Rapport projet Programmation impérative

Axel DEORA, Clément HERBIN, Ewen LE BIHAN

January 2023

1 Introduction

Le but du projet est d'implémenter en Ada un routeur avec cache. Un routeur est un élément d'un réseau qui a pour objectif de transmettre les paquets qu'il reçoit sur une interface d'entrée vers la bonne interface de sortie en fonction des informations qui sont stockées dans sa table de routage.

La première partie du projet était d'implémenter une routeur simple, c'est-à-dire un routeur sans cache.

La deuxième partie du projet consistait à implémenter un routeur avec cache. Il y a deux types de caches :

— Cache LA, qui est un cache représenté par une liste chainée,

— Cache LL, qui est un cache représenté quant à lui par un arbre binaire.

De plus, pour les deux types de caches, il y a différentes politique de remplacement à mettre en place. Elles définissent la donnée qui sera supprimée du cache lorsque celui-ci sera plein :

— FIFO (First In, First Out) : la donnée la plus ancienne du cache,

— LRU (Least Recently Used) : la donnée la moins récemment utilisée,

— LFU (Least Frequently Used) : la donnée la moins utilisée.

Table des matières

2 Architecture de l'application en modules

2.1 Routeur simple

Pour implémenter le routeur simple, nous avons défini plusieurs modules :

1. `Parsing` qui permet d'analyser les différents fichiers d'entrées ainsi que la ligne de commande pour lancer le programme,
2. `Table` qui défini des types de données et des fonctions et procédures permettant de manipuler des adresses IP,
3. `RouteurSimple` qui défini un routeur simple,
4. `Exceptions` qui défini différentes exceptions utilisés dans le programme.

2.2 Routeur avec cache

Chaque type de cache à besoin de stocker les demandes et les défauts de caches. Nous avons donc créé un module `Caches` où nous définissons une structure de données représentant les statistiques d'un cache.

Ensuite, il y a un module `Routeur_la` pour le routeur avec un cache de type arbre et un module `Routeur_ll` pour le routeur avec un cache de type liste chainée.

2.2.1 Routeur LL

Pour implémenter les caches avec différentes politique de remplacement, nous avons décidé de créer trois modules :

1. `Lca_Fifo` qui représente le cache avec une politique FIFO
2. `Lca_Lru` qui représente le cache avec une politique LRU
3. `Lca_Lfu` qui représente le cache avec une politique LFU

2.2.2 Routeur LA

Nous avons choisi d'implémenter les trois politiques avec un seul type. Chaque nœud stockant une interface stocke également un nombre flottant appelé `critere_politique`, qui prend des valeurs différentes selon la politique choisie :

FIFO Ancienneté d'insertion : Le noeud le plus récemment inséré dans l'arbre est à 0. Les autres nœuds voient cette valeur incrémentée à chaque insertion d'un nouveau nœud.

LRU Ancienneté d'utilisation : Le nœud le plus récemment utilisé (celui dont l'interface a été choisie) est à 0. Les autres nœud voient cette valeur incrémentée à chaque demande au cache.

LFU Fréquence d'utilisation : Chaque nœud a sa valeur mise à jour à chaque demande d'accès.

3 Présentation des principaux choix réalisés

3.1 Routeur simple

Pour l'implémentation du routeur simple, nous avons décidé de réutiliser le module LCA du PRO2, en faisant quelques modifications, nous avons rendu ce module non générique, et remplacer l'unique donnée stockée par un triplet comprenant l'adresse, le masque ainsi que l'interface.

3.2 Routeur avec cache

3.2.1 Routeur LL

Pour l'implémentation du cache avec des listes chainées, nous avons décidé d'arrêter d'utiliser le module LCA du PRO2. En effet, nous avons essayé d'utiliser la généricité de ce module, en le modifiant, pour représenter les différentes politiques de remplacement du cache. Mais il y a eu des problèmes, nous avons donc défini de nouveau type de données pour pouvoir représenter les listes chainées. Cela rendait plus facile la mise en place des algorithmes permettant la vérification et l'ajout d'élément dans le cache.

3.2.2 Routeur LA

Nous avons choisi de gérer les trois politiques en un seul paquet et un seul type : la configuration de la politique est gérée par la valeur d'un *enum* `T_Politique`, stockée dans un champ de l'enregistrement `T_Cache_ABR`.

Les nœuds stockant des interfaces ont aussi un champ `critere_politique`, un nombre flottant, qui stocke une donéee différente selon la politique choisie :

FIFO Ancienneté d'insertion : le noeud dernièrement inséré est à 0, les autres s'incrémente à chaque ajout de noeud.

LRU Ancienneté d'accès : le noeud dernièrement accédé (utilisé, celui dont l'interface a été choisie) est à 0, les autres s'incrémentent à chaque demande.

LFU Fréquence d'accès : chaque noeud voit sa valeur mise à jour à chaque demande.

Nous avons choisi d'opérer sur des chaînes de caractères des représentation binaires des adresses plutôt que des entiers dans $\mathbb{N}/(2^{32}-1)\mathbb{N}$: les algorithmes sur cette arbre requiert de nombreuses opérations sur des suffixes, préfixes et sous-chaînes d'une chaîne, il était plus intéréssant de manier des caractères que de devoir traduire ces opérations en opérations sur des entiers, surtout quand on remarque que *Ada* ne propose pas les opérations `shift_right` et `shift_left` (décalages bit-à-bit) sur des entiers (il faut utiliser un type `bytesIO` que nous n'avons pas choisi d'utiliser). Le code est rendu également plus lisible.

4 Présentation des principaux algorithmes et types de données

4.1 Routeur avec cache

Pour représenter les statistiques d'un cache, nous avons défini un type de donnée qui contient un entier représentant le nombre de demandes et un entier représentant le nombre de défaults de cache.

4.1.1 Routeur LL

Pour représenter le cache avec une liste chainée et les différentes politiques de remplacement, il faut définir différents types de données. Les 3 structures de données représentant chaque politique de cache ont des champs en communs :

- `capacite` : la capacité du cache,
- `taille` : la taille actuelle du cache,
- `lca` : la liste chainée,
- `stat` : la structure de donnée représentant les statistiques.

Le cache avec la politique LRU aura quant à lui un entier `compteur` représentant un compteur en plus.
Ensuite, pour représenter les différentes listes chainées, il y a aussi une base commune :

- `ip` : l'adresse IP,
- `masque` : le masque,
- `eth` : l'interface,
- `suivant` : un pointeur vers l'élément suivant.

Pour la politique LRU, il y a un entier `compteur` représentant un compteur en plus, ce qui permettra de trouver l'élément le moins récemment utilisé en cherchant l'élément de la liste contenant le plus petit compteur, et donc de remplacer cette élément par le nouvel élement.

Pour la politique LFU, il y a un entier `frequence` représentant la fréquence, ce qui permettra de trouver l'élément le moins fréquemment utilisé en cherchant l'élément avec la plus petite fréquence, et donc de remplacer cette élément par le nouvel élement.

4.1.2 Routeur LA

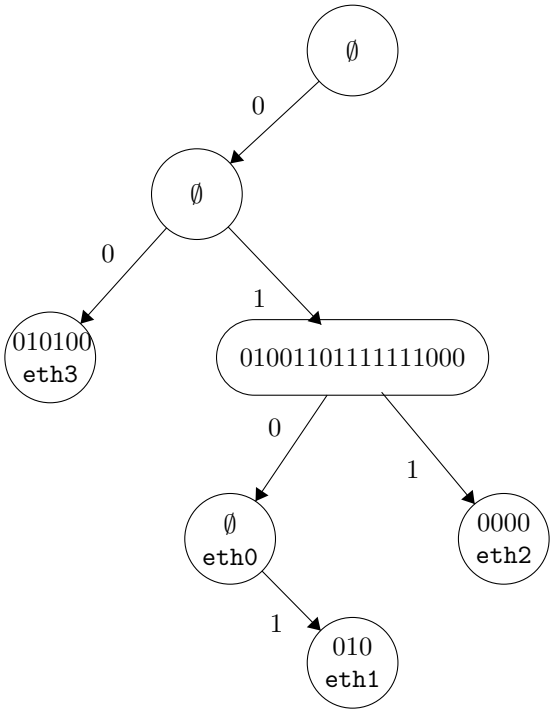
Pour l'implémentation du cache avec des arbres binaires de recherche, nous avons choisi de représenter les nœuds par des enregistrements, avec pour champs :

- `fils_gauche`, `fils_droit` : pointeurs vers les nœuds enfants
- `eth` : l'interface associée à ce nœud. Une chaîne vide correspond à l'absence de règles correspondant au chemin jusqu'à ce nœud.
- `fragment` : fragment du préfixe d'URL. Peut être vide.
- `critere_politique` : nombre utilisé pour la gestion de la politique de suppression (voir 2.2.2)

L'arbre stocke des préfixes d'URLs, qui sont les URLs présents dans la table de routage, sans la partie dont le masque est fait de zéros (on ne stocke que la partie "utile" de la règle)

Une règle est ainsi représentée par une chaîne de caractères `'1'` ou `'0'`, ainsi que l'interface vers laquelle le paquet doit être routé.

Pour trouvé ladite interface, ont descend dans l'arbre, et l'adresse est la concaténation des fragments des nœuds, sans oublier que la descente dans un sous-arbre gauche encode la concaténation d'un `'0'`, et celle dans un sous-arbre droit encode la concaténation d'un `'1'` :



Par exemple, les paquets routés à `eth2` doivent avoir une adresse IP commençant par `01010011011111111000010000`

5 Tests du programme

Pour chaque module, nous avons fait un programme de test :
— `cache_abr_test` teste le module `cache_abr`,
— `cache_lca_test` teste les modules `lca_fifo`, `lca_lru` et `lca_lfu`,
— `parsing_test` teste le module `parsing`,
— `table_test` teste le module `table`,
— `routeur_simple_test` teste le module `routeur_simple`.

6 Difficultés rencontrées

6.1 Analyse du fichier table

La première difficulté rencontrée est l'analyse du fichier table. En effet, le fait de devoir transformer un string représentant l'adresse IP en un entier était assez difficile à mettre en place.

6.2 Les tests

Les tests concernant les fonctions de parsing étaient compliqués à mettre en place. Pour les tests de parsing de fichier, il fallait créer un fichier au début du test et écrire des données à l'intérieur, puis ensuite réaliser les tests et enfin supprimer le fichier. Cela était assez long à mettre en place.

7 Organisation de l'équipe

Pour l'organisation de l'équipe, nous avons décidé de garder le découpage donner dans le sujet :

- Axel Deora a fait les raffinages ainsi que l'implémentation des routeurs,
- Clément Herbin a fait les raffinages ainsi que l'implémentation des caches du routeur LL, de plus, il a fait l'analyse du fichier table,
- Ewen Le Bihan a fait les raffinages ainsi que l'implémentation des caches du routeur LA, de plus, il a fait l'analyse du fichier paquets ainsi que l'analyse de la ligne de commande.

Concernant le test, chacun a testé la partie de code qu'il a implémenté, sauf pour quelques exceptions. Cependant, nous avons chacun relu le code des autres.

8 Bilan technique

8.1 État d'avancement du projet

Le routeur simple est fonctionnel.

Le routeur avec cache marche aussi, dans les deux cas (Arbre et Liste chaînée). Cependant, il y a des fuites de mémoires avec le cache LA.

Concernant les tests, tous les modules sont testés, sauf les modules `routeur_la` et `routeur_ll`.

8.2 Amélioration et évolution

- La partie du code sur le cache LCA est assez redondante, un axe d'amélioration est donc de modifier cette partie du code pour enlever cette redondance.
- Le cache à arbre utilise des `Unbounded_String` par facilité, mais il aurait été intéréssant de gérer la manipulation des fragments d'IP avec des `String(32)`, comme les addresses IP ne peuvent par définition pas dépasser les 32 bits.

9 Bilan personnel et individuel

9.1 Bilan de Clément

Premièrement, ce projet m'a permis de découvrir le principe d'une table de routage et comment cela fonctionne. Nous nous avons utilisé git pour gérer les différentes versions du programme, cela m'a donc permis d'utiliser git dans un contexte de groupe. Ensuite, ce projet m'a permis de mettre en pratique les raffinages pour la conception des algorithmes puis d'implémenter cela en Ada.

Ayant fait les raffinages du cache avec des listes chainées, j'ai passé environ 4 heures à faire des raffinages. Ensuite, j'ai passé environ 12 heures pour la partie sur le routeur simple et environ 10 heures pour la partie cache avec liste chainée. J'ai enfin passé quelques heures pour la rédaction du rapport et la mise au point de différentes parties du code.

9.2 Bilan de Ewen

J'avais déjà effectué des projets traitants les arbres en classe préparatoire, mais avec le langage de programmation fonctionnelle *OCaml*. Implémenter la même structure de données dans un langage de programmation impératif m'a permis de découvrir comment les pointeurs peuvent être utilisés pour réaliser ce genre de structure récursives d'une autre manière.

Il m'a fallu comprendre que, contrairement à l'approche fonctionnelle de *OCaml*, on ne retourne pas de nouveau arbres lors d'appels à des fonctions récursives, mais l'on *modifie l'arbre passé en argument sur-place*. Je me suis rendu compte de la nécéssité de ce changement de paradigme pour éviter les fuites de mémoires, hélas trop tard pour le rendu du projet.

J'ai au total passé environ 30 heures sur ce projet (traqué par *WakaTime* : approximativement 24 heures à coder (`https://wakatime.com/@ewen_lbh/projects/wgwibtqnra?start=2023-01-08&end=2023-01-14`), et 6 heures à réfléchir sur les algorithmes portant sur les arbres via des schémas)

9.3 Bilan de Axel

Ce projet m'a permis d'améliorer mon autonomie et mon efficacité dans un contexte de travail de groupe, surtout sur un sujet qui m'était jusqu'alors moyennement familier. Ce projet d'Ada et l'utilisation intensive de Git m'ont rendu plus efficace tant sur les points de conception logicielle que de communication des problèmes et résultats.

Le langage Ada étant assez différent du C/C++, Python de part son typage fort, mais aussi de sa structure en général il demande une grande rigueur que j'ai pu acquérir grâce à cet ouvrage.

J'ai passé 2 heures environ à faire les raffinages. j'ai passé en tout environ 22 heures sur le routeur simple, le routeur LL et le routeur LA, tests compris, et 1h sur la rédaction du rapport et du manuel.