Projet – **IN407** – **Gestion de flux de données**

Desmares Loïc - Desfontaines Alexia

Ce projet a pour vocation de développer une application capable d'analyser et comparer diverses stratégies de gestion de flux de données dans un réseau modélisé par des *Buffers*. L'enjeux est donc de déterminer quelle stratégie est la plus optimale pour limiter le temps d'attente et la perte de paquets.

L'architecture de notre projet repose sur la programmation orientée objet en python et la philosophie de parallélisme des processus (sur laquelle nous reviendrons plus tard).

Ainsi nous avons implémenter une première classe nommé ***Paquet*** afin de modéliser les données manipulées dans le réseau de communication. La classe **Paquet** dispose d'une instance de classe comptant le nombre de paquets générés modifié uniquement par le constructeur de la classe, cette valeur sert à différencier chaque paquet et à faciliter l'analyse du taux de perte de paquets. De plus la classe est munie de mutateurs et d'accesseurs permettant de manipuler les données temporelles de chaque objet de la classe. Ces mêmes accesseurs sont utilisés dans la méthode « Calcule_attentede() » permettant de calculer le temps d'attente d'un objet de la classe entre sa source (temps émis) et sa destination (temps arrivé). Une dernière méthode « Réinitialiser() » permet de réinitialiser à 0 l'instance de la classe (« nombre_paquets ») et cette dernière méthode est une classmethod, ce qui permet de l'appeler sans disposer d'un objet de la classe.

Une deuxième classe nommé ***Buffer*** permet de modéliser les files d'attentes dans lesquelles les paquets sont stockés en attendant d'être transmis. La classe dispose de trois instances de classes : « nombre_buffers », « Capacité » et « liste_buffers » permettant respectivement de compter le nombre de buffers, stocker la capacité maximale de paquets qu'un buffer peut contenir et la liste de tous les buffers. Le buffer en lui-même est modélisé par une liste nommé « liste_attente ». La classe possède aussi les attributs d'instance : « predecesseur », « successeur » et « capacite_locale » qui comme leurs noms l'évoquent permettent respectivement de gérer la liste des buffers predecesseurs/successeurs et la capacité locale du buffer (donc de la « liste_attente »). Tous les attributs sont munis d'accesseurs et de mutateurs, parmi eux la méthode « setListe_attente() » permet de manipuler la liste d'attente du buffer et les diverses opérations qui lui sont rattachées à l'aide d'un opcode (sous forme de 'str'). Enfin la classe ***Buffer*** possède deux méthodes clés, la première « Insertion() » permet d'ajouter un paquet au buffer à condition qu'il ne soit pas déjà plein. La seconde méthode « Transmission() » prend en entrée un nombre modélisant le débit du buffer et permet à celui-

ci de transmettre le plus vieux paquet qu'il stocke à son successeur. Et de nouveau une classmethod « Réinitialiser() » permet de réinitialiser les valeurs initiales des instances de la classe.

Une troisième classe ***Source*** hérité de la classe ***Buffer*** permet de modéliser les sources générant les paquets. L'héritage de la classe ***Buffer*** permet à chaque objet de la classe ***Source*** de disposer de son propre buffer et d'ainsi directement stocker les paquets générés. La classe dispose de deux attributs de classes : « nombre_sources » et « liste_sources » qui est notamment utilisée pour itérer sur toutes les sources créées (chaque source étant ajoutée à la liste par le constructeur de la classe). De plus la classe dispose d'un attribut d'instance : « numéro » permettant de modéliser l'identifiant de la source. Cet attribut est muni d'un accesseur et d'un mutateur. Enfin la classe est munie d'une méthode « Generateur_paquet() », prenant en entrée un nombre permettant d'appliquer un processus de poisson au temps d'attente entre la génération de deux paquets. Cette même méthode insère le paquet créé dans le buffer de la source dont il est issu. Et de nouveau une classmethod « Réinitialiser() » permet de réinitialiser les valeurs initiales des instances de la classe. Bien sûr la classe ***Source*** dispose des méthodes de la classe ***Buffer*** grâce à l'héritage.

Une quatrième classe ***Stratégie*** permet d'encapsuler l'exécution des diverses stratégies de gestion de flux de données tout en permettant de modifier les paramètres de la simulation. Les objets de la classe prennent donc en entrée tous les paramètres pouvant être modifiés : « numéro » ('int' qui correspond à l'opcode de la stratégie), « nombre_source », « échantillon » (détermine le nombre de paquets que le buffer Destination doit recevoir, une fois ce nombre atteint la simulation prend fin) et « parametre_poisson » (un nombre permettant de faire varier la loi de poisson). La méthode principale de la classe « Update() » utilise des Threads (permettant le parallélisme de l'exécution de plusieurs processus), afin de lancer la génération de paquets par les diverses sources et la transmission d'un paquet du buffer principal vers le buffer Destinataire. La notion de parallélisme des tâches dans leur exécution permet d'optimiser le temps d'exécution des diverses fonctions (notamment la génération de paquets et son « time.sleep() ») et de permettre une modélisation plus fidèle du réseau de communication. La méthode « Update() » est utilisée dans une boucle directement dans le constructeur de la classe. Enfin la classe est munie de deux autres méthodes, la première « Analyse_Temps() » renvoie le temps moyen d'attente des paquets contenus dans le buffer Destination. Et la dernière méthode « Analyse_Taux() » renvoie le taux de perte de paquets.

Enfin, nous avons créé une dernière classe : la classe **Interface()**. Pour la réaliser, nous avons utilisé le module CustomTkinter, qui est en réalité un dérivé du module Tkinter mais qui permet de réaliser des interfaces graphiques plus jolies et avec plus d'options. La classe **Interface()** a pour fonction de gérer absolument tout l'aspect visuel de notre code. Dans le constructeur de la classe, se trouve en premier lieu les caractéristiques de bases tel que le titre et les dimensions de la fenêtre. Puis viennent tous les composants qui apparaissent au fur et à mesure de la démonstration et qui ne sont donc pas visibles dès le début. Nous commençons notre affichage avec une page d'accueil simple, contenant un bouton " Commencer !" qui lance l'exécution de la méthode "next_accueil()" quand on clique dessus. Cette dernière permet de changer le contenu de la page d'accueil en effaçant le bouton et le texte initial, pour laisser place à la page permettant à l'utilisateur de choisir la stratégie pour laquelle il veut voir la démonstration de fonctionnement. L'utilisateur a juste à entrer un chiffre entre 1 et 3 inclus et appuyer sur le bouton "Lancer la démonstration". Si toutefois, l'utilisateur fournit une entrée invalide comme un chiffre supérieur à 3 ou bien une chaine de caractères, aucune démonstration n'est lancée et un message d'erreur s'affiche demandant à l'utilisateur de réessayer. Cela est rendue possible grâce à la méthode "next_explications()". Viennent ensuite les méthodes "demo_strat1()", "demo_strat2()" et "demo_strat3()" qui suivent toutes trois le même fonctionnement, en faisant appel aux méthodes "same_place()" et "deplacer()". La méthode "same_place()" sert à marquer une pause dans le mouvement de chaque paquet, une fois arrivé à mi-écran. Quant à la méthode "deplacer()", elle permet aux paquets de se déplacer horizontalement de 40 pixels à la fois. Par ailleurs il y a sur la page d'explication, un second bouton "Comparer les 3 stratégies", qui lance la méthode "next_analyses()" qui affiche principalement les frames de la page d'analyses, qui elle-même appelle la méthode "recup_data()". Cette dernière méthode lance 3 tests avec des stratégies différentes, stocke et affiche les résultats d'analyses pour permettre une comparaison sur l'efficacité de chaque stratégie. Et pour finir il existe une méthode "quit_app()", qui comme son nom l'indique, permet de quitter l'interface graphique.

Annexe : Code

En ligne : https://github.com/LugolBis/Projet-IN407/tree/main