
Système de gestion d'un parc informatique

On s'attend à ce que les candidats abordent chacune des trois parties : programmation en C (en page 1), programmation en OCaml (en page 6) et audit de code (en page 10).

I. PROGRAMMATION

Nous nous intéressons à la mise en place d'un système de gestion d'un parc informatique. Le parc étant composé d'un ensemble de machines interconnectées, sa gestion est assurée par des processus démons qui s'exécutent sur ces machines. Il y a un démon par machine. Chaque démon est identifié de manière unique par un entier compris entre 0 et 999. La communication entre démons est bi-directionnelle.

La première partie de ce TP (en page 1), en langage C, concerne l'implémentation de la communication *distribuée* entre deux démons. La deuxième partie (en page 6), en langage OCaml, propose une modélisation *centralisée* qui illustre certains aspects algorithmiques de la communication à l'échelle du système.

1 Programmation en C : communication entre deux démons

Dans cette section, nous envisageons le cas simple de communication entre *deux* démons. Chaque démon est un processus indépendant à qui l'on permet d'effectuer des actions en parallèle en utilisant des *threads* (questions 2 à 7). La communication à mettre en place entre les deux démons est *distribuée* à base de *sockets* (questions 8 à 10).

1.1 Messages échangés entre démons

Les processus démons communiquent en échangeant des messages. Chaque message contient l'identité du démon émetteur, indique le type du message et transporte un contenu utile (*payload*). L'identité d'un démon est un entier positif, le type de message distingue entre requêtes (REQ) et acquittements (ACK) et le contenu est une suite d'octets de taille fixe. La structure de données utilisée pour les messages est définie comme suit :

```
#define PAYLOAD_SIZE 10

// types de messages, REQ = requête, ACK = acquittement
typedef enum {REQ,ACK} msgtype_t;

//type identifiant d'un démon
typedef unsigned int did_t;

// structure de message
typedef struct msg_s
{
    did_t id;                // identifiant du démon émetteur
    msgtype_t type;          // type de message
    char payload[PAYLOAD_SIZE]; // contenu du message
} msg_t;
```

Question 1. Définir une fonction

```
char* msg_to_string(msg_t* msg)
```

qui renvoie une représentation sous forme de chaîne de caractères d'un message passé en paramètre. Cette fonction vous sera utile pour afficher les messages échangés dans vos réponses aux questions à venir. Vous pouvez faire appel à des fonctions de la famille `printf` rappelées en Annexe A (page 11).

1.2 Gestion de tampon pour messages reçus

Un démon D_1 peut envoyer des messages à un démon D_2 à une vitesse qui est supérieure à la vitesse de traitement de ce dernier. Pour gérer ce phénomène, le démon D_2 gère un tampon de messages et fait usage du schéma *producteur-consommateur* à deux *threads*. Un premier thread se charge de réceptionner les messages et de les stocker dans le tampon ; le deuxième thread récupère les messages depuis le tampon et les traite. Le premier thread est *producteur* vis à vis du tampon puisqu'il y dépose des messages. Le deuxième thread est un *consommateur* puisqu'il récupère des messages du tampon.

Le producteur et le consommateur modifient tous les deux le tampon qui est une structure de données partagée. Pour garantir un état cohérent du tampon, les deux *threads* doivent impérativement *synchroniser* leurs accès. Dans le cas où le tampon est plein, le thread producteur doit attendre qu'un emplacement se libère pour pouvoir produire. Symétriquement, un consommateur ne peut consommer et doit attendre si le tampon est vide.

Le schéma producteur-consommateur est illustré à la figure 1.

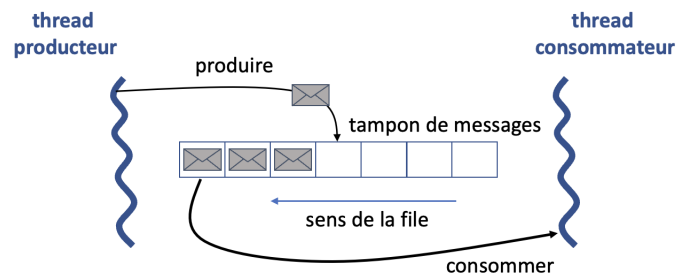


FIGURE 1 – Schéma producteur-consommateur pour la gestion d'un tampon à messages.

Dans la suite, vous allez d'abord définir la structure de données (question 2) et les fonctions (question 3) pour la manipulation *sans synchronisation* d'un tampon à messages. Ensuite, vous allez mettre en place le schéma producteur-consommateur en définissant les structures de synchronisation nécessaires (question 4) et le comportement des threads producteurs (question 5) et consommateurs (question 6). Enfin, vous allez assembler les différents éléments dans un test global créant plusieurs threads producteurs et consommateurs travaillant avec un tampon partagé (question 7).

Question 2. Un tampon à messages est une structure de file : les messages sont consommés dans l'ordre de leur production (arrivée). Pour des raisons de sécurité, le tampon a une taille fixe pré-définie et est géré à l'aide d'un tableau circulaire. Les éléments du tableau référencent les messages i.e sont de type `msg_t*`.

Définir la structure de données `C buffer_t` pour représenter un tampon à messages. Utiliser une constante `BUFFER_SIZE` pour la taille du tampon.

Question 3. *Définir les fonctions*

```
buffer_t* init_buffer();  
void put_message(buffer_t* T, msg_t* M);  
msg_t* get_message(buffer_t* T);
```

qui permettent de respectivement initialiser un tampon vide, produire et consommer un message dans/depuis le tampon sans considérer les aspects de synchronisation.

Synchronisation entre producteur et consommateur. La synchronisation entre threads producteurs et consommateurs doit utiliser des structures de verrous et de sémaphores. Pour la suite des questions, vous trouverez en Annexe B (page 12) une liste de fonctions de programmation et de synchronisation de threads.

Question 4. *Définir un type de structure `synchro_t` qui contient les données nécessaires à la synchronisation entre threads producteurs et consommateurs.*

Dans la suite vous allez mettre en place un programme C complet (question 7) avec plusieurs threads producteurs et consommateurs accédant à un tampon partagé. Le nombre de threads producteurs, n_P , et de threads consommateurs, n_C , sera à passer en ligne de commande (voir la fonction `atoi` en Annexe A, page 11). Les threads seront numérotés de 1 à n_P pour les producteurs, et de n_P+1 à n_P+n_C pour les consommateurs. Chacun des threads devra recevoir son numéro en paramètre, en plus des informations sur le tampon partagé et sur les structures de synchronisation. Le comportement des threads sera celui défini par les fonctions `produce` et `consume` des questions 5 et 6.

Question 5. *Réaliser la fonction*

```
void* produce(void*);
```

qui définit le comportement des threads producteurs. Un thread producteur ne doit produire qu'un seul message à la fois et se terminer. Le message à produire est à générer par le producteur comme suit. L'identifiant du message sera le numéro `NUM` du thread passé en paramètre. Le type du message sera à choisir arbitrairement entre `REQ` et `ACQ`. Enfin, le contenu utile du message sera une chaîne de la forme `prod_NUM`.

Question 6. *Réaliser la fonction*

```
void* consume(void*);
```

qui définit le comportement des threads consommateurs. Un thread consommateur ne doit consommer qu'un seul message à la fois, l'afficher et se terminer. Pour savoir quel thread consomme quel message, l'affichage indiquera également le numéro `NUM`, passé en paramètre, du thread.

Question 7. *Mettre en place le programme C principal qui se charge de l'initialisation du tampon partagé et des structures de synchronisation, ainsi que du lancement des threads producteurs et consommateurs. Le programme devra se terminer après la terminaison de tous les producteurs et consommateurs.*

1.3 Mise en place d'une communication par sockets

Cette sous-section a pour objectif de vous guider à travers les étapes d'interconnexion par *sockets* entre deux processus démons D_1 et D_2 . Les identifiants du démon D_1 et du démon D_2 sont les entiers 1 et 2.

Nous démarrons en question 8 par l'établissement d'une architecture client-serveur classique, puis nous évoluons en question 9 et 10 vers une interaction symétrique où les deux

démons sont des clients et des serveurs en même temps. Dans la version, les démons pourront envoyer et recevoir des messages en parallèle. Pour ce faire ils intégreront chacun une gestion de messages utilisant le schéma producteur-consommateur. Dans tous les échanges que vous mettrez en place le contenu utile des messages ne sera pas pris en compte et par conséquent pourra être choisi librement.

La bibliothèque CSAPP. Nous vous fournissons la bibliothèque CSAPP (`csapp.h`, `csapp.c`) qui simplifie grandement la programmation par sockets. Vous n'avez pas besoin de lire et de comprendre son fonctionnement en détail pour pouvoir l'utiliser.

L'annexe C (page 14) donne la liste des fonctions CSAPP disponibles pour le travail avec des sockets. L'annexe D (page 15) donne un exemple simple de système client-serveur utilisant ces fonctions. L'annexe E (page 18) donne des indications sur l'exécution de l'exemple.

Question 8. Adapter l'exemple client-serveur fourni afin de mettre en place une première version simple de communication entre les démons D_1 et D_2 sans tampon. Le démon D_1 doit prendre le rôle de client, alors que le démon D_2 sera le serveur. Le démon D_1 doit se connecter au démon D_2 , lui envoyer un message de type REQ et recevoir un message de type ACK en réponse. Après l'échange, les deux processus doivent se terminer.

Le fonctionnement est schématisé dans la figure 2.

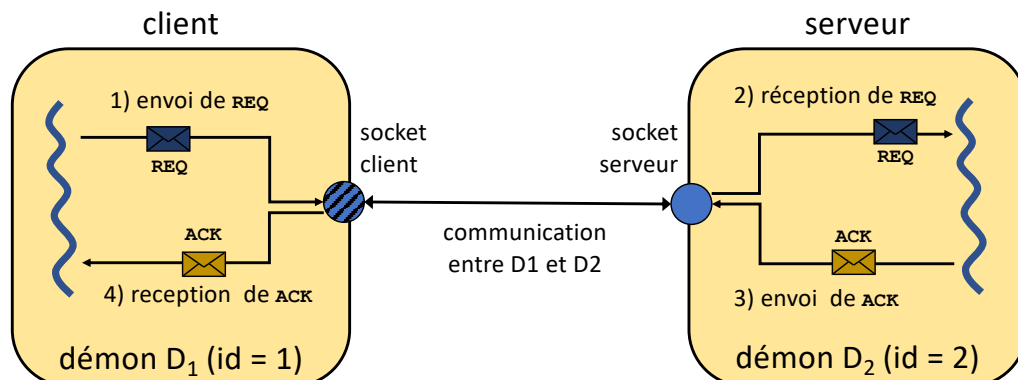


FIGURE 2 – Communication client-serveur entre deux démons D_1 et D_2 (cf. question 8).

Question 9. Enrichir la version fournie à la question 8 pour rendre les démons D_1 et D_2 symétriques : les deux doivent jouer le rôle de client et de serveur en même temps. Le démon D_1 doit se connecter en tant que client au démon D_2 , comme le démon D_2 doit se connecter en tant que client au démon D_1 . Un démon client doit gérer les échecs des tentatives de connexion le temps que le deuxième démon se mette à l'écoute.

L'échange à mettre en place est le suivant. Le démon D_1 doit envoyer, en tant que client, un message de type REQ au démon D_2 . Pour répondre, le démon D_2 doit envoyer un message de type ACK. Toutefois, pour ce faire, le démon D_2 doit utiliser sa connexion en tant que client du démon D_1 . Après cet échange, les deux processus doivent se terminer.

Le fonctionnement est schématisé dans la figure 3.

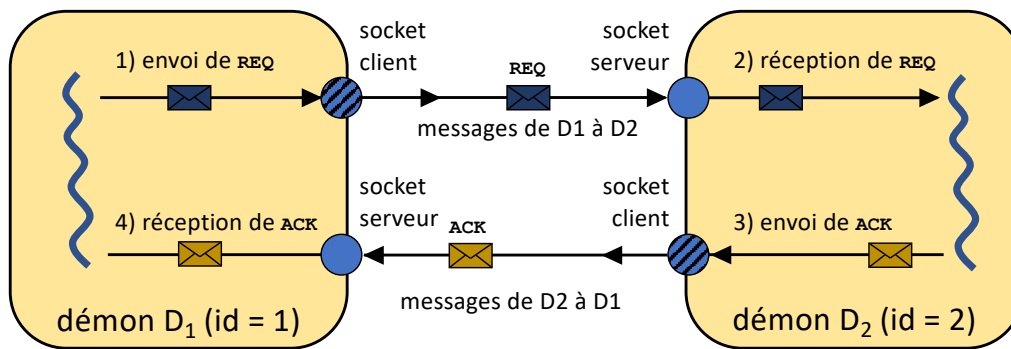


FIGURE 3 – Communication *symétrique* entre deux démons D_1 et D_2 (cf. question 9).

Question 10. Pour permettre aux démons D_1 et D_2 de s'envoyer des requêtes à n'importe quel moment, adapter la solution de la question 9 pour y intégrer une gestion de tampon avec le schéma producteur-consommateur. Les démons D_1 et D_2 doivent chacun avoir leur propre tampon, un thread producteur et un thread consommateur. Pour un démon, ses threads producteur et consommateur se rajoutent au fil d'exécution (thread) du main. Les envois de messages d'un démon à l'autre démon doivent toujours se faire dans le sens client-serveur d'une connexion par sockets. Tout message reçu par un démon, qu'il soit de type REQ ou de type ACK, doit être déposé dans le tampon par son thread producteur. Le thread consommateur s'occupe du traitement d'un message reçu après l'avoir consommé depuis le tampon. Mettre en place le fonctionnement suivant où les deux démons ont le même comportement. Chaque démon commence par envoyer à l'autre démon un message de type REQ. À la réception d'un message REQ, un démon renvoie un message de type ACK. Un démon se termine après avoir traité le message REQ de l'autre démon et après avoir reçu un ACK.

2 Programmation en OCaml : modélisation d'échanges de messages

Dans toute la section 2, l'ensemble des démons et leurs liens sont représentés par un graphe non orienté. Ce graphe est fixé une fois pour toutes et c'est cet unique graphe qui est manipulé à travers le module `Graph` fourni et présenté ci-dessous.

Un processus démon ne connaît pas le graphe du système en sa totalité. Chaque démon, ou sommet du graphe, dispose seulement de quatre attributs dont le rôle est clarifié au fur et à mesure du sujet. Les quatre attributs sont une liste de démons `neighbours`, une valeur entière `root`, un identifiant entier de démon `parent` et un dictionnaire `table` qui associe deux identifiants de démons. Les identifiants des démons sont uniques et à valeur entre 0 et 999.

Le fichier `partie2.ml`, disponible sur les machines de TP et listé également en annexe F (page 19), définit le type `did_t` et le module `Graph` signé comme suit :

```
type did_t = int

module type GraphType = sig
  val create : unit -> unit
  val add_vertex : did_t -> did_t list -> unit
  val show : unit -> unit
  val vertices : unit -> did_t list
  val get_neighbours : did_t -> did_t list
  val get_root : did_t -> int
  val set_root : did_t -> int -> unit
  val get_parent : did_t -> did_t
  val set_parent : did_t -> did_t -> unit
  val get_table : did_t -> did_t -> did_t
  val mem_table : did_t -> did_t -> bool
  val set_table : did_t -> did_t -> did_t -> unit
  val get_table_keys : did_t -> did_t list
end
```

Le type `did_t` est le type des identifiants des démons.

La fonction `Graph.create ()` initialise le graphe du système comme un graphe vide. La fonction `Graph.add_vertex v neigh` ajoute au graphe un démon d'identifiant `v` et relié aux démons de la liste `neigh`. La fonction `Graph.show ()` affiche les données du graphe construit.

Les fonctions `get_neighbours v`, `get_root v` et `get_parent v` renvoient respectivement la liste des voisins, l'attribut `root` et l'attribut `parent` du démon `v`. La fonction `get_table v u` renvoie la valeur associée à la clé `u` dans l'attribut `table` du démon `v`.

La fonction `set_root v w`, respectivement `set_parent v w` et `set_table v u w`, a pour effet de donner à l'attribut `root`, respectivement `parent` et `table u`, du démon `v`, la valeur `w`.

La fonction `mem_table v u` teste si la clé `u` appartient à l'attribut `table` du démon `v`. La fonction `get_table_keys v` renvoie l'ensemble des clés de l'attribut `table` du démon `v`.

La fonction `vertices ()` renvoie la liste des démons du système : il ne sera autorisé de l'utiliser qu'à l'initialisation dans les questions 13 et 15.

Question 11. *Écrire un programme OCaml permettant d'initialiser le module `Graph` avec le graphe de la figure 4.*

Vous pouvez proposer des illustrations supplémentaires sur des graphes plus élaborés.

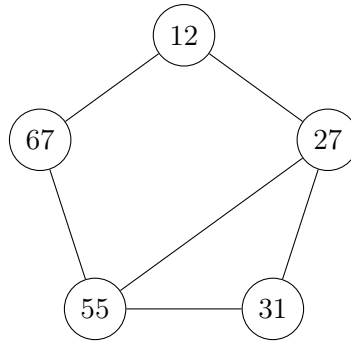


FIGURE 4 – Exemple de système formé de 5 démons.

Afin de modéliser les communications entre démons, on considère que les démons s'échangent des messages de type `msg_t` par le biais d'un système de communication global accessible via le module `Channel`. Le type `msg_t` et le module `Channel` sont aussi fournis dans le fichier `partie2.ml` et sont signés comme suit :

```

type msg_t = did_t * int

module type ChannelType = sig
  val is_empty : unit -> bool
  val produce : did_t -> did_t -> int -> unit
  val consume : unit -> did_t * msg_t
end

```

Le type `msg_t` définit un message comme un couple contenant l'identifiant du démon émetteur et un contenu utile représenté par un entier.

La fonction `Channel.produce u v x` simule l'envoi par le démon `u` au démon `v` d'un message de contenu utile `x`.

La fonction `Channel.consume ()` a pour valeur de retour un couple `(v, (u, x))` et simule la réception par le démon `v` d'un message encore en attente émis par le démon `u` et de contenu utile `x`. Il est garanti que, pour un démon `v` donné, `consume` respecte l'ordre dans lequel les messages ont été envoyés à `v`.

La fonction `Channel.is_empty ()` renvoie `true` si tous les messages produits ont été consommés et renvoie `false` sinon.

Question 12. *Écrire une fonction OCaml*

```

send_to_all_neighbours (v:did_t)(x:int): unit

```

qui simule l'envoi du contenu utile `x` à l'ensemble des voisins du démon `v`.

Dans chacune des questions 13, 15, 16 et 17 à venir, nous simulerons une série d'interactions entre démons par un traitement séquentiel des messages qu'ils s'échangent. On utilisera systématiquement le même modèle de "code à trous" présenté en figure 5. On préservera la modélisation de démons communiquant entre eux sans connaître l'ensemble (le graphe) du système en n'utilisant que les *getters* et *setters* prévus par le module `Graph`.

Arbre couvrant. Dans la question 13, nous souhaitons simuler les échanges qui permettent aux démons de s'entendre sur un *arbre couvrant commun*. L'arbre couvrant à construire sera enraciné dans le nœud d'identifiant de valeur minimale. Il sera représenté par l'ensemble des attributs `root` et `parent` des démons du système. Pour chaque démon, l'attribut `root` sera un entier dont le nombre des milliers reflétera l'identifiant du démon racine de l'arbre et le nombre des unités le nombre de sauts pour atteindre la racine. L'attribut `parent` indiquera le début du chemin orienté menant à la racine.

```

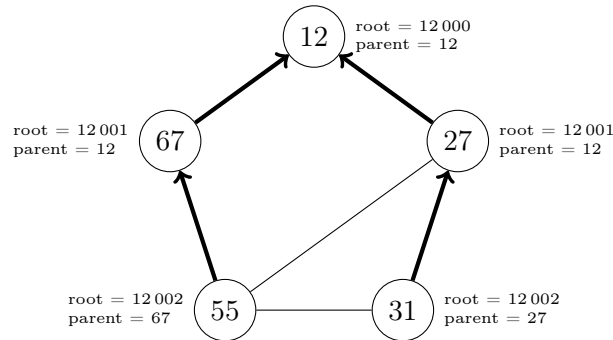
let function_name some_arg1 some_arg2 ... =
  (* phase d'initialisation A COMPLETER *)
  while not (Channel.is_empty ()) do
    let v, (u, x) = Channel.consume () in
    (* phase de dialogue A COMPLETER
       n'utilisant que les getters et setters
       liés au démon v *)
  done

```

FIGURE 5 – Modèle de code pour la simulation d’une phase de communications.

L’algorithme de construction que nous voulons mettre en place est le suivant. Au démarrage, chaque démon u communique à ses voisins la valeur courante de son attribut $root$, lequel est initialisé à $1000 \cdot u$. Lorsqu’un démon v , possédant un attribut $root\ r_v$, est informé par message qu’un de ses voisins u détient un attribut $root\ r_u$ tel que $r_u + 1 < r_v$, le démon v met à jour son propre attribut $parent$ à la valeur u et corrige son attribut $root$ à la valeur $r_u + 1$. De plus, il informe ses voisins de sa nouvelle situation en leur communiquant la valeur r_v mise à jour.

Dans le cas de la figure 4, on peut par exemple obtenir l’arbre suivant :



Question 13. Compléter le modèle de code de la figure 5 afin d’écrire une fonction OCaml

`build_tree () : unit`

dont l’effet est de simuler la construction d’un arbre couvrant par les démons.

Dans le reste de cette section, nous supposons que les démons se sont accordés sur un arbre couvrant.

Question 14. Écrire une fonction OCaml

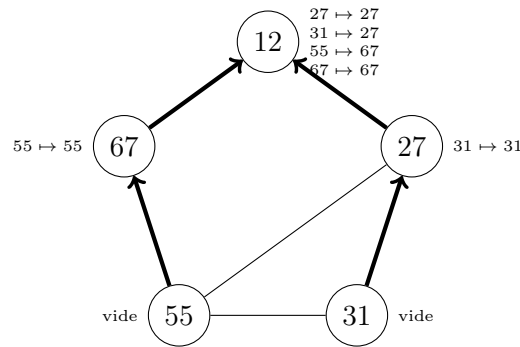
`send_to_parent (v:did_t) (x:int) : unit`

qui simule l’envoi du contenu utile x au parent du démon v .

Routing. Nous souhaitons que chaque démon construise sa table de routage (son attribut `table`) permettant d’envoyer des messages le long de l’arbre couvrant. Ainsi, une fois ces tables construites et pour tous démons v et z distincts, si le démon v veut envoyer un message à z , `get_table v z` renverra le premier démon rencontré immédiatement après v sur l’unique chemin allant de v à z .

Nous allons considérer une construction des tables de routage en deux phases. Dans la phase 1, chacun des démons fait remonter son identifiant vers la racine de l’arbre couvrant pour se signaler. Pour ce faire, tout démon z envoie initialement son identifiant à son parent. Si un démon reçoit un message, il le renvoie vers son parent. Lors de cette phase, les sommets v ainsi traversés peuvent calculer la valeur de `get_table v z`.

Dans le cas de la figure 4, à l’issue de la phase 1, on obtient les tables partielles suivantes :



Question 15. Compléter le modèle de code de la figure 5 afin d'écrire une fonction OCaml

`build_table_phase1 (r : did_t) : unit`

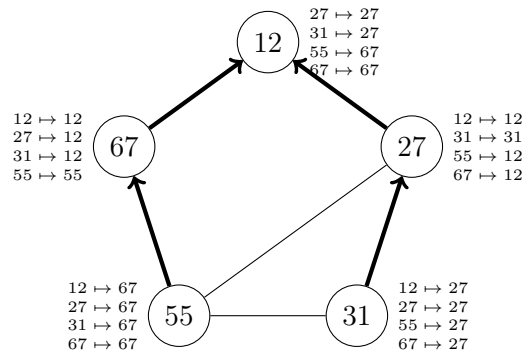
qui simule la phase 1 de la construction des tables.

Précondition : Un arbre couvrant a été calculé et est enraciné en le démon r . L'attribut `table` de tout démon est le dictionnaire vide.

Dans la suite du sujet, nous supposons que la première phase de construction des tables a été exécutée et que chaque démon sait qu'elle s'est terminée.

Dans la seconde phase de construction des tables de routage, le démon à la racine de l'arbre couvrant diffuse un à un l'identifiant de chacun des démons qu'il a identifié au terme de la phase 1. Le déroulement est le suivant. La racine commence par diffuser à tous ses voisins les identifiants des démons. Quand un démon reçoit un message depuis son parent, il l'envoie à tous ses voisins. Ces diffusions permettent à tous les démons du système de compléter leurs tables de routage entièrement.

Dans le cas de la figure 4, on obtient les tables de routage complètes suivantes :



Question 16. Compléter le modèle de code de la figure 5 afin d'écrire une fonction OCaml

`build_table_phase2 (r : did_t) : unit`

qui simule la phase 2 de la construction des tables.

Précondition : Un arbre couvrant a été calculé et est enraciné en le démon r , la phase 1 a été calculée.

Dans la dernière question, nous supposons que les deux phases de construction des tables de routage ont été exécutées.

Question 17. Compléter le modèle de code de la figure 5 afin d'écrire une fonction OCaml

`send_point_to_point (a : did_t) (b : did_t) : unit`

qui affiche la route suivie par un message de contenu utile b envoyé par le démon a vers le démon b .

II. AUDIT DE CODE

Dans cette partie de l'épreuve, il est demandé d'étudier un fichier source qui comporte des erreurs ou des maladdresses, 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 modifications afin de corriger les erreurs ou maladdresses 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.

Vous recevez un rendu de TP par courriel. Le listing du fichier joint (tp.py) est donné ci-dessous.

```
1 def f(p) :
2     if len(p) <= 1:
3         return p
4     else:
5         e = p[0]
6         p1 = []
7         p2 = []
8         for i in range(1, len(p)) :
9             if i < e:
10                p2.insert(0, i)
11            else:
12                p1.insert(0, i)
13            return f(p1) + [e] + f(p2)
14
15 def f2(p) :
16     d=0
17     for i in p:
18         d += 1
19     n = d
20
21     for i in range(n) :
22         todo = True
23         for j in range(n) :
24             if p[j] > p[j + 1]:
25                 p[j], p[j + 1] = p[j + 1], p[j]
26                 todo = False
27         if todo==True:
28             break
29
30     return p
31
32 t1 = [4, 2, 1, 3, 5]
33 t2 = f(t1)
34 t3 = f2(t1)
35 print(t2)
36 print(t3)
```

Question 18. Quel retour feriez-vous ?

ANNEXE A : Fonctions utiles en C

Pour plus de détails voir man.

NOM

printf, fprintf, sprintf, snprintf - Formatage des sorties

SYNOPSIS

```
#include <stdio.h>
```

```
int printf(const char *format, ...);  
int fprintf(FILE *stream, const char *format, ...);  
int sprintf(char *str, const char *format, ...);  
int snprintf(char *str, size_t size, const char *format, ...);
```

DESCRIPTION

Les fonctions de la famille **printf()** produisent des sorties en accord avec le format spécifié.

La fonction **printf()** écrit sa sortie sur stdout.

sprintf() et **snprintf()** écrivent dans la chaîne de caractères str.

La fonction **snprintf()** écrit au plus size octets (octet nul ('\0') inclus) dans str.

VALEUR RENVOYÉE

En cas de succès, ces fonctions renvoient le nombre de caractères affichés (sans compter l'octet nul final utilisé pour terminer les sorties dans les chaînes).

La fonction **snprintf()** n'écrit pas plus de size octets (y compris l'octet nul final). Si la sortie a été tronquée à cause de la limite, la valeur de retour est le nombre de caractères (octet nul final non compris) qui auraient été écrits dans la chaîne s'il y avait eu suffisamment de place. Ainsi, une valeur de retour size ou plus signifie que la sortie a été tronquée. Si une erreur de sortie s'est produite, une valeur négative est renvoyée.

NOM

atoi - Convertir une chaîne en entier

SYNOPSIS

```
#include <stdlib.h>
```

```
int atoi(const char *nptr);
```

DESCRIPTION

La fonction **atoi()** convertit le début de la chaîne pointée par nptr en entier de type *int*.

VALEUR RENVOYÉE

Le résultat de la conversion.

ANNEXE B : Threads et synchronisation en C

Compiler et effectuer l'édition des liens avec l'option **-pthread**.
Pour plus de détails voir man.

Threads

SYNOPSIS

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t*attr,
                   void *(*start_routine)(void*),
                   void *arg);

int pthread_join(pthread_t thread, void **retval);

void pthread_exit(void *retval);
```

DESCRIPTION

pthread_create() crée un nouveau thread s'exécutant simultanément avec le thread appelant. Le nouveau thread exécute la fonction start_routine en lui passant arg comme premier argument. Le nouveau thread s'achève soit explicitement en appelant **pthread_exit**, ou implicitement lorsque la fonction start_routine s'achève.

En cas de succès, l'identifiant du nouveau thread est stocké à l'emplacement mémoire pointé par l'argument thread, et 0 est renvoyé. En cas d'erreur, un code d'erreur non nul est renvoyé.

La fonction **pthread_join()** attend que le thread spécifié par thread se termine. Si ce thread s'est déjà terminé, **pthread_join()** revient tout de suite. Si retval n'est pas NULL, **pthread_join()** copie la valeur de sortie du thread cible (c'est-à-dire la valeur que le thread cible a fournie à **pthread_exit**) dans l'emplacement pointé par retval.

La fonction **pthread_exit()** termine le thread appelant et renvoie une valeur à travers retval qui, si le thread est joignable, est rendue disponible à un autre thread dans le même processus s'il appelle **pthread_join()**.

Verrous

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION

Un mutex est un objet d'exclusion mutuelle (MUTual EXclusion).

pthread_mutex_init initialise le mutex pointé par mutex selon les attributs de mutex spécifié par mutexattr. Si mutexattr vaut NULL, les paramètres par défaut sont utilisés.

Les variables de type `pthread_mutex_t` peuvent aussi être initialisées de manière statique, en utilisant `PTHREAD_MUTEX_INITIALIZER`.

pthread_mutex_lock verrouille le mutex.

pthread_mutex_trylock se comporte de la même manière que **pthread_mutex_lock**, excepté qu'elle ne bloque pas le thread appelant si le mutex est déjà verrouillé par un autre thread.

pthread_mutex_unlock déverrouille le mutex.

pthread_mutex_destroy détruit un mutex, libérant les ressources qu'il détient.

Sémaphores

SYNOPSIS

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

DESCRIPTION

sem_init() initialise le sémaphore non nommé situé à l'adresse pointée par sem. L'argument value spécifie la valeur initiale du sémaphore.

L'argument pshared indique si ce sémaphore sera partagé entre les threads d'un processus (pshared vaut 0) ou entre processus (pshared est différent de 0).

sem_wait() décrémente (verrouille) le sémaphore pointé par sem.

sem_post() incrémente (déverrouille) le sémaphore pointé par sem.

ANNEXE C : Bibliothèque CSAPP pour les sockets

Établissement de connexion

```
/*
 * Création et initialisation de socket serveur en écoute sur un port.
 */
int open_listenfd(char *port);

/*
 * Connexion à un socket serveur d'écoute sur une machine et un port.
 * Renvoie le descripteur de socket pour des lectures/écritures.
 */
int open_clientfd(char *hostname, char *port)
```

Fonctions de lecture/écriture

Ne jamais mélanger des lectures avec tampon avec des lectures sans tampon depuis le même fichier.

```
/*
 * Lit n octets depuis le fichier avec descripteur fd
 * et les stocke à l'emplacement mémoire usrbuf.
 * Lecture sans tampon.
 * Renvoie le nombre d'octets effectivement lus ou -1 en cas d'erreur.
 */
ssize_t rio_readn(int fd, void *usrbuf, size_t n);

/*
 * Transfère n octets depuis l'emplacement mémoire usrbuf
 * vers le fichier avec descripteur fd.
 * Renvoie le nombre d'octets effectivement écrits ou -1 en cas d'erreur.
 */
ssize_t rio_writen(int fd, void *usrbuf, size_t n);

/*
 * Initialise la structure rio_p pour une lecture avec tampon
 * depuis le fichier avec descripteur fd.
 */
void rio_readinitb(rio_t *rp, int fd);

/*
 * Lit n octets depuis le fichier avec descripteur fd
 * et les stocke à l'emplacement mémoire usrbuf.
 * Lecture avec tampon.
 * Renvoie le nombre d'octets effectivement lus ou -1 en cas d'erreur.
 */
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);

/*
 * Lit une ligne de texte (y compris le caractère de fin de ligne)
 * depuis le fichier rp, la copie dans l'emplacement mémoire usrbuf
 * en terminant la ligne avec le caractère de fin de chaîne (zéro).
 * Lecture avec tampon.
 * Renvoie la longueur de la ligne.
 */
ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
```

ANNEXE D : Exemple de client-serveur avec CSAPP

Programme serveur

```
1  /*
2  * serveur.c - Serveur echo.
3  * Le serveur n'accepte qu'une connexion de client à la fois.
4  * Une fois un client connecté,
5  * le serveur traite ses requêtes avec la fonction echo.
6  * Une fois le client déconnecté,
7  * le serveur est prêt à accepter une nouvelle demande de connexion.
8  */
9  #include <assert.h>
10 #include "csapp.h"
11
12 void echo(int connfd);
13
14 int main(int argc, char **argv)
15 {
16     int listenfd, connfd;
17     socklen_t clientlen = sizeof(struct sockaddr_storage);
18     struct sockaddr_storage clientaddr;
19     char client_hostname[MAXLINE], client_port[MAXLINE];
20
21     if (argc != 2) {
22         fprintf(stderr,
23             "Le lancement du serveur exige un numéro de port > 1024\n : %s <port>\n",
24             argv[0]);
25         exit(0);
26     }
27
28     // mise en place du socket d'écoute
29     listenfd = open_listenfd(argv[1]);
30
31     // si échec, message d'erreur et arrêt.
32     if (listenfd < 0)
33         unix_error("Erreur lors de l'ouverture du socket d'écoute.");
34
35     printf("Le serveur est prêt. Ctrl-C pour l'arrêter.\n");
36
37     while (1) {
38         // accepter une demande de connexion
39         connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);
40
41         // si la connexion a bien été établie
42         if (connfd > 0) {
43
44             // afficher les informations de connexion
45             Getnameinfo((SA *) &clientaddr, clientlen, client_hostname,
46                 MAXLINE, client_port, MAXLINE, 0);
47             printf("Connecté à (%s, %s)\n", client_hostname, client_port);
48
49             // traiter l'interaction avec le client
50             echo(connfd);
51
52             // fermer la connexion
53             close(connfd);
54
55         } else { // échec de connexion, arrêt.
56             unix_error("Erreur lors de la connexion du client.");
57         }
58     }
59     exit(0);
60 }
```

Traitement côté serveur

```
1  /*
2  * echo.c : Traitement de requêtes client sur le serveur.
3  */
4  #include "csapp.h"
5
6  void echo(int connfd)
7  {
8      size_t n;
9      char buf[MAXLINE];
10     rio_t rio;
11
12     // initialisation pour lecture avec tampon (voir remarque)
13     rio_readinitb(&rio, connfd);
14
15     // tant que le client renvoie des chaînes de caractères
16     while((n = rio_readlineb(&rio, buf, MAXLINE)) > 0) {
17
18         // une chaîne de caractères de longueur n a été reçue
19         // afficher la chaîne
20         printf("SERVEUR : Reçu %d octets\n", (int)n);
21
22         // renvoyer la chaîne au client (echo)
23         rio_writen(connfd, buf, n);
24
25         // envoyer la longueur de la chaîne
26         rio_writen(connfd, &n, sizeof(size_t));
27     }
28     printf("SERVEUR : Le client s'est déconnecté.\n");
29 }
30
31 /*
32 * Remarque : ATTENTION
33 * ne jamais mélanger des lectures avec tampon
34 * (fonctions rio_readlineb et rio_readnb)
35 * avec
36 * des lectures sans tampon (fonction rio_readn)
37 * depuis la même source.
38 */
```

Programme client

```
1  /*
2  * client.c - Client echo.
3  * L'utilisateur saisit une chaîne sur le terminal.
4  * Le client envoie la chaîne au serveur.
5  * Le client reçoit depuis le serveur la même chaîne (echo) et
6  * un entier (la longueur de la chaîne).
7  */
8  #include <assert.h>
9  #include "csapp.h"
10
11 int main(int argc, char **argv)
12 {
13     int clientfd;
14     char *host, *port, buf[MAXLINE];
15     size_t n,r;
16
17     if (argc != 3) {
18         fprintf(stderr, "usage: %s <host> <port>\n", argv[0]);
19         exit(0);
20     }
21     host = argv[1];
22     port = argv[2];
23
24     // connexion au serveur
25     if ((clientfd = open_clientfd(host, port)) < 0) {
26         fprintf(stderr,
27             "Le serveur sur <%s, %s> n'est pas encore disponible.\n", host, port);
28         exit(-1);
29     }
30
31     printf("Le client est prêt. Ctrl-D pour l'arrêter.\n");
32     printf("Veuillez saisir une chaîne. \n");
33
34     while (Fgets(buf, MAXLINE, stdin) != NULL) { //lire depuis le clavier
35         // la chaîne lue contient le caractère de fin de ligne
36
37         // envoyer au serveur ce qui a été saisi au clavier
38         r=rio_writen(clientfd, buf, strlen(buf));
39         // s'assurer que toutes les données ont été envoyées
40         assert(r==strlen(buf));
41
42         // recevoir une chaîne du serveur du même nombre d'octets
43         r = rio_readn(clientfd, buf, strlen(buf)) ;
44         printf("CLIENT : Reçu la chaîne suivante du serveur (%d octets) \n", (int)r);
45
46         // afficher la chaîne reçue
47         Fputs(buf, stdout);
48
49         // recevoir du serveur un entier non signé
50         r=rio_readn(clientfd, &n, sizeof(size_t));
51         assert(r==sizeof(size_t));
52
53         // afficher l'entier reçu
54         printf("CLIENT : Reçu du serveur l'entier %d\n", (int)n);
55     }
56
57     // déconnexion du serveur
58     printf("CLIENT : Déconnexion du serveur.\n");
59     close(clientfd);
60     exit(0);
61 }
```

ANNEXE E : Lancement de l'exemple CSAPP

Compilation.

Nous vous fournissons un Makefile pour la compilation. La compilation des sources (`client.c`, `serveur.c` et `echo.c` exigent la prise en compte de `csapp.c`. Pour compiler et obtenir les exécutables pour le client et le serveur, effectuer :

TERMINAL

```
> make all
```

Exécution.

Le serveur peut être lancé dans un terminal comme ceci ¹ :

TERMINAL 1

```
> ./serveur 1234  
Le serveur est prêt. Ctrl-C pour l'arrêter.
```

Le client est à lancer dans un autre terminal de la manière suivante :

TERMINAL 2

```
> ./client localhost 1234  
Le client est prêt. Ctrl-D pour l'arrêter.  
Veuillez saisir une chaîne.
```

1. Le numéro de port peut être librement choisi entre 1024 et 65535.

ANNEXE F : Détail des modules OCaml fournis

```
1  type did_t = int
2
3  module type GraphType = sig
4      val create : unit -> unit
5      val add_vertex : did_t -> did_t list -> unit
6      val show : unit -> unit
7      val vertices : unit -> did_t list
8      val get_neighbours : did_t -> did_t list
9      val get_root : did_t -> int
10     val set_root : did_t -> int -> unit
11     val get_parent : did_t -> did_t
12     val set_parent : did_t -> did_t -> unit
13     val get_table : did_t -> did_t -> did_t
14     val mem_table : did_t -> did_t -> bool
15     val set_table : did_t -> did_t -> did_t -> unit
16     val get_table_keys : did_t -> did_t list
17 end
18
19 module Graph : GraphType = struct
20     type dinfo_t = { neighbours : did_t list;
21                     mutable root : int;
22                     mutable parent : did_t;
23                     table : (did_t, did_t) Hashtbl.t
24                     }
25     let g = (Hashtbl.create 1 : (did_t, dinfo_t) Hashtbl.t)
26
27     (** [create ()] (ré)-initialise le graphe au graphe vide *)
28     let create () = Hashtbl.clear g
29
30     (** [add_vertex v neigh] ajoute un nouveau sommet [v] et le relie à
31         tous les sommets de la liste de sommets [neigh].
32         Précondition : le sommet [v] n'appartient pas encore au graphe,
33         les sommets de la liste [neigh] appartiennent déjà au graphe. *)
34     let add_vertex (v:did_t) (neigh:did_t list) : unit =
35         let info = { neighbours = neigh;
36                     root = 1000 * v;
37                     parent = v;
38                     table = Hashtbl.create 1 }
39         in Hashtbl.add g v info ;
40         let update w =
41             let info_old = Hashtbl.find g w in
42             let info_new = { neighbours = v::info_old.neighbours;
43                             root = info_old.root;
44                             parent = info_old.parent;
45                             table = info_old.table} in
46             Hashtbl.replace g w info_new
47         in List.iter update neigh
48
49     (** [show ()] affiche le graphe sur la sortie standard. *)
50     let show () =
51         let display_vertex v info =
52             Printf.printf "Démon %d : root = %d, parent = %d.\n"
53                 v info.root info.parent
54         in
55         Hashtbl.iter display_vertex g
56
57     (** [vertices ()] est une liste des sommets du graphe. *)
58     let vertices () =
59         let cat k _ acc = k::acc in
60         Hashtbl.fold cat g []
61
```

```

62  (** [get_neighbours v] est la liste des sommets (ou démons) reliés
63      au sommet [v]. *)
64  let get_neighbours v = (Hashtbl.find g v).neighbours
65
66  (** [get_root v] est l'attribut `root` du sommet [v]. *)
67  let get_root v = (Hashtbl.find g v).root
68  (** Après exécution de [set_root v r], l'attribut `root` du sommet [v]
69      égale l'entier [r]. *)
70  let set_root v r = (Hashtbl.find g v).root <- r
71
72  (** [get_parent v] est l'attribut `parent` du sommet [v]. *)
73  let get_parent v = (Hashtbl.find g v).parent
74  (** Après l'exécution de [set_parent v w], l'attribut `parent` du
75      sommet [v] égale le sommet [w]. *)
76  let set_parent v w = (Hashtbl.find g v).parent <- w
77
78  (** [get_table v a] est le premier sommet après [v] sur la route allant
79      du sommet [v] au sommet [a]. *)
80  let get_table v a = Hashtbl.find (Hashtbl.find g v).table a
81  (** [mem_table v a] vérifie si le premier sommet après [v] sur la route
82      allant du sommet [v] au sommet [a] a été défini. *)
83  let mem_table v a = Hashtbl.mem (Hashtbl.find g v).table a
84  (** Après l'exécution de [set_table v a b], le premier sommet après [v]
85      sur la route allant du sommet [v] au sommet [a] vaut [b]. *)
86  let set_table v a b = Hashtbl.replace (Hashtbl.find g v).table a b
87  (** [get_table_keys v] est la liste des sommets [a] tels que premier
88      sommet après [v] sur la route allant du sommet [v] au sommet [a]
89      a été défini. *)
90  let get_table_keys v = let cat k _ acc = k::acc in
91      Hashtbl.fold cat (Hashtbl.find g v).table []
92 end
93
94 type msg_t = did_t * int
95 module type ChannelType = sig
96     val is_empty : unit -> bool
97     val produce : did_t -> did_t -> int -> unit
98     val consume : unit -> did_t * msg_t
99 end
100
101 module Channel : ChannelType = struct
102     type t = (int, msg_t Queue.t) Hashtbl.t
103
104     let chan = Hashtbl.create 1
105
106     (** [is_empty ()] est [true] si et seulement s'il ne reste plus
107         aucun message en attente. *)
108     let is_empty () =
109         let f k q acc = acc && Queue.is_empty q in
110         Hashtbl.fold f chan true
111
112     (** [produce u v x] simule la production d'un message de contenu utile
113         l'entier [x] envoyé par le démon [u] au démon [v]. *)
114     let produce (from_u:did_t) (to_w:did_t) (x:int) : unit =
115         Printf.printf "Le démon #%d dit '%d' au démon #%d.\n" from_u x to_w;
116         if not (Hashtbl.mem chan to_w) then
117             Hashtbl.add chan to_w (Queue.create ());
118             Queue.push (from_u, x) (Hashtbl.find chan to_w)
119
120     (** [consume ()] retire et renvoie l'un des messages de tête en attente
121         de réception. La valeur de retour [(v,(u,x))] est formée d'un
122         démon [v] qui a reçu le contenu utile [x] du démon [u] *)
123     let consume () : int * msg_t =
124         let _, v = let select k q (n, acc) =

```

```

125         if Queue.is_empty q then (n, acc)
126         else (n+1, if Random.int n = 0 then k else acc)
127         in Hashtbl.fold select chan (1,-1)
128     in
129     let u,x = Queue.pop (Hashtbl.find chan v)
130     in
131     Printf.printf "Le démon #%d acquitte le démon #%d d'avoir dit '%d'.\n"
132         v u x;
133     (v, (u,x))
134 end

```