `nc` n'est pas disponible) :
   — Dans un nouveau terminal, lancez `nc -l -p XXXX` où `XXXX` est un port de votre choix. Cela dit à `nc` d'écouter sur ce port. Vous pouvez alors tester votre code en mode client en vous connectant à `localhost`.
   — Après cela, lancez cette fois votre code en mode serveur sur le port de votre choix. Vous pouvez alors tester votre code en vous y connectant avec `nc localhost XXXX`, où `XXXX` est le port que vous avez choisi.

Exercice 2.

Copie de flux en assurant des fins de lignes CRLF.

1. → Dans votre fichier, définissez une fonction font le prototype est `int copy (int src, int dst);`. Cette fonction considère `src` (source) et `dst` (destination) comme des descripteurs de fichier. Elle copie tout ce qu'elle lit sur `src` sur `dst`, et si tout s'est bien passé retourne la taille en octet de ce qu'elle a lu puis écrit, ou -1 en cas d'erreur.
2. On veut maintenant vérifier s'assurer que les fins de ligne utilise bien la convention CRLF (`\r\n`). Pour simplifier les choses on fait la supposition raisonnable que les fin de ligne se trouve à la fin du buffer après la lecture.
   → Si le dernier caractère lu est `\n` et que c'est nécessaire, faites le nécessaire pour être en CRLF.
   Attention, pensez bien aux cas limites !
3. → Proposez un code qui permet de tester votre fonction et utilisez le pour confirmer que votre implémentation est correcte.

Exercice 3.

Multiplexage des entrées/sorties.

1. Maintenant, on souhaiterait pouvoir écrire et recevoir des données en même temps.
   Il existe un mécanisme pour cela : `poll`. Le principe de `poll` est qu'on lui passe un ensemble de descripteurs de fichier à surveiller en lui disant le type d'évènement qui nous intéresse (selon qu'on veut lire ou écrire), et il nous indique quand l'un d'entre eux est prêt pour.
   → Allez regarder la page de manuel de `poll` :
   (a) Quel est le prototype de la fonction `poll` ?
   (b) À quoi correspondent chacun de ses arguments ?
   (c) Que retourne la fonction ?
   (d) Quels sont les noms et les rôles respectifs des membres de la structure `pollfd` ?
2. → Dans le `main`, déclarez un tableau de deux `struct pollfd`. L'une sera la socket et l'autre l'entrée standard. Pour les deux, nous sommes intéressé par savoir si des données à lire sont présentes (`POLLIN`).
3. → Après avoir rempli le tableau, dans une boucle infinie, appelez `poll` (en gérant les erreurs). En cas de retour sans erreur, vérifiez pour chaque flux surveillé si l'évènement qui nous intéresse est activé dans le masque `revents` correspondant. Si c'est le cas, appelez votre fonction `copy` de façon appropriée. Si cette dernière renvoie 0 ou moins, sortir de la boucle avec `break`.
4. → Commentez le code de test que l'on avait ajouté à la fin de l'exercice 1 pour qu'il ne soit plus pris en compte par le compilateur.
5. → Testez votre programme, par exemple en vous connectant les uns aux autres, ou à un serveur web :

```
$ gcc ncat.c -o ncat

$ ./ncat www.ai.univ-paris8.fr 80
HEAD / HTTP/1.1
Host: www.ai.univ-paris8.fr
```

```
HTTP/1.1 200 OK
…

$ ./ncat www.ai.univ-paris8.fr 80
HEAD /existepas.html HTTP/1.1
Host: www.ai.univ-paris8.fr

HTTP/1.1 404 Not Found
…
```