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() » la classe pour 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.