PPC : Rapport de Simulation d'Intersection

Table des matières

I. Introduction :....2
II. Conception et Choix Techniques :....2
III. Architecture et Protocoles d'Échange :....2
III.1. Gestion des Directions de Circulation....2
III.2. Gestion des Feux de Circulation....2
Communication Inter-processus....3
III.3. Rétablissement du Trafic Normal....3
III.4. Affichage et Gestion des Données....3
IV. Algorithmes Importants (Pseudo-code) :....3
IV.1. Gestion des Feux de Circulation (`TrafficLights`)....3
IV.2. Génération du trafic normal (`NormalTrafficGen`)....4
IV.3. Génération du trafic prioritaire (`PriorityTrafficGen`)....4
IV.4. Gestion du temps (`Timemanipulator`)....4
IV.5. Coordination des Véhicules (`Coordinator`)....4
IV.6. Déplacement des Véhicules (`move_vehicle`)....4
IV.7. Envoi des états de coordination à Display avec Socket (`send_updates_to_display`)....5
IV.8. Réception des états de coordination avec Socket (`receive_from_coordinator`)....5
V. Plan d'Implantation et Tests :....5
V.1. Nous avons suivi un plan d'implémentation progressif :....5
V.2. Tests effectués :....5
VI. Problèmes Rencontrés et Solutions :....5
VI.1. Synchronisation des processus....5
VI.2. Gestion des feux de circulation....6
VI.3. Blocage des files de messages....6
VII. Exécution du Programme :....6
VII.1. Démarrage de la simulation....6
VIII. Conclusion :....6
IX. Annexe :....6

I. Introduction :

La gestion du trafic à une intersection est un défi complexe qui nécessite une coordination précise entre les feux de circulation et les véhicules en mouvement. Ce projet a pour objectif de simuler une intersection régulée par des feux de circulation, en tenant compte du trafic normal et des véhicules prioritaires.

La simulation est réalisée en Python à l'aide de la bibliothèque multiprocessing pour gérer plusieurs processus en parallèle. Elle permet d'observer le comportement des véhicules et des feux en fonction des règles établies, offrant ainsi un environnement de test et d'optimisation pour la gestion du trafic.

II. Conception et Choix Techniques :

Nous avons décidé, pour ce projet de simulation d'intersection, d'afficher la simulation dans le terminal tout en laissant les commandes à l'utilisateur. Nous voulions que l'utilisateur puisse décider de stopper ou non l'exécution de la simulation et qu'il puisse également ajuster la vitesse de simulation. Pour cela, tous les processus participant à la simulation ont besoin d'être synchronisés entre eux.

C'est pourquoi nous avons implémenté un objet spécifique permettant de gérer ces différents états et de manipuler le déroulement de la simulation. À cela, nous avons ajouté des événements permettant de communiquer aux autres processus l'état du processus actuel et d'indiquer s'il a terminé l'exécution de son code ou non.

L'ensemble du code est organisé sous la forme d'un module Python, ce qui permet une meilleure modularité et offre la possibilité d'étendre la simulation à un cadre plus large si nécessaire. Un élément central de cette approche est la classe `Timemanipulator`, qui sert de classe abstraite pour des entités telles que `Coordinator` et `TrafficLights`. Elle impose une méthode `next()` qui doit être implémentée pour avancer dans l'état de la simulation.

III. Architecture et Protocoles d'Échange :

III.1. Gestion des Directions de Circulation

Le projet utilise la classe `Direction`, qui définit les quatre directions possibles à l'intersection : **Nord, Est, Sud et Ouest**. Chaque véhicule est associé à une direction d'entrée et une direction de sortie. La classe fournit également des méthodes permettant de déterminer la direction à droite ou à gauche d'une direction donnée, ce qui est essentiel pour la gestion des trajectoires dans l'intersection.

III.2. Gestion des Feux de Circulation

Le processus `TrafficLights` est responsable du fonctionnement des feux de circulation. Il fonctionne en deux modes :

- **Mode normal** : Les feux opposés partagent le même état (Nord-Sud et Est-Ouest alternent entre vert et rouge).
- **Mode prioritaire** : Lorsqu'un véhicule prioritaire est détecté, un seul feu passe au vert pour permettre son passage.

Le processus utilise des **verrous (`multiprocessing.Lock`)** pour garantir l'accès exclusif aux états des feux et éviter les conflits entre processus.

Les événements (`multiprocessing.Event`) permettent de synchroniser le `TrafficLights` avec le `Coordinator`, s'assurant que les feux sont mis à jour avant que de nouvelles décisions de circulation soient prises.

Communication Inter-processus

Le processus de génération des véhicules prioritaires ajoute, dans une file disponible en mémoire partagée du processus `Lights`, la source du véhicule prioritaire généré et envoie ensuite un signal à ce même processus de type **SIGUSR1**. Le processus de génération de trafic veille à obtenir au préalable un verrou sur l'objet en mémoire partagée. Une fois que le processus `Lights` reçoit ce signal, il tente de vider la file contenant les sources des différents véhicules prioritaires envoyés et modifie la couleur des feux pour ne garder qu'un seul feu vert, permettant ainsi le passage du véhicule prioritaire.

Cette implémentation permet de générer plusieurs véhicules prioritaires en même temps grâce à l'existence de la file, ce qui assure un passage fluide des véhicules un par un.

III.3. Rétablissement du Trafic Normal

Le coordinateur vérifie si le véhicule passant dans l'intersection est un véhicule prioritaire ou non. Si c'est le cas, il envoie un signal au processus `Lights`, cette fois-ci de type **SIGUSR2**, indiquant ainsi au processus de passer au véhicule suivant dans la file ou de rétablir le cycle normal des feux.

Le coordinateur est informé de l'état des feux via une variable en mémoire partagée. En fonction du nombre de feux allumés en vert, il sait s'il y a un véhicule prioritaire ou non et décide si les véhicules peuvent circuler librement.

Le coordinateur utilise les files de messages **SysV** pour communiquer avec les générateurs de trafic. Chaque direction de l'intersection possède une file de messages indépendante, permettant au coordinateur d'accéder aux véhicules en attente et de réguler leur passage en fonction de l'état des feux.

III.4. Affichage et Gestion des Données

Le processus `display`, permettant l'affichage de l'état actuel de la simulation, récupère les informations concernant :

- Les véhicules présents sur les quatre sections de route
- L'état des feux de circulation

Pour ce faire, des **sockets** sont utilisées pour établir une communication en temps réel avec les processus `Coordinator`.

L'envoi des véhicules s'effectue via des **files de messages SysV**, une par section de route. Le coordinateur consulte et récupère les informations des véhicules générés. Chaque véhicule est représenté par la classe `Vehicle`, qui définit son type (`normal` ou `priority`), ainsi que sa source et sa destination.

IV. Algorithmes Importants (Pseudo-code) :

IV.1. Gestion des Feux de Circulation (`TrafficLights`)

```
Tant que la simulation est active:
    Si un véhicule prioritaire est détecté:
        Activer le mode prioritaire
        Modifier les feux pour n'avoir qu'un feu vert
        Attendre le passage du véhicule prioritaire
        Revenir au mode normal
    Sinon:
        Alterner entre Nord-Sud et Est-Ouest selon le cycle normal
        Attendre un temps défini avant de basculer à l'autre état
```

IV.2. Génération du trafic normal (`NormalTrafficGen`)

```
Tant que la simulation est active:
    Acquérir le verrou pour l'accès à la file de messages
    Si la file de messages contient moins de véhicules que le seuil
maximal:
        Générer un véhicule normal avec une source et une destination
aléatoires
        Envoyer le véhicule dans la file de messages correspondante
        Déclencher un événement indiquant un nouvel ajout
    Sinon:
        Attendre que la file ait de l'espace
    Libérer le verrou
    Attendre un temps aléatoire avant de générer un nouveau véhicule
```

IV.3. Génération du trafic prioritaire (`PriorityTrafficGen`)

```
Tant que la simulation est active:
    Acquérir le verrou pour l'accès à la file de messages
    Si la file de messages contient moins de véhicules que le seuil
maximal:
        Générer un véhicule prioritaire avec une source aléatoire
        Envoyer le véhicule dans la file de messages correspondante
        Déclencher un signal SIGUSR1 au processus `Lights`
        Déclencher un événement indiquant un nouvel ajout
    Sinon:
        Attendre que la file ait de l'espace
    Libérer le verrou
    Attendre un temps aléatoire avant de générer un nouveau véhicule
prioritaire
```

IV.4. Gestion du temps (`Timemanipulator`)

```
Définir une classe abstraite `Timemanipulator` :
    Méthode `next(unit: int = 1)`:
        Déplacer la simulation à l'étape suivante
        (À implémenter dans les classes enfants comme `Coordinator` et
`TrafficLights`)
```

IV.5. Coordination des Véhicules (`Coordinator`)

```
Tant que la simulation est active:
    Vérifier les files de messages de chaque direction
    Si un véhicule est présent et que le feu est vert pour sa direction:
        Ajouter le véhicule à la liste des véhicules en attente
    Si un véhicule prioritaire est détecté:
        DéclencherEnvoyer un signal SIGUSR1 au processus `Lights`
        Ajuster les feux pour permettre son passage
    Identifier les routes avec feux verts et autoriser le passage des
véhicules selon les règles de priorité
    Vérifier si un feu doit repasser en mode normal
    Attendre un court instant avant de refaire le cycle
```

IV.6. Déplacement des Véhicules (`move_vehicle`)

```
Attendre la mise à jour de l’état des feux
Récupérer les directions avec feux verts
Si un seul feu est vert:
    Laisser passer le premier véhicule de la file correspondante
    Si le véhicule est prioritaire, envoyer SIGUSR2 au processus `Lights`
Si deux feux sont verts simultanément:
    Vérifier si les véhicules ont des trajectoires conflictuelles
    Si non, permettre le passage des deux véhicules
```

```
    Sinon, appliquer une règle de priorité aléatoire pour laisser passer un
véhicule en premier
```

IV.7. Envoi des états de coordination à Display avec Socket (`send_updates_to_display`)

```
Créer un socket pour la communication
Essayer de connecter à Display sur un port et un hôte
Boucle infini :
    Envoyer les véhicules avec ses directions à l'aide du socket
    Envoyer les états des feux avec ses directions à l'aide du socket
    Interrompre la connexion si KeyboardInterrupt dectecté
```

IV.8. Réception des états de coordination avec Socket (`receive_from_coordinator`)

```
Créer un socket pour écouter les connexions entrantes
Essayer d'écouter les connexions via le port et l'hôte
Boucle infini :
    Recevoir les données du Coordinator
    Décoder les données en véhicule et en états des feux
    Enregistrer les données décodées
    Interrompre la connexion si KeyboardInterrupt dectecté
```

V. Plan d'Implantation et Tests :

V.1. Nous avons suivi un plan d'implémentation progressif :

1. **Développement indépendant de chaque processus** (tests unitaires)
2. **Implémentation des protocoles d'échange**
3. **Connexion des processus et tests d'intégration**
4. **Développement d'un script global de simulation**

V.2. Tests effectués :

Tests unitaires sur `Vehicle.py` : Vérification de la génération correcte des véhicules, en s'assurant que les sources et destinations attribuées respectent les contraintes définies dans la classe `Vehicle`. Cela inclut des tests pour gérer les erreurs lorsqu'une source et une destination identiques sont attribuées.

Tests d'intégration sur `NormalTrafficGen.py` et `PriorityTrafficGen.py`: Validation du fonctionnement des files de messages en vérifiant que les véhicules sont bien ajoutés aux files correspondantes et que les générateurs respectent les règles de congestion, notamment en bloquant l'ajout de nouveaux véhicules lorsque la file est pleine.

Tests sur `Coordinator.py` : Vérification de la gestion des feux de circulation et du passage des véhicules. On s'assure que les véhicules normaux ne traversent que lorsque le feu est vert et que les véhicules prioritaires déclenchent correctement un changement de feu en envoyant le signal approprié au processus des feux.

VI. Problèmes Rencontrés et Solutions :

VI.1. Synchronisation des processus

Solution : L'utilisation de verrous (`multiprocessing.Lock`) et d'événements (`multiprocessing.Event`) a permis d'assurer une exécution ordonnée des processus impliqués

dans la simulation. Ces mécanismes garantissent que les différentes entités ne modifient pas simultanément des ressources partagées, évitant ainsi des conditions de course et des incohérences dans les données. L'ajout des événements a également facilité la coordination entre les processus, permettant de signaler lorsqu'un événement spécifique, comme la mise à jour d'un feu de circulation, s'est produit.

VI.2. Gestion des feux de circulation

Solution : Initialement, plusieurs feux pouvaient être allumés en vert simultanément, créant des conflits potentiels dans l'intersection. Pour corriger ce problème, un mécanisme a été mis en place afin de garantir qu'un seul feu soit actif en mode priorité, tout en respectant les règles de circulation normales en temps normal. Une mémoire partagée stocke l'état des feux et permet au coordinateur de s'assurer que la transition entre les états se fait correctement. Cela évite tout risque de collision et garantit que les véhicules prioritaires puissent circuler sans perturber le reste du trafic.

VI.3. Blocage des files de messages

Solution : Les files de messages peuvent rapidement se remplir si le taux de génération des véhicules dépasse la capacité de traitement du coordinateur. Pour éviter ce blocage, un mécanisme de régulation du trafic a été mis en place. Lorsque la file atteint un seuil critique, les générateurs de véhicules sont temporairement mis en pause en attendant que des véhicules quittent la file. Cela garantit que la simulation continue de fonctionner sans congestion excessive et sans perte de messages.

VII. Exécution du Programme :

VII.1. Démarrage de la simulation

Le programme s'exécute en lançant les processus en parallèle :

1. **Démarrage des feux de circulation (`TrafficLights`)**
2. **Lancement du générateur de trafic normal (`NormalTrafficGen`)**
3. **Lancement du générateur de véhicules prioritaires (`PriorityTrafficGen`)**
4. **Lancement du coordinateur (`Coordinator`)**
5. **Affichage de la simulation en temps réel (`Display`)**

À tout moment, l'utilisateur peut **stopper la simulation** ou **ajuster la vitesse** en interagissant avec le terminal.

VIII. Conclusion :

Ce projet a permis d'expérimenter la gestion des **processus concurrents**, la **communication inter-processus**, et la **synchronisation des événements** en Python. Grâce à une approche modulaire et bien structurée, nous avons obtenu une simulation fluide et extensible pour la gestion du trafic à une intersection.

IX. Annexe :

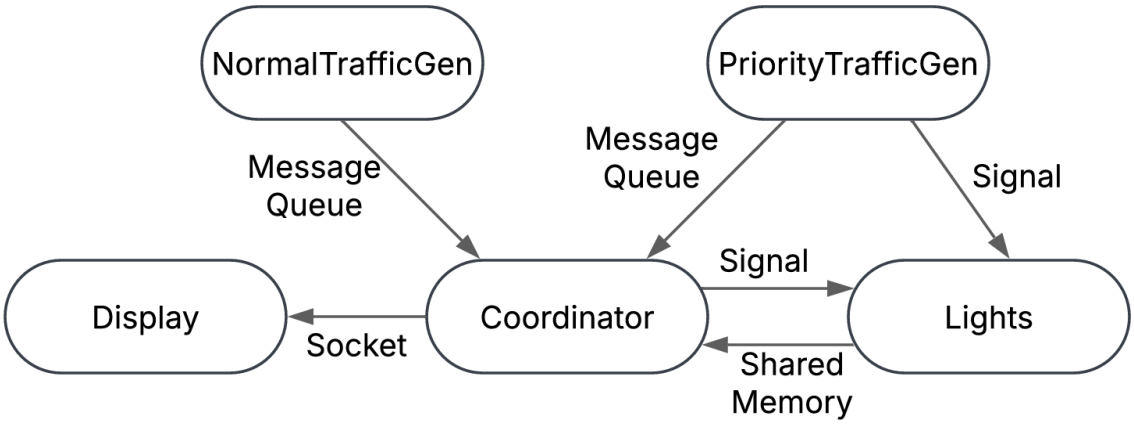

Figure 1 : Les 5 principaux processes

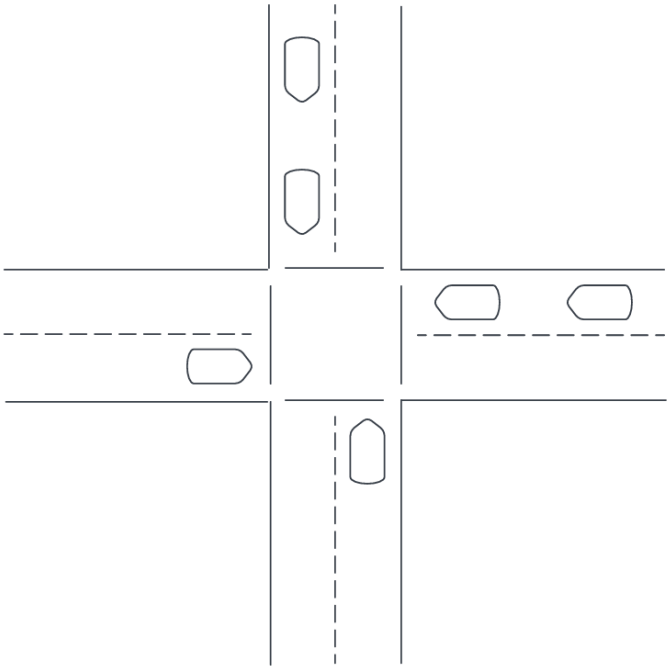

Figure 2 : Carrefour implémenté dans le projet

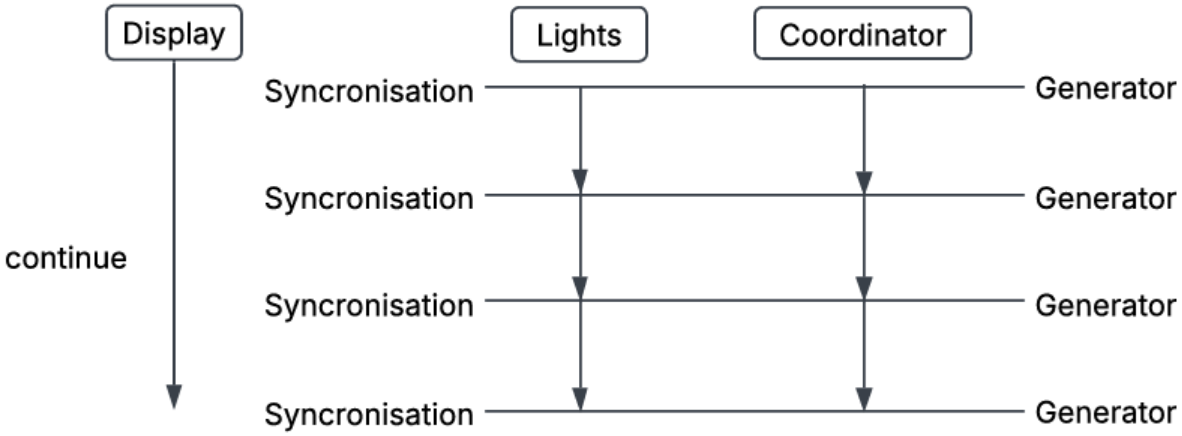

Figure 3 : Synchronisation par rapport au temps
(La flèche représente l'écoulement du temps)

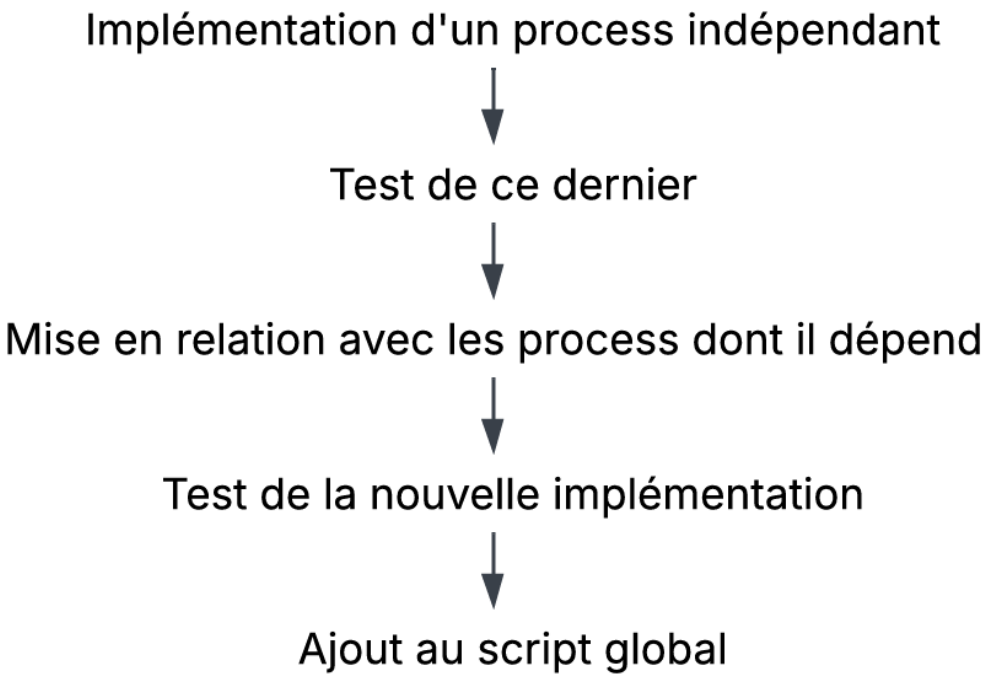

Figure 4 : Déroulement du projet