TP5

October 2018

1 Objectifs

Durant ce TP, vous allez créer un serveur de bytes aléatoire. Ce programme utilisera le générateur de nombes aléatoires du noyau Linux (`/dev/urandom`) et transmettra les données via le protocole TCP/IP. Ce type de serveur est utilisé en cryptographie lorsqu'il est couplé à une source aléatoire matérielle.

Le but principal de ce TP est de vous familiariser avec les sockets POSIX

2 Description

2.1 Génération de nombres aléatoires

Le noyau Linux est équipé de deux générateurs de nombres aléatoires de qualité cryptographique. Le premier (`/dev/random`) utilise des événements externe au système (délai entre les frappes du clavier, packets réseaux, etc.) pour remplir un *réservoir d'entropie*. Ce réservoir ne permet de générer qu'un nombre limité de bytes aléatoires, mais il est constamment rempli par le noyau. Si un utilisateur essaie de lire plus de bytes qu'il est actuellement possible, la lecture bloquera jusqu'à ce qu'assez d'entropie soit à nouveau disponible.

Le deuxième générateur de nombre aléatoire (`/dev/urandom`) est non bloquant, en recyclant le reservoir d'entropie disponible lorsqu'il est trop petit. La qualité de ce générateur est donc moins élevée mais reste suffisante pour des opérations cryptographique. Nous utiliserons ce deuxième générateur durant le TP.

Ces deux générateurs sont représentés par des fichiers synthétiques situés dans le répertoire `/dev`. Ont peut les ouvrir et lire comme n'importe quel fichier. Cependant, il n'est pas possible d'utiliser `lseek` sur ce descripteur de fichiers.

2.2 Protocole

Le protocole de communication est très simple:

1. Dès que la connection est acceptée, le client envoie au serveur le nombre de bytes demandés.
2. Le serveur envoie le nombre de bytes demandés, éventuellement en *plusieurs* envois.

Le fichier `randClient.c` implémente ce protocole, côté client.

2.3 Fonctionnement du serveur

Le serveur fonctionne selon les principes suivants:

- Le serveur possède un buffer B contenant les nombres aléatoires à envoyer (buffer de taille fixe N, comprise entre 512 et 4096 bytes).
- Le serveur connaît le nombre de bytes déjà envoyés (s).
- Si un client demande un nombre de bytes aléatoires n
  - Si $n < N - s$, le serveur envoie les bytes de B_s à B_{s+n}.
  - Si $n > N - s$, le serveur envoie les bytes de B_s à B_N, puis rempli le buffer.
- Chaque fois que le buffer est rempli, s est remis à zéro.
- Chaque fois que x bytes sont envoyés, s est incrémenté de x.
- Si le serveur ne peut pas envoyer toutes les données envoyées en une fois, il effectue plusieurs envois, jusqu'à ce que toutes les données demandées ont été envoyées.

Dans ce TP il vous est demandé d'implémenter une version *bloquante*. Cependant si vous avez le temps et que ça vous intéresse, vous pouvez essayer d'écrire une version non-bloquante une fois que votre implémentation bloquante fonctionne (pas de points supplémentaires).

2.4 Consignes et conseils pratiques

- L'archive fournie contient le code de la partie client ainsi qu'un *Makefile*. Il faudra modifier ce Makefile pour qu'il compile également le code du serveur.
- L'exécutable du serveur aura pour nom `randServer`.
- L'exécutable ne prend qu'un argument, le numéro de port.
- Evitez les allocations de mémoire en dehors de l'initalisation du serveur.
- Pour débugger votre serveur, vous pouvez le lancer avec `strace`, qui vous permet d'afficher les appels systèmes utilisés.
- Lorsque vous terminez le serveur, son port peut rester bloqué pendant quelques secondes. Dans ce cas lancez le serveur suivant sur un autre port.