Quorums et arguments de la ligne de commande

Luis D. Pedrosa & Jean-Cédric Chappelier 2017

Après quelques outils « externes » (manipulation des contenus), nous revenons cette semaine sur le cœur du système avec :

1. R et W quelconques (inférieurs à N), et N différent (mais inférieur ou égal) à S ;
2. implémentation du système de quorum (« vote » en cas de valeurs différentes quand $R > 1$) ;
3. (maintenant que vous avez vu `argc` et `argv` en cours) changement de « l'interface utilisateur ».

Les points 1. et 3. sont liés : en effet, pour le moment nous avions R=1, codé implicitement « en dur » par le comportement de l'algorithme de « get », et W=N=S simplement déterminé par le nombre (S) de lignes du fichier `servers.txt`. Si l'on veut les rendre modifiables, il va falloir un moyen à l'utilisateur de les communiquer. Comme vous venez justement d'apprendre comment passer des arguments sur la ligne de commande, c'est le moyen idéal (point 3). On va alors en profiter pour laisser tomber l'interface utilisateur via des `scanf`.

Commençons donc par le point 2, plus simple. Si vous souhaitez partager le travail, ou simplement tester ce point 2 indépendement du point 3, nous vous conseillons de simuler artificiellement le point 1 par des `#define` (genre « `#define R 2` »). Vous supprimerez ces `#define` (et ferrez le lien) lorsque le point 3 sera opérationnel.

I. Quorum

Le système va maintenant vraiment devenir « *distribué* » en ce sens que les clés vont être réparties sur plusieurs serveurs, possiblement même plusieurs serveurs pour la même clé, suivant l'ordre spécifié dans le fichier `servers.txt` (du répertoire local d'où sont lancées les commandes). Cela signifie que maintenant à la fois N, W **et** R sont définis par l'utilisateur. A ce stade, nous n'avons pas encore la notion de « nœud » (revoir si nécessaire le fichier de description principal) en ce sens que la distribution des clés se fait pour le moment toujours suivant les N serveurs différents consécutifs trouvés par ordre de lecture dans le

fichier `servers.txt` (**sans** répétition, donc. La notion de « nœud » introduira la répétition possible d'un même serveur à plusieurs endroits. Ce n'est pas encore le cas ici).

Comme une même clé peut maintenant être répartie sur différents serveurs pour lesquels les mises à jour de la valeur peuvent avoir été différentes (et donc ils ne donnent pas la même valeur pour une même clé), il faut mettre en place un système de décision. Ce sera au client de garder trace des réponses reçues et décider quoi répondre au final. Comme nous avons bien modularisé le code, le seul changement à faire se situe dans la fonction `network_get()` de `network.c`.

La modification à ce stade est assez simple dans son principe : 1. ajouter de quoi compter le nombre de fois que chacune des valeurs reçues l'a été ; 2. à chaque fois que l'on reçoit une valeur, si c'est la R-ième fois qu'on la reçoit, on s'arrête (et c'est elle la valeur rendue). Et bien sûr (comme avant), si une fois les N serveurs contactés, aucune valeur reçue n'a atteint le quorum de R fois, alors le « get » est un échec.

(il y a donc en fait très peu de chose à changer dans `network_get()` lui-même si l'on a bien modularisé son code ; uniquemen 3 lignes en fait !)

Par exemple, si pour la clé pour laquelle on est en train de faire `network_get()`, on reçoit « A » du premier serveur, « B » du second et à nouveau « A » du troisième (supposons N supérieur ou égal à 3 ici) et si R=2 alors la « réponse » de `network_get()` sera la valeur « A » dès réception du message provenant du troisième serveur.

Pour garder trace du compte de chaque valeur reçue, vous êtes libre d'utiliser la structure de données que vous voulez, mais pourquoi ne pas utiliser une hashtable locale dans laquelle les clés sont les valeurs reçues et les valeurs sont les comptes associés.

Mais vous vous demandez alors peut être : comment stocker un compte sous forme d'une chaîne de caractères ?
On fera l'hypothèse (tout à fait acceptable ici) que R est plus petit que 255, donc le nombre de comptes d'une valeur donnée sera forcément plus petit que 255 et peut donc « tenir sur » un `char` : vous pouvez alors représenter ce compte par la chaîne `{ (char) compte, '\0' }` ou encore plus simplement : initialiser à la chaîne `"\x01"` (qui correspond à `{ '\1', '\0' }`) et faire ensuite simplement des `++chaine[0]`.
Bref, libre à vous..

Autre conseil enfin : modularisez en implémentant *par exemple* deux fonctions une pour augmenter de 1 le compte associé à une valeur et une pour tester si cette valeur a atteint le quorum (son compte atteint (ou dépasse, mais bon...) R). On peut aussi faire avec une seule fonction qui incrément et retourne la valeur, par exemple.

II. R, W et N quelconques

Pour revenir au point 1 et de ce qui est de W différent de S, il suffit normalement de changer simplement 1 ligne dans `network_put()`.

De même, si vous n'avez pas déjà introduit la distinction : faites maintenant attention à ce les boucles de parcours des serveurs se fassent bien sur N et non pas sur S (jusqu'ici N=S mais on veut maintenant les distinguer). A priori cela ne change que 2 lignes, une dans `network_get()` et une dans `network_put()`.

III. Arguments de la ligne de commande

[NOTE : cette section est longue car elle a été re-rédigée dans le but d'être plus claire et plus pédagogique, donc plus facile... **]**

Puisque vous venez tout juste de voir en cours les arguments de la ligne de commande (`argc`, `argv`) et que nous devons justement maintenant passer plusieurs argument supplémentaires à nos commandes (N, R et W), c'est le moment parfait d'en profiter ! Le traitement des arguments de la ligne de commande est un sujet assez impressionnant la première fois qu'on le rencontre, et nous pensions justement que le cadre de ce projet était un bon endroit pour le pratiquer. Nous allons pour le coup en profiter pour faire passer tous nos paramètres de toutes nos commandes (lesquels étaient jusque maintenant lus sur l'entrée standard via des `scanf`) à des paramètres de la ligne de commande. Le tout n'en sera que plus homogène. D'ailleurs, cela simplifiera d'autant certains outils qui ne tourneront plus en boucle, mais ne feront qu'une seule chose à chaque appel.

L'idée est donc de transformer tous nos outils suivant le schéma suivant :

```
pps-client-put [-n N] [-w W] [--] <key> <value>
pps-client-get [-n N] [-r R] [--] <key>
pps-list-nodes
pps-dump-node <IP> <Port>
pps-client-cat [-n N] [-w W] [-r R] [--] <input-key1> <input-key2> ... <output-key>
pps-client-substr [-n N] [-w W] [-r R] [--] <input-key> <position> <length> <output-key>
pps-client-find [-n N] [-w W] [-r R] [--] <key-to-search> <key-to-search-for>
```

La convention de notation est la suivante :

- `[X]` signifie que `X` est optionnel
- `<Y>` signifie que `Y` est obligatoire
- `...` signifie un nombre variable d'arguments

Pourquoi ce `[--]` optionnel ?
Pour désambiguiser entre une option et une clé qui commencerait par un « - ».
Le meilleur exemple pour illustrer l'ambiguité possible est :

```
pps-client-cat -n 5 a b c
```

Est-ce que cela veut dire :

- que l'on veut concaténer les valeurs des clés `a` et `b` pour mettre le résultat sour la clé `c` avec N=5 pour les protocole de communication
- ou bien que l'on veut concaténer les valeurs des clés `-n` (oui ça peut très bien être une clé !), `5` (oui aussi), `a` et `b` (en utilisant la valeur de N par défaut) ?

C'est pour cela que nous avons introduit l'option `--`. Le premier cas s'écrira :

```
pps-client-cat -n 5 a b c
```

et le second :

```
pps-client-cat -- -n 5 a b c
```

et si l'on veut faire ce second cat avec N=3, on écrira :

```
pps-client-cat -n 3 -- -n 5 a b c
```

Pour simplifier, nous avons généralisé cette option `[--]` à toute les commandes à options, même si pour certaines il n'y a pas de risque d'ambiguité. Plutôt que de traiter ces cas particuliers, ce qui serait fastidieux, on adoptera pour toutes nos commandes la convention que, sauf si elles arrivent après un `--`, les chaînes `-n`, `-r` et `-w` seront toujours considérées comme des options. Ainsi la commande

```
pps-client-put -n 10
```

même si elle peut nous sembler non ambigüe à nous lecteurs humains, elle sera, pour simplifier, interprétée comme : on donne la valeur 10 à N et on n'a pas donné de clé ni de valeur (donc échec : `FAIL`). Si l'on veut associer la valeur `10` à la clé `-n`, il faudra écrire :

```
pps-client-put -- -n 10
```

Ce qui simplifie grandement l'algorithme de lecture des arguments :

- tant que l'on n'a pas eu `--`, tout argument commençant par `-` qui peut être une option en est une ; échec si elle n'est pas complète (p.ex. `-n` tout seul au lieu de `-n 4`) ; les arguments qui commence par `-` mais ne peuvent pas être des options (p.ex. `-x`) ne posent pas de problème et font directement quitter le traitement des options ;
- après avoir eu `--`, aucun argument n'est une option. (`--` fait donc quitter directement le traitement des options.)

De même (pour simplifier), on supposera que les options viennent toujours en premier dans les arguments, jamais après. Donc :

```
pps-client-put key value -n 10
```

n'a pas de sens et donc échoue (`FAIL`). Pour passer l'option `-n`, il eût fallu écrire :

```
pps-client-put -n 10 key value
```

Les arguments N, R et W sont donc optionnels. S'ils ne sont pas fournis, leur valeur par défaut respective seront N=3, W=2, R=2, mais N ne doit jamais être plus grand que S (nombre de lignes de `servers.txt`) et W et R ne doivent jamais être plus grands que N (on peut faire ça simplement avec une fonction `min()` (ou des `if`) une fois toutes les options traitées *et* le fichier `servers.txt` lu, à l'endroit qui vous semble le plus approprié).

Comment donc traiter ces arguments ?
De façon générale, inspirez vous pour cela du cours et ayez une approche modulaire, typiquement en écrivant des fonctions outils dans `args.c`.
L'idée que nous proposons est de traiter tous les arguments optionnels dans `client_init()` (puisque c'est lui qui en a besoin) au travers d'une fonctions dédiée à cela : `parse_opt_args()` de `args.[ch]`. Les arguments non optionnels seront ensuite traités de façon chaque fois spécifique dans chaque des commandes `pps-client-ETC` (puisque chacune a des besoins spécifiques).

Commençons par regarder le `args.h` proposé : le type `args_t` va nous servir à garder en mémoire les valeurs de tous les arguments optionnels.
De quoi avons-nous besoin ?
De N, R et W.
Commencez donc par définir une structure ayant ces trois champs (`size_t` ira bien pour eux).

Puis on voit un `enum` incompréhensible… soit !
Et le prototype de la fonction `parse_opt_args()` : elle retourne un `args_t` (les valeurs des trois options), et prend en argument un indicateur des arguments recherchés (voir paragraphe suivant) ainsi qu'un `argv` passé par référence : `argv` est de type `char**` (tableau de chaînes de caractères) donc passé par référence cela donne un `char***`

Le sens du premier argument de `parse_opt_args()` est d'indiquer si l'on cherche N et/ou W et/ou R comme option (car toutes les commandes n'acceptent pas toutes les options). On utilisera pour cela l'`enum args_kind` précédent :

- `TOTAL_SERVERS` est pour dire que l'on accepte l'option `-n` (pour changer N) ;
- `GET_NEEDED` est pour dire que l'on accepte l'option `-r` (pour changer R) ;
- et `PUT_NEEDED` est pour dire que l'on accepte l'option `-w` (pour changer W).

Il suffit de les combiner avec un « OU » (`|`). Par exemple pour dire que l'on accepte les arguments `-n` et `-w`, on fera l'appel

```
parse_opt_args(TOTAL_SERVERS | PUT_NEEDED, &argc)
```

Et pour dire que l'on accepte les arguments `-n`, `-r` et `-w` :

```
parse_opt_args(TOTAL_SERVERS | GET_NEEDED | PUT_NEEDED, &argc)
```

Et si l'on ne veut aucun argument optionnel, simplement :

`parse_opt_args(0, &argc)`

Pour récupérer cette information dans le corps de `parse_opt_args()`, il suffit d'utiliser un « ET » (`&`) :

- `if (supported_args & TOTAL_SERVERS)` permet de savoir si l'on veut traiter `-n` ;
- `if (supported_args & GET_NEEDED)` si l'on veut traiter `-r` ;
- et `if (supported_args & PUT_NEEDED)` si l'on veut traiter `-w`.

Pour parser les options, procédez assez simplement comme présenté en cours. Ici, c'est même plus simple en fait : dans cette fonction nous savons que nous n'attendons que des options avec argument complémentaire (exemple : `-n 10`) ou alors `--` qui nous fait terminer cette fonction (sans oublier d'avancer `argv`). Récupérez les valeurs que vous pouvez/devez ou retournez `NULL` en cas d'erreur (option incomplète par valeur manquante, option non compatible avec `supported_args`, ...). Notez qu'une « option » inconnue, n'est pas une erreur : ce n'est juste pas une option et nous fait terminer `parse_opt_args()`. Par exemple on peut très bien appeler, sans échec :

```
pps-client-put -a bb
```

qui associe la valeur « `bb` » à la clé « `-a` ».

Passons maintenant à `client_init()`, qui appelle cette fonction `parse_opt_args()`. Du coup, `client_init()` a besoin de plus d'arguments. Pour ne pas changer son prototype au cours du projet, nous avons (dans `client.h`) introduit le type `client_init_args_t` : cela nous permet de rester générique (toujours le même type) tout en évoluant. Changez maintenant le type `client_init_args_t` en :

1. supprimant le nom du programme (de type `const char*`) introduit en semaine 5 et qui ne nous sert plus puisque `argv[0]` joue le même rôle ;
2. ajoutant un *pointeur* sur `argv` (que l'on passera en second argument à `parse_opt_args()`)
3. et tout ce que vous pourrez trouver d'utile à y mettre. De notre coté, nous proposons d'y mettre trois `size_t` : le nombre d'arguments requis, les argument optionnels acceptés (même format que ci-dessus, premier argument de `parse_opt_args()`) et `argc` (la taille de `argv`).

Par ailleurs, il faut aussi modifier la structure `client` pour y mettre la valeur de retour de `parse_opt_args()` : un pointeur sur `args_t`.

Vous avez maintenant toutes les « armes » pour réviser `client_init()`. Mais avant cela, commençons par le plus simple : dans `client_end()`, n'oubliez pas de libérer les `args_t` (alloué par `parse_opt_args()` et stockés dans la structure `client`).

Ensuite, dans `client_init()` :

1. commencez par sauter le premier argument (qui est le nom du programme ; garder un pointeur dessus si vous en avez besoin ; pas de copie !) ;

2. vérifiez ensuite la comptatibilité entre `argc` (reçu au travers de `client_init_args`) et le nombre d'arguments requis (aussi dans `client_init_args`; il ne peut pas être plus grand) ; sortez avec `ERR_BAD_PARAMETER` en cas d'incompatiblité ;

3. de façon générale, pensez à traiter tous les cas d'erreur que vous pouvez imaginer (mais n'y passez pas 3 heures non plus) et surtout *dès que* vous les envisagez ; il y a vraiment plusieurs façon de s'y prendre ici ; choisissez la vôtre, mais surtout pensez à arrêter (= sortir avec erreur) *dès que* vous avez détecté un cas d'erreur.

4. Appelez ensuite `parse_opt_args()` avec les deux champs correspondant dans `client_init_args` (`supported_args` et pointeur sur `argv`).

   Notez bien que l'on ne passe pas `argc` à `parse_opt_args()` parce que la vérification de la bonne taille (bon nombre d'arguments au minimum) doit avoir été effectuée **avant** l'appel. `parse_opt_args()` « sait » donc que l'`argv` qu'il reçoit (par référence) est de taille (au moins) correcte.

   Et stockez le résultat dans le `client` des `client_init_args`.

Passez ensuite à la modification des différentes commandes de client. Nous vous conseillons grandement de procéder ici étape par étape, un par un, en vérifiant pleinement (plein de tests) que le passage des argument de la ligne de commande se fait correctement dans tous les cas de la commande traitée.

Commencez par exemple par `pps-client-get` :

1. complétez/modifiez son `client_init_args` passé à `client_init()` ; par exemple ici, on fera un appel comme :

   ```
   err = client_init( (client_init_args_t) { &client, 1, TOTAL_SERVERS | GET_NEEDED, (size
   ```

2. supprimez toute la boucle de lecture **(note 1** ci-dessous **)** ;

3. la clé recherchée devient donc simplement `argv[0]`, car, à ce stade :

4. le nombre d'arguments voulus doit avoir été correctement vérifié par `client_init()`

5. lequel a de plus, au travers de `parse_opt_args()`, déplacé `argv` vers le bon endroit en cas d'arguments optionnels

6. (et n'oubliez pas de supprimer le `free()` de la clé, devenu inutile.)

Procédez ensuite de même, commande par commande. Vous n'êtes pas obligés de tout faire d'un coup. Pour les commandes ayant un nombre quelquonque d'arguments, on prendra la convention que le nombre d'arguments requis (dans `client_init_args`) est mis à `SIZE_MAX`.

[**Note 1 :** Si vous avez envie de garder les deux versions et avez compris les directives de pré-compilation (cours de cette semaine), ajoutez des `#ifdef` appropriés pour garder les deux versions du code ; mais ceci est *vraiment*

optionnel, **hors du cadre du projet** ; ne vous lancez pas là dedans si vous ne voyez pas bien de quoi je parle ou si cela ne vous motive pas ; vous ne feriez que perdre du temps sans gagner de points.)

IV. Rendu

Le travail à ce stade correspond au cinquième rendu du semestre (second rendu sur le projet). Il compte pour 15 % 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 `projet02` (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 13 mai 23:59.

ATTENTION ! Pour pouvoir être accepté, votre rendu devra : 1. être proprement tagé (tag `projet02`) ; 1. se trouver dans le répertoire `done/` ; 1. compiler avec `make`.