Simulation de Trafic aux Feux de Signalisation

1. Introduction

Ce projet vise à simuler le comportement d’un carrefour doté de feux de circulation, en intégrant la gestion du trafic normal et de véhicules prioritaires (ambulances, pompiers, etc.). L’objectif principal est de mettre en œuvre une architecture multi‑processus en Python, en utilisant la communication inter‑processus (queues de messages, mémoire partagée, sockets et signaux) afin de coordonner les différentes tâches (génération de trafic, gestion des feux, calcul des trajectoires, affichage, etc.).

1. Conception Globale et Architecture
   1. Processus et Relations

Le système comporte cinq processus principaux :

* **Lights (Feux) :** Gère l’alternance des états des feux (vert, rouge, orange) en mode normal et bascule en mode prioritaire dès qu’un véhicule prioritaire est détecté.
* **Normal Traffic Generator (ntg)** : Génère à intervalles réguliers des véhicules normaux dont les points de départ et d’arrivée sont choisis aléatoirement.
* **Priority Traffic Generator (ptg) :** Génère des véhicules prioritaires. En plus d’envoyer le véhicule dans la file de messages, il notifie le processus lights par un signal (SIGINT) et transmet la source du véhicule via une file dédiée.
* **Coordinator :** Lit les files de messages pour récupérer les véhicules et met à jour leurs coordonnées en fonction de leur trajectoire, du statut des feux et de la priorité (en appliquant la règle « droit → tout droit → gauche »). Il communique les mises à jour de positions à display via un socket.
* **Display :** Affiche en temps réel l’état de l’intersection (feux et positions des véhicules) via une interface Tkinter, et reçoit les données de position envoyées par le processus coordinator via un socket.

Les processus sont lancés depuis le module principal main.py qui gère la création d’un événement « stop » partagé, permettant une terminaison propre de l’ensemble des processus.

* 1. Communication Inter‑Processus et Protocoles

La communication entre les différents processus repose sur plusieurs mécanismes complémentaires. Les files de messages (via SysV IPC) sont utilisées pour transmettre les objets véhicules entre les générateurs de trafic et le coordonnateur. Chaque véhicule, encapsulé dans un objet sérialisé avec pickle, contient des informations telles que son identifiant, son origine, sa destination, sa position ainsi que le type de virage à effectuer. Par ailleurs, l’état des feux est conservé en mémoire partagée à l’aide d’un Manager de multiprocessing, ce qui permet à plusieurs processus d’y accéder de manière synchronisée. Pour assurer la transmission des mises à jour de position des véhicules entre le coordonnateur et le module d’affichage, nous avons opté pour une communication par sockets TCP. Nous avons opté pour ce protocole afin d'être sûrs que chaque message soit reçu en entier, en ajoutant un petit en-tête de quatre octets qui indique la taille du message. Enfin, la gestion des événements urgents liés aux véhicules prioritaires s’effectue via l’envoi de signaux (SIGINT et SIGTERM) qui déclenchent, par exemple, le passage en mode prioritaire dans le processus des feux et la coordination du retour au cycle normal après le passage sécurisé du véhicule.

1. Pseudo‑code des Algorithmes Importants
   1. Processus Lights

|  |
| --- |
|  |

* 1. Processus Coordinator

|  |
| --- |
|  |

* 1. Processus Display

|  |
| --- |
|  |

1. Plan d’Implantation et de Tests

Notre démarche de développement s’est déroulée en plusieurs étapes progressives. Dans un premier temps, nous avons développé et validé individuellement chaque processus. Par exemple, nous avons simulé la génération de trafic en utilisant des données statiques pour vérifier que les véhicules étaient correctement sérialisés et envoyés via les files de messages, en veillant à ce que la lecture non bloquante permette de détecter rapidement l’absence de message sans bloquer l’exécution. Parallèlement, le module lights a été soumis à des tests de gestion des signaux, afin de s’assurer que la réception de SIGINT et SIGTERM modifie immédiatement l’état des feux et que le passage en mode prioritaire se fasse sans latence perceptible. Lors de l’intégration, nous avons connecté deux processus à la fois — par exemple, l’envoi des messages du générateur de trafic vers le coordinator, puis la transmission des mises à jour de positions au display via socket. Ensuite, nous avons élaborés des scénarios de test pour simuler différentes conditions : par exemple une forte densité de véhicules (jusqu’à MAX\_TOTAL\_VEHICLES), la coïncidence de plusieurs événements prioritaires et la vérification de la fluidité de l’affichage graphique sous charge. Nous avons également mesuré les temps de réponse lors de l’activation du mode prioritaire et aussi observé « l’efficacité » de la réinitialisation des ressources (fermeture des sockets et suppression des files) lors de la terminaison. En faisant ainsi, nous avons pu régler pleins de problèmes que nous n’aurions peut-être pas pu voir, cela nous a permis aussi d’optimiser la robustesse et la réactivité du système.

1. Problèmes Rencontrés et Solutions

Au cours du développement, nous avons rencontré plusieurs défis techniques majeurs qui nous ont obligés à repenser certains aspects de la synchronisation et de la communication. L’un des principaux problèmes était la gestion des signaux. L’envoi simultané de SIGINT pour déclencher le mode prioritaire et de SIGTERM pour signaler la fin de passage a initialement créé des conflits dans l’actualisation de l’état des feux, en particulier lorsque le processus lights devait basculer rapidement entre les modes. Pour résoudre ce problème, nous avons renforcé nos gestionnaires de signaux en intégrant des blocs try/except et nous vérifions systématiquement l’état des variables partagées avant de procéder à toute modification, garantissant ainsi une transition fluide. Un autre défi a été de s’assurer que la lecture des messages via les sockets ne génère pas de trames incomplètes. L’implémentation d’un protocole préfixé par la taille des messages a permis de résoudre ce problème, assurant que le module display reçoive toujours des données complètes et cohérentes. De plus, la gestion concurrente des files de messages a nécessité l’utilisation de lectures non bloquantes, accompagnées de la gestion d’exception (notamment en capturant les erreurs de type BusyError), afin d’éviter les blocages et de permettre un traitement en temps réel. Enfin, l’intégration de tous ces mécanismes a mis en lumière la nécessité de tests approfondis pour valider la stabilité du système en conditions extrêmes, ce qui nous a conduit à mettre en place des scénarios de test détaillés et à documenter minutieusement chaque problème rencontré et la solution apportée. Ces améliorations techniques ont vraiment renforcé la solidité de notre architecture et illustrent bien notre capacité à prévoir et surmonter les défis complexes liés aux systèmes multi‑processus.

1. Conclusion

Ce projet nous a permis de concevoir et de mettre en œuvre une simulation d’intersection qui intègre de manière efficace la gestion simultanée du trafic normal et des véhicules prioritaires au moyen d’une architecture multi‑processus en Python. Au-delà de l’aspect technique, ce travail de groupe a été l’occasion de conjuguer nos compétences pour relever des défis complexes tels que la synchronisation des signaux, la gestion concurrente des files de messages et la communication fiable par sockets. Grâce à l’utilisation d’une combinaison de mécanismes (files de messages, mémoire partagée, sockets et signaux), nous avons pu créer un système robuste, réactif et évolutif, capable de fonctionner en temps réel même dans des conditions de charge extrême.

La démarche incrémentale adoptée nous a permis d’identifier et de résoudre rapidement les problèmes, tout en garantissant une intégration harmonieuse des différents modules. Les nombreux tests effectués ont confirmé la stabilité et la cohérence de notre solution, et ont mis en lumière notre capacité à anticiper et surmonter des problématiques techniques pointues. Par ailleurs, les perspectives d’évolution identifiées, notamment en matière d’optimisation de la gestion des erreurs et d’enrichissement de l’interface graphique, ouvrent la voie à de futures améliorations qui pourraient encore renforcer les performances et la fiabilité de la simulation.