Université Joseph Fourier, Grenoble
Licence d'Informatique L3

Introduction aux Systèmes et Réseaux

TD n°6 : Client-serveur avec *sockets*

Le but de ce sujet de TD est d'étudier les primitives principales pour la communication par sockets afin de préparer le TP7 sur le thème du client-serveur réparti.

La documentation technique *"Une bibliothèque C pour les sockets"* (disponible dans le placard électronique) décrit en détails les primitives `Open_listenfd` et `Open_clientfd` (sections 1 à 3) ainsi que les principales structures de données et fonctions utilitaires nécessaires pour programmer des communications par *sockets* en langage C.

1 Serveur itératif

Reprendre l'exemple du client-serveur simple (itératif) vu en cours en identifiant le rôle joué par les différentes primitives. Quelles sont les principales restrictions de ce serveur ? On trouvera ci-après le code correspondant aux trois fichiers `echoserveri.c`, `echoclient.c` et `echo.c`.

Fichier echoserveri.c :

```
1   #include "csapp.h"
2
3   #define MAX_NAME_LEN 256
4
5   void echo(int connfd);
6
7   /*
8    * Note that this code only works with IPv4 addresses
9    * (IPv6 is not supported)
10   */
11  int main(int argc, char **argv)
12  {
13      int listenfd, connfd, port;
14      socklen_t clientlen;
15      struct sockaddr_in clientaddr;
16      char client_ip_string[INET_ADDRSTRLEN];
17      char client_hostname[MAX_NAME_LEN];
18
19      if (argc != 2) {
20          fprintf(stderr, "usage: %s <port>\n", argv[0]);
21          exit(0);
22      }
23      port = atoi(argv[1]);
24
25      clientlen = (socklen_t)sizeof(clientaddr);
26
```

```
27      listenfd = Open_listenfd(port);
28      while (1) {
29
30          connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
31
32          /* determine the name of the client */
33          Getnameinfo((SA *) &clientaddr, clientlen,
34                      client_hostname, MAX_NAME_LEN, 0, 0, 0);
35
36          /* determine the textual representation of the client's IP address */
37          Inet_ntop(AF_INET, &clientaddr.sin_addr, client_ip_string,
38                    INET_ADDRSTRLEN);
39
40          printf("server connected to %s (%s)\n", client_hostname,
41                 client_ip_string);
42
43          echo(connfd);
44          Close(connfd);
45      }
46      exit(0);
47  }
```

Fichier echoclient.c :

```
1   #include "csapp.h"
2
3   int main(int argc, char **argv)
4   {
5       int clientfd, port;
6       char *host, buf[MAXLINE];
7       rio_t rio;
8
9       if (argc != 3) {
10          fprintf(stderr, "usage: %s <host> <port>\n", argv[0]);
11          exit(0);
12      }
13      host = argv[1];
14      port = atoi(argv[2]);
15
16      /*
17       * Note that the 'host' can be a name or an IP address.
18       * If necessary, Open_clientfd will perform the name resolution
19       * to obtain the IP address.
20       */
21      clientfd = Open_clientfd(host, port);
22
23      /*
24       * At this stage, the connection is established between the client
25       * and the server OS ... but it is possible that the server application
26       * has not yet called "Accept" for this connection
27       */
28      printf("client connected to server OS\n");
```

```
29
30      Rio_readinitb(&rio, clientfd);
31
32      while (Fgets(buf, MAXLINE, stdin) != NULL) {
33          Rio_writen(clientfd, buf, strlen(buf));
34          if (Rio_readlineb(&rio, buf, MAXLINE) > 0) {
35              Fputs(buf, stdout);
36          } else { /* the server has prematurely closed the connection */
37              break;
38          }
39      }
40      Close(clientfd);
41      exit(0);
42  }
```

Fichier echo.c :

```
1   #include "csapp.h"
2
3   void echo(int connfd)
4   {
5       size_t n;
6       char buf[MAXLINE];
7       rio_t rio;
8
9       Rio_readinitb(&rio, connfd);
10      while ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
11          printf("server received %u bytes\n", (unsigned int)n);
12          Rio_writen(connfd, buf, n);
13      }
14  }
```

2 Serveur concurrent

Considérer ensuite le principe du serveur concurrent avec création dynamique de processus exécutants (dont le principe a été vu en cours) et réfléchir aux modifications à apporter au code du serveur. Quel est le principal inconvénient de cette approche ?

Considérer enfin le cas d'un serveur concurrent avec un ensemble prédéfini de processus exécutants. Quelle est la répartition précise du travail à effectuer entre le processus veilleur et les processus exécutants (plusieurs réponses sont envisageables) ? En termes de programmation, quelles sont les principales difficultés introduites par rapport aux architectures des serveurs précédents ? Quelles sont les informations à échanger entre le veilleur et les exécutants ? Comment peut-on y parvenir ?