UTBM

Rapport de projet

LO41

Etienne Gartner

Printemps 2016

MODELISATION

RESEAU DE PETRI

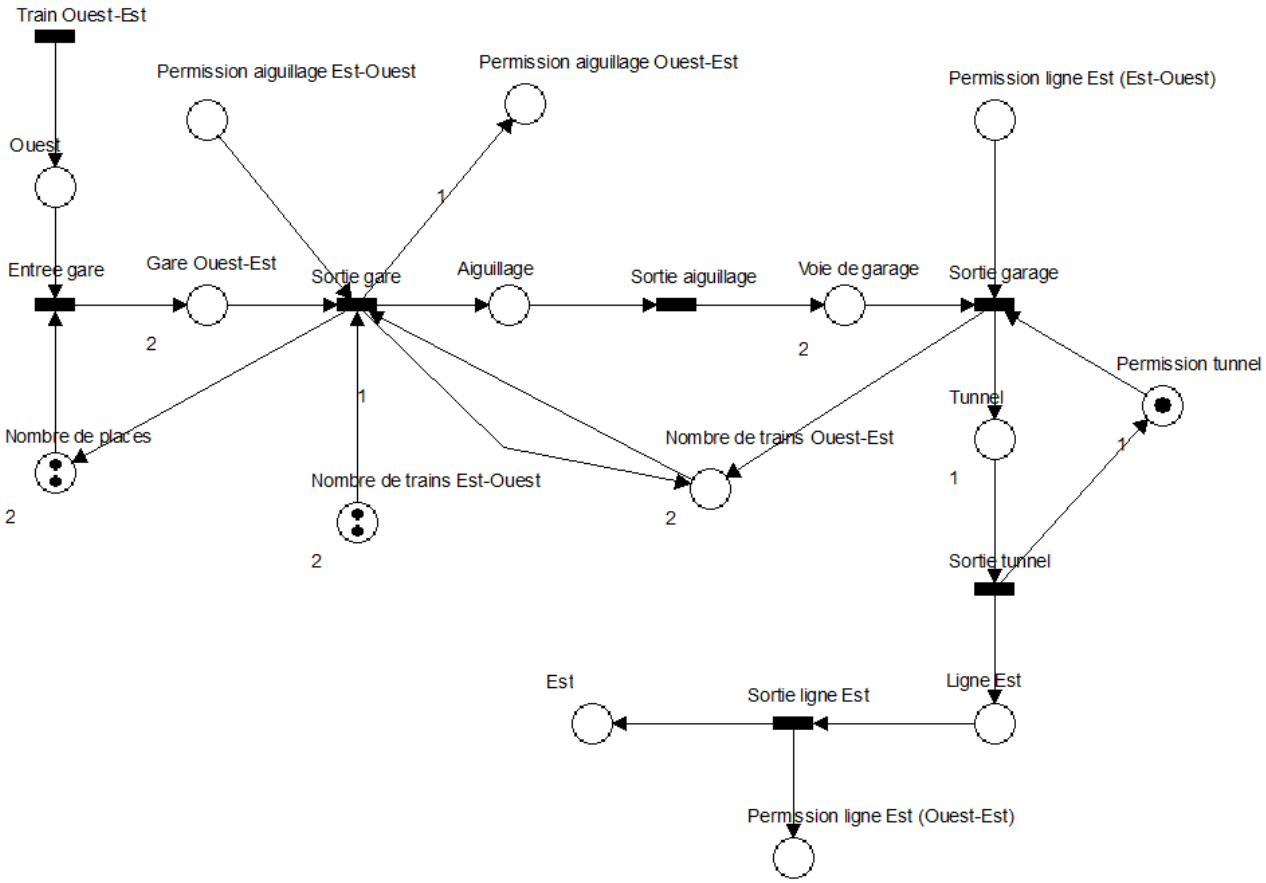

FIGURE 1 : RESEAU DE PETRI POUR UN TRAIN ALLANT VERS L'EST

Ci-dessus un réseau de pétri simpliste pour un train TGV ou Grande Ligne se dirigeant vers l'Est. Pour commencer, le train entre en gare lorsque le nombre de places est inférieur à 2. Puis, pour sortir de la gare, il doit attendre 3 autorisations. En effet, il faut qu'il n'y ait aucun train dans l'aiguillage en sens inverse (Est-Ouest), il ne doit pas non plus y avoir de trains dans le garage correspondant dans le sens inverse et il ne doit pas y avoir plus de deux trains dans le garage correspondant dans le même sens. Une fois ces conditions vérifiées, le train peut entrer sur l'aiguillage et met à jour la permission de l'aiguillage correspondant au sens inverse (qui n'est pas représenté ici). De plus, il augmente directement le nombre de trains dans le garage pour son sens : il s'agît d'une réservation. En effet, s'il met à jour la permission lorsqu'il sort de l'aiguillage seulement, un train en sens inverse pourrait arriver en gare pendant qu'il est sur l'aiguillage.

Pour finir, pour sortir d'une voie de garage, un train doit attendre qu'il n'y ait plus personne sur le tunnel et qu'il n'y ait personne sur les lignes Est arrivant en sens inverse. Une fois le train sorti de la ligne Est, on peut mettre à jour la permission de cette ligne pour les trains de sens contraire.

IMPLEMENTATION

INTRODUCTION

Pour ce problème, j'ai décidé d'utiliser des threads pour représenter les trains ainsi que des moniteurs pour assurer leur synchronisation. Chaque thread dispose de sa propre structure *train* contenant le numéro du train, son sens, son type et sa position actuelle. Cette structure est initialisée aléatoirement au début du programme.

Ensuite, j'ai implémenté plusieurs fonctions gérant chacune une partie du système (gare, aiguillage P0, aiguillage P1, tunnel, voies). Ces fonctions disposent de traitements différents selon le sens du train et parfois selon son type. Par exemple, si un train va en direction de l'Est, il fera appel aux fonctions *gare*, *aiguillage_P0, aiguillage_P1, tunnel* puis *voies*. Un train en direction de l'Ouest fera appel à ces fonctions dans l'ordre inverse.

MECANISMES DE SYNCHRONISATION

Pour synchroniser les trains, j'utilise plusieurs compteurs implémentés en tant que variable globales. Ils représentent les ressources critiques. Pour garantir le bon fonctionnement du programme, j'utilise des mutex et des moniteurs.

Chacune des permissions présentes sur le réseau de pétri ci-dessus correspond à un compteur. Par exemple, un TGV en gare en direction de l'EST sera mis en attente selon le nombre de trains arrivant en sens inverse dans l'aiguillage et du nombre de train attendant en sens inverse dans la voie de garage pour TGV. Cette mise en attente est effectuée à l'aide de la fonction *pthread_cond_wait*. Une fois que le train a l'autorisation de passer, il verrouille un mutex spécifique sur l'aiguillage et incrémente le compteur correspondant à son sens. Il incrémente également le nombre de train sur la voie de garage (il réserve sa place).

Lorsqu'une ressource critique change de valeur et lorsqu'il est nécessaire, j'effectue un appel à la fonction *pthread_cond_signal* afin de réveiller les threads en attente sur la condition correspondante.

PRIORITE DES TRAINS

Dans un premier temps, j'ai choisi d'implémenter la priorité en ajoutant de nouveaux compteurs pour les TGV et les trains grandes lignes. Ainsi, tant qu'il restait au moins un tgv en gare dans son sens, un train grande ligne attend (via un *pthread_cond_wait*). Tant qu'il reste au moins un train grande ligne en gare dans son sens, un train marchand attend. Il en va de même en gare dans le sens Ouest-Est.

Cependant, une fois cette méthode implémentée, j'ai constaté des problèmes d'interblocage dans le programme. Ne trouvant pas la cause exacte de ce problème, j'ai donc opté pour une autre méthode.

Finalement, j'ai décidé d'utiliser la priorité des threads eux-mêmes. Ainsi, lorsque plusieurs threads attendront sur la même ressource critique, le thread ayant la priorité la plus haute y accédera en premier (il sera sélectionné en priorité par le scheduler). J'ai donc utilisé la fonction *pthread_setschedprio* pour fixer la priorité des threads contenant un TGV à 30, de ceux contenant un train grande ligne à 20 et ceux représentant un train marchand à 10.

DEMONSTRATION DU PROGRAMME

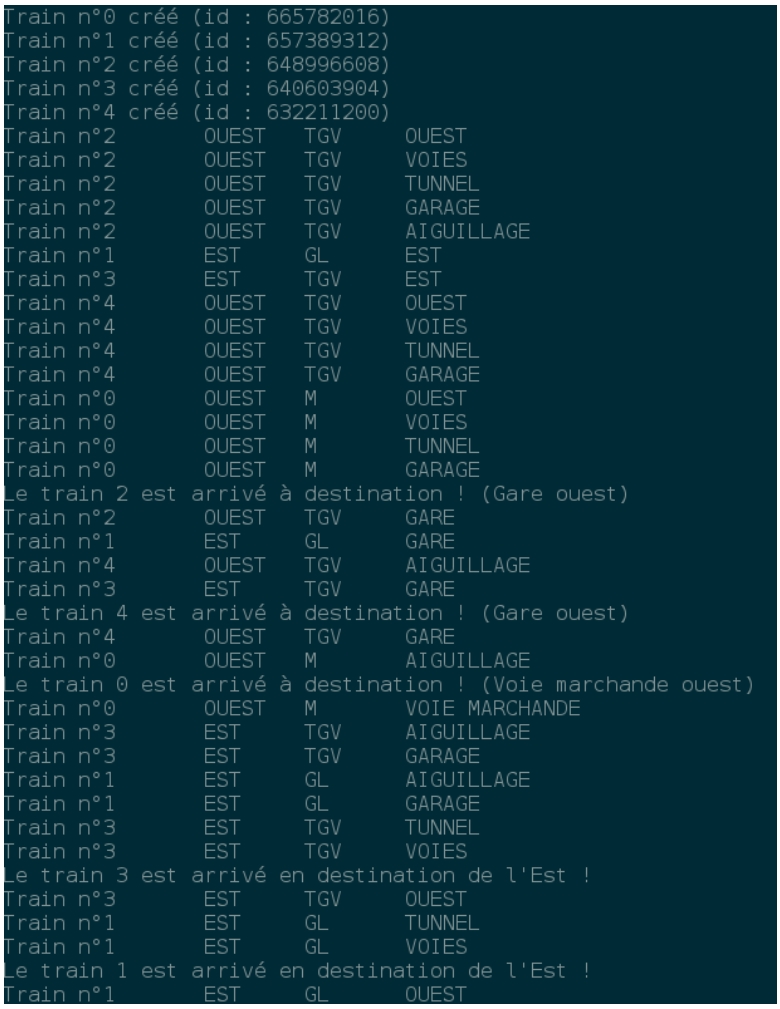

FIGURE 2 : DEMONSTRATION DU PROGRAMME POUR 5 TRAINS

Sur cette instance aléatoire, on est en présence de 5 trains :

- Le train n°0 est un train marchand allant vers l'Ouest
- Le train n°1 est un train Grande Ligne allant vers l'Est
- Le train n°2 est un TGV allant vers l'Ouest
- Le train n°3 est un TGV allant vers l'Est
- Le train n°4 est un TGV allant vers l'Ouest

Ici, on observe que le train n° 2 est le premier à commencer. Il débute à la position Ouest. Comme il n'y a encore personne sur le garage TGV en sens inverse, ni dans le tunnel, ni sur la voie en sens inverse, il effectue donc le trajet Voies -> Tunnel -> Garage. En d'autres mots, les conditions ne sont pas vérifiées pour qu'il se mette en attente avec un *pthread_cond_wait*.

Comme il n'y a encore aucun train dans la gare ni sur l'aiguillage, il enchaîne directement en se positionnant sur l'aiguillage. Le train n°4 suit la même logique et s'arrête dans le garage approprié. Il en va de même, ensuite, pour le train n°0. A ce moment, le train n°2 arrive à destination (c'est-à-dire à la gare ouest). Les trains n°1 et n°3 décident de passer en gare est (rien ne leur en empêche) pendant que le train n°4 prend à son tour l'aiguillage pour arriver à destination. Le train n°0 qui était en gare passe alors par l'aiguillage pour arriver en voie marchande et donc arriver à destination.

Jusque-là, le train n°3 était en attente à la gare. En effet, le train n°4 empruntait la voie de garage TGV dans l'autre sens, puis l'aiguillage pour la gare. A son arrivé en gare, il émet un *pthread_cond_signal* qui va réveiller le train n°3 en attente ainsi que le train n°1 (qui attendait lui la libération de l'aiguillage).

Les trains n°3 et n°1 passent donc tour à tour dans l'aiguillage pour arriver en voie de garage. Le train n°3 étant prioritaire, c'est donc lui qui passe en premier. Il est ensuite à nouveau prioritaire et passe dans le tunnel en premier pour finalement arriver à destination. En réalité, comme il est le premier arrivé en voie de garage, il verrouille ici le mutex en premier. Comme il n'y a plus personne dans le tunnel, le train n°1 peut à son tour passer et finir sa route.

Le programme se termine donc sans incident : il n'y a pas eu de collisions, ni d'interblocage.

CONCLUSION

Ce projet m'a permis de mettre en pratique les notions vues en cours de LO41 et de les approfondir. Plus particulièrement, les notions concernant les mutex et les moniteurs. En effet, dans un système aussi complexe, un seul mutex oublié ou mal positionné peut remettre en question toute l'intégrité du système. Il est également important de bien utiliser les moniteurs pour éviter le phénomène d'interblocage, rencontré plusieurs fois au cours du projet.

Pour finir, le programme fonctionne correctement pour un nombre de train relativement important (testé jusqu'à 500). Le programme prend également en charge la gestion de la priorité des trains.

Néanmoins, il n'est pas exempt de défaut. Il sera par exemple possible d'améliorer l'affichage qui est actuellement peu lisible. Une idée d'amélioration serait d'utiliser des caractères spéciaux pour modifier des lignes déjà affiché : ainsi, on pourrait imaginer un affichage dynamique où les trains seraient placés en temps réel dans les différentes zones.

Une autre amélioration consisterait à gérer les signaux d'interruptions envoyés au programme. Il serait également possible d'affecter des vitesses aux trains pour obtenir une simulation plus proche de la réalité.