HEIG-VD | PCO - Laboratoire 4

Gestion de ressources

Auteurs : Harun Ouweis & Christen Anthony
Date : 29.11.2023

Introduction

Dans le cadre de ce laboratoire, nous sommes chargés de compléter un programme C++ simulant la gestion de deux locomotives sur un circuit de train modélisé. Chaque locomotive est contrôlée par un thread et doit suivre un itinéraire prédéterminé comportant une section de voie partagée qui nécessite une coordination minutieuse pour éviter les collisions. De plus, les locomotives doivent coordonner leurs arrêts en gare pour permettre aux passagers de changer de train. Un arrêt d'urgence a également dû être implémenté.

Fonctionnalités clés du logiciel :

- Multi-threading : Chaque locomotive est "matérialisée" par un thread, permettant des opérations parallèles et synchronisées.
- Synchronisation : Les locomotives doivent synchroniser leur accès à la section partagée et leurs arrêts en gare.
- Gestion des ressources : Contrôle des états des locomotives et gestion des conflits d'accès aux sections critiques.
- Interface de suivi : Vue d'ensemble en temps réel du parcours des locomotives et de leur état via les différentes consoles (console générale et consoles propores à chaque locomotive).
- Arrêt d'urgence : La simulation peut être arrêtée à tout moment.

Objectif

L'objectif de ce projet est de mettre en œuvre la logique de contrôle des locomotives qui permet à la simulation de fonctionner de manière cohérente et de protéger les potentiels accès concurrents aux sections critiques de la simulation, comme la section partagée et la gare.

Implémentation de notre solution

Nous avons commencé ce laboratoire en analysant le code fourni afin d'identifier les zones où des modifications seraient nécessaires. Nous avons déterminé que les interactions entre les locomotives dans la section partagée et la gare sont les points clés du bon fonctionnement de la simulation.

Comme vous pouvez le constater sur la `figure 1`, nous avons décidé de placer la zone de départ de la `locomotive A` entre les contacts 7 et 14 et celle de la `locomotive B` entre les contacts 4 et 10. Leur point d'entrée en gare sont respectivement les contacts 5 et 2. La section partagée que nous avons choisie se trouve entre les contacts 24 et 15. Par rapport aux points de contacts de la section partagée, nous avons décidé d'utiliser un offset de 3 pour la demande d'accès à la section paratgée, un offset de 1 pour l'adaptation des aiguillages et également un offset de 1 pour la sortie de la section partagée. Ainsi, nous disposons des contacts suivants pour chaque locomotive :

- `Locomotive A` :
  - Contact d'entrée en gare : 5
  - Contact d'accès à la section partagée : 33
  - Contact d'adaptation des aiguillages : 25
  - Contact de sortie de la section partagée : 14
- `Locomotive B` :
  - Contact d'entrée en gare : 2
  - Contact d'accès à la section partagée : 29
  - Contact d'adaptation des aiguillages : 22
  - Contact de sortie de la section partagée : 10

L'obtention de ces contacts en fonction de la section partagée et des offsets est détaillé dans la section `TrainTrack`.

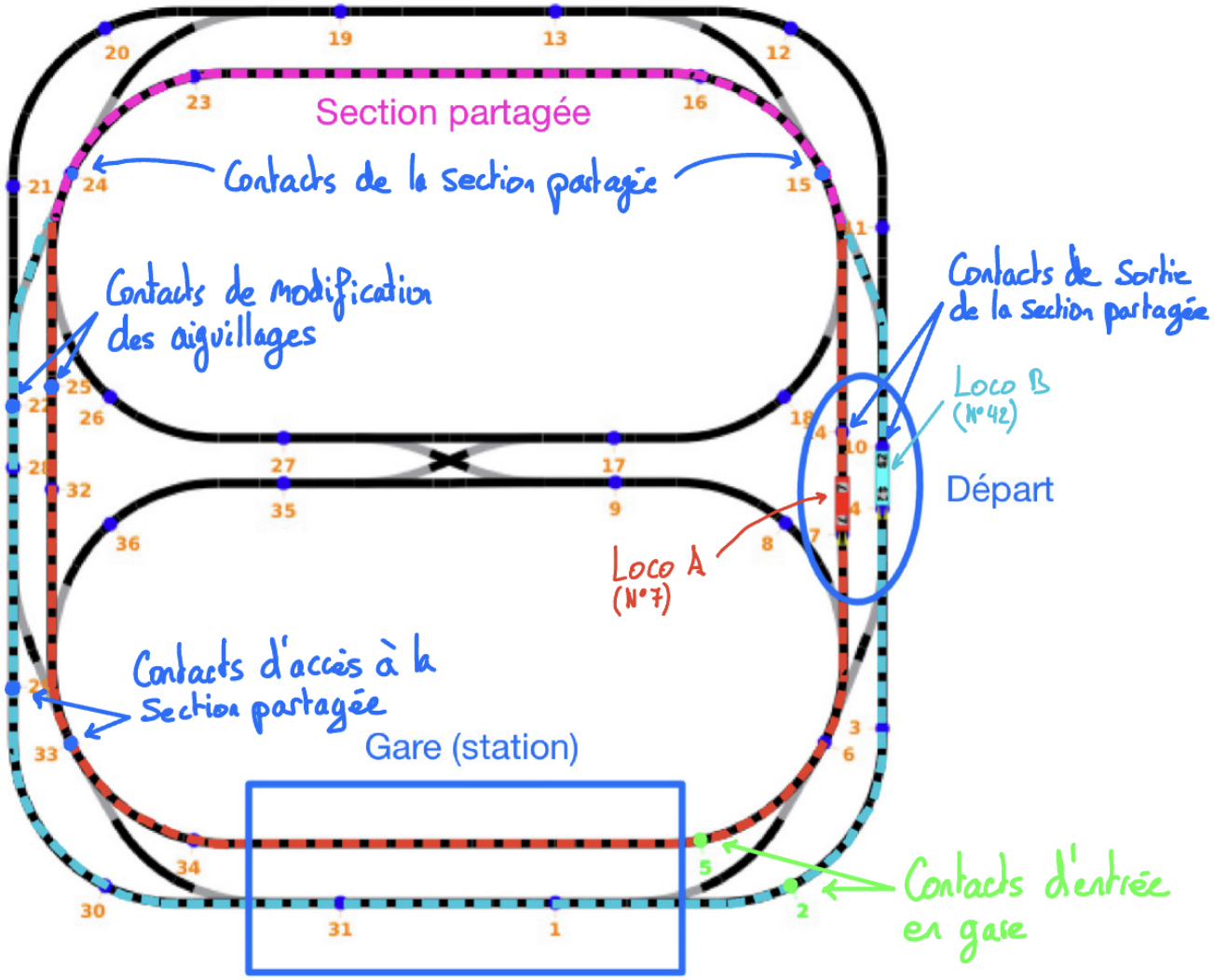

Figure 1 : Circuit

`TrainTrack`

Il s'agit d'une classe que nous avons décidé d'ajouter afin d'encapsuler la logique relative au parcours d'une locomotive. Une instance de cette classe regroupe la voie de chemin de fer de la locomotive, son contact d'entrée en gare, sa section partagée et ses aiguillages devant être adaptés afin d'accéder à la section partagée et d'en sortir.

Nous avons défini les types suivant afin de rendre le code plus compréhensible :

- `Contact` : un entier non-signé représentant un point du circuit et permettant de savoir où la locomotive se trouve afin d'effectuer certaines actions.
- `Railway` : Un vecteur (`std::vector<>`) de `Contact` représantant la voie de chemin de fer sur laquelle évolue la locomotive.

- `Section` : Une paire (`std::pair<>`) de pointeurs sur des `Contact` servant à "marquer" les contacts représentant la section partagée sur la voie de chemin de fer (`Railway`). Le premier élément correspond au contact de début et le deuxième au contact de fin de la section partagée.
- `Switch` : Une paire (`std::pair<>`) de valeurs entières non-signées. Le premier élément correspond à l'aiguillage (numéro) et le second à sa position.

Cette classe est composée des attributs suivants :

- `track` : Il s'agit du circuit de la locomotive qui est de type `RailWay` .
- `station` : Un pointeur sur le contact d'entrée en gare.
- `sharedSection` : La section partagée qui est de type `Section` .
- `switches` : Un vecteur de `Switch` contenant tous les aiguillages devant être adaptés sur le circuit de la locomotive.
- `sharedSectionAccessOffset` : Un offset permettant de faire évoluer le pointeur sur le contact de début de la section partagée afin de définir un contact sur lequel la locomotive effectue une demande d'accès à la section partagée.
- `sharedSectionLeaveOffset` : Un offset permettant de faire évoluer le pointeur sur le contact de fin de la section partagée afin de définir un contact à partir sur lequel la locomotive indique qu'elle a quitté la section partagée.
- `switchesUpdateOffset` : Un offset permettant de faire évoluer le pointeur sur le contact de début de la section partagée afin de définir un contact pour lequel les aiguillages doivent être adaptés lorsque la locomotive l'atteint.

Cette classe dispose des méthodes suivantes :

- `TrainTrack(...)` : Un constructeur prenant en paramètre un pointeur sur une voie de chemin de fer (`Railway*`), le numéro du contact d'entrée en gare, les numéros des contacts de début et de fin de la section partagée et un vecteur d'aiguillages (`std::vector<Switch>`). Son rôle est de localiser (`std::find()`) les contacts mentionnés précédemment dans la variable `track` afin d'obtenir des pointeurs vers ces contacts. Cette approche nous permet de récupérer les contacts situés avant ou après ceux pour lesquels nous avons des pointeurs en faisant évoluer ces pointeurs. Il initialise également les offsets décrits précédemment.
- `travelToStation()` : Cette méthode permet d'attendre que la locomotive atteigne la gare.
- `travelToSharedSectionStart()` : Cette méthode permet d'attendre que la locomotive atteigne le contact de demande d'accès à la section partagée. Ce contact consiste à faire évoluer le pointeur sur l'entrée de la section partagée de `sharedSectionAccessOffset` contacts en arrière.
- `travelToSharedSectionEnd()` : Cette méthode permet d'attendre que la locomotive atteigne le contact de sortie de la section partagée. Ce contact consiste à faire évoluer le pointeur sur la sortie de la section partagée de `sharedSectionLeaveOffset` contacts en avant.

- `updateSwitches()` : Cette méthode permet d'attendre que la locomotive atteigne le contact d'adaptation des aiguillages et d'effectuer cette. Ce contact consiste à faire évoluer le pointeur sur l'entrée de la section partagée de `switchesUpdateOffset` contacts en arrière.

A l'aide des propriétés listées ci-dessus, nous épurons le code de la méthode `run()` de la classe `LocomotiveBehavior` en déléguant la logique relative au circuit à la classe `TrainTrack` .

`Synchro`

Cette classe implémente toute la logique de synchronisation/coordination des locomotives utile à certains tronçons du circuit. Nous avons ajouté à la classe `Synchro` des sémaphores et des variables booléennes afin de gérer/coordonner les accès de locomotives à la gare et à la section partagée. La gare et la section partagée sont gérées comme des entités distinctes. En effet, elles disposent chacunes de leurs propres sémaphores et de leurs propres variables booléennes. Ceci permet d'éviter tout éventuel conflit d'accès n'ayant pas lieu d'être et de permettre d'adapter le code plus facilement si le parcours devait changer ou si des contraintes supplémentaires devaient être ajoutées.

Comme demandé dans le cahier des charges, nous avons implémenté une logique de priorité entre les locomotives. La deuxième locomotive à arriver en gare possède un priorité supérieure à la première locomotive arrivée. Les priorités sont utilisées lors de l'accès à la section partagée.

Signification des valeurs des priorités :

- `-1` : La locomotive est secondaire
- `0` : La locomotive ne possède pas de priorité particulière.
- `1` : La locomotive est prioritaire.

Lors de notre première implémentation de cette classe, nous avons observé un cas limite lorsque la locomotive prioritaire appelait la méthode `leave()` de sortie de la section critique alors que la locomotive secondaire n'avait pas encore appelé la méthode `access()` de demande d'accès à la section critique. Dans ce cas, la locomotive secondaire, ayant une priorité négative, était bloquée jusqu'à ce qu'une locomotive prioritaire la libère en sortant de la section critique. Mais, étant donné que la locomotive prioritaire devait d'abord passer par la gare et attendre l'autre locomotive, nous nous retrouvions dans une situation d'interblocage. Nous avons donc ajouté la variable booléenne `considerPriority` afin de déterminer si la priorité doit être considérée selon la situation et ainsi résoudre ce cas limite. Lorsqu'une locomotive quitte la section partagée, il n'est pas nécessaire de considérer la priorité (`considerPriority = false`). En revanche, lorsque les locomotives quittent la gare, il faut à nouveau considérer la priorité (`considerPriority = true`) jusqu'à ce qu'une locomotive quitte la section partagée.

`Gare`

Les locomotives commencent par passer par la gare possédant un contact d'entrée. Les trains s'immobilisent progressivement après avoir franchi ce contact. La classe `TrainTrack` possède la méthode `travelToStation()` permettant de bloquer le thread jusqu'à ce que sa locomotive ait atteint ce contact. Du point de vue de la synchronisation, la première locomotive arrivant en gare est bloquée à l'aide d'un sémaphore (`stationStop`). La variable booléenne `trainWaitingAtStation` indique si une locomotive attend en gare. Lorsqu'une seconde locomotive entre en gare, elle s'immobilise, attend 5 secondes et libère la première locomotive arrivée en effectuant un `relase()` sur le `stationStop` pour que les 2 locomotives puissent reprendre leur voyage. Au moment de repartir, les locomotives se voient attribuer une priorité qui sera ensuite utilisée lors de l'accès à la secion partagée. La locomotive prioritaire (`1`) est la dernière arrivée en gare et la secondaire (`-1`) est donc la première arrivée.

`Section partagée`

Lorsque les locomotives quittent la gare, les prochains contacts à atteindre sont les contacts de demande d'accès à la section partagée. Lorsque la locomotive atteint son contact, la classe de synchronisation vérifie si la section partagée est occupée ou si la priorité doit être considérée (`considerPriority`) et que le train a une priorité inférieure à 0. Dans ces cas, la locomotive est mise en attente à l'aide d'un sémaphore (`sharedSectionStop`) et la variable boléenne `trainWaitingAtSharedSection` est mise à jour. Nous avons décidé d'utiliser une boucle while forcant la locomotive libérée de son attente à revérifier les conditions d'accès à la section partagée décrites précedemment afin de s'assurer qu'elle ne l'accède pas si elle venait à être préemptée par une autre locomotive lors de la récupération du mutex (`sharedSectionMutex`). Une fois que la locomotive peut accéder à la section partagée, ses aiguillages sont adaptés en conséquence par la méthode `updateSwitches()` de la classe `TrainTrack` . Lorsque la locomotive quitte la section partagée, la variable booléenne `sharedSectionAvailable` est mise à jour et s'il y a une locomotive en attente à l'entrée, elle est libérée.

Les variables relatives aux sections décrites ci-dessus (`Gare` et `Section partagée`) sont protégées par des mutex (sémaphores initialisés à 1).

`LocomotiveBehavior`

Nous avons ajouté un attribut de type pointeur sur un objet `TrainTrack` et nous avons adapté le constructeur en conséquence. La variable `sharedSection` a été renommée en `sharedSectionSync` afin d'expliciter le fait qu'il s'agisse d'un objet de synchronisation et pas de la section partagée en elle-même.

Le cycle de fonctionnement d'une locomotive a été implémenté dans la méthode `run()`. Etant donné que la logique relative au circuit a été encapsulée dans la classe `TrainTrack`, cette méthode dispose de relativement peu de code et il est assez explicite concerant les différentes étapes par lequelles passe la locomotives.

`main`

Nous avons implémenté la création des circuits des locomotives, la gestion des éventulles erreurs en résultant et la libération de la mémoire allouée. La variable `sharedSection` a été renommée en `sharedSectionSync` afin d'expliciter le fait qu'il s'agisse d'un objet de synchronisation et pas de la section partagée en elle-même.

Nous avons également implémenté la fonction `emergency_stop()` afin de permettre à l'utilisateur de stopper la simulation à tout moment. En plus d'appeler la méthode `arreter()` de chaque locomotive, nous fixons leur vitesse à 0 afin de s'assurer que la locomotive ne redémarrera pas. En effet si nous ne rendons pas la vitesse nulle, lorsque `emergency_stop()` est appelée, si la locomotive, en s'arrêtant avec son inertie, traverse un contact qui la place dans un état d'attente, il existe un risque qu'elle redémarre à la même vitesse qu'elle possédait avant de s'arrêter. Nous avons pu observer ce cas limite lors d'un arrêt d'urgence effectué juste avant l'entrée en gare des locomotives.

Remarques

Notre simulation est fonctionelle tel quel. En effet, si la vitesse des locomotives venait à être augmentée de manière considérable et que l'option d'inertie est activée, les distances de freinage des locomotives seraient trop longues pour que les accès aux sections critiques puissent être gérés correctement.

Tests effectués

Des tests variés ont été réalisés dans différentes conditions opérationnelles pour confirmer la fiabilité de la simulation :

- Comportement des Locomotives : Tests pour s'assurer que les locomotives suivent leur parcours prévu et réagissent correctement aux contacts et aux aiguillages.
- Synchronisation et conflits : Tests pour s'assurer de la bonne gestion des accès des sections critiques (section partagée et gare).
- Arrêt d'Urgence : Vérification de la réponse du système à l'activation de l'arrêt d'urgence.
- Longue Durée : Des tests prolongés ont également été effectués pour garantir la stabilité de la simulation sur de longues périodes.

Ces tests ont été effectués en diminuant la vitesse des locomotives, en l'augmentant légèrement (voir `Remarques`), en mettant en pause l'une des locomotives ou les deux à différents endroits du circuit afin de s'assurer que la simulation réponde bien à toutes les situations pouvant être rencontrées. C'est d'ailleurs lors de ces tests que nous avons constaté le cas limite décrit dans la section `Synchro` de ce rapport et que nous avons adapté notre code en conséquence.

Conclusion

Ce laboratoire a été l'occasion de consolider notre compréhension des techniques de synchronisation et de gestion des accès concurrents dans un contexte multi-threadé. Notre solution, fruit d'une conception minutieuse et d'une série de tests rigoureux, démontre l'importance d'une approche méthodique dans le développement de systèmes parallèles fiables.