



PCO - Labo4 Gestion de ressources

Auteurs: Lazar Pavicevic et Valentin Kaelin

[Description des fonctionnalités du logiciel](#)

[Choix d'implémentation](#)

[Choix du parcours](#)

[Programme 1](#)

[Programme 2](#)

[Tests Effectués](#)

[Programme 1](#)

[Programme 2](#)

[Conclusion](#)

Description des fonctionnalités du logiciel

Pour ce laboratoire, nous travaillons sur deux programmes Qt contrôlant deux locomotives de manière concurrente. Chaque programme propose la possibilité de se connecter à une maquette et de gérer des locomotives miniatures ou d'utiliser un simulateur.

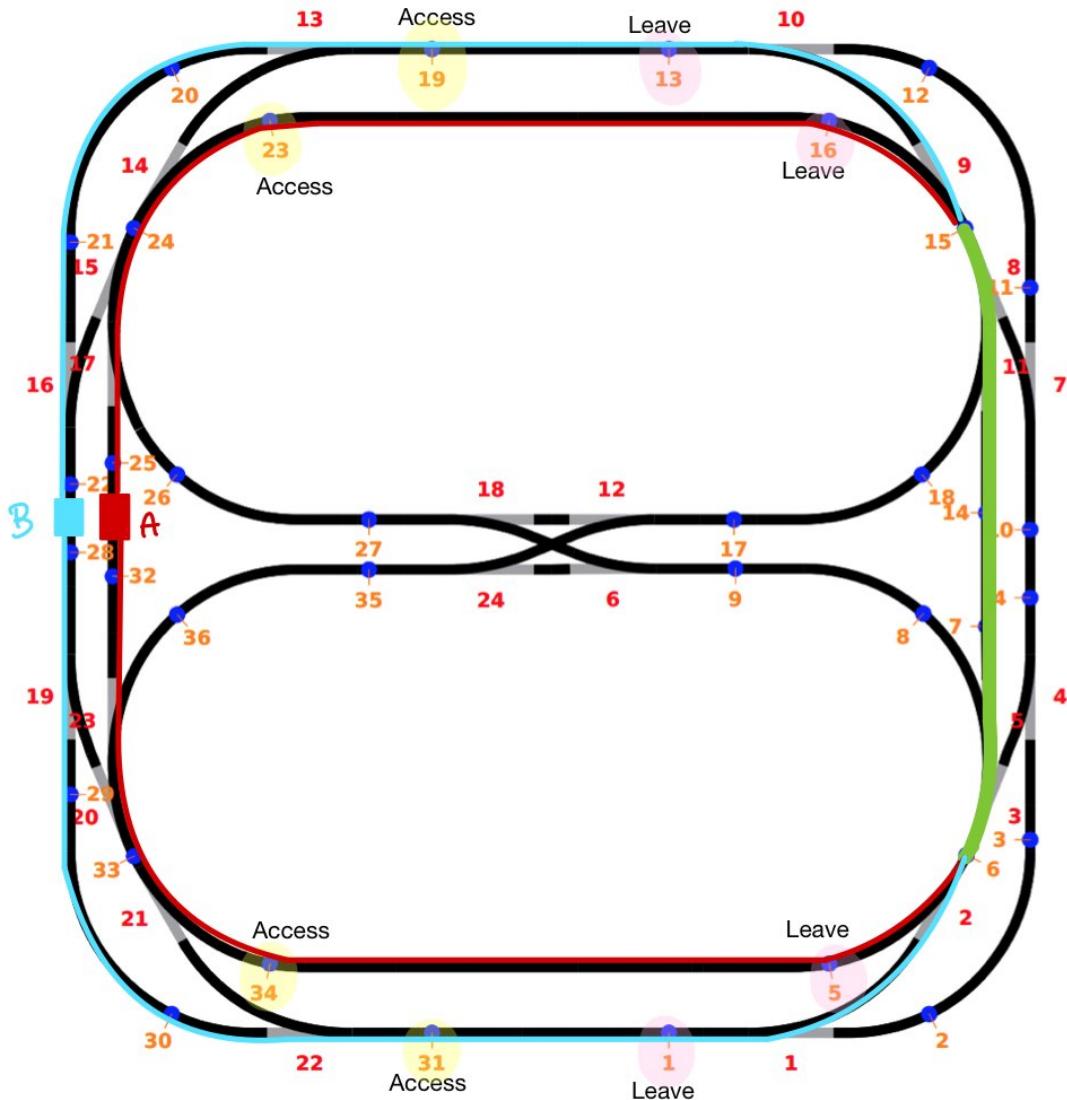
Pour le programme n°1, les locomotives ont un tracé circulaire et partagent un tronçon commun qu'il nous est demandé de gérer correctement pour éviter d'éventuelles collisions. Les locomotives font deux fois leur tracé avant de changer de sens, puis répètent l'opération jusqu'à l'arrêt du programme. Le programme comprend aussi un bouton d'arrêt d'urgence arrêtant immédiatement les deux locomotives.

Le programme n°2 fonctionne sur le même principe, mais possède en plus un système de priorité en fonction de l'id des locomotives et de leur position à l'arrivée d'un tronçon partagé. Concrètement, le système de priorité est géré à travers de requêtes émises par les locomotives approchant un tronçon partagé.

Choix d'implémentation

Choix du parcours

La première étape a été de choisir un parcours. Nous avons choisi d'utiliser le même tronçon partagé pour les deux programmes. Il est tracé en vert sur le graphique ci-dessous. Le parcours de B est en bleu clair et celui de A en rouge.



Programme 1

Les positions initiales des deux locomotives sont les positions proposées par le programme de test.

Parcours

Nous avons créé un vecteur d'entiers par locomotives afin de représenter leur parcours.

```
std::vector<int> pointsA = {25, 24, 23, 16, 15, 14, 7, 6, 5, 34, 33, 32};
std::vector<int> pointsB = {22, 21, 20, 19, 13, 15, 14, 7, 6, 1, 31, 30, 29, 28};
```

Ce qui nous a permis par la suite d'appeler la fonction `attendre_contact()` sur chaque point du vecteur afin de pouvoir suivre la progression de la locomotive. Afin de faire des tours du parcours à l'infini, nous remettons l'index parcourant le vecteur à 0 lorsque nous avons atteint le dernier élément du vecteur.

Par la suite, nous avons remarqué qu'appeler la fonction `attendre_contact()` sur tous les points du parcours était inutile et pouvait même poser des soucis dans certains cas. En effet, il est plus intéressant d'attendre le contact uniquement des points qui nous intéressent, comme l'entrée ou la sortie du tronçon partagé (nous en parlerons plus en détails dans la prochaine section).

Attendre tous les points posait des soucis si les vitesses des locomotives étaient trop élevées (ex : 16-18). Il arrivait que la distance de freinage soit plus élevée que la distance avec le prochain point de contact. La fonction

`attendre_contact()` attendait donc dans le vide un point que la locomotive avait en réalité déjà dépassé, ce qui cassait bien entendu la gestion de la section critique.

Tronçon partagé

La seconde étape a été de pouvoir annoncer à la locomotive où se trouvait la section critique et quand devait-elle s'arrêter si celle-ci était déjà occupée. Pour ce faire, nous avons décidé de créer une nouvelle classe que nous avons appelée `Route` afin de ne pas surcharger la classe `LocomotiveBehavior`. Chaque locomotive contient sa propre instance de la classe `Route` via sa classe `LocomotiveBehavior`. Cette classe `Route` contient plusieurs informations que nous allons explorer au long de ce rapport. Le vecteur de points de contacts décrit précédemment est notamment passé en paramètre du constructeur de cette classe.

En plus de ce vecteur, le constructeur de la classe deux vecteurs supplémentaires en paramètre :

- Un vecteur contenant les points du tronçon partagé
- Un vecteur des aiguillages à modifier (nous en parlerons plus tard)

Le constructeur s'occupe de calculer et de stocker plusieurs attributs dans la classe `Route` :

- Contact où demander l'accès au tronçon partagé
- Contact de sortie du tronçon partagé

Ces attributs sont stockés à double, car ils sont différents selon le sens de parcours de la locomotive. Afin d'éviter de devoir les passer en dur dans le constructeur, nous les calculer via des recherches sur les deux vecteurs :

```
// route: vecteur du parcours complet
// shared: vecteur du tronçon partagé
auto sectionStart = std::find(route.begin(), route.end(), shared.at(0));
auto sectionEnd = std::find(route.begin(), route.end(), shared.at(shared.size() - 1));

// Sens normal
contactStartShared = *std::prev(sectionStart, 2);
contactEndShared = *std::next(sectionEnd, 1);

// Sens inversé
contactStartSharedInversed = *std::next(sectionEnd, 2);
contactEndSharedInversed = *std::prev(sectionStart, 1);
```

Nous avons dû faire l'accès au tronçon partagé deux points de contact avant le véritable début de la section partagé à cause de la latence qu'ont les locomotives lors des freinages. En effet, si nous le faisons un point de contact après, les locomotives se rentrent dedans malgré l'arrêt bien demandé.

Section partagée

Notre code pour la section partagée n'est pas différent des différents exercices du cours de PCO. Le seul choix que nous avons fait a été d'utiliser un entier `nbwaiting` pour connaître le nombre de locomotives en attente d'accès à la section critique. Il aurait également été possible d'utiliser un simple booléen étant donné que nous avons que deux locomotives. Ce choix a été fait afin de faciliter l'évolutivité.

Aiguillages

Lorsque la locomotive accède un tronçon partagé, l'aiguillage se met à jour en fonction du chemin à prendre. L'information sur les aiguillages à changer est passée dans le constructeur de `Route` grâce à un vecteur de paires d'entiers. Par soucis de simplicité, nous avons utilisé un `using` pour exprimer la paire.

```
// Dans route.h
using RailwaySwitch = std::pair<int, int>;
```

```
// Dans cppmain.cpp
std::vector<RailwaySwitch> switchesA = {
    {10, DEVIE},
    {2, DEVIE},
    {9, DEVIE},
};

std::vector<RailwaySwitch> switchesB = {
    {10, DEVIE},
    {2, TOUT_DROIT},
    {9, TOUT_DROIT},
};
```

Changement de sens

Le changement de sens se fait, comme voulu, après deux tours complets du parcours. Le nombre de tours à faire est spécifié avec une macro `NB_TURNS` située dans `locomotivebehavior.h`. Elle est décrémentée à chaque fois qu'une locomotive passe sur le dernier contact (identifié avec la méthode `getTurnEnd()`) de son parcours. Lorsqu'elle arrive à 0, le sens est inversé. Le code suivant explique simplement le processus :

```
// Dans locomotivebehavior.cpp
...
attendre_contact(route.getTurnEnd());
nbTurns--;
if (!nbTurns) {
    inverse();
    nbTurns = NB_TURNS;
}
```

La méthode `LocomotiveBehavior::inverse()` effectue la procédure de changement de sens, c'est-à-dire, qu'elle appelle à la suite les méthodes `arreter()`, `inverserSens()` et `demarrer()` de la locomotive. Elle appelle aussi `inverse()`, de `Route` cette fois-ci, qui va indiquer le changement de sens à la classe `Route`.

Arrêt d'urgence

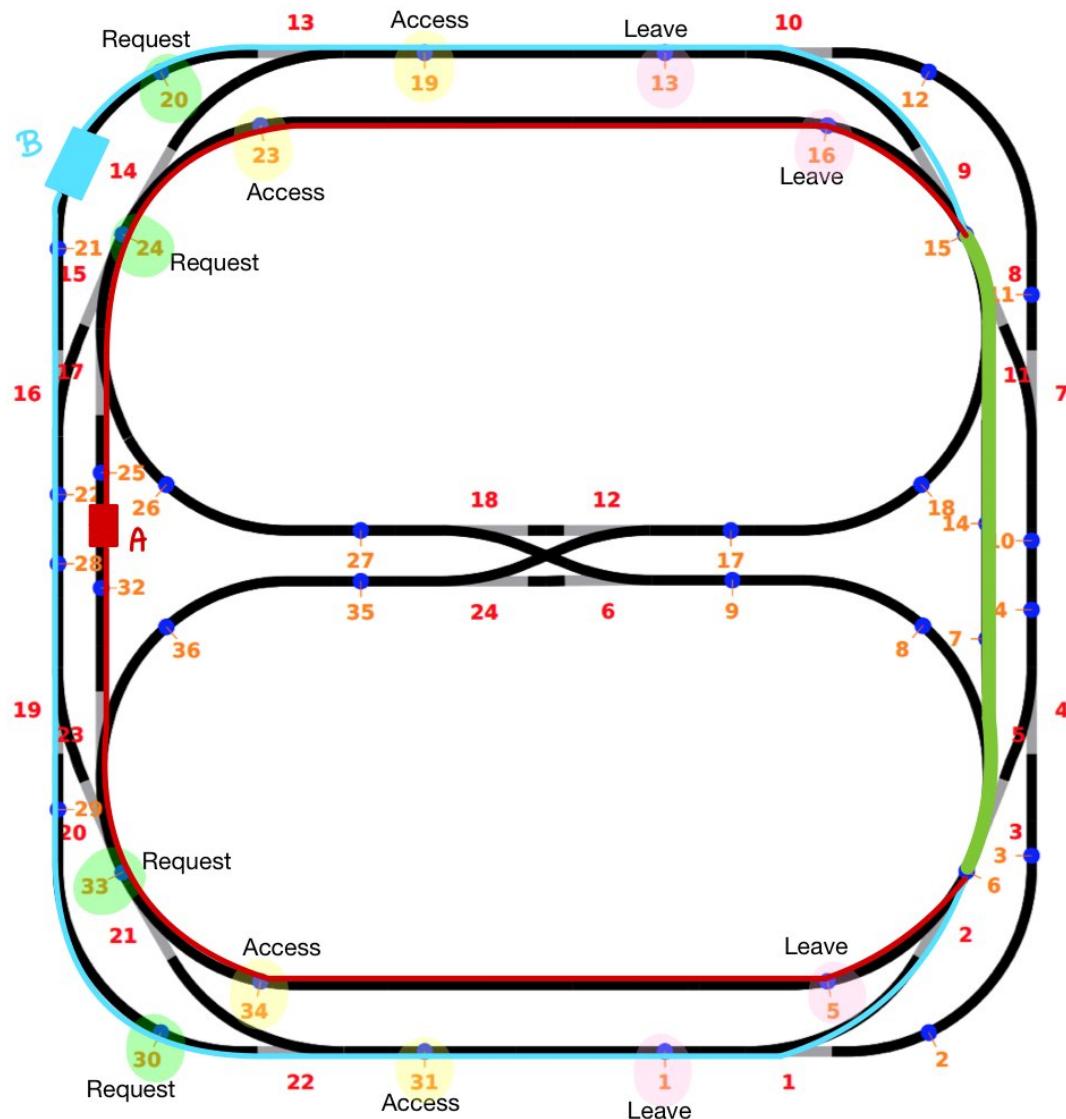
L'arrêt d'urgence des deux locomotives lors de l'appui sur le bouton a été implémenté de manière très triviale:

```
locoA.arreter();
locoB.arreter();
locoA.fixerVitesse(0);
locoB.fixerVitesse(0);
```

Nous mettons également la vitesse à 0 afin de régler un cas assez embêtant : si l'arrêt d'urgence est enclenché lorsqu'une locomotive se trouve à la fin du tronçon partagé et qu'elle le quitte avec l'inertie qu'elle a avant de s'arrêter, la locomotive en attente redémarre. Il aurait également été possible de faire un booléen `stop` vérifié avant de redémarrer la locomotive que nous mettions à `true` lors de l'arrêt d'urgence. Malheureusement, nous avons accès à aucune instance de classe dans la fonction d'arrêt d'urgence et réaliser cette fonctionnalité en static nous a semblé peu élégant.

Programme 2

Le code du programme 2 ressemble fortement à celui du premier. Nous allons donc nous contenter d'expliquer les modifications que nous avons dû apporter par rapport au premier programme.



Changement dans le parcours

La position des locomotives est légèrement décalée par rapport au premier programme pour faciliter les tests des requêtes et la bonne application des priorités.

Ajout des contacts de requêtes et points d'entrées

Le premier besoin a été de stocker les points de requêtes au tronçon partagé dans la classe `Route`. Comme on le voit sur le schéma ci-dessus, ces points se trouvent un point avant les points d'accès. Il a donc été facile de les calculer comme les autres points de contact:

```
// Sens normal
contactRequestShared = *prev(sectionStart, 3);

// Sens inversé
contactRequestSharedInversed = *next(sectionEnd, 3);
```

Il a également fallu stocker l'id de la locomotive et son point d'entrée au tronçon critique. L'id est passé directement au constructeur de la classe `LocomotiveBehavior` et le point d'entrée est calculé dans la boucle de la méthode `run()` de la classe `LocomotiveBehavior`. A chaque fois qu'une locomotive change de sens, on inverse son point d'entrée.

Système de priorité

Les principaux changements ont été réalisés dans la classe de la section partagée `SharedSection`. Nous stockons plusieurs informations supplémentaires dans la classe :

```
bool locoARequest, locoBRequest;
EntryPoint locoAEntry, locoBEntry;
```

Ces variables sont assignées dans la méthode `request()` en fonction de l'id de la locomotive passé en paramètre.

Les booléens servent à savoir si les locomotives spécifiques ont fait une requête d'accès à la section partagée et les `EntryPoint` sont là pour vérifier si les locomotives arrivent du même sens ou non. Nous remettons à `false` les booléens une fois l'accès à la locomotive autorisé et effectué (fin de la méthode `getAccess()`).

Afin de donner accès à la bonne locomotive, nous appelons une méthode privée dans `getAccess()` :

```
bool canAccess(LocoId locoId) {
    if (occupied) return false;

    if (!locoARequest || !locoBRequest) return true;

    return locoId == LocoId::LA ?
        locoAEntry == locoBEntry :
        locoAEntry != locoBEntry;
}
```

Si une seule des deux locomotives a fait une requête, elle obtient directement accès. Si par contre une locomotive se trouve déjà dans la section critique, l'autre doit forcément attendre. Pour finir, si les deux locomotives viennent de faire une requête, on applique les règles de priorité de l'énoncé.

Tests Effectués

Programme 1

Description du test	Résultat attendu et observé
Lancement de la simulation en chargeant le parcours de chaque locomotive	Les locomotives passent sur les points de leur parcours.
Appel de la fonction <code>attendre_contact()</code> sur le prochain contact	Le prochain point de contact du parcours de la locomotive s'allume en vert. L'exécution du code est bloquée tant que la locomotive n'a pas atteint le point.
Comportement après deux tours complets	La locomotive s'arrête et repart dans l'autre sens.
Une locomotive approche un tronçon partagé libre	La locomotive n'effectue pas d'à-coup, elle traverse simplement le tronçon sans s'arrêter.
Une locomotive approche un tronçon partagé occupé	La locomotive s'arrête et attend que le tronçon se libère. Une fois le tronçon dégagé, la locomotive repart.
Appui sur le bouton d'arrêt d'urgence	Les locomotives en mouvement s'arrêtent.
Arrêt d'urgence alors qu'une locomotive se trouve dans le tronçon partagé et en sort avec son inertie restante	La locomotive s'arrête et celle attendant son accès au tronçon partagé ne redémarre pas.

Description du test	Résultat attendu et observé
Comportement des aiguillages des tronçons partagés	Les aiguillages sont changés lorsqu'une locomotive a eu l'accès à un tronçon partagé.

Programme 2

Comme pour les détails d'implémentation, nous nous contentons de lister les tests qui diffèrent par rapport au programme 1. Nous avons bien entendu testé les cas du programme 1 avec le code du programme 2.

Description du test	Résultat attendu et observé
Une locomotive fait une requête pour un tronçon partagé libre	La locomotive se voit attribuer l'accès et ne s'arrête pas avant de rentrer au tronçon.
Deux locomotives venant du même point d'entrée font une requête en même temps	La locomotive qui a la priorité (LA) obtient l'accès au tronçon. L'autre locomotive s'arrête en attendant.
Deux locomotives venant de points d'entrées opposés font une requête en même temps	La locomotive qui a la priorité (LB) obtient l'accès au tronçon. L'autre locomotive s'arrête en attendant.

Conclusion

Nous sommes plutôt satisfaits d'avoir finalement un code fonctionnel. Avoir pu tester son code sur une maquette réelle est très satisfaisant (une fois celle-ci prise en main). Nous voyons cependant plusieurs points d'amélioration qu'il aurait été possible d'implémenter :

Notre code ne gère pas bien le fait d'avoir plus de deux locomotives ou plus d'une section partagée. Nous pourrions par exemple utiliser des vecteurs à la place de variables dédiées par locomotive dans la classe `SharedSection`. Pour gérer plusieurs sections critiques, il aurait fallu bien remanier la classe `Route` afin qu'elle prenne des vecteurs à deux dimensions ainsi que modifier les getters en précisant quel tronçon partagé nous intéressait.