TD 2 - ASR5 système d'exploitation

Protocols

5 mars 2018

I Lire jusqu'à un fanion

De nombreux protocoles de communication ont besoin d'échanger des données de taille variable. Cela pose problème car lors de l'échange, celui qui lit les données doit avoir un moyen de reconnaître la fin du message.

L'une des méthodes pour résoudre ce problème est de choisir un fanion. C'est-à-dire un octet spécial (ou un groupe d'octets) qui marque la fin du message.

Q.I.1) - Donnez 2 autres méthodes permettant résoudre le même problème.

Solution: On peut :
- Transmettre un entête de taille connue contenant la taille totale du message. Celui qui lit peut alors lire l'entête, extraire la taille du reste depuis l'entête puis lire le reste. Par exemple, cette technique est utilisée dans les websockets, les paquets IPs, ...
- Fermer la socket à la fin du message. Celui qui lit est toujours prévenu de la fermeture de cette socket et est donc informé qu'il n'y a plus rien à lire. Par exemple, cette technique est utilisée par le protocole HTTP dans la première version, le protocole FTP lors de la transmission de données, ...
- Découper le message en petits morceaux de taille fixes. Il y a alors un morceau spécial qui marque la fin du message.

Pour pouvoir utiliser les fanions, il faut être capable de lire les données jusqu'au fanion lui-même, sans en oublier et sans lire de données en trop.

Q.I.2) - Donnez le code d'une fonction :

```
vector<char> read_until(int sock, char until);
```

Qui lit exactement de qui est envoyé sur la socket `sock` jusqu'au prochain caractère `until`

Solution: Cette fonction est difficile à écrire car les primitives qui permettent de lire sur le *file descriptor* `sock` consomment les données. C'est-à-dire que si on lit par erreur les données qui se trouvent après le fanion de fin.Une fois lu, on ne peut plus revenir en arrière.
Dans le cas de certains *file descriptor*, il est possible de trouver une solution. Par exemple, dans le cas des fichiers, les données n'ont pas disparues, seul le marqueur de la lecture dans le fichier a été déplacé. Il suffit de revenir en arrière (voir la fonction `lseek()`). Dans le cas des sockets, il est possible de tester la lecture en laissant les données dans le flux (voir l'option MSG_PEEK de la fonction `recv()`). Mais dans le cas général, on ne peut que lire caractère par caractère.

```
vector<char> read_until(int sock, char fanion) {
  // le résultat est un tableau vide au départ
```

```
  vector<char> res;

  while(1) {
    char b;

    int r = read(sock, &b, 1);
    if (r == -1) {// c'est une erreur de lecture
    }
    if (r == 0) {
      // la socket est fermée avant le fanion
      cerr << "Attention, fermeture de la socket" << endl;
      break;
    }
    res.push_back(b);
    if (b == fanion) {
      // c'est fini
      break;
    }
  }

  return res;
}
```

Q.I.3) - La solution proposée n'est pas très efficace. Comment peut-on faire mieux ?

Solution: La lecture caractère par caractère demande beaucoup d'appels système pour une simple lecture d'un message complet. Ce n'est donc pas très efficace.
On peut utiliser la solution de java, c'est-à-dire créer une classe qui englobe les sockets et contient un buffer (voir les `java.io.BufferedReader`. Les octets lus en trop sont stockées dans ce buffer jusqu'à la prochaine lecture.
Cette méthode a un gros inconvénient, la prochaine lecture **doit** être faite en tenant compte du buffer. Donc elle rend inutilisable la socket elle-même.

II Protocole HTML

Un grand nombre de protocoles réseaux sont basés sur un échange de données sous la forme de chaines de caractères (`ftp, http, smtp, ...`). Dans ce cas, les messages sont souvent composés de lignes et l'une des fonctions de base pour reconnaitre la fin d'un message justement de lire correctement une ligne. pour pouvoir analyser chaque ligne séparément.

Par exemple, un client et un serveur web (`HTTP/1.1`) échangent des lignes terminées par \r\n. La requête `GET` est composée de plusieurs lignes et est terminée par une ligne vide. La réponse comporte un entête (terminé par aussi une ligne vide). Après cet entête, le serveur envoie le contenu qu'il faut lire en totalité avant de faire d'autre requête. Pour cela il est bien entendu nécessaire de connaitre la taille du contenu.

Q.II.1) - Donnez l'algorithme pour lire l'ensemble de l'entête.

Solution:

```
début
    lignelue ← "non"
    entete ← ""
    // Lecture entête  tant que lignelue ≠ "" faire
        lignelue ← RecoieLigne(sock)
        entete ← entete+lignelue
fin
```

Q.II.2) - Comment peut-on connaitre la taille du contenu ?

Solution: Il y a 3 méthodes :
- Le serveur peut couper la communication à la fin de la page. C'est ce qui était fait avec la première version du protocole. Mais ce n'est pas très efficace car une page web comporte des images, des fichier de présentation CSS, ... Il faut donc faire beaucoup de connexion pour tout télécharger.
- Le serveur peut se servir de l'entête pour donner la taille, c'est ce qui est fait via le champs `content-length`. C'est efficace pour les petites pages web.
- Le serveur peut découper la page en morceaux et envoyer chaque morceau séparément avec un petit entête donnant sa taille. C'est ce qui est fait par le mode `chunked`.

Le protocole `HTTP/1.1` propose 2 méthodes :
- il peut être transmis en un seul morceau, sa taille est alors signifiée directement dans l'entête par le champ `Content-length` ;
- il peut être en plusieurs morceau, l'entête contien alors la ligne Transfer−Encoding: chunked. Chaque morceau est alors composé d'un d'une ligne avec sa taille (en hexadécimal), puis du contenu. On reconnait la fin du contenu par un morceau de taille 0.

Q.II.3) - En supposant que vous disposez d'une fonction `RecoieLigne` capable de lire une ligne entière sur la socket et une fonction `RecoieTailleFixee` capable de lire un tableau de données de taille connu, proposez un algorithme pour lire la totalité d'une réponse web.

Solution:

```
début
    lignelue ← "non"
    entete ← ""
    contlength ← 0
    chunk ← false
    // Lecture entête  tant que lignelue ≠ "" faire
        lignelue ← RecoieLigne(sock)
        entete ← entete+lignelue
        si lignelue débute par "Content-Length" alors
            contlength ← fin de lignelue convertie en int
        si lignelue ressemble à "^Transfer-Coding *: *chunked$" alors
            chunk ← true ;
    si chunk alors
        // lire chunks
        taillechunk ← converti(RecoieLigne(sock))
        tant que taillechunk ≠ 0 faire
            chunk ← RecoieTailleFixee(sock, taillechunk)
            taillechunk ← converti(RecoieLigne(sock))
    si contlength ≠ 0 alors
        // lire une valeur
        contenu ← RecoieTailleFixee(sock, taillechunk)
    sinon
        erreur
fin
```