Pablo Rauzy
pr@up8.edu
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 3 (TP) : Trace ta route !

N'oubliez pas :
- Les TPs doivent être rendus par courriel au plus tard la veille de la séance suivante avec "[rmpp]" suivi du numéro de la séance et de votre nom dans le sujet du mail, par exemple "[rmpp] TP3 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-tp3").
- 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 :
- Manipulation de *raw sockets*.
- Programmation d'un traceroute.

Exercice 0.

Récupération des fichiers nécessaires.

1. Pensez à organiser correctement votre espace de travail, par exemple tout ce qui se passe dans ce TP pourrait être dans `~/rmpp/s3-tp/`.
2. Récupérez les fichiers nécessaires depuis la page web du cours, ou directement en ligne de commande avec `wget https://pablo.rauzy.name/teaching/rmpp/seance-3_tp.tgz`.
3. Une fois que vous avez extrait le dossier de l'archive (par exemple avec la commande `tar xzf seance-3_tp.tgz`), renommez le répertoire en votre nom (avec la commande `mv rmpp_seance-3_files` *`votre-nom`*). Si vous ne le faites pas tout de suite, pensez à le faire avant de rendre votre TP.
4. Ce TP va consister en la réalisation d'un petit utilitaire réseaux dont le rôle est de donner le chemin emprunté par les paquets entre votre machine et un hôte de destination passé en argument à l'utilitaire.

   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.

   Aussi, n'hésitez pas à consulter les pages de `man` ou le web pour vous documenter.
   Bien qu'un peu désuet, ce guide est très clair par exemple : `https://beej.us/guide/bgnet/`.

Exercice 1.

État des lieux du code fourni.

1. Le fichier fourni, `tracetaroute.c` contient peu de choses :
   - des directives préprocesseur `#include`,
   - une fonction de calcul de somme de contrôle pour entêtes ICMP,
   - un main.

   → Chaque fois qu'on utilisera dans ce TP une nouvelle fonction, allez voir sa page de `man` et dites dans votre fichier `reponses.txt` dans quel fichier entête (`.h`) elle est déclarée (regroupez ces réponses au niveau de cette question pour tout le TP plutôt que de les éparpiller).
2. → Expliquez la boucle dans la fonction `checksum`.
3. → Compilez le programme et vérifiez qu'il fait bien ce à quoi vous vous attendez.

Exercice 2.

Récupérer l'adresse IP de la destination.

1. On passe un nom d'hôte au programme comme destination. Ce nom peut être du type `www.univ-paris8.fr`, mais ce dont on a besoin est de son adresse IP.

   On va pour cela utiliser la fonction `gethostbyname` (n'oubliez pas la question 1 de l'exercice 1!) qui va faire la requête DNS pour nous.

   → Commencez par déclarer dans le main les deux variables dont on va avoir besoin :
   — un pointeur sur une `struct hostent` (ce que retourne `gethostbyname`),
   — une `struct sockaddr_in` qui servira à stocker l'adresse.
2. → Appelez `gethostbyname` en lui passant le premier l'argument reçu par le programme et en gérant correctement le cas où une erreur se produit. On a déjà utilisé cette fonction en cours, n'hésitez pas à aller regarder comment.
3. → Remplissez la `struct sockaddr_in` en affectant :
   — comme famille (champ `sin_family`) la valeur `AF_INET` (on est sûr d'être en IPv4 à Paris 8, l'IPv6 n'est pas encore installé...),
   — comme port (champ `sin_port`) la valeur `0` (on utilise pas encore de port au niveau où on est),
   — comme adresse (champ `sin_addr`) celle renvoyée dans la `struct hostent` par `gethostbyname`.

   Le membre `sin_addr` de `struct sockaddr_in` est de type `struct in_addr` qui ne contient qu'un membre `s_addr` de type `uint32_t *`.

   Le membre `h_addr` de `struct hostent` est un pointeur vers la zone mémoire qui contient l'adresse ip sous forme de nombre.
4. On veut maintenant afficher pour vérification le nom d'hôte et l'ip pour laquelle on va faire le traceroute.

   → Utilisez pour cela les fonctions `printf` et `inet_ntoa`.

Exercice 3.

Architecture pour traceroute.

1. Le but de traceroute est de visualiser le chemin parcouru par les paquets entre votre machine et leur destination. Pour cela, on va utiliser les protocoles IP et ICMP : quand le TTL d'un paquet IP tombe à zéro alors que celui-ci n'a pas encore atteint sa destination, le routeur qui le réceptionne doit répondre à l'émetteur du paquet avec un datagramme ICMP de type 11 ("time exceeded") avec le code 0 (pour signifier que c'est bien le TTL du paquet IP qui est tombé à 0).

   Évidemment, cette réponse ICMP est elle même envoyée encapsulée quand un paquet IP qui contient l'adresse de son émetteur, c'est à dire le routeur sur lequel le TTL est arrivé à 0.

   On peut donc envoyer une succession de paquets IP (en pratique on utilisera des ICMP type 8 "echo request", comme dans ping) avec un TTL partant à un et incrémenté à chaque fois, et regarder d'où revient chaque paquet qui n'a pas pu arriver à destination.

   Une fois qu'on sera bien arrivé à destination, celle-ci répondra avec un datagramme ICMP de type 0 ("echo reply") et on saura alors qu'on a parcouru le chemin (déso pas déso).

   → Déclarez une fonction `icmp_request_echo` qui ne retourne rien et qui prend en argument un pointeur sur une `struct sockaddr` (l'adresse de l'hôte destination du traceroute) dont on se servira comme processus enfant du `fork` dans le main, et une fonction `icmp_listen` qui sera le processus parent et qui prend en argument un `pid_t` pour recevoir le pid du processus enfant.
2. → Remplacez les `printf` dans les processus enfant et parent après le `fork` par les appels aux deux fonctions qu'on vient de déclarer (n'oubliez pas de faire un cast vers le type de l'argument lors de l'appel de `icmp_request_echo`).

Exercice 4.

L'écoute des réponses, première partie.

1. On va commencer par écouter les réponses reçues. Pour cela on a besoin :
   — d'une socket,
   — d'un buffer d'octets (disons de 1024 octets) dans lequel on va stocker les paquets reçus sur la socket, et d'un entier pour stocker sa longueur,
   — d'un pointeur sur une `struct iphdr` pour lire les entêtes du paquet IP dans le buffer,
   — et d'un pointeur sur une `struct icmphdr` pour lire les entêtes du datagramme ICMP encapsulé dans le paquet IP.

   → Déclarez ces variables dans la fonction `icmp_listen`.
2. → Avec la fonction `socket`, ouvrez une socket *raw* pour ICMP sur IPv4, c'est à dire sur le domaine `AF_INET`, de type `SOCK_RAW`, et comme protocole `IPPROTO_ICMP`. Pensez évidemment à gérer les erreurs.
3. On va maintenant faire une boucle qui va consister en un appel à la fonction `recv` pour récupérer dans le buffer un paquet IP, puis du traitement de ce paquet. Cette boucle s'arrêtera au maximum après 64 tours (décision arbitraire, pour éviter que le processus tourne à l'infini en arrière plan).

La fonction `recv` est *bloquante*, c'est à dire qu'elle attend de recevoir quelque chose avant de retourner. Si tout s'est bien passé, elle renvoie le nombre d'octets écrit dans le buffer, sinon elle renvoie 0 ou moins.

→ Commencez par écrire la boucle, en ne faisant à l'intérieur que remettre à zéro le buffer avec la fonction `bzero`.

4. → Après avoir mis à zéro le buffer, appelez la fonction `recv` sur notre socket et avec notre buffer, sans lui passer de flags (c'est à dire mettre 0 pour cet argument). N'oubliez pas de gérer les erreurs (mais sans quitter le programme en cas d'erreur cette fois).
5. → Dans le cas où tout s'est bien passé, faites pointer votre pointeur sur une `struct iphdr` sur le buffer (affectez au pointeur l'adresse du buffer, vous aurez besoin de faire un cast explicite).
6. La source du paquet reçu se trouve dans le champs `saddr` de la `struct iphdr`. Vous pouvez la convertir en chaîne de caractères comme suit (en admettant que votre pointeur le `struct iphdr` s'appelle `ip`) : `inet_ntoa((struct in_addr) {ip->saddr})`.

   → Affichez l'adresse IP de la source du paquet reçu à l'aide de `inet_ntoa`.
7. On peut commencer à tester que votre programme marche bien.

   → Lancez votre exécutable dans un terminal, et dans un autre terminal lancez par exemple un ping. Vérifiez que votre programme reçoit bien les paquets (pour l'instant on se fait aucune sélection dans ce qu'on reçoit).

Exercice 5.

L'envoie des echo requests.

1. Pour envoyer les datagrammes ICMP "echo request" on a besoin :
   — d'une socket,
   — d'un compteur (pour incrémenter le TTL des paquets qu'on envoie au fur et à mesure),
   — et évidemment d'un paquet ICMP, c'est à dire d'une instance de la `struct icmphdr`.
   → Déclarez ces variables dans la fonction `icmp_request_echo`.
2. → À nouveau, ouvrez une socket *raw* similaire à celle ouverte dans la fonction `icmp_listen`.
3. On va préremplir le datagramme ICMP à envoyer, puisque celui-ci ne changera pas (c'est au niveau des entêtes IP qu'on précisera le TTL) :
   — dans le champ `type`, mettre `ICMP_ECHO`,
   — dans le champ `code`, mettre `0`,
   — dans le champ `un.echo.id`, mettre `getppid()` (pour que le processus parent puisse vérifier que le "echo reply" est a priori bien pour lui),
   — dans le champ `un.echo.sequence`, mettre `0`,
   — dans le champ `checksum`, mettre `0` puis mettre ce que renvoie la fonction `checksum` (déjà présente dans le fichier).
4. On va maintenant faire une boucle (tant que le compteur est plus petit que 64, même si le TTL d'un paquet IP peut au maximum être 255, en pratique 64 est déjà une valeur assez grande) qui va à chaque tour :
   — mettre le TTL par défaut de la socket à la valeur de notre compteur,
   — envoyer le paquet ICMP,
   — incrémenter le compteur,
   — attendre une seconde avant d'envoyer le suivant (pour essayer que les réponses ne se mélangent pas).
   → Commencez par écrire la boucle en pensant bien à avoir initialisé le compteur à 1 et à l'incrémenter à chaque tour, jusqu'à 64 maximum. Mettez déjà en fin de boucle un appel à la fonction `sleep` pour attendre une seconde.
5. On peut changer les options d'une socket avec la fonction `setsockopt`.

   Cette fonction prend en argument :
   — la socket à manipuler,
   — le niveau auquel on veut toucher (pour nous c'est au niveau IP, donc la constante `SOL_IP`),
   — le nom de l'option à changer (pour nous, `IP_TTL`),
   — un pointeur vers la nouvelle valeur de l'option (notre compteur),
   — la taille de la valeur en question.

   Quand tout s'est bien passé, `setsockopt` renvoie 0, sinon une autre valeur.

   → Au début de la boucle faites l'appel nécessaire à `setsockopt` en gérant les cas d'erreur.
6. On va maintenant utiliser la fonction `sendto` pour envoyer une ICMP echo request vers notre destination. Les arguments à cette fonction sont, dans l'ordre :
   — la socket à utiliser,
   — un pointeur vers les données à envoyer (pour nous, le datagramme ICMP),
   — la taille des données à envoyer,
   — les flags (aucun pour nous, donc 0),
   — un pointeur vers une `struct sockaddr` qui est l'adresse de destination (on l'a reçu en argument de la fonction `icmp_request_echo`),
   — la taille de cette adresse.

Quand tout s'est bien passé, `sendto` renvoie le nombre d'octets envoyés, sinon 0 ou moins en cas d'erreur.
→ Appelez `sendto` en gérant les erreurs.

7. Vous pouvez maintenant tester votre code et normalement observer que votre traceroute fonctionne mais qu'il ne s'arrête pas une fois arrivé à destination…
   → Pourquoi ?

Exercice 6.

L'écoute des réponses : s'arrêter quand on est arrivé.

1. On revient au cœur de notre fonction `icmp_listen`. Après avoir affiché l'IP de l'émetteur, on veut s'arrêter si on est arrivé à destination.
   → Au niveau du datagramme ICMP reçu, qu'est ce qui change quand on est effectivement arrivé à destination ?
2. → Affectez à votre pointeur sur une `struct icmphdr` l'adresse de début du datagramme ICMP encapsulé dans le paquet IP reçu. Pour rappel, votre buffer est un tableau d'octets, les lignes d'entête IP font 4 octets chacune et il y a un champ `ihl` dans la `struct iphdr` qui indique le nombre de lignes d'entête du paquet IP.
3. → Faites un test sur le type du datagramme ICMP, et si vous êtes bien arrivé à destination :
   — signalez le (par exemple avec un `printf`),
   — tuez le processus enfant avec la fonction `kill` en lui envoyer un `SIGINT`,
   — puis sortez de la boucle (et même de la fonction) avec un `return` pour terminer le processus.
   Vérifiez aussi au passage que c'est bien le datagramme que vous avez envoyé vos (souvenez-vous, on a rempli le champ `un.echo.id` de l'entête ICMP).

Exercice 7.

Bonus. Amélioration de notre traceroute.

1. → À l'aide de la fonction `getnameinfo`, affichez en plus des IP les noms des serveurs par lesquels passent nos paquets pour arriver à destination.