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>

En texte :

**### Projet IN407 import time import random import threading import customtkinter class Paquet: nombre\_paquets = 0 def \_\_init\_\_(self,source=0): Paquet.nombre\_paquets += 1 # On compte au fur et à mesure le nombre de paquet créés poids = 1.5 # On initialise par défaut le poids à 1,5 étant donné qu'il n'est pas obligatoire de s'en préoccuper self.valeur = Paquet.nombre\_paquets # Chaque paquet est différencié par sa valeur, qui est simplement le n° du n-ième paquet self.source = source self.poids = poids temps\_émission = time.time() # On initialise le temps auquel le paquet est généré self.\_temps\_émision = temps\_émission temps\_arrivé = None # On initialise le temps d'arrivé a sa destination du paquet self.\_temps\_arrivé = temps\_arrivé def getTemps\_émission(self): return self.\_temps\_émision def setTemps\_émission(self,nouveau\_temps=None): """Il est fortement conseillé de passer 'time.time()' en argument de 'nouveau\_temps'.""" self.\_temps\_émision = nouveau\_temps def getTemps\_arrivé(self): return self.\_temps\_arrivé def setTemps\_arrivé(self,nouveau\_temps=None): """Il est fortement conseillé de passer 'time.time()' en argument de 'nouveau\_temps'.""" self.\_temps\_arrivé = nouveau\_temps temps\_émision = property(getTemps\_émission, setTemps\_émission) temps\_arrivé = property(getTemps\_arrivé, setTemps\_arrivé) def Calcule\_attente(self): """Cette méthode renvoie le temps d'attente du paquet entre son émission et son arrivée dans à Destination.""" return self.getTemps\_arrivé() - self.getTemps\_émission() @classmethod def Réinitialiser(cls): Paquet.nombre\_paquets = 0 def \_\_repr\_\_(self): return str(self.valeur) # Chaque paquet est simplement représenté par sa valeur class Buffer: nombre\_buffers = 0 # Cette variable de classe permet de compter le nombre de Buffer Capacité = 100 # Cette variable de classe initialise la capacité maximale d'un Buffer (donc 100 paquet pour l'heure) liste\_buffers = [] def \_\_init\_\_(self,successeur=None): # L'idée derrière prédescesseur et successeur est de lier les buffer entre eux, mais aussi à leur Source notamment pour simplifier la transmission des paquets # Remarque : un buffer peut avoir plusieurs prédecesseurs Buffer.nombre\_buffers += 1 # On incrémente la variable de classe comptant le nombre de buffer dès la création d'un nouveau buffer Buffer.liste\_buffers.append(self) # On ajoute à la variable de classe le Buffer en lui même predecesseur=[] # On peut ici stocker les prédecesseurs du Buffer, ce qui pourra servir plus tard pour l'interface graphique self.\_predecesseur = predecesseur liste\_attente = [] # On initialise le coeur du Buffer sous la forme d'une liste, les paquets transmis au Buffer seront stockés ici self.\_liste\_attente = liste\_attente capacite\_locale = 0 # On initialise la capacité locale du Buffer self.\_capacite\_locale = capacite\_locale self.\_successeur = successeur # Bien sûr on initialise une méthode permetant d'accéder au successeur du Buffer def getPredecesseur(self): return self.\_predecesseur def setPredecesseur(self,nouvelle\_valeur=None): if nouvelle\_valeur != None: self.\_predecesseur = nouvelle\_valeur def getListe\_attente(self): return self.\_liste\_attente def setListe\_attente(self,element): """L'élément en entrée doit être un tuple (OPCODE,Variable) tel que : - l'OPCODE indique l'opération à effectuer : ['ECRASE','AJOUT','DEPOP'] - la Variable est l'objet en lui même \n 'ECRASE' -> écrase la liste par la liste en entrée \n 'AJOUT' -> ajoute l'objet à la liste \n 'DEPOP' -> on renvoie l'élément que l'on à dépop de la liste à l'indice de la Variable""" OPCODE = element[0] Variable = element[1] if OPCODE == "ECRASE": self.\_liste\_attente = Variable elif OPCODE == "AJOUT" : self.\_liste\_attente.append(Variable) elif OPCODE == "DEPOP": return self.\_liste\_attente.pop(Variable) def getCapacite\_locale(self): return self.\_capacite\_locale def setCapacite\_locale(self,ajout=0): self.\_capacite\_locale += ajout def getSuccesseur(self): return self.\_successeur def setSuccesseur(self,buffer:'Buffer'): assert isinstance(buffer,[Buffer,list]), "Le successeur d'un buffer doit être un Buffer ou une liste." self.\_successeur = buffer predecesseur = property(getPredecesseur, setPredecesseur) liste\_attente = property(getListe\_attente,setListe\_attente) capacite\_locale = property(getCapacite\_locale,setCapacite\_locale) successeur = property(getSuccesseur,setSuccesseur) def Insertion(self,paquet:'Paquet'): """ Cette méthode permet d'insérer un Paquet dans le buffer """ if self.capacite\_locale** < Buffer.Capacité: # On s'assure ici que le buffer n'est pas déjà plein #self.liste\_attente.append(paquet) self.setListe\_attente(('AJOUT',paquet)) self.setCapacite\_locale(ajout=1) def Transmission(self,débit=0): """ Cette méthode permet de transmettre un paquet au buffer successeur du buffer avec lequel cette méthode est appelé \n En résumé : bufferA [Paquet1,Paquet2,...,PaquetN] et bufferB [ ] -> **bufferA [Paquet2,...,PaquetN] et bufferB [Paquet1] """ time.sleep(débit) # On fait attendre le Buffer pour simuler son débit if isinstance(self.getSuccesseur(), Buffer) and (self.getCapacite\_locale()>0): # On s'assure que le successeur est bien un objet de type 'Buffer' et que le buffer ""source"" n'est pas vide self.getSuccesseur().Insertion(paquet=self.setListe\_attente(('DEPOP',0))) # On insère dans le buffer successeur le premier paquet du buffer self.setCapacite\_locale(ajout = -1) @classmethod def Réinitialiser(cls): Buffer.nombre\_buffers = 0 # Cette variable de classe permet de compter le nombre de Buffer Buffer.Capacité = 100 # Cette variable de classe initialise la capacité maximale d'un Buffer (donc 100 paquet pour l'heure) Buffer.liste\_buffers = [] class Source(Buffer): nombre\_sources = 0 liste\_sources = [] # On stock dans une variable de la classe tous les objets 'Source' def \_\_init\_\_(self,successeur=None): # successeur est le buffer principal super().\_\_init\_\_(successeur) # Héritage -> chaque source à son propre Buffer intégré Source.nombre\_sources += 1 numéro = Source.nombre\_sources # On initialise le numéro de la source Source.liste\_sources.append(self) # On ajoute à la variable de classe la Source en elle même self.\_numéro = numéro def getNuméro(self): return self.\_numéro def setNuméro(self,nouveau\_numéro=0): self.\_numéro = nouveau\_numéro numéro = property(getNuméro,setNuméro) def Generateur\_paquet(self,lambda\_poisson=0.5): """ Générateur de paquet""" assert isinstance(self.getSuccesseur(), Buffer), f"La source n°{self.getNuméro()} n'a pas de successeur valide." temps\_delta = random.expovariate(1/lambda\_poisson) # On choisit le délait d'attente avant de générer un nouveau paquet time.sleep(temps\_delta) paquet = Paquet(source=self.getNuméro()) # On génère un paquet print(f"paquet n°{paquet} provenant de {self.getNuméro()} -- temps d'attente : {temps\_delta}") self.Insertion(paquet) # On insère le paquet directement dans le Buffer rataché à la source def AfficheTest(self): """ Cette fonction est temporaire elle ne sert qu'à afficher l'évolution de la Source pour faciliter les tests sur la classe """ print(f"Buffer Source {self.getNuméro()} : {self.getListe\_attente()}") @classmethod def Réinitialiser(cls): super().Réinitialiser() Source.nombre\_sources = 0 Source.liste\_sources = [] class Stratégie: # Cette classe à pour but d'encapsuler les stratégies de gestion de flux de données def Update(self,Buffer\_Principal,file\_attente,indice\_max): liste\_threads = [] # On initialise la liste des threads qui vont être éxécutés durant la boucle liste\_threads.append(threading.Thread(target=Buffer\_Principal.Transmission(), args=(1,5))) liste\_threads[0].start() # On démarre la transmission du Buffer Principal for source\_ in Source.liste\_sources : liste\_threads.append(threading.Thread(target=source\_.Generateur\_paquet(), args=(self.parametre\_poisson,))) # On génère de nouveaux paquets pour chaque source for thread in liste\_threads[1:]: thread.start() for thread in liste\_threads[1:]: thread.join() if self.numéro == 1: source\_Transmission = None # On initialise la source dont on va transmettre un paquet à None capacite\_max = 0 # On initialise la capacité locale de la source transmission à 0 for source\_ in Source.liste\_sources : if source\_.getCapacite\_locale() > capacite\_max: # On choisit ici la source\_ dont la capacite locale est maximale capacite\_max = source\_.getCapacite\_locale() source\_Transmission = source\_ elif self.numéro == 2: source\_Transmission = file\_attente.pop(0) # On retire le premier élément de la file d'attente des sources file\_attente.append(source\_Transmission) # On lance le thread de tranmsission de la source choisit en amont vers le Buffer principal else : source\_Transmission = file\_attente[random.randint(0,indice\_max)] # On prend aléatoirement une source dans la file d'attente liste\_threads.append(threading.Thread(target=source\_Transmission.Transmission())) # On lance le thread de tranmsission de la source choisit en amont vers le Buffer principal liste\_threads[-1].start() liste\_threads[-1].join() liste\_threads[0].join() # Destination.Transmission() if self.Destination.getListe\_attente() != []: self.Destination.getListe\_attente()[-1].setTemps\_arrivé(time.time()) # Dès qu'un paquet arrive on stock son temps d'arrivé def \_\_init\_\_(self,numéro,nombre\_source=2,échantillon=20,parametre\_poisson=0.5): assert numéro in [1,2,3], "Il n'existe que 3 stratégie différente qui sont : [1,2,3]." assert isinstance(nombre\_source,int), "Le nombre de sources utilisées dans la simulation doit être un entier." # On réinitialise les instances des classes Paquet.Réinitialiser() Buffer.Réinitialiser() Source.Réinitialiser() Destination = Buffer() Destination.setCapacite\_locale(ajout=Buffer.Capacité - échantillon) # On initialise la capacité local du Buffer Destination pour décider de la taille de l'échantillon de paquets sur lequel nous allons baser nos analyses self.numéro = numéro self.parametre\_poisson = parametre\_poisson self.Destination = Destination DEBUT\_TEMPS\_TEST = time.time() # Initialisation des objets Buffer\_Principal = Buffer(Destination) # On initialise le Buffer principal for s in range(nombre\_source): # A chaque itération on crée un objet source, qui sera directement stocké dans 'Source.liste\_sources' lors de l'initialisation de ceux-ci Source(Buffer\_Principal) Buffer\_Principal.setPredecesseur(Source.liste\_sources) file\_attente = Source.liste\_sources # On initialise une file d'attente qui sera utilisée pour faire alterner le choix de la source par le Buffer principal indice\_max = len(file\_attente)-1 # On initialise l'indice max pouvant être tiré au hasard pour accéder à la file d'attente while Destination.getCapacite\_locale()** < Buffer.Capacité: self.Update(Buffer\_Principal,file\_attente,indice\_max) # Affichages de contôle pour s'assurer du bon fonctionnement du script print(f"Buffer Principal : {Buffer\_Principal.getListe\_attente()}") print(f"Buffer Destination : {Destination.getListe\_attente()}") for source\_ in Source.liste\_sources : source\_.AfficheTest() self.temps\_total = time.time() - DEBUT\_TEMPS\_TEST print(f"\nFin du test !\nLe test a duré : {self.temps\_total}") def Analyse\_Temps(self): """Cette méthode renvoie le temps moyen d'attente des paquets contenu dans le 'Buffer\_Destination' qui modélise le destinataire des paquets.""" if isinstance(self.Destination, Buffer): Contenu = self.Destination.getListe\_attente() # On récupère le Buffer\_destination créé par le constructeur temps\_attente = 0 for paquet\_ in Contenu : temps\_attente += paquet\_.Calcule\_attente() return round(temps\_attente/len(Contenu), 2) # On renvoie un float arrondi à la 2ème décimale, contenant le temps moyen d'attente des paquets def Analyse\_Taux(self): """Cette méthode calcule le taux de perte de paquets.""" nombre\_paquets\_générés = Paquet.nombre\_paquets # On récupère le nombre total de paquets générés nombre\_paquets\_stockés = 0 for buffer\_ in Buffer.liste\_buffers : nombre\_paquets\_stockés += len(buffer\_.getListe\_attente()) # On ajoute le nombre de paquets stockés dans chaque buffer résultat = round(nombre\_paquets\_stockés/nombre\_paquets\_générés, 2) # On renvoie un float arrondi à la 2ème décimale, contenant le taux de perte des paquets if résultat == 1: return 0.0 else: return résultat class Interface(customtkinter.CTk): ### Classe qui gère tout l'aspect visuel du code def \_\_init\_\_(self): super().\_\_init\_\_() ### On définit les caractéristiques de bases self.title("Projet 2024 : Stratégies de gestion de flux.py") largeur = 680 longueur = 1400 self.geometry(f"{longueur}x{largeur}") ### On s'occupe ici des éléments de la page d'acceuil self.frame\_accueil = customtkinter.CTkFrame(self, width=longueur, height= largeur) self.frame\_accueil.pack\_propagate(False) self.frame\_accueil.pack() self.label\_commencer = customtkinter.CTkLabel(self.frame\_accueil,font=("Helvetica", 40), text=" Projet 2024 : \n Stratégies de gestion de flux à l’entrée d’un un réseau decommunication.\n\n DESFONTAINES - DESMARES", anchor="center") self.label\_commencer.pack(pady = 120) self.bouton\_accueil = customtkinter.CTkButton(self.frame\_accueil, corner\_radius= 40, text= "Commencer !",anchor="center", command= self.next\_accueil, font=("Arial", 27), height= 30) self.bouton\_accueil.pack() #### Eléments présents sur la page intermerdiaire self.strat = customtkinter.CTkEntry(self.frame\_accueil, placeholder\_text="Entrer un numéro de stratégie") self.bouton\_demo = customtkinter.CTkButton(self.frame\_accueil, text= "Lancer la démonstration ", command= self.next\_explications, height= 30, width= 150, corner\_radius= 30, font= ("Arial", 20)) self.erreur = customtkinter.CTkLabel(self.frame\_accueil, text = "Saisie invalide. Ré-essayez !", font = ("Arial", 40), text\_color="red") self.bouton\_analyses = customtkinter.CTkButton(self.frame\_accueil, text= "Comparer les 3 stratégies", command= self.next\_analyses,font= ("Arial", 20), corner\_radius= 30, height= 30, width= 150) ### On s'occupe de la page où la démonstration se fait self.frame\_principal = customtkinter.CTkFrame(self, width=1000, height=400) self.frame\_principal.pack\_forget() self.frame\_principal.grid\_columnconfigure(0, weight=1) self.frame\_principal.grid\_columnconfigure(1, weight=1) self.frame\_principal.grid\_rowconfigure(0, weight=1) self.frame\_principal.grid\_rowconfigure(1, weight=1) ### On s'occupe des composants de la page d'analyses self.frame\_analyses = customtkinter.CTkFrame(self, width= longueur, height= largeur) self.frame\_analyses.grid\_propagate(False) self.frame\_test1 = customtkinter.CTkFrame(self.frame\_analyses, width = 400, height= 500) self.frame\_test2 = customtkinter.CTkFrame(self.frame\_analyses, width = 400, height= 500) self.frame\_test3 = customtkinter.CTkFrame(self.frame\_analyses, width = 400, height= 500) self.frame\_test1.grid\_propagate(False) self.frame\_test2.grid\_propagate(False) self.frame\_test3.grid\_propagate(False) self.label\_test1 = customtkinter.CTkLabel(self.frame\_test1, text="", bg\_color= "transparent", anchor = "center") self.label\_test2 = customtkinter.CTkLabel(self.frame\_test2, text="", bg\_color= "transparent", anchor = "center") self.label\_test3 = customtkinter.CTkLabel(self.frame\_test3, text="", bg\_color= "transparent", anchor = "center") self.bouton\_quitter = customtkinter.CTkButton(self.frame\_analyses, text= "Quitter", command= self.quit\_app, font= ("Arial", 20), corner\_radius= 30, height= 30, width= 150 ) ### On créée des boites pour chacun des buffers et pouvoir illustrer l'ajout, le retrait et la transmission d'un paquet mes\_buffers = [] buffer = Buffer.liste\_buffers[::-1] for k in range(4): ### 4 car on fait nos tests avec deux buffers sources + un buffer principal + un buffer destination ### On créée les "boites" à tour de rôle et on les positionnes de différentes manières, selon le buffer auquel correspond la "boite" self.frame\_source = customtkinter.CTkFrame(self.frame\_principal,width=100, height= 300, corner\_radius=10, fg\_color= "black") if k == 0 : self.frame\_source.grid(column = 0, row = 0, pady = 10, padx = 10, sticky = "nsew") self.frame\_principal.grid\_columnconfigure(0, weight=1, uniform="buffers") self.frame\_principal.grid\_rowconfigure(0, weight=1, uniform="buffers") elif k == 1: self.frame\_source.grid(column = 0, row = 1, pady = 10, padx = 10, sticky = "nsew") self.frame\_principal.grid\_columnconfigure(0, weight=1, uniform="buffers") self.frame\_principal.grid\_rowconfigure(1, weight=1, uniform="buffers") elif k == 2: self.frame\_source.grid(column = 1, row = 0, rowspan = 2, pady = 10, padx = 10, sticky = "nsew") self.frame\_principal.grid\_columnconfigure(1, weight=1, uniform="buffers") self.frame\_principal.grid\_rowconfigure(0, weight=1, uniform="buffers") self.frame\_principal.grid\_rowconfigure(1, weight=1, uniform="buffers") else : self.frame\_source.grid(column = 2,row = 0, rowspan = 2, pady = 10, padx = 10, sticky = "nsew") self.frame\_principal.grid\_columnconfigure(2, weight=1, uniform="buffers") self.frame\_principal.grid\_rowconfigure(0, weight=1, uniform="buffers") self.frame\_principal.grid\_rowconfigure(1, weight=1, uniform="buffers") mes\_buffers.append(self.frame\_source) ### Méthode qui récupère que le chiffre saisi par l'utilisateur, et lance en conséquence la démonstration correspondante def next\_explications(self): if not isinstance(self.strat,int): self.erreur.place( x = 440, y = 530) strategie = int(self.strat.get()) if strategie not in [1,2,3]: self.erreur.place( x = 440, y = 530) else: self.frame\_accueil.destroy() self.frame\_principal.pack\_propagate(False) self.frame\_principal.pack(pady = 20, fill = "both", expand = True) if strategie == 1: self.demo\_strat1() elif strategie == 2 : self.demo\_start2() else : self.demo\_strat3() def next\_accueil(self): ### Méthode qui permet d'afficher la page précédent la page d'accueil, la page où l'on demande à l'utilisateur quelle stratégie adopter. self.label\_commencer.configure(text = "Il vous est demandé ici de choisir une stratégie à adopter pour la démonstration.\n" " Vous avez 3 stratégies de gestion de la file d'attente possibles : \n" " – La file d’attente choisie est celle contenant le plus grand nombre de paquets.\n" " – Un paquet est pris de chaque file d’attente, à tour de rôle.\n" " – La file d’attente est choisie de manière aléatoire.\n" "Entrez 1, 2 ou 3 puis appuyez sur le bouton.", font=("Helvetica", 30)) self.bouton\_accueil.destroy() self.strat.place(x = 630, y = 380) self.bouton\_demo.pack() self.bouton\_analyses.place(x = 570, y = 500) ### Méthode qui permet de lancer 3 tests avec des stratégies différentes, récupérer les données et les afficher afin de permettre une comparaison def recup\_data(self): liste\_data = [] ### On lance 3 stratégies différentes, sur un même nombre d'échantillon pour pouvoir les comparer par la suite Strat1 = Stratégie(1,2,1,0.5) Strat2 = Stratégie(2,2,1,0.5) Strat3 = Stratégie(3,2,1,0.5) ### On stocke ici les données d'analyses obtenues, à la liste liste\_data liste\_data.append((Strat1.Analyse\_Temps(), Strat1.Analyse\_Taux(), Strat1.temps\_total)) liste\_data.append((Strat2.Analyse\_Temps(), Strat2.Analyse\_Taux(), Strat2.temps\_total)) liste\_data.append((Strat3.Analyse\_Temps(), Strat3.Analyse\_Taux(), Strat3.temps\_total)) ### On affiche les données self.label\_test1.configure(text = f"Résultats Strat 1 \n\n\n\n temps moyen d'attente : {round(liste\_data[0][0], 2)} s \n\n " f"taux de perte : {round(liste\_data[0][1],2)} \n\n durée du test :{round(liste\_data[0][2],2)} s", font = ("Arial", 20)) self.label\_test2.configure(text = f"Résultats Strat 2 \n\n\n\n temps moyen d'attente : {round(liste\_data[1][0], 2)} s \n\n " f"taux de perte : {round(liste\_data[1][1],2)} \n\n durée du test :{round(liste\_data[1][2],2)} s", font = ("Arial", 20)) self.label\_test3.configure(text = f"Résultats Strat 3 \n\n\n\n temps moyen d'attente : {round(liste\_data[2][0], 2)} s \n\n " f"taux de perte : {round(liste\_data[2][1],2)} \n\n durée du test :{round(liste\_data[2][2],2)} s", font = ("Arial", 20)) self.label\_test1.grid(padx = 55, pady = 150) self.label\_test2.grid(padx = 55, pady = 150) self.label\_test3.grid(padx = 55, pady = 150) def next\_analyses(self): ### Méthode qui permet de changer de page, et d'afficher celle qui présente les résultats d'analyses self.frame\_accueil.destroy() self.frame\_analyses.grid() self.frame\_test1.grid(column = 0, row = 0, padx = 40, pady = 20) self.frame\_test2.grid(column = 1, row = 0, padx = 20, pady = 20) self.frame\_test3.grid(column = 2, row = 0, padx = 40, pady = 20) self.bouton\_quitter.grid(column = 1, row = 1, pady = 20) self.recup\_data() def quit\_app(self): ### Méthode qui détruit complétement la fenetre d'interface graphique. self.destroy() def same\_place(self, paquet\_a\_deplacer, my\_x, x\_destination): ### Méthode qui permet juste de faire une pause dans l'animation du paquet\_a\_deplacer app.after(300, lambda : self.deplacer(paquet\_a\_deplacer, my\_x, x\_destination)) ### Cette ligne permet de relancer le déplacement du paquet\_a\_deplacer ### Méthode qui change la coordonnée x du paquet\_a\_deplacer, et le ré-affiche en fonction de sa nouvelle position. def deplacer(self, paquet\_a\_deplacer, my\_x, x\_destination): my\_x += 40 if my\_x < 740: ### Cela correspond au moment où le paquet se trouve entre sa position initiale et sa position intermediaire (dans le buffer principal) paquet\_a\_deplacer.place(x = my\_x) app.after(300, lambda : self.deplacer(paquet\_a\_deplacer, my\_x, x\_destination)) elif my\_x == 740: ### Cela correspond au moment où le paquet se trouve à sa position intermediaire (dans le buffer principal) app.after(200, lambda : self.same\_place(paquet\_a\_deplacer, my\_x, x\_destination)) elif 741 < my\_x < x\_destination: ### Cela correspond au moment où le paquet se trouve entre sa position intermediaire et sa position finale (dans le buffer Destination) paquet\_a\_deplacer.place(x = my\_x) app.after(300, lambda : self.deplacer(paquet\_a\_deplacer, my\_x, x\_destination)) ### Méthode qui lance l'animation de la stratégie 1 def demo\_strat1(self): paquet1\_source1 = customtkinter.CTkLabel(app, width= 30, height= 30, bg\_color="blue", text = "") paquet1\_source1.place(x = 200, y = 100) paquet2\_source1 = customtkinter.CTkLabel(app, width= 30, height= 30, bg\_color="blue", text = "") paquet2\_source1.place(x = 150, y = 100) paquet3\_source1 = customtkinter.CTkLabel(app, width= 30, height= 30, bg\_color="blue", text = "") paquet3\_source1.place(x =100, y = 100) paquet4\_source1 = customtkinter.CTkLabel(app, width= 30, height= 30, bg\_color="blue", text = "") paquet4\_source1.place(x = 150, y = 150) paquet5\_source1 = customtkinter.CTkLabel(app, width= 30, height= 30, bg\_color="blue", text = "") paquet5\_source1.place(x = 100, y = 150) paquet1\_source2 = customtkinter.CTkLabel(app, width= 30, height= 30, bg\_color="red", text = "") paquet1\_source2.place(x = 150, y = 400) paquet2\_source2 = customtkinter.CTkLabel(app, width= 30, height= 30, bg\_color="red", text = "") paquet2\_source2.place(x = 100, y = 400) self.deplacer(paquet1\_source1, 200, 1300) app.after(3500,lambda : self.deplacer(paquet2\_source1, 150, 1250)) app.after(7000,lambda : self.deplacer(paquet3\_source1, 100, 1200)) app.after(10500,lambda : self.deplacer(paquet4\_source1, 150, 1250)) app.after(14000,lambda : self.deplacer(paquet1\_source2, 150, 1250)) app.after(17500,lambda : self.deplacer(paquet5\_source1, 100, 1200)) app.after(21000,lambda : self.deplacer(paquet2\_source2, 100, 1200)) ### Méthode qui lance l'animation de la stratégie 2 def demo\_start2(self): paquet1\_source1 = customtkinter.CTkLabel(self.frame\_principal, width= 30, height= 30, bg\_color="blue", text = "") paquet1\_source1.place(x = 200, y = 100) paquet2\_source1 = customtkinter.CTkLabel(self.frame\_principal, width= 30, height= 30, bg\_color="blue", text = "") paquet2\_source1.place(x = 150, y = 100) paquet3\_source1 = customtkinter.CTkLabel(self.frame\_principal, width= 30, height= 30, bg\_color="blue", text = "") paquet3\_source1.place(x =100, y = 100) paquet1\_source2 = customtkinter.CTkLabel(self.frame\_principal, width= 30, height= 30, bg\_color="red", text = "") paquet1\_source2.place(x = 200, y = 400) paquet2\_source2 = customtkinter.CTkLabel(self.frame\_principal, width= 30, height= 30, bg\_color="red", text = "") paquet2\_source2.place(x = 150, y = 400) self.deplacer(paquet1\_source1, 200, 1300) app.after(3500,lambda : self.deplacer(paquet1\_source2, 200, 1300)) app.after(7000,lambda : self.deplacer(paquet2\_source1, 150, 1250)) app.after(10500,lambda : self.deplacer(paquet2\_source2, 150, 1250)) app.after(14000,lambda : self.deplacer(paquet3\_source1, 100, 1200)) ### Méthode qui lance l'animation de la stratégie 3 def demo\_strat3(self): paquet1\_source1 = customtkinter.CTkLabel(app, width= 30, height= 30, bg\_color="blue", text = "") paquet1\_source1.place(x = 200, y = 100) paquet2\_source1 = customtkinter.CTkLabel(app, width= 30, height= 30, bg\_color="blue", text = "") paquet2\_source1.place(x = 150, y = 100) paquet3\_source1 = customtkinter.CTkLabel(app, width= 30, height= 30, bg\_color="blue", text = "") paquet3\_source1.place(x =100, y = 100) paquet1\_source2 = customtkinter.CTkLabel(app, width= 30, height= 30, bg\_color="red", text = "") paquet1\_source2.place(x = 200, y = 400) paquet2\_source2 = customtkinter.CTkLabel(app, width= 30, height= 30, bg\_color="red", text = "") paquet2\_source2.place(x = 150, y = 400) paquet3\_source2 = customtkinter.CTkLabel(app, width= 30, height= 30, bg\_color="red", text = "") paquet3\_source2.place(x = 100, y = 400) self.deplacer(paquet1\_source2, 200, 1300) app.after(3500,lambda : self.deplacer(paquet2\_source2, 150, 1250)) app.after(7000,lambda : self.deplacer(paquet1\_source1, 200, 1300)) app.after(10500,lambda : self.deplacer(paquet3\_source2, 100, 1200)) app.after(14000,lambda : self.deplacer(paquet2\_source1, 150, 1250)) app.after(17500,lambda : self.deplacer(paquet3\_source1, 100, 1200)) if \_\_name\_\_ == "\_\_main\_\_": app = Interface() app.mainloop()