Projet Système :
Gestion d'un dictionnaire de données

FRANCES Tom, BOU-SERHAL Jean

October 22, 2021

Contents

Le but de ce projet est de créer une application permettant la gestion d'un dictionnaire de données. Ces données seront des chaînes de caractères, que nous devrons pouvoir stocker et consulter.
La gestion de ces données sera répartie entre plusieurs processus créés par l'application.

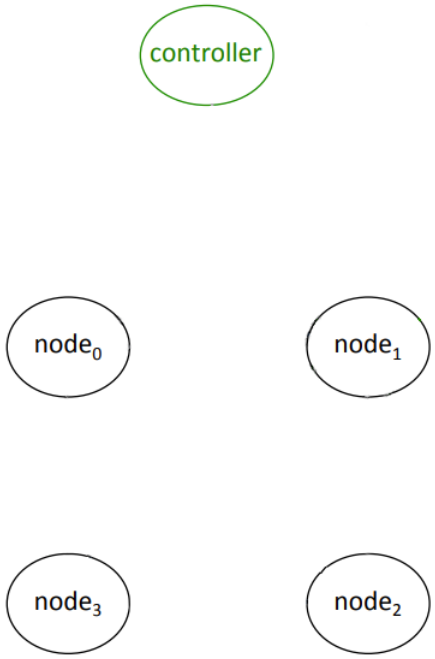
1 Architecture

Dans cette section, nous allons détailler comment l'application est concue à sa base, en détaillant la gestion des processus et des canaux de communication entre ces mêmes processus.

1.1 Les processus fils, ou nodes

L'application va créer ***N*** processus, que nous appellerons ***node*** par la suite. Chacun de ces nodes s'occupera de la gestion d'une partie du dictionnaire de données. Ces nodes seront tous les fils d'un même processus principal, appelé ***controller***, qui assurera l'interface avec l'utilisateur. Le nombre de *node* que l'application créera sera passé en paramètre à la commande.
Les *nodes* seront donc tous créés par le *controller*, dans notre cas le programme principal.

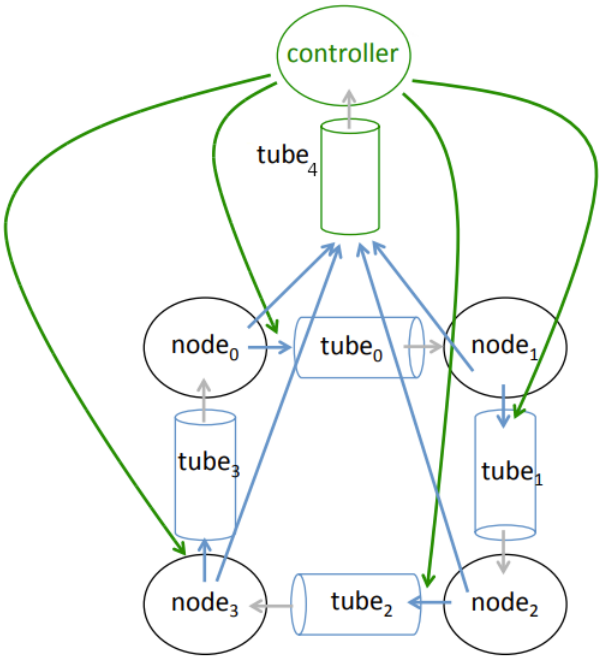
Nous allons donc nous retrouver avec un système de processus comme imagé ci-dessous, dans le cas où $N = 4$, avec les nodes indicés de 0 à $N - 1$



1.2 Communication entre les processus

Afin que l'application fonctionne correctement, nous allons utiliser une **communication par tubes** grâce à des pipes :

- les *nodes* seront connectés entre eux par une structure en anneau, c'est-à-dire qu'il y aura un tube entre chaque $node_i$ et $node_{i+1}$, et entre $node_{N-1}$ et $node_0$, afin de pouvoir faire circuler une requête jusqu'à ce qu'elle atteigne le bon *node*. Le *controller* pourra écrire dans chacun de ces tubes.
- tous les *nodes* pourront également écrire dans un même tube afin de communiquer directement avec le *controller*



Par exemple, si le *controller* souhaite envoyer une commande que le $node_2$ devra exécuter :

1. Il envoie la commande au $node_0$ via le $tube_3$
2. Ce dernier la reçoit, ne la traite pas car ce n'est pas son rôle, et la transmet au *node* suivant via le $tube_0$
3. Le $node_1$ reçoit la commande, ne la traite pas et la transmet au *node* suivant via le $tube_1$
4. Le $node_2$ reçoit la commande, la traite, puis envoie un message au *controller* via le $tube_4$ pour lui dire que tout s'est bien passé

2 Conception

2.1 Mise en place, création des nodes et des tubes

Pour la création des *nodes*, nous nous servons d'une boucle *for* à indice i, de 0 à N-1 inclus. A chaque itération, un $fork()$ est réalisé, qui est la fonction créatrice d'un processus fils par clonage du processus père. Ainsi avec N itérations, nous obtenons N processus fils.
Pour s'assurer de la bonne création de ces derniers, chaque fork se fait au sein d'un *switch* qui interprète la valeur renvoyée par la fonction $fork()$ dans le processus père :

- 1er cas : la valeur renvoyée est -1. Cette valeur signifie que le processus fils n'a pas été créé (erreur affichée, stderr).
- 2e cas : la valeur renvoyée est 0. Cette valeur signifie que la création du processus fils s'est achevée avec succès. Ce cas dicte en conséquent le comportement du fils.
- 3e cas : la valeur renvoyée est le PID du fils (comportement du père, qui ici ne fait rien de particulier).

Mode DEBUG :
Après la fin de la boucle for de création des fils, nous utilisons la commande *pgrep -P "pid"*, qui affiche tous les processus fils du processus dont le pid est donné en paramètre. Cette commande est exécutée à l'aide de la fonction *execlp*. Cependant, puisque la fonction *execlp* se termine par

un *exit*(), il est nécessaire de créer un dernier processus fils, qui sera chargé d'exécuter cette commande (sinon, si c'est le père qui effectue la commande, il mourra à la fin et le programme se terminera). Le pid du père (*controller*) est récupéré à l'aide de la fonction *getppid*().

Pour la création des *tubes*, nous utilisons une boucle *for* avant la création des *nodes*, afin que tous les *nodes* les connaissent et puissent les utiliser par la suite. Ils sont stockés dans un tableau à 2 dimensions (2, N+1) et initialisés par la fonction *pipe*() :

- les N premiers sont les *tubes* utilisés dans la structure en anneau de communication entre les *nodes*
- le dernier *tube* est le *tube controller*, qui sert à la communication entre les *nodes* et le *controller*

Dans le cas d'une erreur à la création d'un *tube*, le programme se termine immédiatement.

Mode DEBUG :
A chaque appel de la fonction *pipe*(), un affichage indique si la création du *tube* s'est bien déroulée ou non.

2.2 Structure du dictionnaire de données

Comme dictionnaire, nous allons utiliser une collection associative sous la forme d'une liste chainée d'éléments. Ces éléments sont un couple clé/valeur (plus un pointeur vers l'élément suivant), de type $Table\_entry$. La tête du dictionnaire est défini par le type $PTable\_entry$, un pointeur vers le premier élément du dictionnaire.
Cette structure est mise à notre dispostion, ainsi que plusieurs fonctions de manipulation :

- une fonction $store(PTable\_entry * table,\ int\ k,\ char\ v[])$, qui prend en paramètre un entier *clé* et une chaine de caractère *valeur*, et qui insère un nouvel élément constitué de la clé et de la valeur à la fin du dictionnaire
- une fonction $*lookup(PTable\_entry\ table,\ int\ k)$, qui prend en paramètres la référence vers la table et un entier clé, et retourne la valeur associée à cette clé (si la valeur n'est pas trouvée, retourne *null*)
- une fonction $display(PTable\_entry\ table)$ qui prend en paramètre la référence d'une table et affiche toutes ses associations clé-valeur. Nous avons modifié la fonction pour qu'elle renvoie 1 à la fin de son exécution, pour des questions de synchronisation (cf 2.6)

Nous avons ajouté 2 fonctions sur la structure du dictionnaire :

- une fonction *init_table*(), qui initialise et alloue la mémoire pour un dictionnaire et renvoie ce dernier
- une fonction $key\_exists(PTable\_entry\ table,\ int\ key)$, qui prend en paramètres la référence vers la table et un entier clé, et retourne vrai si la clé existe dans le dictionnaire, faux sinon

2.3 Structure des commandes

Nous avons décidé, afin de faciliter les échanges de commandes entre le *controller* et les *nodes*, de créer une structure Request, qui contient tous les éléments nécéssaires à l'exécution d'une commande saisie par l'utilisateur.
Cette structure est définie par un entier commande (qu'on limitera aux cas 0, 1, 2 et 3 dans les fonctions qui suivent), un entier clé (key, strictement supérieur à zéro), une chaîne de caractères data (qui est la valeur de la donnée stockée, et limitée à une taille de 30 caractères), et par un entier réponse. Ce dernier sera modifié par les *nodes* en fonction du résultat de l'exécution de la requète, et interprété par le *controller*.
Tous les entiers de la structure Request sont initialisés à -1, et la chaîne de caractères sera initialisée à une chaine vide.

Nous avons écrit plusieurs fonctions pour la gestion des requêtes :

- une fonction *init_request*(*Request* ∗ *request*), qui initialise tous les champs d'une requête comme décrit ci-dessus
- une fonction *new_request*(*Request*∗ *request*), qui va interagir avec l'utilisateur pour formuler une nouvelle requête, en modifiant les champs de la requête passée en paramètre :
  - saisie de la commande : restriction de la valeur entre 0 et 3
  - saisie de la clé : la valeur saisie doit être strictement positive. Cette saisie de la clé n'est conduite que dans les cas où la commande vaut 1 (fonction *store*()) ou 2 (fonction *lookup*()), car les 2 autres commandes n'utilisent pas de clé
  - saisie de la valeur : la valeur saisie ne doit pas être vide. Cette saisie de la valeur n'est conduite que dans le cas de commande 1 (fonction *store*()), car aucune autre fonction ne la nécessite
- une fonction *print_request*(*Request* ∗ *request*), qui affiche les éléments d'une requête (mode DEBUG)

2.4 Protocole d'échange

Afin que la communication entre les *nodes* et le *controller* soit fonctionnelle, il est nécessaire de fermer convenablement tous les pipes inutilisés, et ce pour chaque *node* et pour le *controller*. Nous avons écrit une fonction *close_unused_pipes*(*int* ∗ ∗ *pipes*, *int node*, *int N*), qui prend en paramètres les *pipes*, l'indice d'un *node*, et le nombre total de *nodes* créés N.

- si le node est le *controller*, on ferme l'extrémité d'écriture du *pipe controller*, et nous fermons les extrémités de lecture de tous les *pipes* des *nodes* (cf figure 1 ci-dessous)
- si le node est l'un des processus fils, nous fermons l'extrémité de lecture du *pipe controller*, les extrémités de lecture des *pipes* de tous les *nodes* fils hormis celle du pipe précédent dans l'anneau, et les extrémités d'écriture des *pipes* de tous les *nodes* fils hormis celle du pipe suivant dans l'anneau (cf figure 2 ci-dessous)

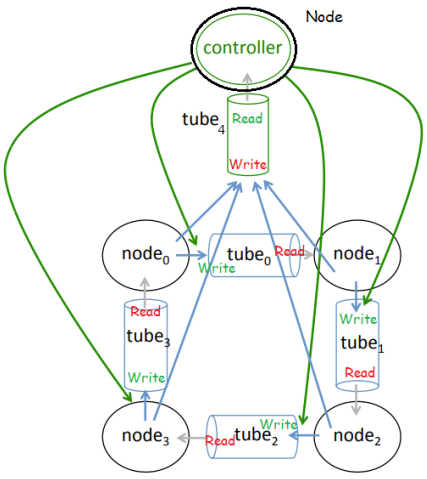


Figure 1: Pipes fermés(rouge)/ouverts(vert) pour le controller

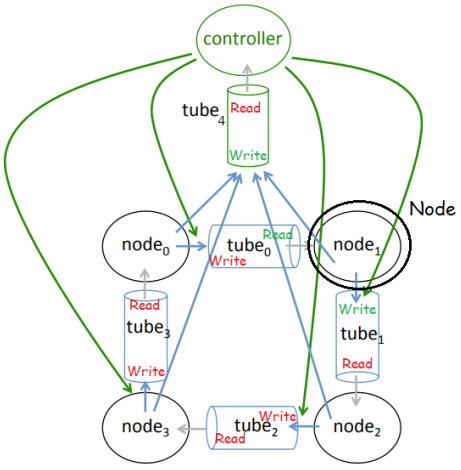


Figure 2: Pipes fermés(rouge)/ouverts(vert) pour un fils

Nous avons écrit 2 fonctions *write_pipe*() et *read_pipe*(), qui permettent respectivement d'écrire et de lire la requête passée en paramètre dans le pipe lui aussi passé en paramètre. Le réel intéret se trouve dans la fonction *read_pipe*, qui contient une boucle *while* avec une condition d'arrêt sur le nombre d'octet lu par la fonction *read* : dès que le nombre d'octet lu est strictement positif, la

boucle se termine. Cela permet de mettre un processus en attente d'une lecture, qui ne continuera pas la suite de son travail tant qu'il n'aura rien lu dans le *pipe*.

2.5 Les fonctions *node()* et *controller()*

Une des plus importantes question que nous nous sommes posé est comment faire tourner le programme tant que l'on ne saisi pas la commande *exit*. Pour y parvenir, nous avons décidé de faire tourner nos *nodes* et le *controller* dans une boucle *while*, avec une condition d'arrêt vraie dès lors que la commande *exit* est saisie par l'utilisateur. Nous allons détailler le fonctionnement des 2 fonctions principales du programme : *node*() et *controller*().

La fonction *controller*(*Request* $*$ *request*, *int* $**$ *pipes*, *int N*) nous permet de gérer la saisie d'une requête, son envoi aux *nodes*, et la réception et traitement de la réponse contenue dans la requête par le *controller*. Elle prend en paramètres la requête, le tableau des *pipes*, et le nombre de *nodes* N. Cette fonction fait les actions suivantes :

1. saisie d'une nouvelle requête avec la fonction *new_request*()
2. l'envoi de la requête au premier *node* (commandes *store* et *lookup*) ou à tous les nodes en même temps (commandes *dump* et *exit*)
3. la réception et le traitement de la réponse

Toutes ces étapes se font en boucle tant que l'utilisateur n'a pas saisi la commande *exit*.

La fonction *node*(*Request* $*$ *request*, *int reading_pipe*[], *int writing*$_p$*ipe*[], *int controller_pipe*[], *int N*, *int node_indice*) nous permet de gérer la réception, le traitement, et/ou la transmission d'une requête (et la réponse modifiée) par un *node* à un autre ou au *controller*. Elle prend en paramètres la requête, le *pipe* où s'effectuera la lecture, le *pipe* où s'effectuera l'écriture, le *pipe controller*, le nombre total de *nodes* N, et l'indice du *node* concerné. Cette fonction fait les actions suivantes :

1. création d'un nouveau dictionnaire à l'aide de la fonction *init_table*()
2. écriture de la requête dans l'extrémité d'écriture (1) de la pipe du controller (fct. *write_pipe*())
3. tant que la commande reçue dans la requête est différente de 0 (commande *exit*) (sous forme d'un do{...}while(), puisqu'on cherche à exécuter le code au moins une fois) :
   - lecture de la requête sur le *reading_pipe*
   - switch sur la commande :
     - cas 0 (*exit*) : la variable fonctionnant comme booléen est mise à 1, sortie de la boucle while, le *node* met à jour la réponse de la requête à -1, qu'il renverra au controller, et meurt
     - cas 1 (*store*) : si c'est au *node* qui vient de recevoir la requête de traiter la commande (c'est-à-dire, si la valeur de la clé modulo N est égale à l'indice du *node*), alors on vérifie si la clé existe déjà dans le dictionnaire du *node* en question, à l'aide de la fonction *key_exists*() : si la clé existe déjà, elle ne sera pas ajoutée, la réponse de la requête est mise à 0 et renvoyée au *controller*; si la clé n'existe pas, on ajoute la donnée correspondante dans la table à l'aide de la fonction *store*(), la réponse est mise à 1 et renvoyée au *controller*
     - cas 2 (*lookup*) : même fonctionnement que la commande *store*, mais si la clé existe la valeur associée est affiché à l'aide de la fonction *lookup*(), la réponse est mise à 1 et renvoyée au *controller* ; si la clé n'existe pas, la réponse est mise à 0 et renvoyée au *controller*
     - cas 3 (*dump*) : le premier *node* affiche sa table puis transmet la requête au *node* suivant; les *nodes* 1 à N-1 se mettent en position d'attente d'une lecture depuis le *node* précédent, avant de pouvoir afficher sa table une fois une réponse reçue du *node* précédent

Une fois sorti de la boucle, le *node* renvoie la requête au *controller* avec la réponse mise à -1, puis meurt.

Mode DEBUG :
A chaque lecture d'une requête par un *node*, le PID de ce dernier, ainsi que le détail de la requête sont affichés à l'aide des fonctions *getpid*() et *print_request*(). Le *node* affichera également un message indiquant si c'est bien lui qui a traité la requête ou s'il l'a transmise au *node* suivant.

2.6 Synchronisation

La question de la synchronisation entre tous les processus se pose dans 3 situations.

La première est lorsque le *controller* doit attendre que tous les fils soient prêts, juste après leur création, avant de commencer l'interaction avec l'utilisateur.
Dans ce cas, nous avons écrit une fonction $wait\_childs\_ready(Request* request, int** pipes, int N)$, qui met en attente le *controller* jusqu'à ce que tous les *nodes* soient créés et prêts. Pour ce faire, cette fonction prend en paramètres une requête, les *pipes*, et N; au début de leur création, tous les *nodes* vont envoyer une première requête au *controller*, et ce dernier va compter le nombre de réponses envoyées par les *nodes* à l'aide d'un compteur dans une boucle *while*, qui incrémente ce compteur à chaque lecture d'une réponse : lorsqu'il atteint N, c'est que tous les fils sont prêts, le *controller* peut alors commencer l'interaction avec l'utilisateur.

La seconde est lorsque le *controller* attend la fin du traitement de la commande *store*() ou *lookup*() par le bon *node*. Dans ce cas, c'est le même principe que dans le précédent, mais avec une seule réponse : le *controller* attend la réponse de la part d'un des fils (depuis le *pipe controller*), et reprend l'interaction avec l'utilisateur dès sa réception et interprétation.

La dernière est lorsque la commande *display*() est exécutée. Dans ce cas, le *controller* envoie la requête à tous les *nodes*, mais les affichages des tables de chacun ne doivent pas se chevaucher. Pour palier à ce problème, nous avons imaginé la solution suivante : à la réception de cette requête, tous les *nodes* hormis le premier (celui d'indice 0) se mettent en attente d'une lecture depuis le *node* précedent. Le premier *node* affiche sa table dès la réception de la commande, et lorsqu'il a terminé, envoie la requête au node suivant, qui pourra alors commencer son affichage, et ainsi de suite pour tous les *nodes*. Le dernier *node* enverra une réponse au *controller* pour que celui-ci recommence l'interaction avec l'utilisateur.

2.7 Programme principal

Le programme principal se déroule en effectuant les actions suivantes :

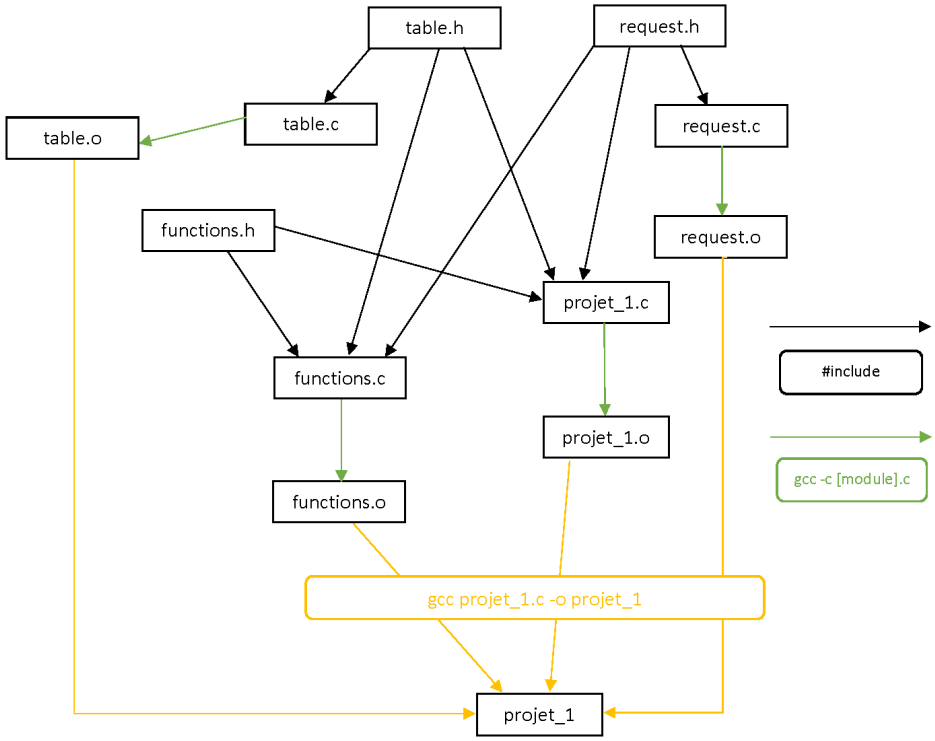
- gestion des paramètres de la commande
- création des *pipes*
- création de la requête
- création des fils, fermeture des *pipes* inutilisés et lancement de la fonction *node*() pour chacun
- fermeture des *pipes* inutilisés pour le *controller* et attente des fils
- lancement de la fonction *controller*()
- fin

3 Modularisation et Makefile

Nous avons séparé notre projet en différents modules, afin de faciliter la lecture du code

- le module TABLE (fichiers table.c et table.h) contient la définition de la structure du dictionnaire (*PTable_entry*) et toutes les fonctions associées à la gestion du dictionnaire (*init_table*(), *lookup*(), *store*(), *display*() et *key_exists*())
- le module REQUEST (ficheirs request.c et request.h) contient la définition de la structure d'une requête (*Request*) et toutes les fonctions associées à la gestion d'une requête (*init_request*(), *new_request*(), *print_request*())
- le module FUNCTIONS (fichiers functions.c et functions.h) contient toutes les fonctions associées la gestion du programme principal, communication entre les *nodes*, lecture et écriture, synchronisation, etc (*close_unused_pipes*(), *write_pipe*(), *read_pipe*(), *interpret_response*(), *node*(), *controller*(), *wait_childs_ready*())
- le module PROJET_1 (fichier projet_1.c) contient le programme principal *main*

Vous pouvez voir dans le schéma ci-dessous comment sont organisés les fichiers.



Les commandes en vert et en orange seront regroupées dans un même Makefile, automatisant la compilation et la création d'un programme exécutable « projet_1 ».

Ce projet peut être compilé en 2 modes différents :

- mode classique : vous pourrez utiliser l'application sans affichage superflu → make
- mode DEBUG : des informations supplémentaires seront affichées, notamment dans le but de vérifier le bon fonctionnement des nodes et pipes → make DEBUG