

# Epreuve *Travaux pratiques de programmation* sujet 0

On s'attendra à ce que les candidats traitent de manière au moins partielles les différentes parties : programmation en OCaml, en C, audit de code.

## 1 Programmation

Dans cette partie de l'épreuve, il est demandé au candidat de concevoir, programmer et tester un répartiteur de charge pour un système de signature de blocs. La première partie sera à implémenter en OCaml, les deux parties suivantes seront à implémenter en C. Vous trouverez un rappel de certaines fonctions en fin du sujet, et pouvez utiliser la commande `man` pour avoir des détails sur ces commandes.

**Livrables :** On rappelle qu'il sera nécessaire lors du rendu de présenter le travail réalisé, les forces et faiblesses des approches utilisées, ainsi que d'explicitier et de commenter les jeux de tests utilisés.

### 1.1 Principe

On considère avoir un certain nombre de clients voulant accéder à un service de signature de blocs. Pour pouvoir gérer le grand nombre de clients, on veut répartir de manière dynamique la charge sur plusieurs serveurs.

Le schéma suivant (Figure 1) montre un exemple d'un tel système.

Les requêtes des clients arrivent au travers du réseau (flèches gauches) vers le répartiteur de charge (au centre). Ce répartiteur va répartir les requêtes (sans les modifier) sur les différents serveurs capables de les traiter.

### 1.2 Gestion de la répartition de charge en *OCaml*

Pour évaluer différents algorithmes de répartition de charge, cette partie se concentre sur l'implémentation de la simulation du cœur du système. Il s'agit de la boîte du centre du schéma de la figure 1. Dans un système réel, cette partie reçoit par le réseau les requêtes et décide sur quel serveur les envoyer, et peut aussi décider de lancer un nouveau serveur.

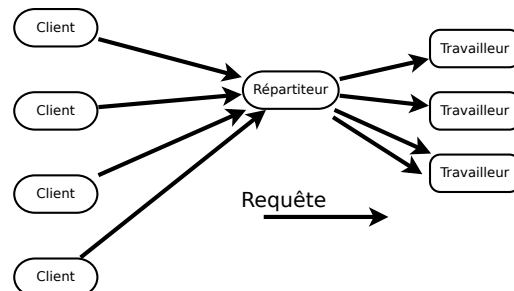


FIGURE 1 – Principe de fonctionnement du répartiteur de charge

Ici, le répartiteur de charge prends une liste de tâches en attente, l'état des serveurs de travail et décide, pour chaque tâche, sur lequel de ces serveurs exécuter cette tâche. Il peut s'agit d'un nouveau serveur. On considère ici qu'il n'est pas possible de déplacer une tâche en exécution. Pour simuler cette décision, les structures de données suivantes vont être utilisées :

**tache** Un triplet (*duree*, *charge*, *data*) représentant la tâche en attente d'exécution. *charge* est un réel entre 0 et 1 et représente la charge nécessaire sur le serveur pour exécuter la tâche sans baisse de qualité de service. *duree* est un réel positif donnant la durée de la tâche en secondes. *data* est la donnée qui est associée à la requête et est de type générique.

**etat\_serveurs** Une liste d'état de serveurs. Un état de serveur est représenté par une liste de tâches en cours d'exécution sur ce serveur. Les triplets sont comme décrits précédemment, hors le fait que *duree* représente alors le temps restant pour cette tâche.

Remarque : un état de serveur peut être une liste vide, signifiant que ce serveur est prêt à gérer des tâches, mais n'en exécute actuellement aucune.

On considèrera qu'un serveur est surchargé si la somme des charges qui lui sont allouées est strictement plus grande que 1.

On définira un répartiteur de charge comme étant une fonction *repartiteur* en OCaml prenant en entrée

- Une tâche (triplet)
- L'état des serveurs (liste de liste de triplets) ;
- Le nombre maximum de serveurs (entier positif). Si ce nombre est 0, il n'y a pas de limite au nombre de serveurs.

La fonction **repartiteur** renverra :

- Le nouvel état des serveurs (liste de liste de triplets). **Remarque** : Il est possible de créer un nouveau serveur (i.e. une liste) pour y placer la nouvelle tâche, mais pas de déplacer des tâches déjà allouées. De même on fera attention à conserver l'ordre des serveurs.

### 1.2.1 Ressources illimitées

L'algorithme *best-fit* consiste à insérer une tâche à l'endroit où la capacité restante est la plus faible après ajout de la tâche examinée. Dans le cas où plusieurs serveurs sont alors en concurrence, on choisira le premier. Dès qu'une tâche est placée, la décision n'est plus changée lors de l'étude des tâches suivantes.

Exemple

---

```
let tache = (1, 0.5, 42) in
let serveur1 = [(2, 0.6, 88)] in
let serveur2 = [(1, 0.2, 11); (19, 0.2, 8)] in
let serveur3 = [(1, 0.3, 27)] in
best-fit(tache, [serveur1; serveur2; serveur3], 0)
```

---

va renvoyer

---

```
[[ (2, 0.6, 88) ]; [(1, 0.2, 11); (19, 0.2, 8); (1, 0.5, 42)]; [(1, 0.3, 27)]]
```

---

car la tâche a une charge de 0.5, les serveurs ont respectivement 0.4, .6 et .7 en place libre. Le premier n'a donc pas assez de place, et parmi les deux autres, celui qui est le plus proche tout en ayant assez de place est le second.

**Question 1** Implémentez un algorithme *best-fit* en considérant que vous avez des ressources infinies (un nombre de serveurs infinis) et que l'on veut garantir qu'aucun serveur ne soit surchargé. Si il n'est pas possible d'ajouter la tâche sans provoquer une surcharge, on procèdera à la création d'un nouveau serveur à la fin de la liste des serveurs.

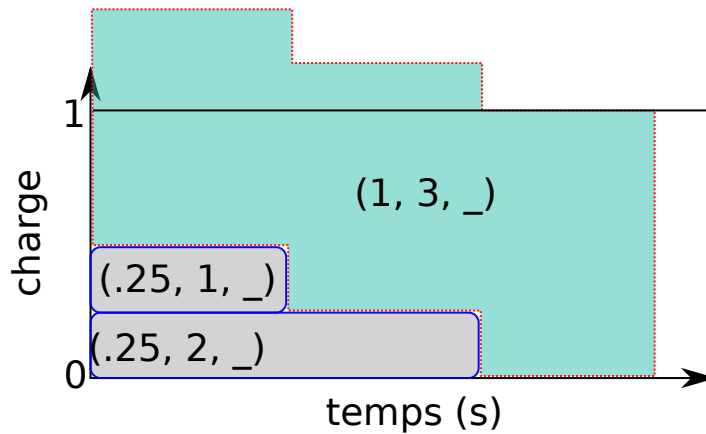


FIGURE 2 – Exemple de surcharge sur un serveur. Pendant la première seconde, la surcharge est de .5, pendant la seconde, elle est de .25 puis il n’y a plus surcharge. La surcharge totale est donc  $.5 \times 1 + .25 \times 1 = .75$ .

### 1.2.2 Ressources limitées

Lorsque les ressources sont limitées, il est parfois impossible de placer une tâche tout en garantissant une parfaite qualité de service. La surcharge d’un serveur est définie comme étant l’intégrale des charges au dessus de 1. Un exemple est montré sur la figure 2.

**Question 2** Implémentez une fonction `repartiteur` qui minimise la surcharge en tenant compte du nombre limité de serveurs.

## 1.3 Gestion du réseau en C

On s’intéresse maintenant aux tâches. Une tâche va être un programme permettant de signer des blocs de données. On considère avoir un tampon de taille constante  $T$  contenant des entiers positifs (`unsigned int`). Initialement ce tampon contient un seul élément 0, et l’on peut effectuer deux opérations : ajouter un entier à la fin, calculer la signature du tampon en cours et le vider en mettant dans la première case la signature. Par simplicité, on considèrera que la signature est le XOR de tous les éléments présents plus la signature précédente. Un exemple de ce comportement est montré sur la figure 3. Lorsqu’on tente d’ajouter un élément lorsque le tampon est plein, il y a automatiquement une signature pour faire de la place.

**question 3** Implémentez un `signeur` qui prend comme argument la taille du tableau ( $T$ ) et le port sur lequel écouter les ordres. Les ordres `add X` et `sign` sont à écouter sur le réseau, en TCP, chaque ordre donnant lieu à une nouvelle connexion.

**question 4** Implémentez un `test_signeur` qui envoie des commandes avec comme arguments : Le nombre de commandes à envoyer ; le ratio entre commande de type `add` et de type `sign` ; l’IP et le port du signeur.

**question 5** Évaluez la performance (en terme de requêtes par secondes) du signeur en fonction de  $T$  pour illustrer l’impact relatif de la performance de la hiérarchie mémoire et du réseau. Vous justifierez le choix des valeurs choisies pour  $T$ .

## 1.4 Gestion de la concurrence C

En réalité le nombre ajouté à la fin du bloc (par la commande `add X`) est le résultat d’un long calcul ne dépendant que de la valeur reçue sur le réseau (ici  $X$ ). Comme les calculs en eux-même sont indépendants, il est possible de paralléliser ces calculs en utilisant plusieurs processus légers (*threads*), un par ordre reçu sur le réseau.

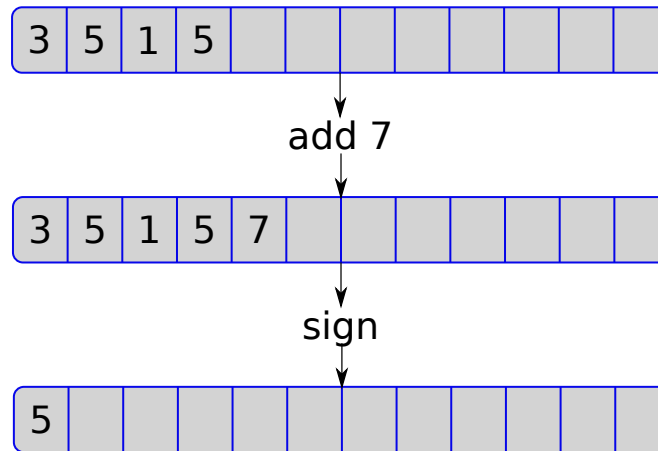


FIGURE 3 – Exemple de l’application de deux opérations **add 7**, puis **sign**. La précédente signature était 3, et le XOR de 3, 5, 1, 5, 7 est 5.

**question 6** en simulant le traitement long par un `usleep()`, proposez une implémentation multi-thread gérant les accès concurrents au tampon.

Remarque : Vous pouvez faire cette question indépendamment de la précédente en remplaçant la partie réseau par une boucle simulant la réception de messages.

## 2 Audit de code

Dans cette partie de l’épreuve, il est demandé au candidat d’étudier un fichier source qui comporte des erreurs ou des maladroresses, de qualité de code ou de fonctionnement, et il est demandé d’auditer ce fichier, c’est-à-dire :

- de comprendre et d’être capable d’expliquer le fonctionnement du code à l’oral ;
- de proposer des corrections en réécrivant certaines parties afin de corriger les erreurs ou maladroresses éventuelles et de rendre le code plus clair, notamment dans une optique pédagogique ;
- de proposer des améliorations de la complexité en temps ou en mémoire, ou de la sûreté du code.

Le temps indicatif de préparation de cette partie est d’une heure.

### 2.1 Code en C

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct a {
5     struct a **f;
6     int n;
7     int e;
8 };
9
10 int f(struct a *a) {
11     int s = a->e;
12     for (int i=0; i<a->n; i++)
13         s += f(a->f[i]);
14     return s;

```

```

15 }
16
17 struct a* cree(struct a **f, int n, int e)
18 {
19     struct a* a= malloc(sizeof(struct a));
20     a->f = f;
21     a->n = n;
22     a->e = e;
23     return a;
24 }
25
26 void supprime(struct a *a)
27 {
28     if (a->f) {
29         for(int i = 0; i < a->n; i++)
30             supprime(a->f[i]);
31         free(a->f);
32     }
33
34     free(a);
35 }
36
37 int main(int argc, char **argv)
38 {
39     struct a *a = cree(NULL, 0, 1);
40     struct a *b = cree(malloc(sizeof(struct a) * 2), 2, 1);
41
42     b->f[0] = a;
43     b->f[1] = a;
44
45     printf("%d\n", f(b));
46     supprime(b);
47
48     return 0;
49 }

```

---

## 2.2 Code en OCaml

---

```

1 let gen n =
2     Array.init n (fun _ ->
3         List.map fst
4             (List.filter snd
5                 (List.init n (fun i -> (i, Random.float 1.0 < 0.1))))))
6
7 let p g =
8     let n = Array.length g in
9     let e = Array.make n 0 in
10    let x = ref 0 in let f = ref true in
11    let c = ref 0 in
12    while !f do while !x < n && e.(!x) < 0 do incr !x done;
13        f := !x < n;
14    if f then begin
15        incr c; let p = ref [!x] in

```

```

16         while !p <> [] do
17             let v = List.hd !p in p := List.tl !p;
18             e.(v) <- c; let vl = ref g.(v) in
19             while !vl <> [] do let y = List.hd !vl in
20                 vl := List.tl !vl;
21                 if e.(y) == 0 then p = y :: !p
22             done
23         done
24     end done;
25     e
26
27 let _ =
28     let g = gen 10 in
29     let c = p g in
30     for i = 0 to 9 do
31         Printf.printf "%d->%d\n" i c.(i)
32     done

```

---

## Référence C

**socket** Créer un point de communication  
**getaddrinfo** Traduction d'adresses et de services réseau  
**connect** Débuter une connexion sur une socket  
**write** Écrire dans un descripteur de fichier  
**read** Lire depuis un descripteur de fichier  
**listen** Attendre des connexions sur une socket  
**accept** Accepter une connexion sur une socket  
**bind** Fournir un nom à une socket  
**recv** Recevoir un message sur une socket  
**close** Envoyer un message sur une socket  
**pthread\_create** Créer un nouveau thread  
**pthread\_join** Attendre la fin d'un autre thread  
**pthread\_mutex\_init** Opérations sur les mutex  
**PTHREAD\_MUTEX\_INITIALIZER** Opérations sur les mutex  
**pthread\_mutex\_lock** Opérations sur les mutex  
**pthread\_mutex\_unlock** Opérations sur les mutex  
**usleep** Interrompre le programme durant un nombre donné de microsecondes

Les documentations complètes de ces commandes sont accessibles en utilisant la commande **man**.

Rappel : **man -k mutex** permet de rechercher toutes les pages man en lien avec les mutex.

rappel sur la structure **struct addrinfo** (voir **man getaddrinfo**)

---

```
struct addrinfo {  
    int                ai_flags;  
    int                ai_family;  
    int                ai_socktype;  
    int                ai_protocol;  
    size_t             ai_addrlen;  
    struct sockaddr *ai_addr;  
    char               *ai_canonname;  
    struct addrinfo *ai_next;  
};
```

---