Auteurs: Théophile Wachtel et Paul Treutenaere

# Projet PPC - At the Crossroad

Nous avons fait le choix de réaliser notre projet en orienté objet : chaque process est un objet issu d'une classe qui lui est propre. Voici nos classes : **Lights**, **Server**, **Display**, **VehicleGen** et **Coordinator**. Chacune de ces classes héritent de la classe **multiprocessing.Process** : les objets ainsi créés sont des process. Notre projet s'articule autour d'un fichier **main.py** qui crée l'ensemble des process fils : **lights**, **display**, **coordinator**, **normal_vehicle_gen** and **priority_vehicle_gen**. Ainsi que l'ensemble des variables partagées entre les process (IPC). Nous avons aussi un fichier **utils.py** qui retrace l'ensemble de nos constantes et de nos fonctions utiles dans plusieurs process. Au sein de ce projet nous nous sommes concentrés sur la fonctionnalité et la simplicité avant tout. L'objectif premier était de créer un algorithme fonctionnel pour l'améliorer si possible. Notre rendu est donc minimaliste et efficace.

Tout d'abord nous avons choisi pour la shared memory d'implémenter 3 arrays partagés entre les process : **vehicle_gen**, **coordinator** & **lights**.
Ces 3 arrays sont :

- **traffic_light_states** : représente l'état des feux avec l'index 0,1,2,3 correspondant respectivement aux feux N,E,S,W. ex: [1,0,1,0]

- **priority_mode_array** : représente le nombre de véhicules prioritaires en attente. ex: [1,1,0,0]

- **priority_direction_array** : représente la source des véhicules prioritaires en attente, avec 0,1,2,3 correspondant respectivement à N,E,S,W. ex : [3,1,2,0]

l'utilisation d'arrays est nécessaire pour garder en mémoire les véhicules prioritaires dans le cas ou il y en aurait plusieurs en même temps. Dans notre code nous avons fixé la taille de ces arrays à 100 car nous supposons que le nombre de véhicules prioritaire en attente en même temps ne va probablement pas dépasser 100 mais il est tout à fait possible d'augmenter la taille des ces arrays dans le fichier **utils.py**.

la manipulation des arrays **priority_mode_array** et **priority_direction_array** est faite avec un multiprocessing.lock pour éviter tout conflit tandis que la manipulation de **traffic_light_states** est faite sans self.lock car le process **lights** est le seul à modifier cet array

Ce sont les fonctions **shift_array_add()** et **shift_array_remove()** qui permettent de manipuler ces array. (respectivement de décaler toutes les valeurs de l'array vers la droite et d'ajouter 'x' en premier élément ou de décaler toutes les valeurs vers la gauche et d'ajouter 'x' en dernier élément)

Voici une présentation du fonctionnement de nos 5 classes :

**VEHICLEGEN**

FONCTIONNEMENT : pour générer du trafic nous avons réalisé une classe de génération de véhicule avec un attribut pour définir si celle-ci génère des véhicules prioritaires ou non. Les véhicules sont générés sous la forme d'un dictionnaire aléatoirement :
{"dest":(R|S|L|U), "source":(N|E|S|W), "priority":(N|P)}
Les lettres représentent :
R : Right, S : Straight, L : Left, U : U-Turn.
N : North, E : East, S : South, W : West.
N : Normal, P : Priority.
Puis sont concaténées en string 3 character : source + dest + priority pour être envoyé dans la MessageQueue correspondant à leur attribut source parmis une des 4 MessageQueue disponible.
A la création d'un véhicule prioritaire, un signal (signal.SIGUSR1) est envoyé au process **lights**. On append 1 à l'array partagée **priority_direction_array**.

IMPLEMENTATION & TEST : nous avons vérifié si chaque véhicule était bien envoyé dans la bonne queue grâce à des prints. Nous avons aussi vérifié qu'à la création de chaque véhicule prioritaire un signal était bien envoyé au process **lights**. Nous avons aussi remarqué que parfois, même lorsque le process **priority_vehicle_gen** est lancé avant le process **lights**, alors si **priority_vehicle_gen** lui envoie un signal, notre programme crash. Nous avons donc implémenté une condition qui vérifie que **lights.pid** est lancé avant de continuer l'exécution de vehiclegen.

**LIGHTS**

FONCTIONNEMENT : pour gérer les feux, il faut vérifier en permanence si un véhicule prioritaire est en attente.
Si un véhicule prioritaire est en attente, alors elle change les feux dans la direction de ce véhicule prioritaire(ex: [1,0,1,0] -> [0,1,0,0]).
Sinon elle change la direction des feux (ex:[1,0,1,0] -> [0,1,0,1]) toutes les X secondes.
Indices des feux : 0 - North, 1 - East, 2 - South, 3 - West

IMPLEMENTATION & TEST : nous avons vérifié si les feux changeaient bien toutes les 5 secondes grâce à des prints. Nous avons aussi

vérifié que lorsqu'un signal du process **priority_vehicle_gen** est reçu, alors l'array **priority_mode_array** est modifié correctement et que le feu correspondant au véhicule prioritaire est mis en vert grâce à des prints comparés aux prints des données du véhicule prioritaire créé.

**COORDINATOR**

FONCTIONNEMENT : nous avons fait le choix de gérer les priorités aux feux (priorité à droite donc right > straight > left > U-turn). À chaque instant, on regarde les véhicules en attente dans les messageQueue dont les feux sont au vert (soit max 2 véhicules) pour déterminer quel véhicule, parmi ces deux là, devrait passer en premier.
Pour avoir accès aux véhicules en attente dans les message queues, nous utilisons la fonction **peek()** (renvoie la tête de la queue). Nous avons fait ce choix car nous affichons le contenu des queues. Ainsi il est intéressant de vider les éléments dedans un à un.

IMPLEMENTATION & TEST : nous avons mis en entrée les 4 messageQueue avec des données initiales choisies pour représenter différents cas de priorités. Comme à chaque itération, ce process ne fait passer qu'un seul véhicule, il est facile avec un print de savoir dans quelle ordre sont passées chaque véhicule et de voir si cet ordre correspond à celui que l'on a trouvé à la main.

**SERVER**

FONCTIONNEMENT : Server crée une connexion TCP client/server. Il envoie l'état des véhicules contenus dans chaque messageQueue ainsi que l'état des feux pour les envoyer via une socket au process display. Il forme les données des queues grâce à la fonction **formatqueues()**. Nous avons aussi implémenté une gestion de Keyboardinterrupt qui permet de libérer les sockets pour pouvoir refaire des tests à la suite sans devoir changer les sockets à chaque fois.

IMPLEMENTATION & TEST : nous avons envoyé notre string formaté et nous vérifions si le code client (dans display) reçoit bien les mêmes données grâce à un string. Nous devions aussi faire attention au nombre de bits envoyés par le process server pour s'assurer que l'entièreté du message était reçue par le client. Nous avons donc choisi une taille fixe pour le string formatée. (en réalité cette valeure pourrait changer mais nous ne la changeons pas par sécurité)

**DISPLAY**

FONCTIONNEMENT : Pour afficher graphiquement le contenu des messageQueue, nous avons réalisé une interface graphique avec pygame. C'est la fonction **parse_message()** qui prend en paramètre le

string envoyé (un string formaté) par le process server et qui la transforme en un dictionnaire. Nous avons besoin de cette fonction car nous ne pouvons pas envoyer de dictionnaire dans une queue. Il permet d'afficher les trois premiers véhicules de chaque file.

IMPLEMENTATION & TEST : dans le fichier **main.py**, avant de lancer le process display, nous vérifions que les process server, **normal_vehicle_gen** et **priority_vehicle_gen** sont déjà en route car sinon le process **display** plante : il ne pourrait alors rien afficher et ne pourrait pas réaliser de connection TCP.

**PROBLÈMES RENCONTRÉS :**

Nous avons rencontré un problème : **priority_direction_array** et **priority_mode_array** prennent une valeur arbitraire dès le lancement du process. Cette valeur indique qu'un véhicule prioritaire a été créé or coordinator n'en était pas informé via le signal. Nous avons essayé de régler ce problème via l'utilisation de timers : sans succès. La cause de cette erreur nous est inconnue, nous avons donc "artificiellement" enlevé l'élément de l'array au début du lancement de **priority_vehicle_gen** pour corriger manuellement ce bug.