Projet PIM : Stockage et exploitation de tables de routage

Auchère, Charles, Pisenti, Trichard

Département Sciences du Numérique - Première année
2022-2023

Sommaire

Table des matières

1 Introduction

Un routeur est un élément d'un réseau qui permet de transmettre les paquets reçus sur une interface d'entrée vers la bonne interface de sortie selon les informations qui sont stockées dans sa table de routage. Dans cette optique, notre premier objectif a été d'implanter un routeur dit simple, c'est-à-dire un routeur qui conserve toutes les routes dans une structure de donnée, nous permettant ainsi de retrouver l'interface de sortie à utiliser selon la destination. Par la suite, nous avons amélioré l'efficacité de notre routeur en utilisant un cache conservant des informations nécessaires dans la trouvaille de la bonne sortie. Pour cela, nous avons implanté des caches avec différentes politiques d'utilisations pour que le cache ne soit pas trop grand et pour un large choix d'utilisation.

Dans un premier temps, nous décrirons l'architecture de l'application. Ensuite, nous présenterons les principaux choix qui ont été réalisés ainsi que les principaux algorithmes et types de données. Nous décrirons par la suite notre démarche de test des programmes et nos principales difficultés et les solutions que nous avons trouvées pour les résoudre. Nous indiquerons également notre organisation d'équipe. Ce rapport sera conclu par un bilan technique du projet et un bilan personnel proposé par chacun des membres du groupe.

2 Résumé

Tout au long de ce projet, nous avons eu pour objectif de réaliser un routeur simple puis un routeur avec un cache. Le développement a été réalisé en respectant la méthode des raffinages pour une meilleure vision de la division du travail par la suite. Des tests unitaires ont été mis en place pour garantir le fonctionnement global. Enfin, à l'heure actuelle, nous avons un routeur qui fonctionne avec un cache pour les différentes politiques souhaitées.

3 Architecture en modules

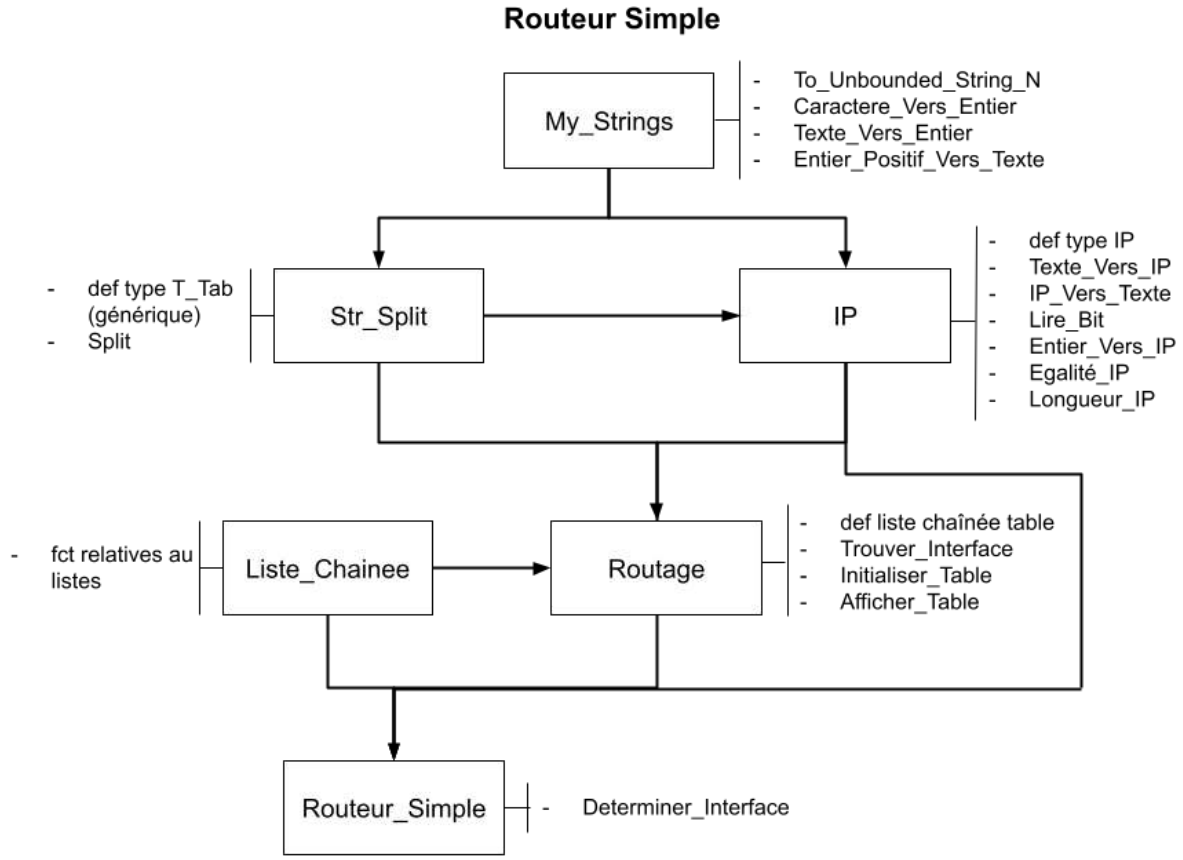

FIGURE 1 – Architecture en module du routeur simple

Pour ce routeur, nous avons créé plusieurs modules qui permettent d'effectuer des actions simples sur les types *strings*, *IP* et *Liste_Chainee*. Ainsi, nous avons juste à les appeler dans le module *Routage* qui est le module qui va trouver l'interface et afficher la table. Enfin, nous avons le module *Routeur_Simple* qui permet d'analyser la ligne de commande à l'aide des modules *Liste_Chainee* et *Routage*.

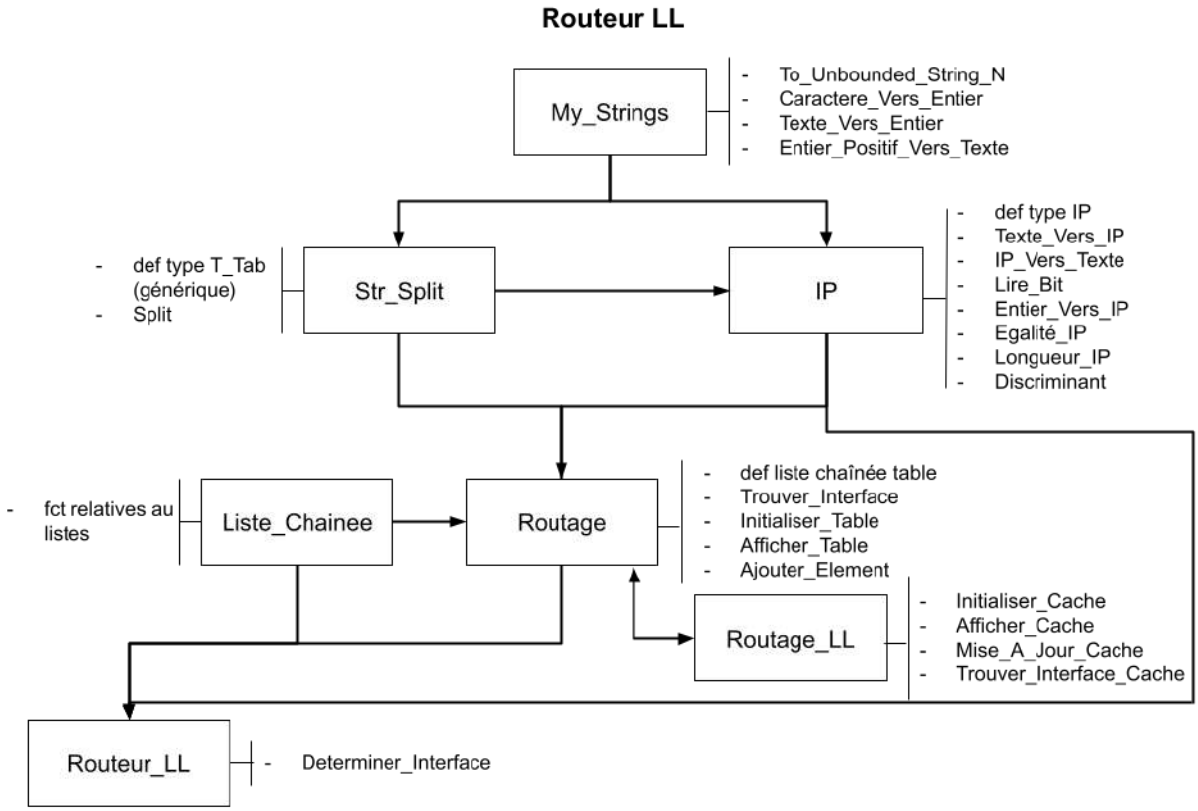

FIGURE 2 – Architecture en module du routeur avec un cache

Pour ce routeur, nous avons créé plusieurs modules qui permettent d'effectuer des actions simples sur les types *strings*, *IP* et *Liste_Chainee*. Ainsi, nous avons juste à les appeler dans le module *Routage* qui est le module qui va trouver l'interface, afficher la table, mais aussi gérer le cache à l'aide du module *Routage_LL*. Enfin, nous avons le module *Routeur_LL* qui permet d'analyser la ligne de commande à l'aide des modules *Liste_Chainee* et *Routage_LL*.

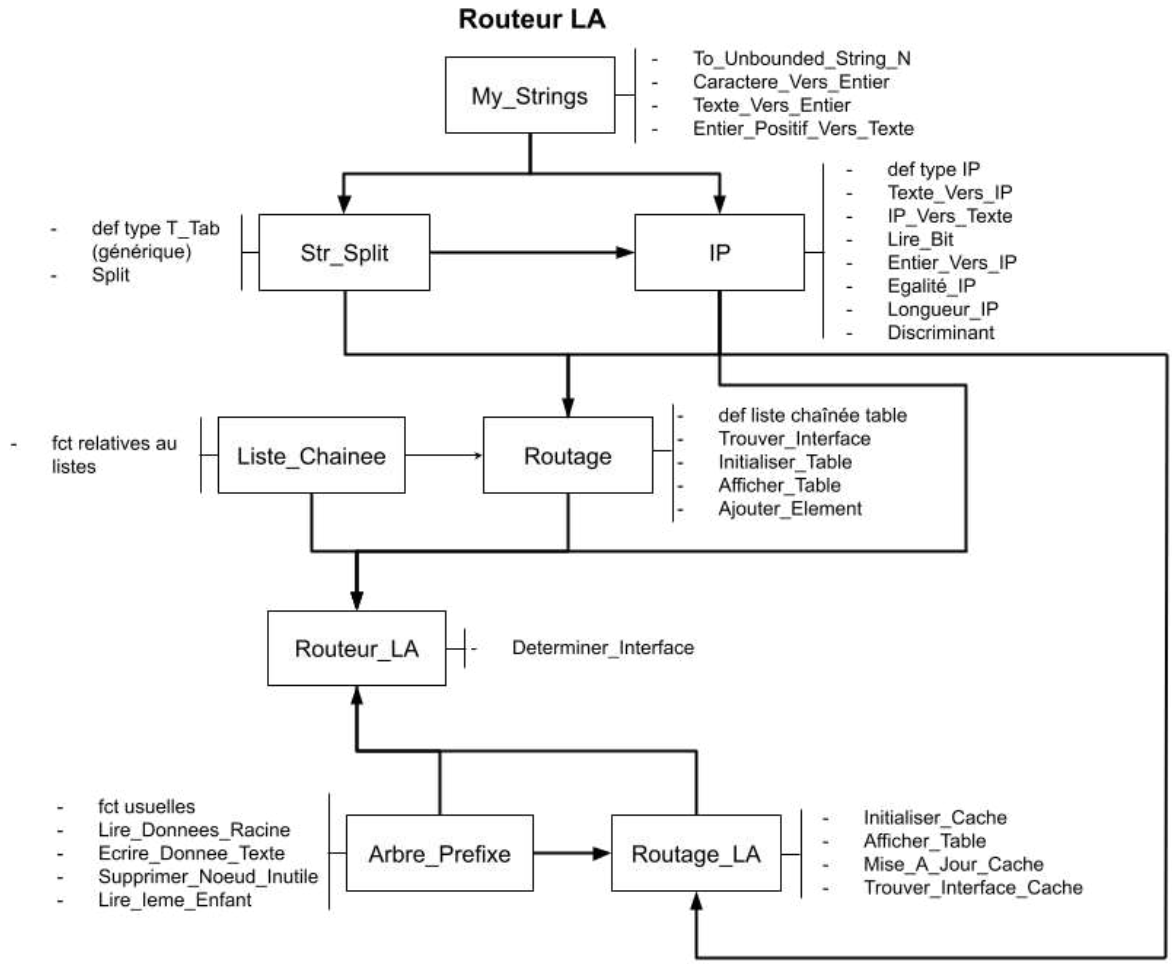

FIGURE 3 – Architecture en module du routeur LA

Pour ce routeur, nous avons créé plusieurs modules qui permettent d'effectuer des actions simples sur les types *strings*, *IP* et *Liste_Chainee*. Ainsi, nous avons juste à les appeler dans le module *Routage* qui est le module qui va trouver l'interface et afficher la table. De plus, le module *Routage_LA* avec l'aide du module *Arbre_Prefixe* qui permet de s'occuper des arbres du cache, permet de gérer le cache du routeur. Enfin, nous avons le module *Routeur_LL* qui permet d'analyser la ligne de commande à l'aide des modules *Liste_Chainee* et *Routage_LA*.

4 Principaux choix réalisés

Système indice dans l'arbre

Afin de savoir quel élément est : le plus ancien, le moins récemment utilisé ou le moins fréquemment utilisé, on a décidé de rajouter un paramètre *Id* de type *Natural* tel que, peu importe la politique utilisée, l'élément à supprimer en cas de dépassement de la capacité du cache soit celui avec la valeur de *Id* minimale.

— FIFO : Id contient le numéro de la i-ème insertion à laquelle la route a été ajoutée au cache

— LRU : Id contient le numéro de la i-ème utilisation réussie du cache pour laquelle la route a été utilisé

— LFU : Id contient la fréquence d'utilisation de la route

Garder une fréquence pour tout type de cache dans le cas LL

Lors de la création du type *T_Cache*, 2 solutions se sont offertes à nous. La première consistait à réutiliser le type *T_Table* pour les politiques FIFO et LFU et utiliser un type *T_Cache* distinct avec la fréquence en plus dans *T_Element*, ou bien d'ajouter la fréquence pour toute politique. Le premier choix permet de sauver de la RAM lors de l'exécution du programme, puisque hors LFU chaque cellule contient un entier de moins. Néanmoins, ce gain est minime et nous avons estimé qu'étant donné qu'un cache a une capacité faible (afin d'être efficace), l'écart de RAM engendré par le second choix est négligeable sur la facilité d'implémentation, lisibilité et compréhensibilité du code.

Limite théorique des indices de l'arbre assez élevée, mais peut être améliorée

Dans la même lignée que le choix précédent, nous avons choisi d'affecter à chaque route enregistrée dans l'arbre un indice qui lui est propre. Cela implique que le routeur finira par atteindre la limite théorique des indices, leur type étant *Natural*, cette limite est de 2^{16}. Puisque cela prendra plus d'une année à atteindre par un routeur recevant 100 requêtes par secondes une fois le cache plein. Nous avons décidé que ce problème était négligeable. Une solution possible pour régler ce problème est de "remettre à 0" les indices à partir d'une certaine limite. C'est-à-dire qu'au-delà d'un seuil fixé (indice $> 2^{15}$ par exemple), un parcours de l'arbre serait lancé de telle sorte que le plus petit indice soit remit à 1 et ainsi de suite en conservant l'ordre déjà établi des indices.

Choix du cache : liste chainée ou tableau

Nous avions envisagé de multiples implémentations possibles pour le cache LL.

— une liste doublement chaînée : algorithmes plus performants pour le tri du cache (pas besoin de reparcourir depuis le début à chaque fois). Algorithmes beaucoup

plus difficiles à écrire (on doit mettre à jour jusqu'à 4 pointeurs dans le bon ordre lors de l'utilisation).

— une liste simplement chaînée avec stockage de l'indice du dernier élément (un FIFO) : algorithme en temps constant pour ajouter/supprimer des éléments dans le cas de la politique FIFO. Pas d'avantage notable pour les autres politiques (LRU, LFU).

— un tableau : cela est possible puisque le cache a une capacité maximale connue. Algorithmes plus simples (pas besoin de gérer dynamiquement la mémoire) mais moins performants (même si on conserve une classe de complexité linéaire).

— une liste simplement chaînée : choix retenu, car imposé par le sujet, et permet un bon équilibre entre complexité temporelle, spatiale et pour écrire le code.

Choix *private* ou *limited*

— Pour les structures chaînées, on a choisi de les mettre en type "limited private" afin de s'assurer de garder une cohérence des listes, du cache et d'éviter des erreurs de la part de l'utilisateur. Cependant, cela nous impose d'écrire beaucoup plus de procédures et de remplacer beaucoup de fonctions par des procédures (puisqu'on ne peut plus faire d'affectations), ce qui est plus lourd à manipuler. Le choix limited nous a également imposé d'écrire beaucoup de procédures / fonctions qui prennent en paramètre des procédures ou fonctions génériques afin de pouvoir réaliser certaines actions n'ayant pas été pensées par le développeur originel (voir module Arbres_Prefixes).

— Pour le type T_IP, on a choisi de mettre le type en "private" et non "limited private" car on considère qu'il est pertinent de pouvoir faire des affectations et des comparaisons d'adresses IP dans le code des utilisateurs. (le type T_IP étant relativement simple, et ne contenant pas d'allocation dynamique par exemple qui pourraient permettre à l'utilisateur de faire des choses incohérentes contrairement à précédemment par exemple.)

Choix de l'arbre préfixe avec plusieurs fils et versatile

Nous avons décidé d'implémenter un module Arbre_Prefixe générique pour la gestion du cache LA. Ce module est capable de gérer tout type d'arbres préfixes avec une Arité/Nombre de Fils arbitraire. Il peut être utilisé pour modéliser/reconnaître des langages avec des alphabets génériques (un mot étant considéré terminal s'il mène jusqu'à une feuille de l'arbre). Pour parvenir à ses fins, ce module prend comme paramètre de généricité une fonction Lire_Préfixe donnée par l'utilisateur qui permet d'associer à chaque symbole de l'alphabet considéré, un unique entier de 1 à Nombre_Prefixes.

Dans le cadre de ce projet, nous devons considérer le langage des adresses IP constitué de l'alphabet "0,1". Nous allons donc l'instancier avec $Nombre\_Prefixes = 2$ (arbre binaire) et "$Lire\_Prefixe(IP, Position) = Lire\_Bit(IP, Position) + 1$"

Afin de convenir à un nombre maximal de cas d'usages, la majorité des procédures du module prennent des procédures génériques "Post_Traitement" en paramètre de

généricité (et doivent donc être instanciées) qui permet à chaque fois d'appliquer un Post_Traitement aux différents nœuds visités par la procédure en question (après que celle-ci s'est terminée, et dans l'ordre de bas en haut). Cela nous sera utile pour maintenir des relations entre les nœuds et leurs fils après chaque modification.

Ce module, bien que suffisant pour notre projet, reste cependant non exhaustif des fonctions sur les Arbres_Prefixes. Et peut-être étoffé de nouvelles procédures lors d'utilisation dans de futurs projets.

Choix de plusieurs politiques pour le cache LA

Nous avons également pris le temps dans ce projet d'ajouter au module Routage_LA la gestion des trois types de cache (LFU, LRU, FIFO) bien que seul la politique LRU soit exigée par le sujet. Nous avons également écrit une fonction Trouver_Interface_Table() qui est inutilisée dans ce projet, mais qui aurait pu servir pour trouver l'interface dans notre arbre binaire, si celui-ci était utilisé directement comme table de routage plutôt que comme un cache. La différence majeure étant que le cache est plus strict sur les masques (en raison de la gestion de la cohérence du cache), tandis que pour une table de routage, on est obligé de parcourir la totalité de l'arbre afin de s'assurer de choisir le masque le plus long en cas de plusieurs interfaces possibles. (On constate d'ailleurs que dans ce cas les arbres binaires n'apportent pas d'avantage pour la complexité temporelle. C'est pour cela qu'ils sont utilisés uniquement pour le cache dans ce projet).

Choix des adresses IP cherchées

Nous avons décidé de mettre dans le cache directement les adresses IP que l'on a routé, plutôt que de masquer celles-ci pour qu'elles ne se finissent pas des "…0.0" comme montré dans le sujet.
Nous avons effectivement jugé cela non utile puisque les bits restants ne sont, de toute manière, jamais regardés par l'ensemble des fonctions du programme.

5 Principaux algorithmes et types de données

5.1 Principaux algorithmes

Routeur_Simple.adb

Determiner_Interface : traite les instructions de gestion du routeur de la ligne de commande.

Routage.adb

function Trouver_Interface(Table_Routage : in T_Table ; IP : in T_IP) return Unbounded_String : trouve l'interface correspondant à une IP dans la table de routage
procedure Initialiser_Table_Vide(Table_Routage : out T_Table) : initialise la table de routage à *Null*
procedure Initialiser_Table(Table_Routage : out T_Table ; Fichier : in File_Type) : initialise la table de routage avec le fichier dédié
procedure Afficher_Table(Table_Routage : in T_Table) : affiche tous les éléments de la table de routage
function Est_Vide(Table_Routage : in T_Table) return Boolean : renvoie si la table est vide ou non

Pour le routeur cache, on ajoute :

function Determiner_Masque_Cache (Table : in T_Table ; IP_A_Router : in T_IP) return T_IP : détermine le masque à mettre dans le cache

Liste_Chainee.adb

Fonctions relatives aux listes chainées. (*Initialiser*, *Vider*, *Ajouter*, *Supprimer*...)

IP.adb

function Texte_Vers_IP(Chaine : in Unbounded_String) return T_IP : convertit une chaine du type 192.168.0.1 en adresse IP
function IP_Vers_Texte(IP : in T_IP) return Unbounded_String : convertit une adresse IP en une chaine du type "192.168.0.1"
function Lire_Bit(IP : in T_IP ; i : in Integer) return Integer : renvoie la valeur du i-ème bit (en partant de la gauche)
function Entier_Vers_IP(n : in Integer) return T_IP : convertit un entier en adresse IP (on ne peut pas représenter toutes les adresses IP par des entiers, car ils sont trop petits)
function Egalite_IP(IP1 : in T_IP ; IP2 : in T_IP ; Masque : in T_IP) return Boolean : teste l'égalité entre 2 adresse IP pour un masque donné
function Longueur_IP(IP : in T_IP) return Integer : renvoie la longueur max de 1 consécutif dans IP en partant de la gauche
function Discriminant(IP1 : in T_IP ; IP2 : in T_IP) return T_IP : renvoie le masque de

longueur minimale qui discrimine IP1 et IP2

Str_Split.adb

procedure Split(Tableau : out T_TAB ; Chaine : in Unbounded_String ; Caractere : in Character) : permet de couper une chaine sur le caractère c et remplit un tableau de *NbrArgs* éléments. Si on a plusieurs fois c d'affilée dans la chaine, on ne coupe qu'une seule fois

My_Strings.adb

function To_Unbounded_String_N(Chaine : in String ; n : in Integer) return Unbounded_String : créée un Unbounded_String avec les n premiers caractères de chaine
function Caractere_Vers_Entier(c : in Character) return Integer : convertit un caractère en chiffre
function Texte_Vers_Entier(Texte : in String) : convertit une string en un entier, les espaces sont ignorés et le signe de l'entier est respecté
function Eniter_Positif_Vers_Texte(n : in Integer) : convertit un entier positif en texte (sans rajouter d'espace avant ou après le nombre)

Routeur_LL.adb

Determiner_Interface : traite les instructions de gestion du routeur de la ligne de commande.

Routage_LL.adb

procedure Initialiser_Table_Vide (Table_Routage : out T_Table) : initialise le cache à *Null*
procedure Vider_Table (Table_Routage : in out T_Table) : vide le cache
procedure Trouver_Interface_Cache (Interface_Nom : out Unbounded_String ; Table_Routage : in out T_Table ; IP : in T_IP ; Politique_Cache : in Unbounded_String) : trouve l'interface correspondant à IP dans la table de routage
procedure Afficher_Cache(Cache : in T_Table ; Politique_Cache : in Unbounded_String) : affiche tous les éléments du cache
function Est_Vide (Table_Routage : in T_Table) return Boolean : renvoie si le cache est vide ou non
procedure Mise_A_Jour_Cache(Cache : in out T_Table ; IP_A_Router : in T_IP ; Capacite_Cache : in Integer ; Politique_Cache : in Unbounded_String ; Taille_Cache_Actuelle : in Integer ; Interface_Nom : in Unbounded_String ; Table : in Routage.T_Table) : met à jour le cache en accord avec la politique et la cohérence du cache

Routeur_LA.adb

Determiner_Interface : traite les instructions de gestion du routeur de la ligne de com-

mande.

Routage_LA.adb

procedure Initialiser_Table_Vide (Table_Routage : out T_Table) : initialise le cache à *Null*
procedure Vider_Table (Table_Routage : in out T_Table) : vide le cache
procedure Trouver_Interface_Cache (Interface_Nom : out Unbounded_String ; Table_Routage : in out T_Table ; IP : in T_IP ; Politique_Cache : in Unbounded_String) : trouve l'interface correspondant à une IP dans la table de routage. Cette fonction ne fonctionne que dans le cas où la Table_Routage est en réalité un cache LA
procedure Afficher_Cache(Cache : in T_Table ; Politique_Cache : in Unbounded_String) : affiche tous les éléments du cache
function Est_Vide (Table_Routage : in T_Table) return Boolean : renvoie si le cache est vide ou non
procedure Mise_A_Jour_Cache(Cache : in out T_Table ; IP_A_Router : in T_IP ; Capacite_Cache : in Integer ; Politique_Cache : in Unbounded_String ; Taille_Cache_Actuelle : in Integer ; Interface_Nom : in Unbounded_String ; Table : in Routage.T_Table) : met à jour le cache en accord avec la politique et la cohérence du cache

Arbre_Prefixe.adb

procedure Initialiser(Arbre : out T_Trie) : initialise l'arbre à *Null*
procedure Vider(Arbre : in out T_Trie) : vide l'arbre
function Est_Vide(Arbre : in T_Trie) return Boolean : renvoie si l'arbre est vide ou non
function Est_Feuille(Arbre : in T_Trie) return Boolean : renvoie si l'élément est une feuille
procedure Ajouter(Arbre : in outT_Trie ; Cle : in T_Cle ; Element : in T_Element) : Ajouter un élément dans l'arbre que l'on peut retrouver en suivant successivement le chemin (regarder récursivement dans les bons fils) associé aux lettres de la clé donné par la fonction Lire_Préfixe
function Lire_Donnee_Racine(Arbre : in T_Trie) return T_Element : lit la donnée contenue dans la racine de l'arbre passé en paramètre
function Lire_Cle_Racine(Arbre : in T_Trie) return T_Cle : lit la clé associée à la donnée contenue dans la racine de l'arbre passé en paramètre
procedure Ecrire_Donnee_Tete(Arbre : in out T_Trie ; Donnee : in T_Element) : modifie la donnée contenue dans la racine de l'arbre passé en paramètre
function Lire_Ieme_Enfant(Arbre : in T_Trie ; i : in Natural) return T_Trie : Obtient une copie du i-ème enfant de l'arbre courant
procedure Supprimer_Selection(Arbre : in out T_Trie avec :
generic procedure Post_Traitement(Arbre : in out T_Trie)
et generic function Selection(Arbre : in T_Trie) return Boolean : supprime toutes les

feuilles de l'arbre F telles que Selection(F) = True. La fonction générique "Selection" renvoie True si le nœud courant est un parent d'au moins un nœud à supprimer. Elle renvoie également True si le nœud courant est une feuille à supprimer de l'arbre. False dans les autres cas. La fonction Post_Traitement permet d'appliquer un post traitement sur tous les nœuds visités après la suppression des éléments à supprimer. Et dans l'ordre inverse dans lequel ils ont été visités.
procedure Parcours_Profondeur_Post(Arbre : in T_Trie) avec :
Traiter(Arbre : in T_Trie) : applique un Traitement à tous les nœuds de l'arbre, en suivant l'ordre d'un parcours en profondeur
procedure Chercher_Et_Verifier_Post(Element_Trouve : out T_Element ; Arbre : in out T_Trie ; Cle : in T_Cle) : cherche une clé dans l'arbre en suivant la méthode habituelle des arbres préfixes (À chaque profondeur. on choisit le fils correspondant au caractère actuel de la clé renvoyé par la fonction Lire_Prefixe). Quand on arrive sur la feuille finale, on regarde si le mot trouvé correspond aux attentes

5.2 Principaux types de données

Pour le routeur simple, nous avons créé les types de données suivants :

type privé *T_IP* : le type de l'adresse IP définit comme *mod* $2 ** 32$

type Generique T_Table : la table de routage de notre routeur. C'est un pointeur sur *T_Cellule*.

type T_Cellule : c'est un enregistrement contenant les informations suivantes :

— *Adresse* : l'adresse IP souhaitée de type *T_IP* ;

— *Masque* : le masque obtenu à partir de l'IP de type T_IP

— *Interface_Nom* : nom donnée à l'interface résultant de cette adresse et de ce masque de type *Unbounded_String*

Pour le routeur avec un cache LL :

type Generique T_Cache : le cache de notre routeur. C'est un pointeur sur *T_Cellule_Cache*.

type T_Cellule_Cache : c'est un enregistrement contenant les informations suivantes :

— *Adresse* : l'adresse IP souhaitée de type *T_IP* ;

— *Masque* : le masque obtenu à partir de l'IP de type T_IP

— *Interface_Nom* :nom donnée à l'interface résultant de cette adresse et de ce masque de type *Unbounded_String*

— *Frequence* : fréquence associée à la cellule type *Integer*

Pour le routeur avec un cache LA :

type Generique T_Trie : le cache de notre routeur. C'est un pointeur sur *T_Noeud*.

type T_Noeud : c'est un enregistrement contenant les informations suivantes :

— *Cle* : la clé du nœud *T_Cle*

— *Element* : l'élément du nœud type *T_Cellule*

— *Enfants* : représente les fils du nœud, ayant pour type un tableau allant de 1 à *Nombre_Enfant_Max* de *T_Trie*

type T_Cellule : c'est un enregistrement contenant les informations suivantes :

— *Masque* : le masque obtenu à partir de l'IP de type *T_IP*

— *Interface_Nom* :nom donnée à l'interface résultant de cette adresse et de ce masque de type *Unbounded_String*

— *Id* : représente le fils le moins récemment utilisé type *Natural*

6 Démarche adoptée pour tester le programme

Dans un premier temps, nous avons créé des fichiers de test pour tous les modules que nous étions en train de coder. Ainsi, nous pouvions vérifier au fur et à mesure que nos programmes fonctionnaient correctement. Nous retrouvons ces fichiers de test sous le nom *test_* puis le nom du module testé. Dans chacun de ces fichiers, il y a une partie dédiée à chacune des fonctions implantées dans le module.

Dans un second temps, des tests unitaires ont été réalisés, notamment pour les routages, que ce soit le routeur simple ou le routeur avec un cache. Pour cela, nous avons écrit de nombreuses lignes de commandes et nous avons vérifié que le programme réagissait de la façon souhaitée.

7 Raffinages

Vous trouverez dans le dossier un fichier nommé *Raffinages.pdf*. Ce fichier contient nos raffinages ainsi que notre grille d'auto-évaluation.

8 Difficultés rencontrées et les solutions adoptées

Familiarisation avec Git

Un point difficile à aborder lors de ce projet fût l'apprentissage de l'utilisation de Git à plusieurs. Bien que la prise en main ait été finalement rapide, réussir à comprendre et gérer les conflits au départ n'a pas été évident. Ainsi, nous avons appris à réfléchir aux impacts des commandes, à la clarté des commits et à entretenir une sauvegarde externe.

Gestion d'un grand nombre de modules au sein du même projet

Ce projet se constitue en un nombre important de modules liés les uns aux autres. La création de ces modules fût une chose, mais les relier proprement et gérer les types privé et limité privé en fût une autre. En effet, cela nous a demandé des réflexions préalables à l'écriture de ces modules puisque les types définis dans les uns devaient ou non pouvoir être accessibles dans les autres.
De même, l'instanciation des différents modules et fonctions n'a pas toujours été des plus évidents.

Trouver rapidement l'élément à supprimer

Nous avons été soumis à un problème lors de l'implémentation des différentes politiques du cache LA. Il fallait pouvoir trouver rapidement l'élément d'indice minimal quand le cache est plein afin de le supprimer. Pour d'éviter de parcourir tout l'arbre (ce qui ferait perdre l'avantage de l'utilisation de structures d'arbres avec une complexité très élevée), nous avons décidé de conserver dans chaque nœud intermédiaire de l'arbre, le numéro "Id" le plus petit numéro de tous ces nœuds enfants.
Ainsi, on connaîtra directement la valeur de l'Id minimal en regardant celui de la racine de l'arbre, et on pourra chercher uniquement dans les branches qui conservent cet Id minimal plutôt que de tout parcourir.

9 Organisation de l'équipe

Quoi ?		Qui ?			
Modules	**Sous-programmes**	**Spécification**	**Codage**	**Test**	**Relecture**
Routeur Simple	Determiner_Interface	I	NA	M	M
Routage	Trouver_Interface	M	NA	M	I
	Initialiser_Table	NA	M	I	M
	Ajouter_Element	M	I	TP	NA
	Afficher_Table	I	M	I	NA
IP	Texte_Vers_IP	TP	TP	NA	I
	IP_Vers_Texte	TP	TP	I	NA
	Lire_Bitit	TP	TP	NA	I
	Entier_Vers_IP	TP	NA	I	TP
	Longueur_IP	I	M	NA	I
	Egalité_IP	TP	NA	I	TP
	Discriminant	I	NA	TP	M
Str_Plit	Split	I	TP	NA	M
My_Strings	To_Unbounded_String_N	M	I	M	NA
	Caractere_Vers_Entier	M	I	NA	TP
	Texte_Vers_Entier	NA	I	TP	M
	Entier_Positif_Vers_Texte	NA	M	TP	TP
Liste Chainees	Fonctions des listes	NA	I	M	TP
Routeur Cache	Determiner_Interface	I	NA	M	TP
Routage LL	Initialiser_Cache_Vide	I	NA	M	TP
	Trouver_Interface_Cache	NA	I	M	TP
	Afficher_Cache	NA	I	TP	M
	Mise_A_Jour_Cache	I	NA	TP	M
Routeur LA	Determiner_Interface	M	TP	I	NA
Arbre_Prefixe	Fonctions usuelles	M	TP	I	NA
	Lire_Donnee_Racine	M	TP	I	NA
	Ecrire_Donnee_Tete	TP	M	I	NA
	Supprimer_Noeud_Inutile (private)	TP	M	NA	I
	Lire_Ieme_Enfant	TP	M	NA	I
Routage LA	Initialiser_Cache	M	TP	NA	I
	Trouver_Interface_Cache	M	TP	NA	I
	Afficher_Table	TP	M	NA	I
	Mise_A_Jour_Cache	TP	M	NA	I

10 Bilan technique

Au moment de ce rendu, nous avons implémenté le routeur simple, mais aussi le routeur avec un cache. Les deux types de cache ont étés implémentés, le routeur LL qui a pour cache une liste chainée et le routeur LA qui a pour cache un arbre préfixe.

Comme il l'était subtilement pointé dans le sujet, nous avons opté pour une implémentation du masque le plus précis dans le cache. Autrement que d'aller au bout de l'idée de la cohérence de cache, ce choix permet un routage "plus efficace" par le fait que le cache est entièrement discriminant. C'est-à-dire qu'il contient le plus de routes possibles différentes. C'est ainsi qu'il est efficace, permettant d'éviter au maximum le parcours peu efficace de la table de routage. De plus, à l'heure actuelle, le cache LL est trié pour avoir affichage plus clair à la suite de son changement.

Enfin, comme expliqué dans la rubrique des choix effectués, nous avons une limite théorique des indices de l'arbre. Elle reste malgré tout assez élevée, mais pourrait être améliorée comme relevé précédemment. C'est une partie sujette a être implantée comme évolution du projet.

11 Bilan personnel

11.1 Nathan Auchère

Temps total : $\sim 30h$

— conception : $\sim 10h$

— implémentation : $\sim 8h$

— mise au point : $\sim 10h$

— rapport : $\sim 2h$

Ce projet a pour moi été intéressant sur plusieurs points. Tout d'abord, le travail d'équipe nécessaire pour le mener à bien a été une redécouverte avec l'utilisation de Git et les discussions techniques ayant eu lieu au cours du projet. En effet, les choix que le groupe a été mené à prendre ont dû être débattus et considérés. Cette expérience m'a permis d'acquérir en confiance et en esprit critique, d'écouter les arguments des uns et des autres pour se faire son propre avis. Ensuite, de manière plus anecdotique, j'ai apprécié comprendre et recoder le principe de fonctionnement d'un routeur, une machine présente au quotidien. Enfin, réaliser petit à petit le projet, assembler chaque module avec chaque module et voir le résultat compiler et fonctionner a été un vrai plaisir.

11.2 Ines Charles

Temps total : $\sim 30h$

— conception : $\sim 6h$

— implémentation : $\sim 7h$

— mise au point : $\sim 10h$

— rapport et autres livrables :$\sim 7h$

J'ai trouvé ce projet très intéressant. Tout d'abord, le travail d'équipe m'a permis de mieux appréhender certaines notions et d'avoir des dialogues avec d'autres pour réaliser les choix les plus pertinents. De plus, la découverte de l'utilisation de Git et ses spécificités m'a beaucoup apporté pour le futur. Ce projet m'a appris à être à l'écoute de mes camarades pour essayer d'être le plus efficace dans le travail, mais surtout de pouvoir écouter les points de vue de chacun et de se mettre d'accord sur la bonne démarche à suivre. Ensuite, j'ai trouvé le sujet plutôt intéressant, car il était en lien direct avec notre quotidien. Enfin, voir le bon fonctionnement du projet est un vrai plaisir.

11.3 Téo Pisenti

Temps total : $\sim 40h$

— conception : $\sim 15h$

— implémentation : $\sim 15h$

— tests : $\sim 4h$

— rapport : $\sim 1h$

— relectures et corrections d'erreurs : $\sim 5h$

Ayant déjà mené des projets, celui-ci demeure toutefois le plus long d'entre eux (environ 3000 lignes, tout compris). La nouveauté fut également pour moi le travail en équipe et la découverte de Git que je n'avais jamais eu l'utilité avant. J'ai beaucoup œuvré à l'élaboration de l'architecture du projet, au choix des structures utilisées, aux spécification et écriture des modules primaires (indépendant de tous les autres modules) afin d'offrir à mes camarades des outils pratiques pour la suite du projet. La difficulté fut notamment de choisir les types d'accès ("limited", "private") les plus stricts, prévenir les erreurs de manipulation des données. Ce qui nécessite d'écrire des procédures suffisamment versatiles pour ne pas engendrer de blocage dans la suite du projet sans influer sur l'efficacité des algorithmes finaux. Le tout en ayant des modules aux rôles clairement identifiés (exemple : pas de fonctions de routage dans le module Arbres_Prefixes). Je pense être parvenu aujourd'hui à un compromis qui me satisfait, et c'est aussi pourquoi je suis fier de notre travail à tous les quatre. J'espère que vous parviendrez à apprécier le temps et les efforts que nous avons consacrés dans ce projet.

11.4 Matthieu Trichard

Temps total : $\sim 25h$

— conception :$\sim 5h$

— implémentation : $\sim 4h$

— tests : $\sim 15h$

— rapport : $\sim 1h$

Pour ma part, ce projet a été très intéressant, car il traite d'un système qui est omniprésent dans notre monde moderne. Il m'a permis de mieux le comprendre et d'appréhender son fonctionnement en profondeur. De plus, j'ai pu comprendre de nouvelle méthode de programmation grâce à mes partenaires. De plus, ce projet en groupe m'a permis d'apprendre à utiliser git pour les projets de code collaboratifs. Il m'a aussi permis d'apprendre à partager des idées à un groupe, mais aussi à débattre des différentes propositions.