PCO Laboratoire 4

Gestion de Ressources

Benoît Delay, Eva Ray

November 28, 2023

Description des fonctionnalités du logiciel

Le programme simule le déplacement de deux locomotives, dont le comportment est exécuté par un thread lancé avec la méthode `LocomotiveBehavior::run()`. Chaque locomotive a un certain point de départ et suit un parcours cyclique. Les deux parcours ont un tronçon commun, auquel une seule locomotive à la fois ne peut accéder. Ce tronçon est appelé "section partagée" ci-après. Si une locomotive est en train d'accéder à la section partagée, l'autre s'arrête et attend son tour.

Chaque locomotive a une gare attirée à laquelle elle doit s'arrêter. Les deux locomotives assurent une correspondance aux passagers, afin qu'ils puissent changer de train si besoin. Ainsi, lorsque la première locomotive arrive en gare, elle s'arrête et attend la seconde. Lorsque les deux locomotives sont en gare, elles attendent encore 5 secondes avant de repartir et continuer leur parcours.

La priorité d'accès à la section partagée est définie comme ceci: la locomotive qui arrive en gare en dernier est prioritaire.

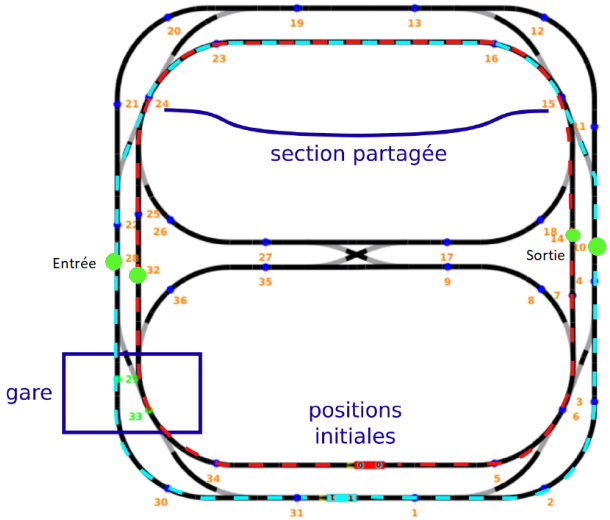
Les locomotives continuent leur parcours de manière infinie. Le seul moyen de les arrêter est d'actionner l'arrêt d'urgence.

Le logiciel simule le déplacement des locomotives par des threads différents, il est donc multi-threadé. Par conséquent, il doit assurer une bonne gestion de la concurrence pour les ressources partagées entre plusieurs threads. En particulier, l'accès à la section partagée doit être géré conformément au exigences citées ci-dessus.

Choix d'implémentation

Choix du parcours

Nous avons décidé de travailler sur la maquette `A`. L'imagination n'étant pas notre point fort, nous avons choisi de définir le parcours des locomotives de la même manière que dans l'exemple présent dans la consigne de laboratoire. Les points de départ des locomotives, les gares et la section critique sont mises en avant dans le schéma ci-dessous.



Gestion section partagée

Pour gérer l'accès à la section partagée, nous avons ajouté un sémaphore `waiting` et un compteur `nbWaitingSharedSec` en attributs privés de la classe `Synchro`. Ce sémaphore est initialisé à 0 et agit comme une barrière. Tout thread qui tente d'acquérir le sémaphore est bloqué tant qu'un autre thread ne l'aura pas relâché. Ce sémaphore est utilisé dans la méthode `Synchro::access()`.

Si une locomotive qui n'a pas la priorité essaie d'accéder à la section partagée, elle va acquire le sémaphore et se mettre en attente. Il faut alors qu'une autre locomomotive release le sémaphore. Cela se fait dans la méthode `Synchro::leave()`, qui est appelée dès qu'une locomotive quite la section partagée. Si une locomotive est en attente, on release le sémpahore `waiting` et elle peut à son tour accéder à la section partagée.

Gestion attente à la gare

Pour gérer l'attente à la gare, nous avons ajouté un sémaphore `station` et un compteur `nbWaitingStation` en attributs privés de la classe `Synchro`. Comme pour la section partagée, ce sémaphore est initialisé à 0 et agit comme une barrière. La gestion de la gare se fait dans la méthode `Synchro::stopAtStation`.

Si la locomotive arrive à la gare en premier, elle doit attendre la deuxième. Pour simuler ce comportement, la première locomotive va acquire le sémaphore `station` et va donc devoir attendre l'arrivée en gare de la seconde locomotive avant de faire sa routine (i.e. attendre 5 secondes). La priorité de la locomotive est alors mise à 0, ce qui signifie qu'elle n'est pas prioritaire pour l'accès à la section partagée. Si la locomotive qui arrive en gare est la dernière à arriver, elle release le sémaphore `waiting` pour libérer l'autre locomotive en attente. La priorité de la locomotive est alors mise à 1, ce qui signifie qu'elle est prioritaire pour l'accès à la section partagée. Une fois les 5 secondes d'attente passées, les deux locomotives peuvent redémarrer.

Gestion variables partagées

Certaines variables sont partagées entre plusieurs threads, il faut donc protéger leur accès. En particulier, les attributs `nbWaitingSharedSec` et `nbWaitingStation` de la classe `Synchro` sont partagés entre les différents threads lancés sur `LocomotiveBehavior::run()`. Afin de les protéger, nous avons ajouté un sémaphore `mutex` comme attribut de la classe `Synchro`. Ce sémaphore est initilaisé à 1 et se comporte donc comme un mutex. Il est acquire dès qu'on doit modifier des variables partagées ou qu'on a besoin d'une lecture protégée, par exemple dans la condition d'un `if`. Le sémaphore est release dès qu'on a fini notre travail. D'une manière générale, nous essayons de réduire la section critique au minimum, afin de de ne pas bloquer des threads pour rien. Le mutex est aussi systématiquement release avant qu'un thread passe en attente sur les sémaphores `waiting` ou `station`, encore une fois afin de ne pas bloquer les autres threads pendant l'attente. Une fois l'attente terminée, le mutex peut être acquire à nouveau, si besoin.

Classe Route

Nous avons défini et implémenté une nouvelle classe `Route`, dont le nom est la traduction de "parcours" en anglais, qui représente les principales cractéristiques du parcours suivi par une certaine locomotive. Les attributs de la classe décrivent les points clés du parcours. Les attributs `station`, `startSharedSection` et `endSharedSection` contiennent le numéro des points de contact de la gare, du début et de la fin de la section partagée du parcours. Dans l'attribut `aiguillages`, on stocke le numéro et l'orientation des aiguillages permettant d'accéder et de quitter la section partagée. L'orientation des aiguillages est propre à chaque locomotive. Tous ces attributs ont une visibilité privée, afin de garantir une bonne encapsulation. Nous avons défini des méthodes publiques qui permettent d'attendre la détection de points de contact cités ci-dessus et une méthode qui permet de changer la direction des aiguillages pour garantir les fonctionnalités qui peuvent être utiles en dehors de la classe. Ces méthodes sont notamment utilisées dans `LocomotiveBehavior::run()`.

Définir une nouvelle classe est utile pour des raisons d'encapsulation des données, de lisibilité et de modularité.

Modularité

Le programme est orienté pour être relativement modulable.

L'utilisation de la classe `Route` permet d'avoir différents types de parcours. Comme les directions des aiguillages sont sockées dans un `std::Vector`, on peut imaginer un parcours qui demande le changement de direction de plus de deux aiguillages. Si on souhaite, par exemple, avoir plusieurs gares ou sections critiques, on peut changer les attributs qui stockent le numéro des points de contact en `std::Vector` et adapter les méthodes correspondantes en

conséquences. (Il faudrait aussi évidemment adapter `LocomotiveBehavior::run()` pour que le comportement des locomotives ait du sens.)

La classe `Route` nous permet aussi d'avoir une méthode `LocomotiveBehavior::run()` propre, simple et sans valeurs hardcodées.

Le programme a aussi été développé en pensant qu'il pourrait potentiellement y avoir plus de deux locomotives sur la maquette. Il faudrait faire quelques ajustements mais la présence des attributs `nbWaitingSharedSec` et `nbWaitingStation` rendent le programme extensible. Si on souhaite avoir plus deux locomotives sur la maquette, il faut modifier la méthode `Synchro::stopAtStation` afin que l'arrivée en gare de la dernière locomotive libère toutes les locomotives en attente, pas seulement une. Ensuite, il faudrait mettre en place un nouveau mécanisme de priorité d'accès à la section partagée, si on le souhaite, avec par exemple un mécanisme de file LIFO. Dans l'état actuel des choses, à part pour la locomotive qui a la priorité, les autres locomotives accèderont à la section partagée selon la loi du "premier arrivé, premier servi".

Tests effectués

Le résultat des tests ci-dessous a notamment pu être vérifié grâce au fait que les locomotives log dans l'interface graphique tous leurs comportements. En particulier, un message apparait lorsqu'une locomotive arrive en gare, accède à la section partagée avec une certain priorité ou quitte la section partagée.

Test: Accès à la section partagée

- Lorsque la locomotive A arrive en dernier à la gare, elle a la priorité pour l'accès à la section partagée.
- Lorsque la locomotive B arrive en dernier à la gare, elle a la priorité pour l'accès à la section partagée.
- Lorsqu'une locomotive qui n'a pas la priorité arrive à la section partagée, elle s'arrête et attend son tour.
- Lorsqu'une locomotive quitte la section partagée, l'autre locomotive qui était en attente est débloquée et redémarre.
- Lorsqu'une locomotive a le droit d'accès à la section partagée, les aiguillages s'orientent dans la direction définie par le parcours de la locomotive.
- Lorsqu'une locomotive accède à la section partagée et s'y arrête, l'autre locomotive continue d'attendre.
- Lorsqu'une locomotive qui n'a pas la priorité attend l'accès à la section partagée mais que la locomotive qui a la priorité n'arrive jamais, elle attend indéfiniment.

Ces tests ont été faits en mettant les locomotives A et B dans chacun des rôles décrits (en utilisant notamment le bouton pause), afin d'être certain que les résultats des tests ne dépendent pas de la locomotive. La priorité des locomotives accédant à la section partagée a pu être vérifié grâce aux logs affichés. Tous les tests produisent le résultat attendu.

Test: Attente à la gare

- Lorsqu'une locomotive arrive au niveau du contact de sa gare, elle s'arrête.
- Lorsqu'une locomotive arrive en gare en premier, elle attend que l'autre arrive avant de faire son décompte de 5 secondes.
- Lorsque la deuxième locomotive arrive en gare, les deux locomotives attendent bien 5 secondes avant de repartir.
- Lorsqu'une locomotive arrive en gare en premier et que la deuxième n'arrive jamais, la locomotive ne redémarre pas.

Ces tests ont été faits en mettant les locomotives A et B dans chacun des rôles décrits (en utilisant notamment le bouton pause), afin d'être certain que les résultatsdes tests ne dépendent pas de la locomotive. Tous les tests produisent le résultat attendu.

Test: Arrêt d'urgence

- Lorsque les deux locomotives sont en train de rouler er qu'on appuie sur le bouton d'arrêt d'urgence, les locomotives s'arrêtent comme attendu.
- Lorsqu'une seule locomotive roule et qu'on a mis l'autre sur pause, la locomotive qui roule s'arrête et la locomotive qui était sur pause ne repart pas quand on appuie sur start. C'est le comportement attendu.
- Lorsqu'on appuie sur l'arrêt d'urgence alors qu'une locomotive se trouve dans la section partagée et a juste assez d'inertie pour déclencher le capteur de sortie de secion partagée, la locomotive s'arrête et l'autre locomotive qui attend l'accès à la section partagée ne redémarre pas. C'est le comportement attendu.

En plus de ces tests, nous avons aussi laissé tourner la simulation pour une vingtaine de tours et les locomotives ont toujours eu le comportement espéré. Nous avons aussi beaucoup joué avec le bouton pause et nous n'avons jamais rencontré de comportements innatendus.

Conclusion

Nous avons implémenté un programme multi-threadé qui simule le déplacement des locomotives conformément au cahier des charges. Nous avons implémenté une gestion de la concurrence et d'accès aux ressources partagées grâce à des sémaphores, qui semblent faire leur travail convenablement. L'accès à la section partagée, la définition des priorités, le changement de direction des aiguillage et l'attente à la gare sont gérés correctement. Le bouton d'arrêt d'urgence fonctionne lui aussi comme attendu et nous permet de mettre fin à la simulation.