Projet programmation système W03

Introduction

Cette semaine, nous voulons vous faire travailler avec les bibliothèques qui nous sera utile pour la suite : les communications réseaux UDP, que nous utiliserons dès la semaine 5 pour communiquer entre clients et serveurs, et la `libssl`, bibliothèque de protocoles crytographiques, que nous utiliserons en semaine 11 pour calculer les empreinte unique (« code SHA ») de serveurs/nœuds. Commençons par cette dernière.

Un « code SHA », ou plus simplement « SHA » (pour « *Secure Hash Algorithm* »), est une représentation compacte, très probablement unique et difficilement inversible de n'importe quelle donnée. Plus concrètement cela veut dire :

- *compacte*: que n'importe quelle séquence de données sera représentée par un nombre unique de bits ; nous utiliserons par exemple dans ce projet un code SHA à 160 bits (nommé « SHA-1 » ; même s'il est considéré comme obsolète depuis 2011 : nous ne faisons pas ici une application critique du point de vue cryptographie) ;

  par exemple, le SHA-1 de « `coucou` » est
  `5ed25af7b1ed23fb00122e13d7f74c4d8262acd8` ;

- *très probablement unique* : deux données différentes auront très certainement deux SHA différents s'il n'y a « pas trop » de données ; ceci n'est évidemment pas totalement garanti (il n'y a par exemple « que » environ 10^{48} SHA-1 différents) mais très probable si le nombre de données reste faible devant le nombre de codes SHA possibles : avec 10^{20} données différentes, on a une probabilité de $10^{10}/2^{161}$=~10^{-9} que deux d'entre elles aient le même SHA-1 ; lorsque deux données différentes ont le même code SHA, on parle de « *collision* ».

- *difficilement inversible* : connaissant un code SHA, il est très difficile (= impossible en pratique) de retrouver les données (la fonction SHA n'est pas injective !) auxquelles il correspond. Nous n'utiliserons pas cette propriété dans notre projet, mais notez cependant une condition nécessaire pour qu'il en soit ainsi est qu'une toute petite variation de la donnée donne un code SHA totalement différent. Par exemple, le SHA-1 de « `coucou!` » est
  `f016edef54ce0db24446034950744aafba3d3ef2`
  (à comparer à celui « `coucou` » de ci-dessus; et celui de « `coucoo` » est `7592f7cec45f818c08a974cfae5cfad922f189fc`.

Pour ceux qui sont intéressés par en savoir plus sur les codes SHA, voir (optionnel) la page Wikipédia correspondante.

Partie I : codes SHA

Préliminaire : la bibliothèque libssl

Si ce n'est pas déjà fait, installez la « version dev(eloppement) » de la bibliothèque sur votre environnement de travail :

```
sudo apt-get install libssl-dev
```

Lorsque vous utiliserez les fonctions offerte par celle bibliothèque pour manipuler des SHA, vous devez ajouter (au moins) l'option `-lcrypto` lors de l'édition de liens ; par exemple :

```
gcc -std=c11 -o mon_petit_SHA mon_petit_SHA.c -lcrypto
```

Premier exercice

Le premier consiste simplement à fournir deux fonctions pour afficher les codes SHA-1.

Dans votre dépôt GitHub, vous trouverez un fichier `exo-sha-1.c` (dans `week03/provided`). Ce fichier compile seul en l'état, mais il lui manque un constituant : la fonction `print_sha_from_content`, à lui fournir lors de l'édition de liens (revoir si nécessaire le tutoriel sur les Makefiles).

Dans ce premier exercice, vous devez donc écrire :

1. un fichier `sha.c` contenant (entre autres) la fonction `print_sha_from_content` ;
2. un fichier `sha.h` (contenant ce qui vous semble nécessaire) ;
3. un `Makefile.sha` permettant de produire un exécutable `exo-sha-1`.

Note : pour utiliser un autre fichier `Makefile` que `Makefile` ou `makefile` avec la commande `make`, utiliser l'option f‘ ; par exemple :

```
make -f Makefile.sha
```

[**fin de note**]

Exemple de déroulement attendu :

```
./exo-sha-1
Entrez une phrase : bonjour tout le monde !
Le SHA1 de
"bonjour tout le monde !"
est :
337b897df213f2f3d7f8bd666cea1589dd386f42
```

En bonne conception modulaire, le fichier `sha.c` devra contenir *deux* fonctions :

- `print_sha` : qui reçoit un code SHA-1 sous forme d'un tableau d'`unsigned char` (`man SHA1` ; ne regarder que la fonction `SHA1()` (la quatrième), pas le reste) et ne retourne rien ;
- `print_sha_from_content` : qui reçoit des données sous forme d'un tableau d'`unsigned char` et de sa taille, et ne retourne rien.

À noter que la fonction `print_sha` n'a (pour une fois) pas besoin de recevoir la taille du tableau code SHA car celle-ci est connue et fournie par la bibliothèque `<openssl/sha.h>` (`man SHA1`) : il s'agit de la constante `SHA_DIGEST_LENGTH`.

La fonction `print_sha` devra :

1. vérifier que son argument n'est pas égal à `NULL` et quitter sinon ;
2. afficher chacun des bouts de code SHA (éléments du tableau) en utilisant le format `%02x` de `printf`.

La fonction `print_sha_from_content` devra :

1. calculer le SHA-1 du contenu reçu en appelant la fonction `SHA1` de la bibliothèque `openssl` (`man SHA1`) ;
2. afficher ce code SHA.

Second exercice

Buts

Le but de ce second exercice est d'explorer (un tout petit peu) la variété des codes SHA. L'idée est de rechercher des variations de contenu qui partagent le même début (quelques octets) de code SHA.

Vous devrez pour cela écrire un fichier `exo-sha-2.c` (et mettre à jour le `Makefile.sha`) dont le but est de :

1. créer un contenu initial, puis calculer, mémoriser et afficher son SHA-1 ;
2. chercher avec deux stratégies expliquées ci-dessous tous les contenus ayant un SHA-1 dont les deux premiers octets sont les mêmes que le SHA-1 du contenu initial ;
3. refaire la même chose pour les quatre premiers octets.

Exemple de déroulement attendu :

```
./exo-sha-2
Initial SHA1:
8cd537a621659c289f0707bad94719b5782ddb1f

====== check size: 2 ========================================
---- Stratégie 1 ----
0 match(s) over 255 tests
---- Stratégie 2 ----
8cd535959f6a431f3cfc63c8d54da9073698887b
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 138 0 0 \
0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 match(s) over 12750 tests

====== check size: 4 ============================================
---- Stratégie 1 ----
0 match(s) over 255 tests
---- Stratégie 2 ----
0 match(s) over 12750 tests
```

NOTE : pour mieux comprendre la suite, il est utile de savoir que le type `unsigned char` permet de représenter un octet ; ce n'est pas ici l'aspect «`char`», c.-à-d. un caractère lisible pour un humain, qui est utilisé, mais le fait qu'il occupe exactement un octet en mémoire ; on utilise donc sa valeur comme un entier (que nous écrirons ici soit en décimal soit en hexadécimal) et non pas comme un caractère au sens usuel du terme.

Fonction à produire

Les fonctions à produire dans `exo-sha-2.c` sont :

- `zero` qui prend un tableau de `unsigned char` et l'initialise à 0 ; elle ne retourne rien ;

  (**Note (que je ne devrais pas avoir besoin de mettre !) :** il est clair que lorsqu'une fonction reçoit un tableau et a besoin de sa taille, elle reçoit *aussi* la taille de ce tableau ; je ne le spécifierai plus.)

- `print_content`, une fonction outil utile pour debuger, qui prend un tableau de `unsigned char` et l'affiche sur une ligne en séparant chaque élément par une espace ;

  par exemple, si elle reçoit le tableau { 3, 12, 255 }, elle affichera (avec un retour à la ligne en fin) :

  ```
  3 12 255
  ```

  cette fonction ne retourne rien ;

- `print_match` qui reçoit deux tableaux de `unsigned char` et retourne un entier ; cette fonction a pour but d'afficher le SHA d'un contenu si celui-ci commence par une séquence d'octets donnée ;

  le premier tableau reçu en paramètre correspond au *début* d'un code SHA recherché (sa taille indiquée est donc le nombre d'éléments à vérifier) et le second tableau le contenu dont on doit vérifier le SHA ;

  par exemple, si cette fonction reçoit { 8c, d5 } (valeurs entières indiquées ici en hexadécimal) et un contenu dont le SHA-1 est `8cd537a621659c289f0707bad94719b5782ddb1f`,

alors cette fonction affichera ce code SHA ; si par contre le premier argument est { 8c, d6 }, elle n'affiche rien (pour le même contenu) ;

en cas de succès (c.-à-d. d'affichage) la fonction retourne `1` et sinon elle retourne `0` ;

- `lookfor` qui reçoit un tableau `unsigned char`, début d'un code SHA recherché (idem que `print_match`), et en plus une taille, disons `generate_size` pour l'algorithme de recherche de données expliqué ici :
  - `lookfor` commence par créer un tableau de contenu (sous forme d'octets, c.-à-d. de type `unsigned char`), de taille `generate_size`, et l'initialiser à 0 ;
  - pour chacun des octets du tableau de contenu, tour à tour, elle :
  - modifie sa valeur, de 1 à 255,
  - teste si le contenu produit un SHA-1 recherché ;
  - et remet à 0 l'octet modifié ;
  - cette fonction devra de plus compter le nombre total d'essais et le nombre de succès, et les afficher ; voir l'exemple de déroulement donné plus haut :

    ```
    1 match(s) over 12750 tests
    ```

  ainsi, la fonction `lookfor` avec une taille `generate_size` de 3 testera respectivement :

  ```
  1 0 0
  2 0 0
  ...
  255 0 0
  0 1 0
  0 2 0
  ...
  0 255 0
  0 0 1
  0 0 2
  ...
  0 0 255
  ```

Déroulement du `main()`

Le `main()` attendu devra reproduire *exactement* l'exemple de déroulement donné plus haut, c.-à.d. :

- produire un contenu initial de taille 50, entièrement nul et afficher son SHA-1 (avec le bon message) ;

- rechercher, avec une première stratégie ne cherchant que parmi les contenus de *1 octet*, un contenu ayant les mêmes *deux* premiers octets que le SHA-1 initial ;
- rechercher, avec une seconde stratégie ne cherchant que parmi les contenus de *50 octets*, un contenu ayant les mêmes *deux* premiers octets que le SHA-1 initial ;
- réappliquer les deux mêmes stratégies de recherche (resp. 1 et 50 octet(s)), mais en recherchant des SHA-1 ayant les *quatre* premiers octets en commun avec le SHA-1 initial : il n'y en a en fait aucun comme le montre l'exemple de déroulement donné plus haut.

Tests

Pensez à tester vos programmes au delà des cas de déroulement donnés ici (cela fait aussi partie des objectifs de ce devoir et devrait être – sinon devenir – un réflexe systématique).
Imaginez non seulement les cas standards, mais aussi tous les cas particuliers.

Partie II : communication UDP

Contexte

Le but de cette seconde partie est de vous faire pratiquer, à partir de fonctions fournies, la communication réseau UDP (protocole de « *transmission sans connexion* » vu en cours de Computer Networks). C'est en effet ce type de communication que nous utiliserons dans le projet pour communiquer entre client(s) et serveur(s).

Les fonctions de base nécessaire à cette communication sont fournies dans le module « système » (`system.[ch]`) :

- `int get_socket(time_t temps)` permettant d'obtenir un socket réseau (représentation interne au programme) pour la communication en précisant son temps d'attente (« timeout », en secondes) ;
- `int get_server_addr(const char* IP, uint16_t port, struct sockaddr_in* adresse)` permettant d'obtenir l'adresse, au sens « objet interne » (`struct sockaddr_in`), d'une adresse IP et d'un port donnés ;
- `int bind_server(int socket, const char *IP, uint16_t port)` permettant d'associer une communication réseau (représentation externe : adresse IP, port) à un socket réseau (représentation interne).

Note : voir si nécessaire le fichier `system.h` pour plus de détails.
[fin de note]

Ces fonctions vous sont fournies en l'état ; vous n'avez pas à les modifier, simplement à les utiliser.

Troisième exercice

Le but de cet exercice assez simple est juste de vous faire lire le code et comprendre les différentes étapes : vous n'aurez que quelques lignes à compléter dans les fichiers `udp-client.c` et `udp-server.c` (et créer le fichier `Makefile.udp` pour les compiler).

Dans ces deux fichiers, cherchez et remplacez les lignes

```
// A COMPLETER ICI
```

mais nous vous conseillons, bien sûr, pour savoir quoi y mettre, de lire le reste du code (et au moins `system.h`).

`udp-server.c` a pour but de démarrer un serveur : boucle infinie qui lit sur un port et répond (à celui qui lui a envoyé quelque chose). Pour cela, il doit commencer par ouvrir un socket (**à faire**), puis l'associer à une adresse IP et port donnés (**à faire**) ; il entre ensuite dans sa boucle infinie qui lit sur le socket ouvert (déjà fait), convertit la donnée reçue (**à faire**, voir paragraphe suivant), la modifie (**à faire**, ajouter simplement +1 à la valeur reçue) et la renvoie (déjà fait).

Les données peuvent être représentées de façon différente entre la mémoire des machines et les communications réseau (little-endian contre big-endian). Il est alors nécessaire de les convertir. Utilisez à ce sujet les fonctions `htonl()` et `nltoh()` (`man htonl`).

De son coté, `udp-client.c` a pour but d'envoyer un seul message (un entier) au serveur : il commence par ouvrir un socket pour communique (**à faire**), puis convertir l'adresse IP et port du serveur en format interne (**à faire**) ; il demande ensuite à l'utilisateur d'entrer un entier positif (**à faire**), convertit cette valeur au format réseau (**à faire**), puis l'envoie au serveur et attend sa réponse (déjà fait), convertit la réponse reçue au format « normal » (format utilisé en mémoire, **à faire**) et l'affiche (déjà fait).

Dans ces deux programmes :

1. l'adresse et le port du serveur sont donnés respectivement par les constantes `PPS_DEFAULT_IP` et `PPS_DEFAULT_PORT` ;
2. en cas d'erreur, sortez simplement avec `die()` comme fait, par exemple, ligne 60 de `udp-client.c` ou ligne 56 de `udp-server.c`.

Tests / Exemple de déroulement

Pour tester vos programmes, nous vous conseillons d'ouvrir DEUX terminaux côte à côte, et de lancer le serveur dans l'un et plusieurs fois le client dans l'autre et observer les messages. Essayez aussi de lancer le client sans avoir lancé le serveur pour observer le timeout.

Pour terminer le serveur, tapez simplement `Ctrl-C` dans le terminal correspondant.

Voici par exemple ce que vous pourriez avoir (on note `TERM1>` les commandes que vous tapez dans le premier, et `TERM2>` celles dans le deuxième) :

```
TERM1> ./udp-server
Server listening on 127.0.0.1:1234.

TERM2> ./udp-client
What int value do you want to send? 42
Sending message to 127.0.0.1:1234: 42
Received response: 43
```

et ce qui aura aussi eu pour effet d'afficher les deux lignes suivantes sur le premier terminal (le numéro de port pouvant être tout à fait différent, bien sûr) :

```
Received message from 127.0.0.1:40477: 42.
Sending message to 127.0.0.1:40477: 43.
```

Rendu

Pour rendre le devoir, ajoutez les sept fichiers `Makefile.sha`, `sha.c`, `sha.h`, `exo-sha-2.c`, `Makefile.udp`, `udp-client.c` et `udp-server.c` à votre GitHub dans un répertoire `done/week03` (à la racine de votre GitHub, mais **PAS** dans `provided` ; vous ne devez **jamais** modifier le répertoire `provided`) puis « tagger » avec le tag `week03` et « pousser » le résultat vers GitHub (avec le tag : `git push --tags`).

Le délai pour faire le `push` de ce rendu est : dimanche 18 mars 23:59.

Et n'oubliez pas : 1. faire le rendu de la semaine passée avant dimanche soir ; 1. de vous inscrire par groupes de deux avant lundi soir (après il sera trop tard) au moyen du lien dans le menu à gauche ci-contre (**Note :** votre inscription est validée manuellement ; cela prend donc « *un peu* » de temps avant que vous ne receviez une confirmation et un dépôt GitHub pour votre groupe).