Auteurs: Auberson Kevin, Alexandre Shyshmarov

# Description des fonctionnalités du logiciel

Ce logiciel implémente la gestion de deux locomotives dans un simulateur de réseau ferroviaire. Chaque locomotive circule sur un parcours défini, passant par des points de contacts spécifiques, une gare, et une section partagée.

- **Contrôle des locomotives** : Les locomotives démarrent, s'arrêtent, changent de vitesse et suivent un parcours cyclique.
- **Arrêts en gare** : Les locomotives attendent en gare 5 secondes pour permettre au passagers de changer de train.
- **Arrêts d'urgence** : Possibilité d'arrêter d'urgence les deux locomotives.
- **Section partagée** : Quand les locomotives doivent emprunter une même route, une section partagée est mise en place sous forme de classe.
- **Aiguillage** : Une classe d'aiguillage est mise en place afin de simplifier le changement d'aiguillage pour chaque train. Elle fonctionne sous le même principe que le chois du parcours.

# Choix d'implémentation

## Approche adoptée

Pour résoudre le problème, nous avons suivi une approche basée sur la définition de trois classes principales : `Synchro`, `LocomotiveBehavior` et `TrainSwitch`.

La classe `Synchro` gère la section partagée en utilisant des sémaphores pour contrôler l'accès des locomotives. Elle possède deux méthodes afin de gérer ça. access() qui permet de donner accès à la section partagée en fonction de l'occupation et de la priorité. Cette dernière est protégée par une Sémaphore (mutex). Si le train à l'autorisation, il entre et relâche le mutex sinon il va dans la file d'attente (waiting) et attend la libération de la section. leave() est appelé lorsqu'un train sort de la section partagée. Si des trains sont en attente, le train sortant release un acces.
Pour l'arrêt des trains en gare la fonction stopAtStation() a été implémenté. Elle permet au premier train qui active cette fonction de se placer dans une file d'attente représentée par une sémaphore "atStation". Un bool trainAtStation a été utilisé pour choisir la partie de code du premier train arrivé en gare. Le second train effectue l'autre partie du code qui va libérer le premier train puis lui affecter une priorité à 1. La priorité est utilisée pour l'entrée du premier

train dans la section partagée. L'attente de 5 sec en gare est effectuée dans la fonction run() des trains (threads) avec la fonction usleep().

La classe `LocomotiveBehavior` gère l'ensemble du parcours ainsi que le comportement adopté par une locomotive. Dans le .h, on a rajouté deux vecteurs qui nous permettent de gérer la logique. Un vecteur de parcours qui stock les capteurs que doit atteindre le train et le deuxième, un vecteur de TrainSwitch qui permet de changer les aiguillages. Le .cpp comporte à proprement parler la logique. Quel capteur parcourir dans quel ordre.

La classe `TrainSwitch` permet une gestion des aiguillages. On l'a crée pour simplifier la déclaration des aiguillages à prendre pour chaque train. Elle stock un pointeur sur la fonction de changement d'aiguillage (on lui a mis une valeur par défaut, car dans notre cas, on connaît cette fonction.) et les informations, dont cette fonction à besoin. La méthode getSwitch() permet de faire le changement. Cette classe est utilisée avec un vecteur dans le cppmain, ce qui nous permet de rajouter des aiguillages facilement (Il faudrait aussi rajouter la logique qui va avec dans locomotivebehavior.cpp).

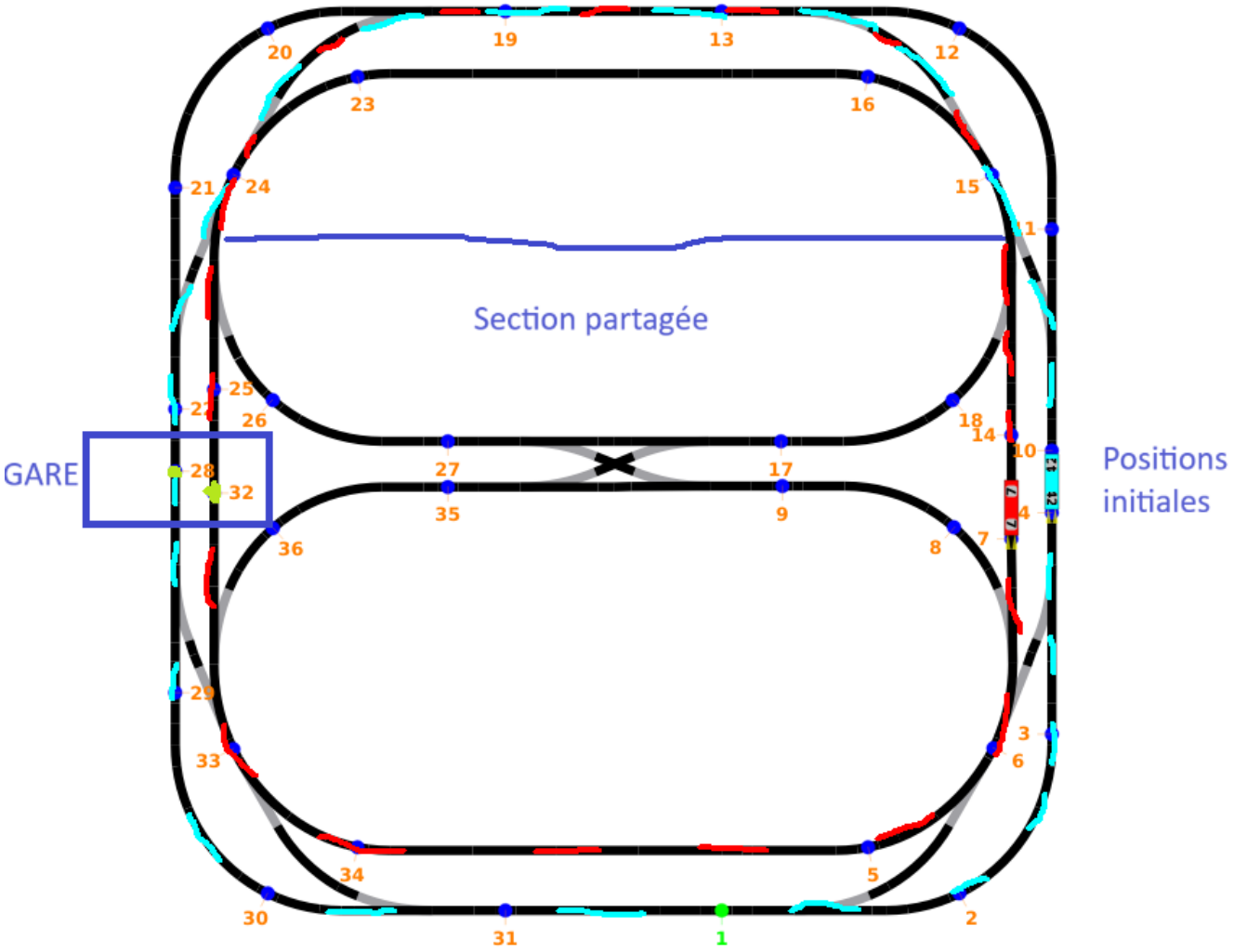
Le vecteur parcours et aiguillage dans le cppmain ont sensiblement la même logique. Chaque train doit avoir son tableau d'aiguillage ainsi qu'un tableau de parcours. Ces derniers sont passés au constructeur de train avant que celui-ci ne commence sont parcours dans locomotivebehavior.
Cela nous permet d'avoir "deux comportements diffèrent" en une seule ligne.

attendre_contact(parcours[0]); Par exemple ici la valeur se trouvant à l'emplacement 0 sera différente dans le thread du train A et dans le thread du train B.

## Protection des variables

Les variables critiques telles que le nombre de trains dans la section partagée, le contrôle des sémaphores pour l'accès concurrent, et le suivi des locomotives en attente ont été protégées par l'utilisation de mutex (sémaphores) pour éviter les conditions de concurrence.

# Tests effectués




Tous les tests ci-dessous sont effectués avec la maquette A.

1. **Test de synchronisation des locomotives :** Vérification que les locomotives accèdent à la section partagée de manière synchronisée en respectant les règles de priorité. Ce test a été réussi en observant que la locomotive ayant la priorité accède en premier à la section partagée.
2. **Test d'arrêt en gare :** Vérification que les locomotives attendent correctement en gare et reprennent leur chemin après l'arrêt requis. Ce test a été validé en observant le comportement des locomotives à l'approche de la gare.
3. **Test d'arrêt d'urgence :** Vérification de la fonctionnalité d'arrêt d'urgence pour stopper instantanément les locomotives. Ce test a été réalisé avec succès en déclenchant l'arrêt d'urgence et en observant l'arrêt immédiat des deux locomotives. Lorsque la locomotive passe sur le contact de sortie de la section partagée. L'autre train arrêté à l'entrée de la section partagée ne démarre pas. Les deux trains arrêtés à la gare ne démarrent pas lors de l'activation de cette fonctionnalité.
4. **Test de temps :** Le programme a été en fonction pendant plus 1h30 afin de vérifier sur une plus grande plage d'exécution si des problèmes surviendraient. Ce test a été validé

avec l'apparition d'aucun problème est un fonctionnement continuel.

# Conclusion

En résumé, le logiciel développé permet de contrôler deux locomotives dans un simulateur de réseau ferroviaire tout en respectant les règles de synchronisation, les arrêts en gare et la gestion d'urgence. Les tests ont été concluants, confirmant le bon fonctionnement du programme selon les spécifications du cahier des charges.
Ce labo nous a entre autres permis de comprendre comme fonctionne les sémaphores, comment les implémenté à travers une section partagée et comment on peut faire pour synchroniser deux threads (simulés avec l'attente d'un train à la gare).