F2B205B - Programmation Système et Réseaux
Partie Unix

C. Lohr

3 Décembre 2010

Modalités

- Document de cours autorisés (polys, sujets de TP, listings de TP)
- Les ordinateurs portables et autres équipements informatiques ne seront pas autorisés dans la salle. Ils devront tout au moins être éteints.
- Réponses à rendre sur copie séparée.
- Durée indicative : 1h30

1 Le Grand Oracle Qui A Réponse À Tout

On souhaite développer une application réseau. Le système est composé d'un serveur (appelé ici le *Le Grand Oracle Qui A Réponse À Tout*), et des clients.

Les clients se connectent au serveur sur le port TCP 5555. Le protocole est un simple échange bidirectionnel de chaînes de caractères. Des commandes Unix telles que `telnet` ou `netcat` conviennent parfaitement. Nous n'aborderons pas cet aspect du système.

Le cerveau de l'affaire sera constitué du programme `megahal` [1]. MegaHAL est un «simulateur de conversation» : un programme informatique qui répond en langage naturel à ce que vous tapez... Un des aspects ludiques de l'intelligence artificielle. Pour la petite histoire, MegaHAL fut l'un des premiers programmes à passer le test de Turing et atteint le point où il n'est plus possible de distinguer si l'interlocuteur est un humain ou une machine (enfin... selon les critères du test). L'algorithme est basé sur des chaînes de Markov. [2]. Il est capable «d'apprendre» quoi dire en observant les phrases qu'on lui écrit. Nous n'aborderons pas la programmation de ce logiciel, qui est disponible sous forme de logiciel libre. (Sur votre Ubuntu préféré : `aptitude install megahal`)

Nous considérons donc que nous disposons déjà du cerveau : le programme `megahal`, installé et configuré sur la machine serveur. Ce programme lit des chaînes de caractères sur son entrée standard, et écrit ses réponses sur sa sortie standard (au moment où l'on tape une ligne vide, mais peu importe).

Puisque nous avons le cerveau, reste à développer les jambes. Dans cet exercice, nous allons nous intéresser à un programme qui va gérer les aspects serveur TCP et lancer l'exécution du programme `megahal` de manière adéquate. (Et oui, dans cet exercice MegaHAL n'est qu'un prétexte, n'importe quoi d'autre aurait plus faire l'affaire, mais MegaHAL est tout de même plus rigolo.)

1. Le système doit pouvoir répondre à plusieurs clients simultanément. On pourrait imaginer qu'à chaque demande de connexion d'un nouveau client, le système exécute une nouvelle

1. `http://megahal.alioth.debian.org/`
2. `http://www.cnts.ua.ac.be/conll98/proceedings.html`

instance du logiciel `megahal` pour parler avec le client en question. Mais nous ne souhaitons pas procéder ainsi. Pour la beauté de l'exercice, et parce que l'on estime que notre grand oracle est omniscient et unique, nous souhaitons n'avoir qu'une seule instance du logiciel `megahal`. Même s'il est sollicité par plusieurs clients à la fois, tous les clients parleront à la même instance de `megahal`. Cette exigence exclue l'une des formes classiques de serveur vue en cours/TP, et nous oriente vers une autre. Laquelle et pourquoi ? L'un des exercices vus en TP met justement en œuvre ce type de serveur ; voyez-vous lequel ?

2. Cependant, le type de connexion est différent : ici on utilise le protocole TCP. Quels sont les changements à apporter pour répondre à cela ?
3. Notre serveur doit exécuter `megahal`. Le programme `megahal` communique classiquement via ses entrées-sorties standards (i.e. clavier-écran). Avant d'exécuter ce programme, notre serveur doit donc préparer le terrain pour s'assurer qu'il pourra communiquer avec lui par la suite pour relayer les messages des clients. Quels sont les mécanismes qu'il doit mettre en place, et avec quels appels systèmes ?
4. D'une manière générale, comment un serveur sait-il qu'un client est partit et a fermé sa connexion ? Sur quel appel système peut-il s'en rendre compte, et de quelle manière ?

2 Rétro-ingénierie d'un code C

L'annexe A donne le code source d'un mystérieux programme. C'est une portion d'un vrai programme, d'une commande Unix standard. [3] J'ai juste extrait et simplifié la partie la plus intéressante.

Étudier ce code, et expliquer ce qu'il fait.

(Question bonus : avez vous reconnu la commande Unix standard dont il est extrait ?)

3 Questions de cours

- Quelles différences faites-vous entre `close()` `unlink()` et `shutdown()` ?
- Lorsque l'on a utilisé les appels systèmes `bind()` `accept()` `connect()`, l'API de ces fonctions nous ammène systématiquement à faire un transtypage (ou *cast* selon la terminologie du langage C) pour le second argument. Pourquoi ?

3. `http://www.gnu.org/software/coreutils/`

Annexe A

Programme à étudier

```
1  bool mysterious_code(int nfiles, const char **files) {
2    FILE **descriptors;
3    char buffer[BUFSIZ];
4    ssize_t bytes_read;
5    int i;
6    bool ok = true;
7
8    descriptors = malloc((nfiles + 1) * sizeof(FILE *));
9
10   /*
11    * ...
12    * un peu de code ici pour rearranger la structure files[]
13    * fournie en parametre
14    * ...
15    * suivi de :
16    */
17   files[0] = "standard output";
18   descriptors[0] = stdout;
19
20   for (i = 1; i <= nfiles; i++) {
21     descriptors[i] = fopen(files[i], "w");
22     if (descriptors[i] == NULL) {
23       perror("fopen");
24       return false;
25     }
26   }
27
28   while (1) {
29     bytes_read = read(0, buffer, BUFSIZ);
30     if (bytes_read <= 0)
31       break;
32
33     for (i = 0; i <= nfiles; i++)
34       if (descriptors[i]
35           && fwrite(buffer, bytes_read, 1, descriptors[i]) != 1) {
36         perror("fwrite");
37         descriptors[i] = NULL;
38         ok = false;
39       }
40   }
41
42   if (bytes_read == -1) {
43     perror("read");
44     ok = false;
45   }
```

```
46
47    for (i = 1; i <= nfiles; i++)
48      if (descriptors[i] && fclose(descriptors[i]) != 0) {
49        perror("fclose");
50        ok = false;
51      }
52
53    free(descriptors);
54
55    return ok;
56  }
```