Révision des hashtable

Jean-Cédric Chappelier 2017

from former document by Laurent Bindschaedler + Jean-Cédric Chappelier 2016

Nous allons cette semaine réviser l'implémentation des *hashtables* utilisées jusqu'ici, et cela de deux manières :

1. en changeant les types des clés et des valeurs vers des types plus génériques ;
2. en permettant (et gérant) les collisions.

Pour rappel (cf. semaine 4), la représentation concrète de la *hashtable* que nous vous proposons d'implémenter ressemble à ceci :

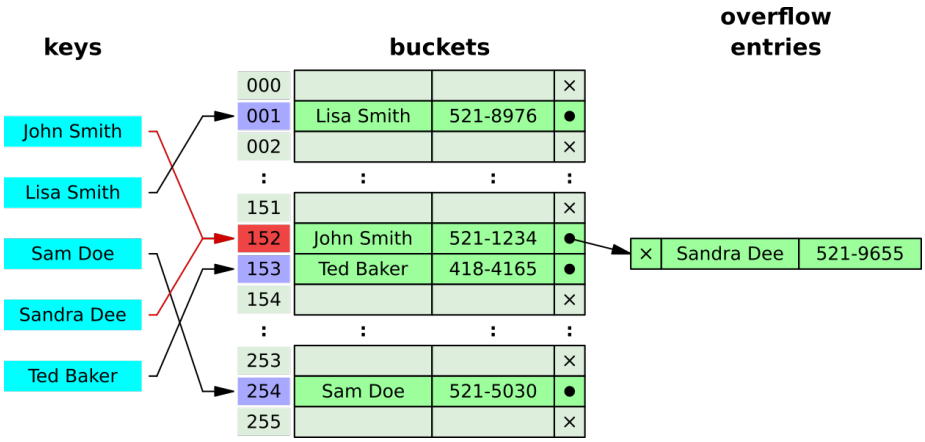

Fig. : *représentation d'une hashtable avec gestion des collisions par listes chaînées.* (CC) Jorge Stolfi@WikiMedia

I. Révision des types `pps_key_t` et `pps_value_t`

Le type des clés et celui des valeurs doivent maintenant être `const char*`. Nous insistons sur les `const` ici : la *hashtable* ne doit pas pouvoir modifier ces contenus. Par ailleurs, le type `bucket_t` devient plus complexe afin de pouvoir gérer les collisions comme illustré dans la figure ci-dessus et expliqué ci-dessous.

Vous devez par ailleurs maintenant effectuer des allocations dynamiques de la mémoire associée à la *hashtable* afin d'éviter de gaspiller inutilement de la mémoire ; le type `Htable_t` ne peut plus être un tableau statique, mais doit lui-même être alloué dynamiquement. Vous devez pour cela définir deux nouvelles fonctions :

- `Htable_t construct_Htable(size_t)` qui prend un nombre « d'alvéoles » (« *buckets* ») en paramètre et retourne une *hashtable* nouvellement créée (à la bonne taille) ;
- `void delete_Htable_and_content(Htable_t*)` qui libère la mémoire associée à la *hashtable* passée en paramètre, **ainsi que tout le contenu mis dans cette table** (cf remarque sur la propriété ci-dessous).

NOTE au sujet de la propriété des contenus de la table : le type des valeurs stockées étant maintenant un pointeur, se pose la question de la propriété du contenu de la *hashtable* (c.-à-d. les éléments contenus dans la table doivent ils être « cachés » ou peuvent ils être partagés à l'extérieur ?).

Le point de vue choisi dans ce projet est que la *hashtable* est propriétaire unique de son contenu. Les valeurs ajoutées (valeurs pointées) doivent donc être copiées (au lieu de simplement copier le pointeur).
[fin de la note]

Adaptez ensuite les fonctions suivantes par rapport à ces nouveaux types :

- `pps_value_t get_Htable_value(Htable_t, pps_key_t)` : cette fonction doit maintenant retourner la valeur associée si elle existe ou `NULL` si elle n'existe pas ; elle retourne également `NULL` en cas de paramètre incorrect ;
- `error_code add_Htable_value(Htable_t, pps_key_t, pps_value_t)` : bien que ce serait tout à fait faisable, nous avons décidé au niveau de ce projet de ne pas accepter les valeurs `NULL`, c.-à-d. que si la `pps_value_t` passée est `NULL`, cette fonction retourne `ERR_BAD_PARAMETER` ; par ailleurs, n'oubliez pas que le contenu de la valeur doit être copié (et non pas simple copie du pointeur).

Pour calculer la valeur de hash d'une clé donnée (`key` ci-dessous) pour une *hashtable* ayant `size` « alvéoles », nous vous demandons d'utiliser la fonction suivante :

```
/** ---------------------------------------------------------------------
 ** Hash a string for a given hashtable size.
 ** See http://en.wikipedia.org/wiki/Jenkins_hash_function
 **/
size_t hash_function(pps_key_t key, size_t size)
{
    M_REQUIRE(size != 0, SIZE_MAX, "size == %d", 0);
    M_REQUIRE_NON_NULL_CUSTOM_ERR(key, SIZE_MAX);
```

```
    size_t hash = 0;
    const size_t key_len = strlen(key);
    for (size_t i = 0; i < key_len; ++i) {
        hash += (unsigned char) key[i];
        hash += (hash << 10);
        hash ^= (hash >>  6);
    }
    hash += (hash <<  3);
    hash ^= (hash >> 11);
    hash += (hash << 15);

    return hash % size;
}
```

II. Gestion des collisions

Le type `Htable_t` (ou plus précisément `bucket_t`) et les fonctions `get_Htable_value()` et `add_Htable_value()` doivent également être réviser pour gérer les collisions de valeurs de hash pour des clés différentes (cf. figure ci-dessus). On utilisera alors une *liste chaînée* pour stocker plusieurs paires clé-valeur en cas de collision de valeur (même « alvéole »).

Pour l'ajout de la valeur (fonction `add_Htable_value()`) : comme toujours, si la clé existe déjà dans la *hashtable*, la nouvelle valeur reçue doit écraser l'ancienne déjà stockée ; mais par contre, si c'est une *nouvelle* clé dont la fonction de hashage donne une collision (même valeur de hash) avec une clé déjà stockée, il faudra alors ajouter la nouvelle clé à la liste chaînée des paires clé-valeur de « l'alvéole » correspondante.

De même, lors de la recherche d'une valeur pour une clé (`get_Htable_value()`), il faudra maintenant parcourir la liste chaînée des paires clé-valeur de « l'alvéole » cible.

Comme d'habitude, n'hésitez pas à modulariser votre code et créer d'autres fonctions utilitaires si nécessaire ! Dans cet esprit, nous vous recommandons *par exemple* de modulariser les types :

1. profiter du type `kv_pair` ;
2. définir un type `struct bucket` dans `hashtable.c` (oui, oui, `.c` et non pas `.h`, il n'y a pas de typo ici : ce type étant strictement interne, il n'y a aucune raison de l'exporter via le fichier `.h` ; vous pouvez par contre mettre une *pré-déclaration* dans le fichier `.h`...).

et créer si nécessaire des fonctions outils associées.

Au final, pensez également à tester correctement les fonctionnalités de votre *hashtable* pour plusieurs combinaisons de tailles, clés et valeurs **avant** de passer

à l'étape suivante. Cela vous évitera bien des tracas...

III. Adaptation des communications client-serveur

Il nous faut maintenant adapter les communications client-serveur (`network.c`) et les outils (`pps-launch-server.c`, `pps-client-put.c` et `pps-client-get.c`) à nos nouvelles structures de données.

Par exemple au niveau du format des communications, nous avions jusqu'ici une taille fixe (clés d'un octet et valeurs de 4 octets), ce qui ne peut plus être le cas. Les messages échangés seront maintenant constitués de chaînes de caractères de longueur potentiellement différentes ; et deux chaînes concaténées dans le cas du `put` (le caractère nul `\0` servant de séparateur). Les requêtes d'écriture auront donc maintenant le format suivant : « `<clé>\0<valeur>` », les requêtes de lecture le format : « `<clé>` », et leur réponse le format « `<valeur>` », où `<clé>` et `<valeur>` représentent la chaîne de caractères *usuelle* de, respectivement, la clé et la valeur, sans caractère nul final. Par exemple, la requête d'écriture de la valeur « xy » pour la clé « abc » enverra sur le réseau la séquence de six caractères « `abc\0xy` », **sans** `\0` final ; et la réponse à une requête de lecture pour cette même clé (une fois celle-ci écrite) enverra « `xy` » (2 caractères) et **non pas** « `xy\0` » (3 caractères ; pour rappel, le nombre total d'octets échangés au niveau réseau est connu, par exemple comme valeur de retour de `recvfrom()`). Attention, donc, lorsque vous convertirez ces clés/valeurs en chaînes C à ne pas oublier les caractères nuls terminaux, si nécessaire.

Comme ces clés et valeurs peuvent en principe contenir n'importe quoi, elles pourraient devenir très longues. Afin de limiter l'impact sur les communications réseaux (et d'éventuels échecs au niveau des protocoles inférieurs), nous avons décidé de limiter la longueur de chacun de ces éléments (clés et valeurs) à `MAX_MSG_ELEM_SIZE` (défini dans `config.h`). Ainsi, si `network_get()` ou `network_put()` reçoivent des argument trop longs, elles doivent sortir avec l'erreur `ERR_BAD_PARAMETER` (voir la macro `M_EXIT_IF_TOO_LONG` de `error.h`).

NOTES :

1. Si jamais, nous avons aussi défini la constante `MAX_MSG_SIZE` (dans `config.h`) pour représenter la taille maximale d'un message réseau, lors d'une commande `put` (« `<clé>\0<valeur>` ») : `2 * MAX_MSG_ELEM_SIZE + 1`.
2. Pour rechercher un caractère particulier, y compris le caractère nul, dans une séquence de caractères quelconque, utilisez la fonction `memchr()` (`man memchr`, similaire à `strchr()`, mais qui ne s'arrête pas sur le premier `\0` rencontré).

[fin de note]

Une autre modification à apporter concerne les clés non existantes. Jusqu'ici nous donnions une valeur par défaut (0) à toute clé non encore insérée dans la DHT ; nous allons maintenant gérer ce cas explicitement en répondant le caractère nul (`\0`) en cas de requête de lecture d'une clé non encore écrite. Même s'il n'est pas encore possible pour le moment avec l'interface disponible d'associer des valeurs vides (chaîne vide) à des clés, cela l'est tout à fait en principe et le sera plus tard. Il faut donc bien comprendre ici la différence entre répondre une valeur vide (chaîne vide) associée à une clé existante et répondre à une clé non existante : * le message réseau envoyé en réponse à une requête de lecture pour une clé connue dont la valeur est vide sera simplement vide (aucun caractère, 0 octet) ; * alors que le message envoyé en réponse à une requête de lecture pour une clé inconnue sera la chaîne vide au sens du C, c.-à-.d la séquence de caractères réduite au seul caractère nul (`\0`, et donc de longueur 1 octet).

En cas de réponse « clé inconnue » de la part d'un serveur, le client doit considérer cette réponse exactement comme s'il n'avait pas eu de réponse du tout (c.-à-d. comme un « *timeout* ») et passer au serveur suivant. Si aucun des N serveurs n'a répondu positivement pour la clé demandé, on doit considérer cela comme un échec, exactement comme si aucun des N serveurs n'avait répondu du tout (N « *timeouts* »).

Au niveau du stockage local des tables de hashage (c.-à-d. sur les serveurs eux-mêmes), ceux-ci alloueront dynamiquement chacun une table de `HTABLE_SIZE` « alvéoles ».

IV. Tests

Il sera bien entendu important de bien tester votre code, tout d'abord au niveau de l'implémentation des tables de hashage (comme déjà suggéré plus haut) et ensuite au niveau des nouveaux protocoles réseaux. Pour ce dernier point, nous vous rappelons que vous pouvez remplacer le `sendto()` système par le nôtre fourni dans `log-packets.c`. Il suffit pour cela de (le compiler et) faire précéder la commande utilisée par « `LD_PRELOAD=./log-packets.so` ». Revoir si nécessaire le sujet de la semaine 05.

Voici par exemple un scénario possible, sur trois terminaux (et avec un fichier local `servers.txt` contenant « `127.0.0.1 1234` ») :

```
TERM1> rm packets.log ## si jamais il existait déjà d'une expérience précédente
TERM1> LD_PRELOAD=./log-packets.so ./pps-launch-server < servers.txt &
TERM1> tail -F packets.log

TERM2> LD_PRELOAD=./log-packets.so ./pps-client-put
abc xy
OK
```

```
TERM3> LD_PRELOAD=./log-packets.so ./pps-client-get
abc
OK xy
not_a_key
FAIL

TERM2 (suite)>
not_a_key some_value
OK
u v
OK
^D

TERM1 (suite)>
not_a_key
OK some_value
u
OK v
v
FAIL
^D
```

Le contenu du fichier `packets.log` devrait alors ressembler à (les numéros de ports peuvent être différents) :

```
0.0.0.0 54275 127.0.0.1 1234 61 62 63 00 78 79
127.0.0.1 1234 127.0.0.1 54275
0.0.0.0 50349 127.0.0.1 1234 61 62 63
127.0.0.1 1234 127.0.0.1 50349 78 79
0.0.0.0 50349 127.0.0.1 1234 6E 6F 74 5F 61 5F 6B 65 79
127.0.0.1 1234 127.0.0.1 50349 00
0.0.0.0 54275 127.0.0.1 1234 6E 6F 74 5F 61 5F 6B 65 79 00 73 6F 6D 65 5F 76 61 6C 75 65
127.0.0.1 1234 127.0.0.1 54275
0.0.0.0 50349 127.0.0.1 1234 6E 6F 74 5F 61 5F 6B 65 79
127.0.0.1 1234 127.0.0.1 50349 73 6F 6D 65 5F 76 61 6C 75 65
0.0.0.0 54275 127.0.0.1 1234 75 00 76
127.0.0.1 1234 127.0.0.1 54275
0.0.0.0 50349 127.0.0.1 1234 75
127.0.0.1 1234 127.0.0.1 50349 76
0.0.0.0 50349 127.0.0.1 1234 76
127.0.0.1 1234 127.0.0.1 50349 00
```

qui se comprend comme suit :

```
client-put -> serveur : 'a' 'b' 'c' 00 'x' 'y'
serveur -> client-put : <rien> (0 octet) ==> OK
client-get -> serveur : 'a' 'b' 'c'
serveur -> client-get : 'x' 'y'
```

```
client-get -> serveur : 'n' 'o' 't' '_' 'a' '_' 'k' 'e' 'y'
serveur -> client-get : 00
client-put -> serveur : 'n' 'o' 't' '_' 'a' '_' 'k' 'e' 'y' 00 's' 'o' 'm' 'e' '_' 'v' 'a' '
serveur -> client-put : <rien> (0 octet) ==> OK
client-get -> serveur : 'n' 'o' 't' '_' 'a' '_' 'k' 'e' 'y'
serveur -> client-get : 's' 'o' 'm' 'e' '_' 'v' 'a' 'l' 'u' 'e'
client-put -> serveur : 'u' 00 'v'
serveur -> client-put : <rien> (0 octet) ==> OK
client-get -> serveur : 'u'
serveur -> client-get : 'v'
client-get -> serveur : 'v'
serveur -> client-get : 00
```

V. Rendu

Le travail à ce stade correspond au quatrième rendu du semestre (premier rendu sur le projet). Il compte pour 30 % de la note finale (comme indiqué dans le barème).

Pour rendre cette partie, mettez (et commit+push) tous votre code correspondant dans le répertoire `done/` de votre dépôt GitHub (de groupe ! et à la racine, **PAS** dans `provided/` ; vous ne devez **jamais** modifier le répertoire `provided/`) puis « tagger » le avec le tag `projet01` (tout en minuscules, avec un zéro ; **AUCUN** autre tag ne sera considéré comme rendu du projet : pas de majuscule, pas de souligné, avec un `0` devant le `1`) et « pousser » le résultat vers GitHub (`git push --tags`).

Le délai pour faire le `push` de rendu est : dimanche 22 avril 23:59.

ATTENTION ! Pour pouvoir être accepté, votre rendu devra : 1. être proprement tagé (tag `projet01`) ; 1. se trouver dans le répertoire `done/` ; 1. compiler avec `make`.