Épreuve de travaux pratiques blanche

préparation à l'agrégation d'informatique
durée 5h00

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

1 Programmation

On s'intéresse dans cette partie à un protocole d'échange de message ressemblant au protocole IRC (*Internet Relay Chat*) ancêtre des applications modernes de messageries. Nous décrivons dans un premier temps le protocole. Le serveur est à implémenter en OCaml, le client en C. Enfin, un « *bot* » (abréviation robot ou client automatique) est à programmer en SQL et Python. Vous trouverez en annexe un rappel des différentes fonctions utiles disponibles dans les bibliothèques standard de ces langages.

L'énoncé fournit aussi des fonctions spécifiques, qui sont expliquées et commentées dans l'énoncé au moment de leur utilisation.

1.1 Protocole

Le protocole est un protocole en mode texte utilisant TCP comme couche de transport. Les messages échangés entre le client et le serveur ont le format

$$tag\texttt{:}payload\texttt{\textbackslash n}$$

où *tag* vaut `CONNECT` ou `TEXT` et *payload* est une chaine de caractères quelconque ne contenant ni le caractère nul (de code ASCII 0) ni le caractère `\n` (de code ASCII 10). Tout autre message envoyé entre le client et le serveur est considéré comme invalide et met fin à la communication.

Les utilisateurs sont identifiés par un *nick* (abréviation de *nickname* ou pseudonyme). Un *nick* valide est une suite d'au plus 20 caractères, composée uniquement de chiffres (0 à 9), lettres majuscules ou minuscules (de a à z ou de A à Z, sans accent) ou des caractères « `-` », « `_` », « `[` » ou « `]` ».

Initialement, le serveur se met en écoute en TCP sur le port 9000. Lorsqu'un client se connecte au serveur, il lui envoie le message

$$\texttt{CONNECT:}s\texttt{\textbackslash n}$$

où s est le *nick* choisi. Si le s n'est pas un *nick* valide ou s'il est déjà utilisé par un autre client, le serveur interrompt la connexion. Sinon, il renvoie au client le message

$$\texttt{CONNECT:OK\textbackslash n}$$

à partir de ce moment, la connexion est établie. Un client dont le *nick* est s peut envoyer au serveur un message

$$\texttt{TEXT:}p\texttt{\textbackslash n}$$

sur réception d'un tel message, le serveur va alors diffuser, à tous les autres clients que s, le message p en le préfixant de s, et d'un caractère `>` suivi d'un espace :

$$\texttt{TEXT:}s\texttt{> }p\texttt{\textbackslash n}$$

1.2 Serveur (programmation réseau et parallèle en OCaml)

Le serveur est à programmer en OCaml, en n'utilisant que des fonctions de la bibliothèque standard, de la bibliothèque `Unix` et de la bibliothèque *thread*. On fournit un code initial (fichier `server/server.ml`), reproduit à la figure 1.

```
1  open Unix
2
3  let addr_to_string = function
4    | ADDR_INET (ia, port) ->
5       Format.sprintf "%s:%d" (string_of_inet_addr ia) port
6    | ADDR_UNIX a -> a
7
8  let handle_client socket inc outc s_addr =
9    failwith "À compléter"
10
11 let main () =
12   let server_socket = socket PF_INET SOCK_STREAM 0 in
13   setsockopt server_socket SO_REUSEADDR true;
14   bind server_socket (ADDR_INET (inet_addr_any, 9000));
15   listen server_socket 100;
16   Format.printf "Démarrage du serveur\n%!";
17   while true do
18     let cl_socket, addr = accept server_socket in
19     let inc = in_channel_of_descr cl_socket in
20     let outc = out_channel_of_descr cl_socket in
21     let s_addr = addr_to_string addr in
22     Format.printf "Connexion depuis %s\n%!" s_addr;
23     ignore
24       (Thread.create (handle_client cl_socket inc outc) s_addr)
25   done
26
27 let () = main ()
```

FIGURE 1 – Le fichier `server.ml`

question 1 On demande de compléter le fichier `server.ml` afin d'ajouter le code de traitement d'un client une fois la connexion établie. On donne les indications suivantes :

- le code de gestion de chaque client (L.8) est démarré dans un *thread* (processus léger) séparé (L.24)
- la fonction de traitement du client prend en argument la socket permettant de parler au serveur, deux canaux et l'adresse du client ainsi que son port sous forme d'une chaine de caractères. Les canaux utilisent la socket comme descripteur de fichier sous-jacent. Ils permettent donc de lire et d'écrire de façon structurée plus agréablement qu'en utilisant directement la socket.
- une écriture sur `outc` doit être suivie de `flush(outc)` afin de garantir que les octets écrits ne restent pas dans un buffer intermédiaire.
- Pour déconnecter le client, on fermera les canaux au moyen de `close_in` et `close_out` et la socket au moyen de `close` (fonction du module `Unix`).

— le fichier doit être compilé avec
`ocamlopt -o server -I +threads unix.cmxa threads.cmxa server.ml`
— il est évidemment possible de rajouter toute autre définition nécessaire au fichier, sans cependant modifier le code des lignes 11 à 27.
— il est aussi possible d'ajouter du code dans des fichiers séparés. On les compilera alors avec :

```
ocamlopt -c -I +threads f1.mli
ocamlopt -c -I +threads f1.ml
ocamlopt -c -I +threads f2.mli
ocamlopt -c -I +threads f2.ml
ocamlopt -o server -I +threads unix.cmxa threads.cmxa f1.cmx f2.cmx server.ml
```

en ordonnant les fichiers `.cmx` sur la ligne de commande finale par ordre de dépendance.

1.3 Client (programmation réseau et parallèle en C)

On demande d'implémenter en C un client en mode texte, fonctionnant dans un terminal. Dans toute cette section, on appelle « chaine de caractères » une suite d'octets (type `char`) terminée par un octet nul.

1.3.1 Gestion du réseau en C

question 2 Programmer le début d'un client en mode texte. Ce dernier doit prendre exactement trois arguments sur la ligne de commande
— l'adresse ou le nom du serveur
— le numéro de port
— le *nick* choisi
Si la ligne de commande n'est pas au bon format, que le *nick* ne respecte pas les contraintes demandées le programme échoue avec un code de sortie 1. Le client essaye ensuite d'établir une connexion TCP avec le serveur, de lui envoyer le message de connexion et de recevoir le message d'acquittement. Si à un quelconque moment un problème se produit, le client quitte avec le code de sortie 2. On pourra utiliser l'adresse locale 127.0.0.1 pour contacter un serveur s'exécutant sur la même machine que le client.

Comme dans la partie en OCaml, il est suggéré d'encapsuler le descripteur de fichier de la *socket* dans un objet `FILE*` de la bibliothèque standard C afin de disposer de fonction de haut niveau pour lire et écrire des chaines de caractères. Attention, les écritures sur de tels `FILE*` sont généralement accumulées dans un tampon interne. On peut forcer leur écriture sur le descripteur sous-jacent avec la fonction `fflush`. Une façon d'encapsuler un descripteur est :

```
1  //un descripteur de fichier fdescr, par exemple une socket
2  FILE * infile = fdopen(dup(fdescr), "rb");
3  FILE * outfile = fdopen(dup(fdescr), "wb");
4  //lire sur infile et écrire sur outfile
```

1.3.2 Gestion de la concurrence en C

Afin d'avoir une interface conviviale, on propose la petite bibliothèque dont l'entête est donnée à la figure 2. Avec cette bibliothèque il est possible de concevoir une interface comme celle de la figure 3. Cette figure représente un écran initialisé avec un appel à `screen_init("charlie> ")`. En effet, la zone se situant sous les « `===...` » est la zone de saisie. L'utilisateur peut à tout moment y écrire des caractères autre qu'un retour chariot (le rectangle noir représente la position du curseur). La fonction `screen_read_input` bloque jusqu'à ce que l'utilisateur valide sa saisie avec la touche entrée. La chaine saisie est alors envoyée. Pour les affichages, la fonction `screen_print_line` prend en argument l'écran, le numéro de la ligne sur laquelle afficher et la chaine à afficher. La fonction `screen_get_size` permet de connaitre le nombre de lignes et de colonnes de l'écran.

question 3 programmer un type de liste chainées pour lesquelles les valeurs stockées sont des chaines de caractères. Une telle liste représente l'historique des messages reçus, la tête de liste étant le message le plus récent.

question 4 programmer une fonction d'affichage de cette liste chainée, depuis la ligne se trouvant juste au dessus du séparateur jusqu'au haut de l'écran.

question 5 programmer l'interface graphique, en utilisant la bibliothèque fournie ainsi que des *threads*. Sur saisie d'un message par l'utilisateur, ce dernier doit être ajouté dans son historique et envoyé au serveur. Sur réception d'un message

`TEXT:foo> ...\n`

la partie suivant les « : » jusqu'au « `\n` » exclu doit être ajouté à l'historique et ce dernier entièrement ré-affiché. Enfin, si le texte de l'utilisateur commence par « `/QUIT` » le programme se termine.

1.4 Programmation en SQL et Python

On considère le code Python du *bot* partiellement donné en figure 4. La fonction `bot` prend en argument :

`serv` une paire de chaines de caractères représentant l'adresse IP du serveur et son nom d'hôte. Si ce dernier n'est pas connu, alors la chaine correspondante est vide.

`socket` une *socket* représentant la connexion au serveur de discussion

`dbcn` un objet `Connection` représentant la connexion à une base de donnée.

La fonction se met en attente de message (de la part du serveur de discussion). Sur réception d'un message valide, il le décompose pour obtenir d'une part le *nick* et d'autre part le message. Il appelle alors la fonction `log_message`, non détaillée, qui les enregistre alors dans une base de données avec l'identifiant du serveur et l'heure courante, au format *Unix* (nombre de secondes écoulées depuis le 1er janvier 1970 00 :00 :00 GMT). De plus, si le message commence par `!STATS` le *bot* appelle la fonction `print_stats`, non-détaillée, qui calcule des statistiques et les écrits sur le serveur de discussion (en simulant un participant qui parle).

Plusieurs instances de ce programme peuvent être exécutées et connectés à des serveurs de discussion différents. On fait l'hypothèse (simpliste) qu'un *nick* représente le même utilisateur sur tous les serveurs de discussion. On suppose aussi que tous les *bots* sont connectés à un unique serveur de bases de données.

question 6 proposer une modélisation de la base de données comportant au moins deux tables. Vous décrirez précisément les tables, les types et les contraintes en donnant les ordres de création de tables. Vous justifierez le choix des types.

```c
#ifndef SCREEN_H
#define SCREEN_H
/* Type abstrait représentant l'écran.
   L'accès concurrent au même écran n'est pas sûr
*/
struct screen;

/* Initialise l'écran. Ce dernier est effacé,
  le séparateur est dessiné et la zone de saisie
  est initialisée avec le prompt donné */
struct screen *screen_init(const char * prompt);

/* Détruit et l'écran */
void screen_free(struct screen * sc);

/* La fonction bloque jusqu'à ce que l'utilisateur
   saisisse une chaine non vide et la valide avec
   la touche entrée. La chaine renvoyée doit être
   désalouée avec free.
*/
char *screen_read_input(struct screen *);

/* Affiche la chaine txt à la ligne line sur l'écran.
   Les numéros de ligne commencent à 1. Si le numéro de ligne
   est invalide, la fonction ne fait rien. Les caractères
   allant au delà de la dernière colonne sont ignorés.
*/
void screen_print_line(struct screen * sc, int line, char * txt);

/* Initialise les pointeurs donnés respectivement avec le nombre
   de lignes et de colonnes de l'écran.
*/
void screen_get_size(struct screen *sc, int *lines, int *cols);


#endif
```

FIGURE 2 – L'interface de la bibliothèque `screen`

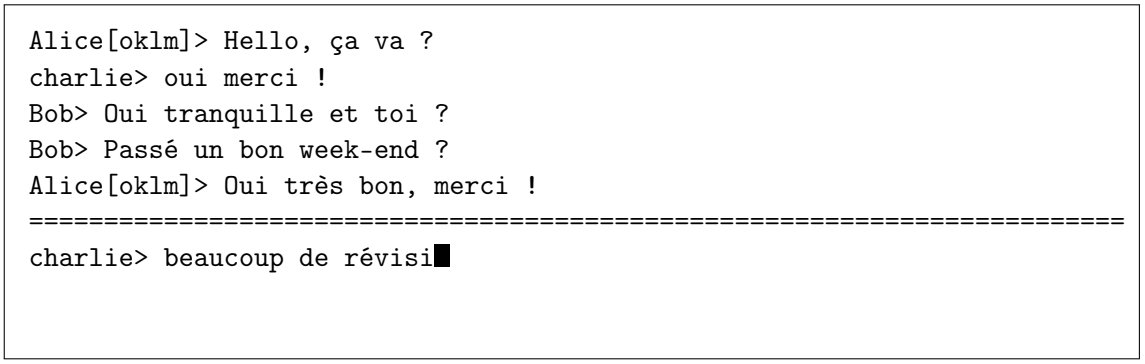

FIGURE 3 – L'interface texte

```python
import socket
from time import time

def bot(serv, socket, dbcn):
    """Enregistre tout ce qui est envoyé par le serveur serv
    sur la socket dans la base de donnée dénotée par dbcn."""
    f = socket.makefile() #transforme la socket en fichier
    while True:
        line = f.readline()
        fields = line.split(':')
        if fields[0] != "TEXT":
            continue
        fields = fields[1].split('>')
        nick = fields[0]
        msg = fields[1]
        log_message(serv, nick, int(time()), msg, dbcn)
        if msg.startswith(" !STATS"):
            print_stats(serv, f, dbcn)

#Le reste du code établit une connexion à la base de données
#(dbcn) et au serveur de discussion (socket) puis appelle
#la fonction bot
```

FIGURE 4 – Le code du *bot* (fichier `bot/bot.py`)

question 7 proposer des requêtes SQL qui utilisent votre modélisation et permettent de calculer les statistiques suivantes

1. Le nombre de messages stockés
2. Le nombre d'utilisateurs distincts
3. Le nombre de serveurs distincts dont le nom d'hôte se termine par `.fr`
4. Le nombre d'utilisateurs distincts sur un serveur `'rezosup.fr'`
5. Le nombre de messages contenant la sous-chaine `informatique` (en minuscules)
6. L'ensemble sans doublon des utilisateurs du serveur s ayant envoyé des messages avant le 1er janvier 2020

7. Pour chaque utilisateur, le nombre de messages envoyés si ce dernier est plus grand que 30, triés par ordre décroissant de nombre de messages
8. Les utilisateurs n'ayant jamais envoyé de message sur le serveur `'rezosup.fr'`
9. Les utilisateurs ayant envoyé globalement plus de 100 messages et qui ont discuté sur le serveur `'rezosup.fr'`

question 8 Un problème peut se produire si deux *bots* sont connectés au même serveur de discussion car les messages de ce dernier vont être comptés deux fois. Proposer une solution à ce problème utilisant la base de données (en fournissant du code SQL d'exemple).

question 9 On détaille maintenant l'API d'une connexion à la base de données par un exemple :

```
1 #dbcn est un object Connection
2 cur = dbcn.cursor()   #objet pour exécuter un ordre SQL
3 tab = cur.execute("SELECT * FROM T")
```

Après exécution de ce code, `tab` est un itérable (utilisable dans une boucle **`for .. in`**) de dictionnaires dont les clés sont les noms des colonnes sélectionnées et les valeurs celles de la ligne en cours. Écrire une fonction Python `print_stats_html` qui écrit sur la sortie standard le résultat d'une requête calculant le nombre de messages par serveur sous la forme d'un fichier HTML. Ce dernier devra utiliser une `<table>` dont les en-têtes sont les noms des colonnes du résultat de la requête et dont les lignes paires ont un fond blanc et les lignes impaires un fond gris.

Annexe

A Référence OCaml

Entrées/sorties sur les cannaux, parallélisme

`in_channel` : type d'un canal de lecture

`out_channel` : type d'un canal d'écriture

`output_char` : écriture d'un caractère sur un canal

`output_string` : écriture d'un caractère sur un canal

`input_line` : lecture d'une ligne (terminée par `'\n'`) sur un canal

`close_in` : fermeture d'un canal de lecture

`close_out` : fermeture d'un canal d'écriture

`flush` : vidage du tampon interne d'un canal d'écriture

`Mutex.create` : création d'un mutex (bibliothèque `threads`)

`Mutex.lock` : verrouillage d'un mutex (bibliothèque `threads`)

`Mutex.unlock` : déverrouillage d'un mutex (bibliothèque `threads`)

B Référence C

Hors fonctionnalités de la bibliothèque standard C, les fonctions ci-dessous peuvent être utiles.

Chaine de caractères, formatage

`strncmp` : comparaison des n premiers caractères de deux chaines données

`fprintf` : écriture formatée dans un `FILE*`

`snprintf` : écriture formatée dans un tableau de caractères de taille au plus n

`fflush` : vidage du tampon interne d'un `FILE*` et écriture sur le descripteur de fichier sous-jacent

`isalnum` : teste si un caractère est alphanumérique

`getline` : lecture d'une ligne (terminée par `'\n'`) d'un `FILE*` avec allocation automatique

`fdopen` : création d'un `FILE*` à partir d'un descripteur de fichier

`fclose` : fermeture d'un `FILE*`

Système,réseau,parallélisme

`read` : lecture sur un descripteur de fichier

`write` : écriture sur un descripteur de fichier

`close` : fermeture d'un descripteur de fichier

`memset` : initialisation d'une zone mémoire

`getaddrinfo` : traduction d'adresses et de services réseaux

`socket` : création d'une *socket*

`bind :` association d'une adresse à une *socket*

`listen :` mise en mode passif d'une *socket*

`accept :` attendre des connexions sur une *socket*

`connect :` débuter une connexion sur une *socket*

`send :` envoi de données sur une *socket*

`recv :` réception de données sur une *socket*

`pthread_create :` création d'un nouveau *thread*

`pthread_cancel :` arrêt d'un *thread*

`pthread_join :` attente de l'arrêt d'un *thread*

`pthread_mutex_t :` type d'un mutex

`pthread_mutex_init :` initialisation d'un mutex (pile ou tas)

`PTHREAD_MUTEX_INITIALIZER :` initialisation d'un mutex (global)

`pthread_mutex_lock :` verrouillage d'un mutex

`pthread_mutex_unlock :` déverrouillage d'un mutex

`pthread_mutex_destroy :` destruction d'un mutex