UNIVERSITÉ PIERRE ET MARIE CURIE

Projet STL

Contrôle de trafic ferroviaire

Auteurs:
AFFES Mohamed Amin
KOBROSLI Hassan

Encadrants:
MANOURY Pascal
LESUEUR Bruno

Table des matières.

Problématique

Dans le monde de l'industrie, nous devons parfois interagir avec des applications distantes stockées sur un serveur afin de leur communiquer des instructions ou d'effectuer une maintenance éventuelle, et nous n'avons pas toujours accès à l'ensemble des fichiers de l'application et dans certains cas n'avons aucun accès interne à celle-ci. Et même si l'accès était possible, ceci ne serait pas une bonne solution car il représenterait une perte de temps considérable qu'on ne peut se permettre, surtout pour des projets de grande importance. La solution est donc de définir des spécifications qu'on fournit par la suite au développeur afin qu'il puisse à travers eux intervenir sur l'application distante.

Introduction

Dans le problème ici présenté, nous devons développer un programme interagissant avec un moniteur (serveur) gérant un circuit ferroviaire distant, en suivant un protocole de communication (fourni en annexe). L'objectif est de définir des règles de sécurité permettant un fonctionnement automatisé du circuit distant, en évitant les collisions des trains et en prenant en compte les contraintes qu'impose l'architecture du circuit, ainsi que les limites de nos solutions qui sont proposées pour une architecture bien définie du circuit. L'ajout de nouveaux types d'éléments au circuit par exemple entraînerait donc le dysfonctionnement de notre solution. Ce dernier point sera détaillé dans la suite du rapport.

Nous allons dans ce rapport tout d'abord présenter l'organisation de notre travail au cours de ce projet, suivie d'une description générale du dispositif et du système de communication contrôleur/moniteur qui sera finalisé par la présentation d'un diagramme de composant représentant l'architecture de notre solution. Puis nous aborderons le système de gestion des règles de circulation et pour finir, parler de l'interface graphique de l'application.

1 Organisation

Cette partie est la plus importante lors de la conception d'une application, car elle permet de faciliter la conception de la solution, et la collaboration avec d'autres personnes pour cette conception en ayant le minimum de problèmes ou retards possibles. Mieux l'organisation est pensée, mieux est le respect des délais définis et la qualité de la solution proposée au client.

1.1 Travail collaboratif

Nous commençons par aborder le premier point dans cette phase d'organisation qui est le travail collaboratif, et donc la façon dont nous avons travaillé pour la conception de ce projet.

Tout d'abord, nous avons choisi comme outil de travail collaboratif « Git », et ceci pour la facilité qu'il procure pour un travail collaboratif grâce à son système de branches et de synchronisation, permettant de travailler en parallèle sur le même projet sans que l'avancement de l'un ne perturbe celui de l'autre.

Concernant notre façon de travailler nous débutions assez souvent à développer ensemble le travail de la partie du projet concerné à ce moment-là. Puis après s'être fixés les grandes lignes et partagés les taches, chacun continuait sur sa partie tout en prévenant l'autre lors d'un ajout d'une partie de code au projet, afin que chacun soit au courant de l'avancement de son collègue et des modifications qu'il a apportées. Cette méthode nous a donc permis de travailler de façon efficace et d'obtenir un gain au niveau du temps de développement et de profiter chacun des remarques de l'autre afin d'améliorer notre solution et de faire en sorte que les parties développées se complètent et s'accordent pour arriver à une solution finale complète et homogène.

1.2 Planning

Nous avons rédigé un planning répartissant la charge de travail à réaliser chaque semaine, ainsi que le pourcentage accompli pour chaque étape. Ceci nous a permis tout d'abord de fixer les objectifs à atteindre à la fin de chaque semaine, et de donner un délai approximatif minimal pour la réalisation de la solution.

Nous avons aussi dû faire face à quelques manquements au niveau du respect de certains délais dus à des erreurs dans les données fournies par le moniteur. Ces retards se sont répercutés sur les semaines suivantes, mais ont finalement pu être rattrapés. Ceci nous a été très formateur car dans la vie professionnelle de telles erreurs peuvent arriver, et donc le programmeur se doit de récupérer le retard survenu sur son planning et d'éliminer la cause du problème.

Le planning permettait aussi à nos encadrant de connaitre notre avancement et de nous réorienter en cas de non-respect de la demande décrite dans le cahier des charges fourni par nos encadrants dans notre solution.

1.3 Réunions

Le dernier pilier de cette organisation est les réunions avec nos encadrants. Celles-ci nous ont permis de présenter notre avancement, prendre en compte leurs remarques et conseils, noter les modifications nécessaires si besoin à réaliser sur notre solution présentée si certains points ne correspondaient pas à la demande ou pouvaient être améliorés. Enfin, après chaque réunion, il nous fallait envoyer un compte rendu à nos encadrants notant les points essentiels relevés, afin qu'ils puissent être pris en compte par nous ainsi que nos encadrants et de garder une trace.

Ceci nous permettait d'aiguiller notre façon de travailler et notre solution afin qu'elle soit la plus proche possible de ce que nos encadrants attendaient de nous. Ces réunions étaient hebdomadaires dans un premier temps, puis convenues selon les besoins.

2 Description générale du dispositif

Le réseau ferroviaire est équipé de capteurs, de feux bicolores, d'aiguillages et naturellement, de trains. Tous ces équipement sont identifiés par un identifiant unique.

Les trains sont soit en marche ou à l'arrêt, et ont une direction de circulation leurs permettant de circuler dans un sens ou dans l'autre (marche avant ou arrière). Les capteurs émettent une impulsion au passage d'un véhicule. Les feux bicolores sont réalisés par deux leds (une rouge, une verte). Un aiguillage correspond à deux voies potentielles de circulation et permet d'ouvrir effectivement l'une ou l'autre. Lorsqu'une des voies d'un aiguillage est ouverte, l'autre est fermée.

Un aiguillage peut être franchi dans un sens ou dans l'autre. De ce point de vue, il y a deux types d'aiguillage : les aiguillages 1-2 qui divisent une voie en deux, et les aiguillages 2-1 qui joignent deux voies en une.

Le système final visé comporte tous les équipements cités précédemment, et en plus de ça :

- Un module « sprog » contrôlant le véhicule, il relaye les commandes de marche et d'arrêt vers les véhicules.
- Un microcontrôleur Arduino pour relayer les commandes d'allumage et d'extinction des feux ainsi que les signaux d'activation des ampoules représentant les capteurs.

Il est à noter que nous n'avons aucun accès direct aux deux éléments précités. Le gestion de ceux-ci ne fait pas partie de nos attributions.

2.1 Le composant moniteur

Le composant moniteur (serveur), est le système gérant le réseau ferroviaire physique, il commande la couleur des feux ainsi que les mouvements (marche, arrêt) et directions (avant, arrière) des trains. Il gère aussi les positions des aiguillages dans le circuit.

Le composant moniteur communique avec le contrôleur par une liaison TCP/IP selon le protocole PCF décrit en annexe, afin de lui communiquer l'état du circuit, ou le notifier d'un changement d'état dans ce dernier. Il reçoit aussi des commandes du contrôleur lui demandant d'effectuer une action.

Le composant moniteur communique avec le microcontrôleur pour changer l'état d'un feu, et avec le module « sprog » pour commander les véhicules (marche, arrêt, ou changement de direction). Il reçoit aussi les notifications du microcontrôleur indiquant le passage d'un train sur un capteur. Le composant moniteur est relié au module « sprog » et au microcontrôleur par des liaisons série (USB).

Le composant moniteur peut aussi simuler à lui seul tous ces événements et les gérer par lui-même, ce qui lui permet de communiquer uniquement avec le module contrôleur. C'est donc avec des circuits simulés par le moniteur cette que nous avons pu tester notre solution, en ayant seulement un accès au moniteur à disposition.

Le composant moniteur ne gère pas lui-même toutes les règles de sécurité du circuit et est totalement soumis aux ordres du composant contrôleur.

2.2 Le module contrôleur

Le module contrôleur est le module gérant les règles de sécurité nécessaires au bon trafic des trains, et censées prévenir tout risque de collision. C'est selon les ordres du contrôleur appliquant ces règles que le module moniteur pourra correctement gérer le circuit.

Le module contrôleur commence par demander l'état du circuit au moniteur, puis lui envoie une suite de commandes pour démarrer le trafic. Il attend ensuite de recevoir des messages provenant du moniteur indiquant l'activation d'un capteur par un train. A chaque notification d'activation de capteur, le contrôleur répond au moniteur une suite de commandes indiquant les actions à effectuer (arrêt, marche d'un train, changement de la couleur d'un feu) pour le bon fonctionnement du circuit et ainsi éviter les collisions de train. Ces deux dernières actions sont renouvelées durant toute la durée de vie de notre programme.

- Le schéma suivant représente ce qui a été présenté dans les section précédentes :

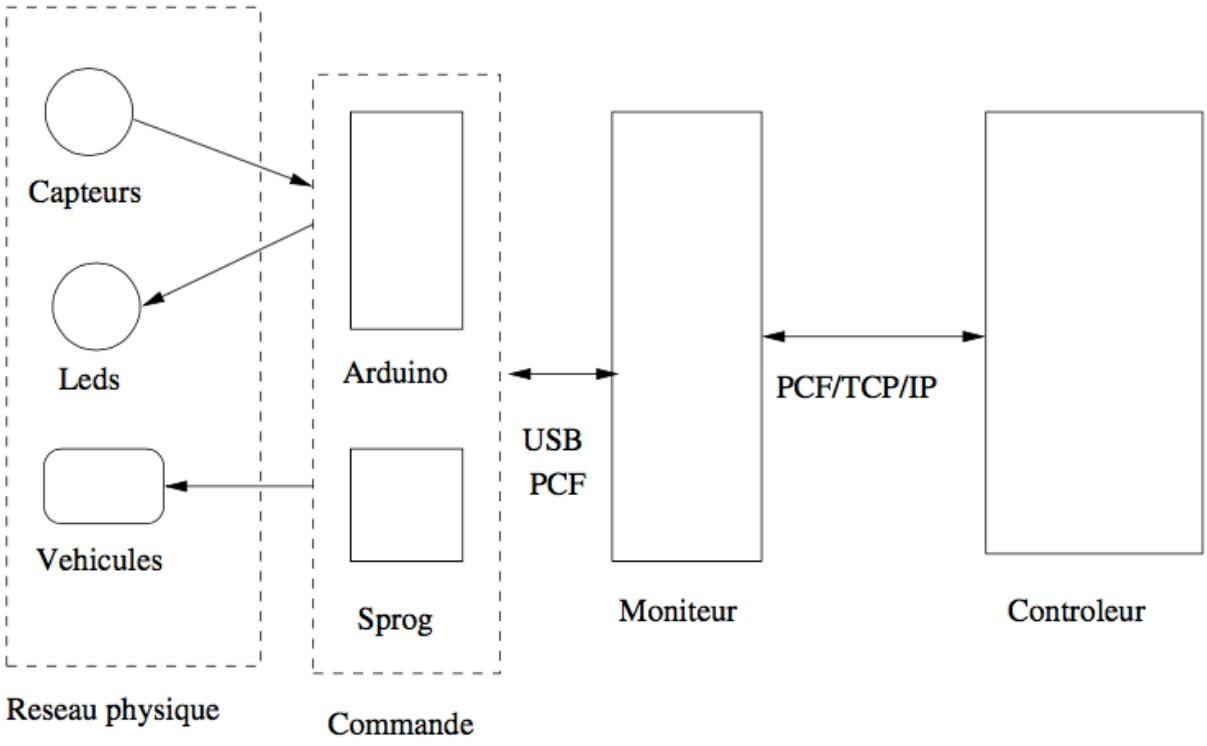
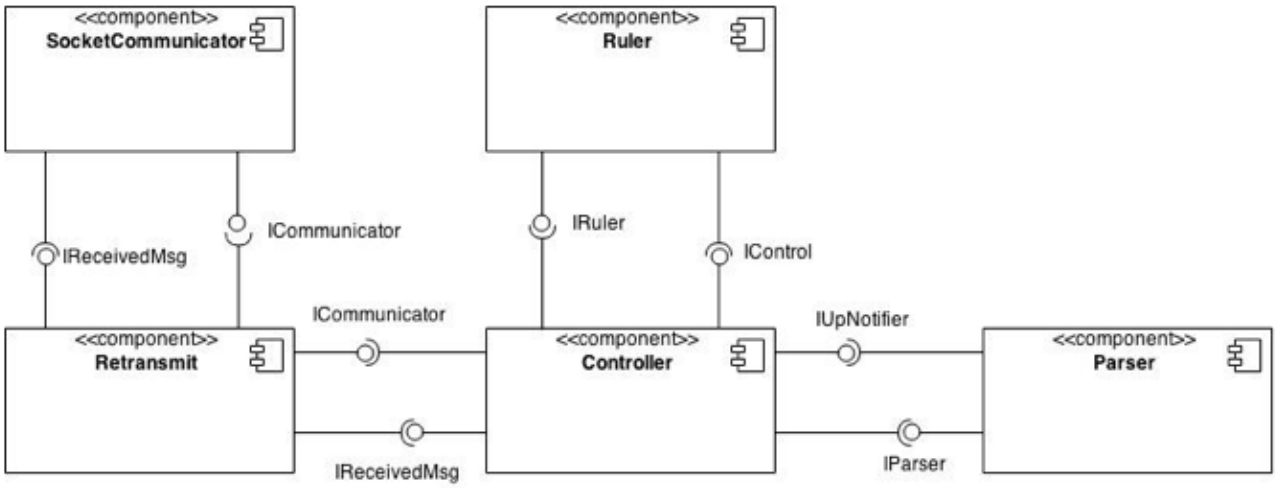


Schéma représentant la communication entre les différents modules.

2.3 Architecture de la solution

Nous allons présenter dans cette partie deux diagrammes de composant, le premier a été établi dans un premier temps. Puis, lors de son implémentation nous nous sommes rendu compte avec l'aide de nos encadrants de certaines modifications qu'il fallait lui apporter et qui seront développées par la suite.

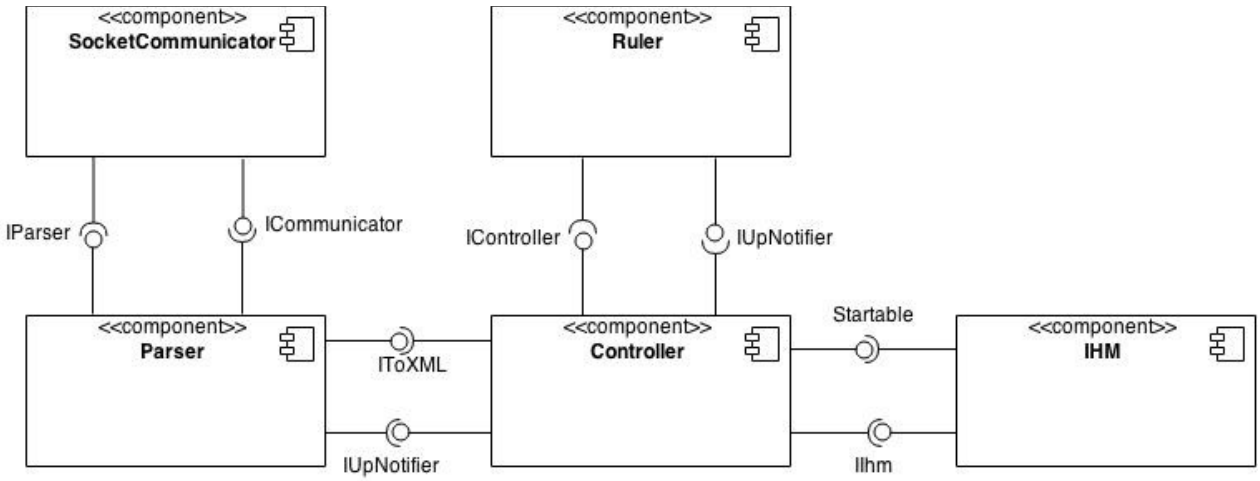
Le premier diagramme établi en début de projet est le suivant :



Le rôle du composant **Retransmit**, supprimé par la suite, était de gérer l'acquittement des messages envoyés : si un message n'était pas acquitté par le serveur au bout d'un temps défini, le message était renvoyé. Ceci a été abandonné car on considérait que le simulateur recevait tous les messages, ce qui s'est avéré vrai pendant tous nos essais.

La disposition du composant **Parser** n'était pas des meilleures car celui-ci est chargé d'analyser les messages reçus. Les messages devaient donc passer par le **Controller** pour aller vers le **Parser** qui les traduisait puis notifiait le **Controller**. Ce composant a donc été déplacé.

Le diagramme final est donc le suivant :



Voici une brève présentation des différents composants et des interfaces qu'ils offrent ou requièrent d'autres composants, en citant pour chacun les raisons de la modification ou suppression lors du passage au deuxième diagramme si concerné.

Composant SocketCommunicator

Ce composant s'occupe de la communication client/serveur entre le contrôleur et le moniteur. Il établit la connexion entre les deux, puis permet l'envoi et la réception des messages entre le client et le serveur. L'échange des messages est asynchrone.

Il offre l'interface **ICommunicator** permettant la communication avec le moniteur (connexion, déconnexion et envoi de messages au moniteur). Il requiert l'interface **IParser** du composant **Parser** permettant de parser les messages provenant du moniteur, et les rendre utilisables et compréhensibles par notre programme.

Composant Parser

Ce composant s'occupe de la traduction des messages du contrôleur pour qu'ils soient compréhensibles par le moniteur et la traduction (parse) des messages de ce dernier pour qu'elles soient compréhensibles par notre contrôleur.

Il offre l'interface **IParser** au composant **SocketComminicator** pour la raison citée précédemment, et requiert de ce dernier l'interface **ICommunicator** lui

permettant l'envoi de messages vers le moniteur. Il offre également l'interface **IToXML** au composant **Controller** lui permettant d'envoyer une action au moniteur tel que le changement de l'état d'un train par exemple. Il requiert du composant **Controller** l'interface **IUpNotifier**, permettant de notifier le composant **Controller** lors de la réception d'un signal provenant d'un capteur du moniteur.

Composant Ruler

Ce composant définit les règles de sécurité liés à la circulation des véhicules ferroviaires, il suit donc une stratégie en contrôlant les couleurs des feux, le changement des états des trains, et des positions des aiguillages afin d'éviter la collision de véhicules.

Il offre l'interface **IUpNotifier**, permettant au composant **Controller** de le notifier lorsqu'il reçoit un signal provenant d'un capteur du moniteur (appel à la méthode « notifyInit() » en cas d'initialisation, ou « notifyUp() » sinon). Il requiert du composant **Controller** l'interface **IController**, permettant au composant **Ruler** de lui communiquer les instructions à envoyer au moniteur.

Composant IHM

Ce composant gère l'interface graphique affichant notre circuit, ainsi que les changements d'états des équipements du circuit en temps réel (changement de couleur de feu, déplacement du train ...).

Il offre l'interface **IIhm** au composant **Controller**, permettant à ce dernier de communiquer les changements des états des équipements du circuit, en lui demandant d'effectuer les changements adéquats sur l'interface comme le changement de la couleur d'un feu par exemple. Il requiert l'interface **IStartable**, lui permettant de demander au composant **Controller** d'effectuer la phase d'initialisation (expliqué dans la suite du rapport), qui se termine par la mise en route du trafic (méthode « start »). Elle permet aussi de demander l'arrêt du trafic et de couper la connexion client/serveur entre le contrôleur et le moniteur (méthode « stop »).

Composant Controller

Ce composant est le composant principal de notre programme, il communique avec les trois composants « **Parser, Ruler** et **IIhm** » en traitant leurs demandes et en envoyant les actions à effectuer au composant concerné, il représente donc le coeur de notre programme qui effectue le lien entre ces trois composants.

Il fournit les interfaces « **IUpNotifier, IController, IStartableStoppable** » et requiert les interfaces « **IToXML, IUpNotifier** et **IIhm** ». Leurs utilisations ont étés expliquées dans la présentation des 3 composants précédents.

Toutes les interfaces citées sont fournis en annexe.

3 Système de communication contrôleur/ moniteur

Il existe dans ce projet deux types de protocole :

- Le premier est un **protocole de commandes ferroviaire** et est défini pour la communication entre le moniteur et le module de contrôle. Nous ne nous intéressons pas à ce protocole dans ce rapport puisqu'il n'est pas de notre ressort. Néanmoins, nous l'avons tout de même mis en annexe.
- Le deuxième est un **protocole de commandes ferroviaire** et est défini pour la communication entre le moniteur et le module de contrôle. Nous ne nous intéressons pas à ce protocole dans ce rapport puisqu'il n'est pas de notre ressort. Néanmoins, nous l'avons tout de même mis en annexe.

3.1 Protocole de contrôle ferroviaire

La solution retenue ici a été de suivre un protocole défini par le moniteur nommé Protocole de Contrôle Ferroviaire « PCF ». Les messages sont au format XML. On peut retrouver en annexe la description de ce protocole avec la DTD correspondante.

Ce protocole est donc constitué d'un ensemble de balises imbriquées à des niveaux différents, ayant chacun un rôle différent et un identifiant permettant de les différencier. Le type de la balise de premier niveau **PCF** permet de déterminer si le message est une requête ou une réponse. Certaines balises servent à décrire l'architecture du circuit (ordres des capteurs, aiguillages, feux), d'autres la position initiale des trains, d'autres signalent l'activation d'un capteur.

Le composant **Parser** permet de traduire une commande du contrôleur au moniteur suivant ce protocole.

Cette façon de faire facilite beaucoup le travail, car permet de communiquer avec le moniteur en ayant à notre disposition le protocole de communication spécifié. Cela limite cependant le champ d'actions réalisables puisque limité aux seules méthodes spécifiées.

3.2 L'objet PCF

Nous avons vu dans la section précédente le protocole utilisé pour la communication contrôleur/moniteur. Cependant, comment faire en sorte afin qu'il soit compréhensible et utilisable par le contrôleur ?

La solution choisie a été de créer un modèle objet constitué de classes représentant les différentes balises du protocole. Les attributs et éléments des balises sont représentés par les champs des classes. Nous avons créé un composant **Parser** dont le rôle principal est de s'occuper de la traduction des messages provenant du moniteur vers le contrôleur et ceux provenant du contrôleur vers le moniteur. Il traduit donc le message reçu du moniteur au format XML, et produit au final un objet PCF déterminé par la balise de premier niveau PCF ainsi que son contenu. De même, à l'inverse il traduit une commande reçue du contrôleur comme par exemple le changement de la couleur d'un feu (commande **set**) en XML et l'envoie au moniteur.

L'objet PCF est une représentation locale de l'état du circuit situé côté serveur. Cette représentation est stockée dans le composant **Controller**. A chaque notification

d'un changement lié à un équipement du circuit, que ce soit une notification provenant du moniteur ou du circuit, le contenu de l'objet PCF (l'objet correspondant à la balise concernée par le changement) est modifié afin qu'il corresponde au nouvel état du circuit.

Pour les différents états liés à certains objets tel que l'action d'un train ou sa direction par exemple, nous avons utilisé des énumérations (**enum** en Java) afin d'être en accord avec la spécification du protocole PCF.

3.3 Les phases des échanges contrôleur/moniteur

Un échange de messages PCF est effectué entre le contrôleur et le moniteur. On peut découper cet échange en deux phases :

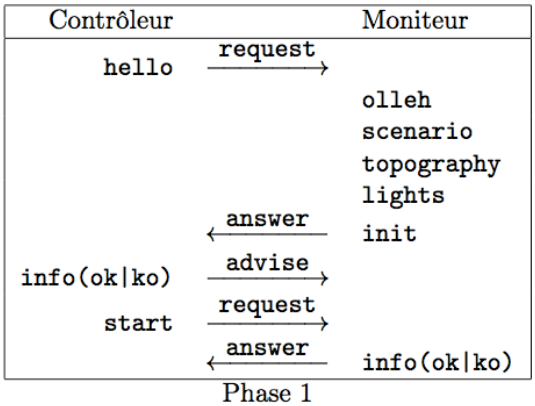
1. La phase d'initialisation.
2. La phase d'exploitation.

3.3.1 Phase d'initialisation

Cette phase a pour but de récupérer du moniteur l'état du circuit puis de démarrer le circuit.

On commence donc par envoyer une commande « **hello** », après s'être connecté au serveur hébergeant le moniteur. Le moniteur nous envoie ensuite une réponse décrivant l'état général du circuit et de ses différents équipements, état traduit par le **Parser** pour donner l'objet PCF correspondant. Puis nous pouvons envoyer une commande info « **ok** » afin de notifier le simulateur de la bonne réception des données envoyés. Enfin, le contrôleur envoie une commande « **start** » au moniteur lui demandant de démarrer l'exploitation du circuit ce qui conclut cette phase. Le moniteur envois une commande de confirmation **info** de type « **ok** » si bonne réception, ou « **ko** » dans le cas contraire ce qui impose un nouvel envoi de la commande de la part du contrôleur.

Voici un schéma représentant les échanges cités:



Phase 1

3.3.2 Phase d'exploitation

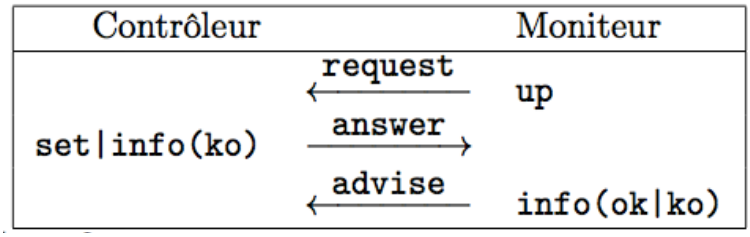
La phase d'exploitation est la phase qui persiste pendant toute la durée de vie du programme. Elle s'occupe de la suite des échanges contrôleur/moniteur veillant au bon déroulement du trafic.

Le contrôleur se met en attente d'une notification du moniteur indiquant le signalement du passage d'un train sur un capteur. Puis par la suite, et selon les règles de sécurité définies, il décide des actions à effectuer et les communique au moniteur afin qu'il l'applique.

Le moniteur informe le contrôleur (en envoyant une commande « **up** » contenant l'identifiant du capteur auteur du signal) des changements d'état du circuit, puis le contrôleur répond par une commande « **set** » définissant l'action à effectuer et l'équipement concerné. En donnant les directives que le moniteur doit exécuter pour tenter d'éviter les collisions, le contrôleur assure le bon fonctionnement du circuit durant toute la période d'exécution du programme.

Le moniteur répond après réception d'une commande « **set** » en envoyant une commande « **info** » de type « **ok** », pour nous confirmer la bonne réception de nos directives, ou dans le cas contraire de type « **ko** ».

Voici un schéma représentant les échanges cités:



Phase 2

4 Système de gestion des règles de circulation

Afin d'assurer un trafic sans accident (collisions de trains), le contrôleur doit veiller à l'application de nombreuses règles de sécurité. Nous disposons pour cela comme cité précédemment du contrôle des feux, de l'état de marche des trains, et de la direction des aiguillages. En plus de ces éléments, des capteurs permettent d'annoncer l'arrivée d'un train à un feu (le capteur est donc situé un peu en amont du feu). Un canton est une zone délimitée par des capteurs sur laquelle un train peut circuler.

Les feux délimitant chaque canton servent à autoriser (feu vert) ou interdire (feu rouge) l'entrée d'un train dans ce canton, mais ceux-ci n'entraînent pas directement l'arrêt ou le départ d'un train par le moniteur. Le contrôleur doit donc en plus de gérer la couleur des feux, ordonner l'arrêt ou le départ des trains. La position précise des trains n'est pas connue, nous savons seulement entre quels capteurs (*before* et *after*) ils sont situés.

La première, et sûrement la plus importante règle est qu'un seul train est autorisé à circuler à la fois dans un canton. Cela implique donc que le ou les feux permettant l'entrée d'un canton sont au rouge lorsqu'un train y est présent. En effet, l'état du système ne nous permet pas de connaître la position relative des trains s'ils sont situés dans un même canton, ce qui pourrait provoquer des collisions.

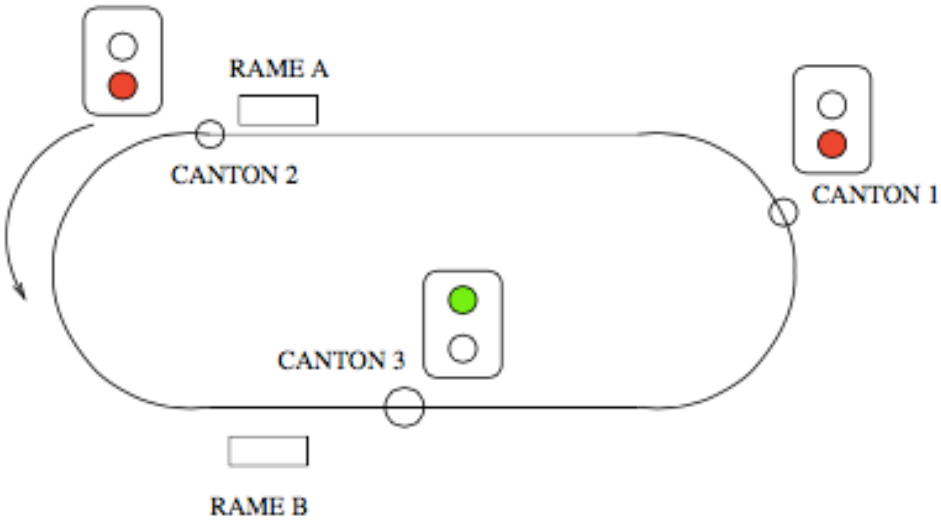
Lors de la phase d'initialisation tous les trains doivent être à l'arrêt et les feux au rouge.

4.1 Les scénarios

Pour faciliter l'implémentation de toutes les règles de sécurité, nous avons procédé pas à pas en se basant sur 3 scénarios différents fournis par le moniteur. Ces scénarios sont des simulations de circuit accessibles chacune par un port différent du serveur hébergeant le programme moniteur.

4.4.1 1er scénario

Ce scénario est le plus simple : le circuit est composé de capteurs, de feux et de trains. Il n'y a pas d'aiguillage.



Phase d'initialisation :

- Les trains sont situés entre 2 limites de canton.
- Tous les feux situés derrière un train sont mis au rouge, tous les autres au vert.
- On démarre tous les trains.

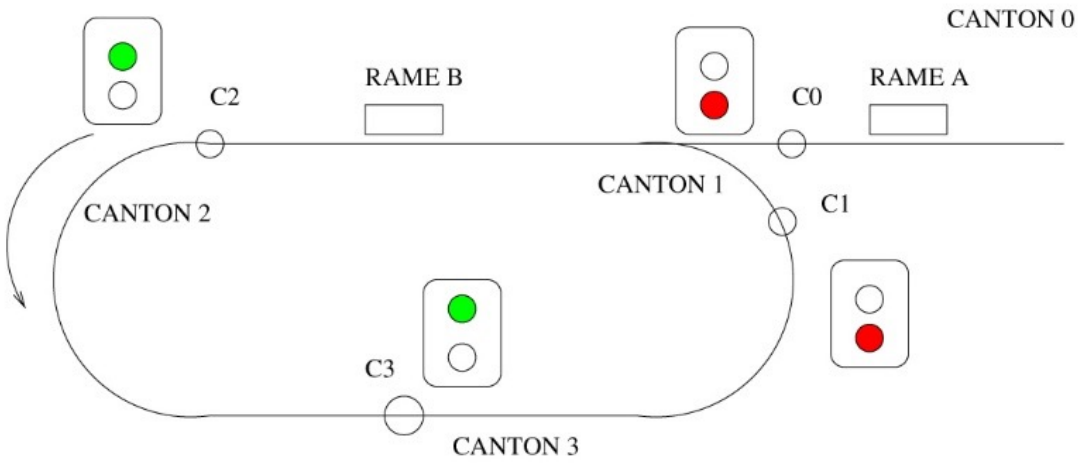
Politique de sécurité :
Lorsqu'un train arrive à un feu (c'est-à-dire que le capteur associé au feu est activé) :

- Si le feu est vert, le train passe et le feu est mis au rouge (pour empêcher un autre train d'entrer). Le feu précédent est mis au vert (car le train est sorti du canton délimité par ce feu). Si un train était à l'arrêt devant le feu précédent, on le redémarre après que le feu ait été mis au vert.
- Si le feu est rouge, cela signifie qu'un train est présent dans le canton se trouvant devant le canton actuel. Le train s'arrête.

Pour les deux scénarios suivants, étant donné que les trains ne circulent que dans un seul sens, nous considérons qu'il existe deux types d'aiguillage : les aiguillages 1-2 et 2-1. L'aiguillage 1-2 a en entrée une voie et en sortie deux voies. L'aiguillage 2-1 a en entrée deux voies et en sortie une seule voie. Il ne peut y avoir qu'un seul train sur un aiguillage.

4.4.1 2ème scénario

Dans ce scénario, nous avons ajouté la gestion des aiguillages de type 2-1 au circuit. L'ajout de cet élément entraîne le fait qu'un canton peut avoir plusieurs entrées.



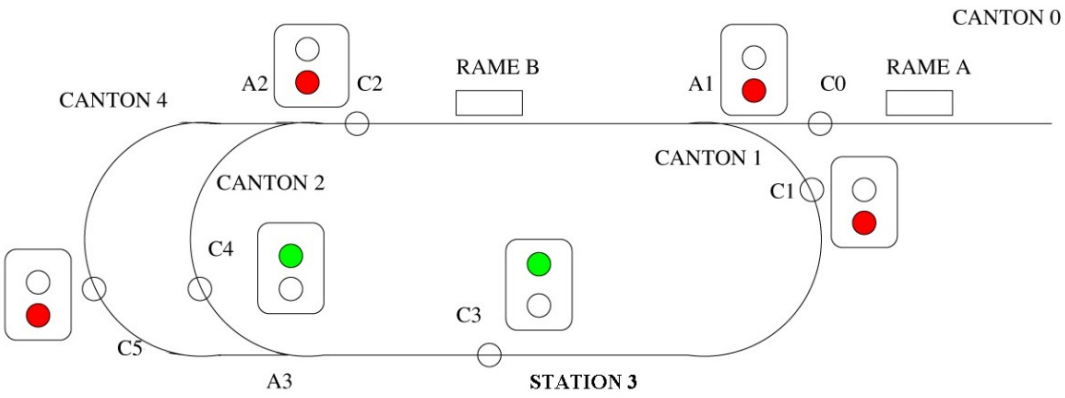
Phase d'initialisation :
Cette phase reprend celle du scénario précédent à la différence qu'un canton peut avoir plusieurs entrées. Un train peut donc avoir plusieurs feux derrière lui qu'il faut mettre au rouge.

Politique de sécurité :
On ajoute aux règles du scénario précédent deux nouvelles règles concernant l'aiguillage :

- Si le train arrive à un aiguillage et que le feu est vert, on dirige l'aiguillage vers la voie sur laquelle le train est situé. Le train passe. Les feux de chaque entrée de l'aiguillage sont mis au rouge.
- Si le train quitte un canton dont l'entrée était un aiguillage, on met les deux feux permettant l'entrée au vert, et si un train est en attente à l'entrée de l'aiguillage, on le démarre.

4.4.1 3ème scénario

Dans ce scénario, on ajoute les aiguillages 1-2 et une autre nouveauté : les stations. Une station, contrairement au canton, est une zone dans laquelle le train doit obligatoirement s'arrêter.



C'est dans ce dernier scénario que se sont concentrées les difficultés. Certains circuits proposés par le moniteur étaient dépourvus d'éléments censés être présents, comme certains aiguillages ou feux. L'absence d'aiguillage empêchait tout bon fonctionnement du contrôleur, il s'agissait d'une erreur du côté du moniteur qui a été corrigée par la suite.

L'absence de feu suivant un capteur nous a poussés à modifier la première règle du contrôleur qui devient : il ne peut y avoir qu'un seul train entre deux feux. Ne pouvant arrêter un train à un endroit précis, nous avons choisi d'arrêter les trains lorsqu'ils arrivent en fin de station, c'est-à-dire lorsque le capteur indiquant l'arrivée au canton suivant est activé.

Concernant l'aiguillage 1-2, la voie de sortie est prise au hasard, mais cela pourrait être modifié, en définissant à l'avance un chemin (liste de cantons et stations) que le train devrait suivre.

Phase d'initialisation :
Cette phase est la même que dans le scénario précédent.

Politique de sécurité :
On ajoute aux règles des scénarios précédents deux nouvelles règles concernant l'aiguillage 1-2 et les stations :

- Si le train arrive à un aiguillage 1-2 et que le feu est vert, on dirige l'aiguillage vers l'une des deux voies de sortie choisie au hasard. Le feu permettant l'accès à l'aiguillage est mis au rouge.
- Si le train arrive en fin de station, il s'arrête. Il ne pourra redémarrer que si le feu devant lui est vert.

4.2 Limites du système

Certaines conditions sont nécessaires afin d'assurer le bon fonctionnement du contrôleur et le respect des règles de sécurité :

- Lors de l'initialisation, il ne doit pas y avoir plus d'un train par canton, car dans le cas contraire, nous ne pouvons savoir quel train devance l'autre, et donc lequel démarrer en premier.
- Tout ordre donné par le contrôleur doit être scrupuleusement appliqué.
- Il doit toujours y avoir au moins un canton libre, c'est-à-dire qu'il faut plus de cantons que de trains. S'il y a autant de cantons que de trains, les trains pourront circuler à l'intérieur du canton dans lequel ils sont initialement, mais ils ne pourront jamais en sortir, puisque le canton suivant sera toujours occupé.

Le contrôleur a aussi certaines limites, il ne peut assurer un trafic sécurisé seulement si les trains se dirigent en marche avant. En effet, faire circuler un train en marche arrière poserait problème notamment au niveau des capteurs de passage qui sont normalement situés un peu en amont des feux. De plus, les feux sont ici censés indiquer l'autorisation ou le passage d'un train se dirigeant en marche avant. Le passage d'un aiguillage serait aussi problématique puisqu'un aiguillage 1-2 deviendrait un aiguillage 2-1 pour ce train, et vice versa.

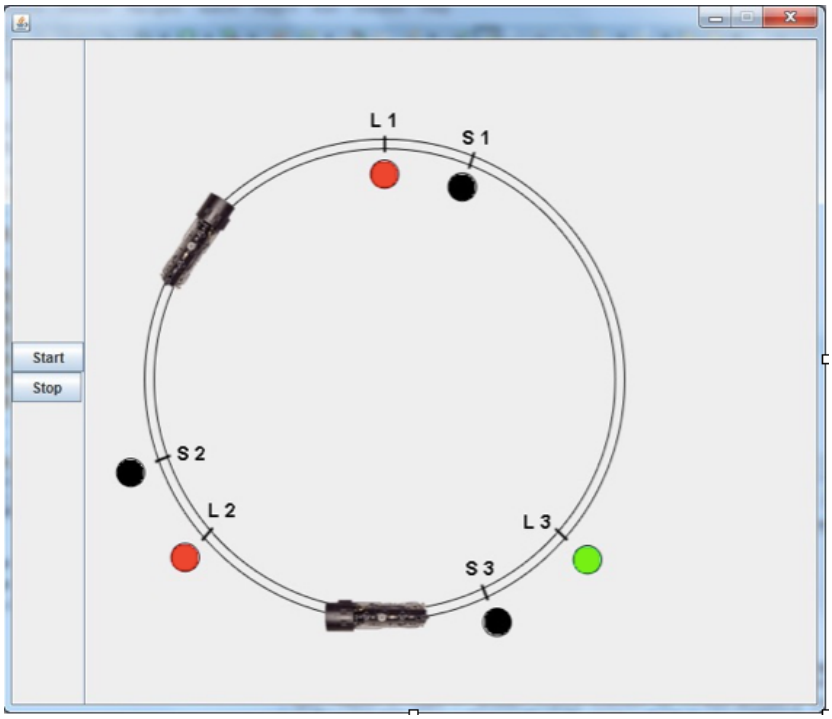
Comme cité précédemment, lorsqu'un train arrive à un aiguillage 1-2, le contrôleur choisit la sortie de ce dernier au hasard, ce qui n'est pas très réaliste. Il serait donc préférable d'attribuer à l'avance un itinéraire à chaque train afin de pouvoir le diriger correctement.

Nous avons aussi dû faire face à d'autres limites imposées par l'architecture du circuit. Nous ne pouvons situer un train précisément à l'intérieur d'un canton, ni gérer sa vitesse. Lors d'une commande d'arrêt d'un train, nous ne pouvons donc pas être absolument sûrs que ce dernier se soit arrêté avant le feu. Gérer la vitesse des trains s'avèrerait utile lors du passage de zones dangereuses comme les virages ou les aiguillages et donc prévenir tout déraillement.

5 Implantation de l'interface graphique

Ne disposant d'aucun réel moyen de test du bon fonctionnement du contrôleur, nous avons décidé de réaliser un dispositif d'affichage du circuit avec tous ses éléments. Celui-ci nous permet de visualiser la position des trains, l'activation d'un capteur, la couleur des feux, et les aiguillages.

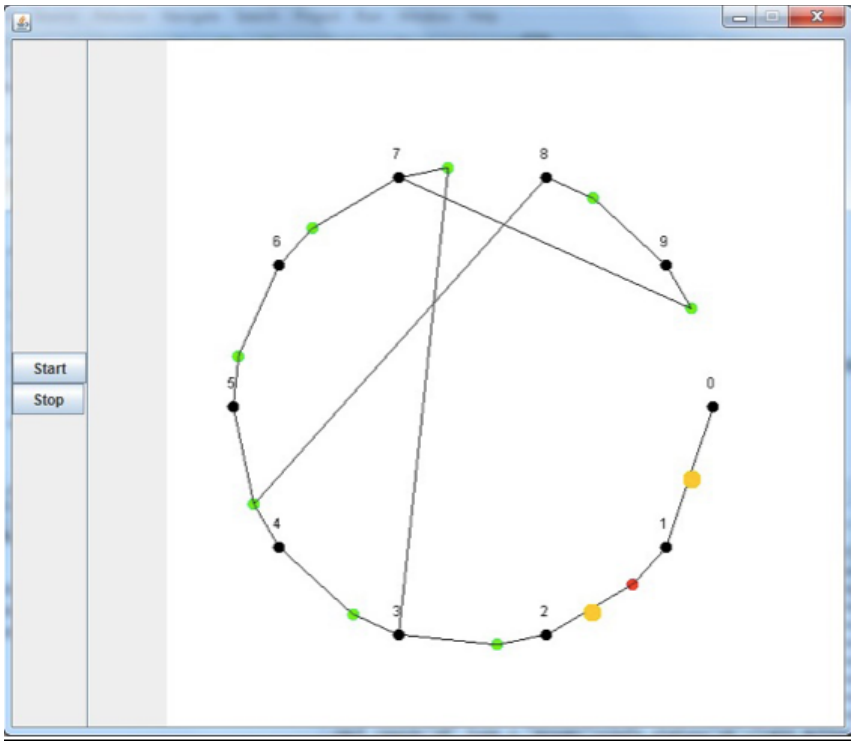
5.1 Solution statique



La première représentation graphique a été réalisée pour le premier scénario (sans aiguillage). Il s'agissait d'une version pré-dessinée du circuit comportant 3 cantons et deux trains (voir annexe ...). Chaque train était à l'intérieur du canton, ou entre un capteur et le feu correspondant. Il y avait donc en tout 6 positions possibles.

L'inconvénient de cette version statique est qu'elle ne fonctionnait qu'avec ce circuit, ce qui la rendait inutile pour un nouveau circuit. La réponse à ce problème a donc été la réalisation d'une représentation du circuit dynamique.

5.2 Solution dynamique



Lors de la réception des informations propres au circuit, la représentation graphique est générée automatiquement en fonction de son architecture. Feux, capteurs et trains sont représentés par des points de couleur (clignotants pour les trains). Un point représentant un capteur devient jaune lors de son activation.

Cette solution, bien que peu esthétique, a pour avantage de pouvoir représenter n'importe quel circuit. Les différents points représentant les feux et capteurs peuvent en outre, être déplacés manuellement afin d'obtenir un affichage plus compréhensible du circuit.

L'interface graphique est dotée enfin de deux boutons permettant respectivement de démarrer et arrêter le contrôleur. L'arrêt du contrôleur entraîne l'arrêt de tous les trains et la mise au rouge de tous les feux du circuit.

L'interface graphique ne permet en aucun cas le contrôle direct des éléments du circuit, feux ou aiguillages ou trains. L'implémentation de cette fonction est bien sûr possible mais ceci irait à l'encontre de la fonction première du contrôleur qui est de veiller à la sécurité du réseau ferroviaire.

Conclusion

Bien qu'ayant subi quelques retards, ce projet a pu être mené à son terme avec en plus l'ajout d'un dispositif de visualisation du circuit.

Nous avons pu réaliser des essais sur un circuit réel de modélisme ferroviaire qui se sont révélés assez concluants. Certains problèmes gênants sont apparus, notamment le fait que certains capteurs ne s'activent pas tout le temps au passage d'un train. Ils nous ont permis de corriger certaines failles de notre programme qui n'étaient pas visible lors des tests sur simulateur.

Certaines fonctionnalités pourraient être ajoutées dans le futur :

- Les trains pourraient suivre un chemin prédéfini, et cela en définissant un chemin constitué d'une suite de capteurs à suivre, puis de l'associer à un train. Ce chemin sera donc l'itinéraire suivi par ce train tout au long de son fonctionnement.
- Les feux et aiguillages peuvent être commandés depuis l'interface graphique, ainsi que l'état du train (marche ou arrêt).

Enfin, ce projet nous a beaucoup apporté sur le plan professionnel, car c'est un projet qui à demandé de la rigueur et du travail tout en respectant un délai et une demande précise du client. C'est aussi un projet qui a demandé un effort personnel pour la quasi totalité du projet, tout en étant aiguillé et guidé par nos encadrants. C'est donc un excellent projet nous préparant au monde de l'entreprise.

Annexes

A Protocole de contrôle ferroviaire

L'échange des messages est asynchrone. Il est à l'initiative du contrôleur ou du réseau. Les messages sont au format XML tel que défini par la DTD donnée page 8. L'élément racine des messages échangés est pcf: on parlera de message pcf. L'élément racine pcf a deux attributs:

- **reqid** un identificateur (voir ci-dessous);
- **type** qui a pour valeur **request** ou **answer** selon qu'il s'agit d'une requête ou d'une réponse. L'attribut peut également prendre la valeur **advise** lorsqu'il s'agit d'un simple message d'information. L'émetteur des requêtes engendre les identificateurs et gère pour son compte leur unicité. Une réponse doit contenir l'identificateur de la requête à laquelle elle répond. Une réponse contenant un identificateur inconnu doit être ignorée. L'identificateur d'un message de type **answer** peut reprendre celui de la requête ou de la réponse qui a provoqué son émission; il peut également utiliser un identificateur unique.

Lorsque le type d'un message pcf est request nous parlerons d'une requête pcf ou, plus simplement de requête, lorsque le type est **answer** nous parlerons d'une réponse pcf ou plus simplement de réponse. Les messages pcf ne contiennent qu'un seul élément. Selon l'élément contenu dans le message pcf et la nature du message (requête ou réponse), nous parlons de requête **eltname** ou de réponse **eltname,** où **eltname** est le nom de cet élément.

Éléments contenus dans un message pcf

- **info** cet élément est principalement destiné aux messages d'informations. En réponse, il peut être utilisé pour accepter ou refuser une requête. En requête, il peut

être utilisé pour signaler un incident de fonctionnement. L'élément **info** doit contenir un attribut **status** qui prend la valeur **ok** ou **ko**. Il peut transporter un message

- **hello** requête d'initialisation d'une session.
  L'élément **hello** doit indiquer un attribut **id** qui permet d'identifier l'émetteur de la requête

- **olleh** réponse d'acquittement de la requête **hello**.
  L'élément **olleh** peut indiquer un attribut **id** qui permet d'identifier l'émetteur de la réponse

- **bye** requête ou réponse d'information de fin de session. L'émetteur d'un message **bye** doit fermer sa connexion après l'émission du message

- **topography** requête destinée à déterminer la topographie du réseau utilisé. La topographie donne la relation de succession entre les capteurs ainsi que la position des aiguillages par rapport aux capteurs (voir éléments **sensor-edges** et **switch-edges**).
  Lorsque cette requête est émise par le réseau, elle doit contenir la description de sa topographie. Le contrôleur doit alors émettre une réponse **advise** avec un statut **ok** ou **ko** selon qu'il accepte ou non la topographie proposée. Si la réponse **advise** a le statut **ok**, la topographie du réseau est dite déterminée. Lorsque cette requête est émise par le contrôleur, elle doit contenir un graphe vide (liste vide). Le réseau doit alors émettre une requête **topography** qui doit contenir la description de son graphe de capteurs.

- **lights** requête ou réponse permettant de définir où sont placés les feux. Le placement des feux est indiqué par une liste de feux: voir élément **light**.

- **scenario** requête permettant de définir quel scénario de circulation va être utilisé. Le **nom (ou numéro)** du scénario peut être indiqué avec l'attribut id.
  Si, dans une requête, l'identificateur est indiqué, le récepteur de la requête doit comprendre que l'émetteur lui indique quel scénario il veut utiliser. Auquel cas, le récepteur doit émettre une réponse **advise** avec un statut **ok** ou **ko** selon qu'il accepte ou non le scénario proposé. Si le statut de la réponse est **ok**, le scénario est dit défini.
  Si, dans une requête, l'identificateur n'est pas indiqué, le récepteur doit comprendre que l'émetteur demande quel scénario il faut utiliser. Auquel cas, le récepteur doit émettre soit une réponse scenario contenant le scénario qu'il veut utiliser soit une réponse de fin de connexion.

- **init** requête destinée à déterminer la configuration initiale du système. La configuration initiale du système est donnée par l'indication de la position des véhicules entre deux capteurs (voir élément **position**).
  Lorsque cette requête est émise avec un ensemble de positions non vide, le récepteur de la requête doit émettre une réponse **advise** avec le statut **ok**, s'il accepte la configuration proposée, ou avec le statut **ko** s'il la refuse.
  Lorsque cette requête est émise avec un ensemble de positions vide, le récepteur de la requête doit comprendre que l'émetteur lui demande de proposer une configuration initiale. Le récepteur peut alors émettre une requête **init** contenant une configuration initiale non vide.
  Lorsqu'une requête proposant une configuration non vide a reçu une réponse positive **(advise/ok)**, que la topographie est déterminée et le scénario défini, le système est initialisé.
  Il faut faire l'hypothèse que le réseau ne propose ou n'accepte que des configurations initiales qu'il peut réaliser.

- **start** requête destinée à démarrer le fonctionnement du système. Cette requête ne peut être émise que par le contrôleur. Elle doit être comprise par le réseau comme la demande de mettre en marche les véhicules présents sur le réseau. Cette requête ne doit pas être acceptée et réalisée tant que le système n'a pas été initialisé. Le réseau doit émettre une réponse **advise** avec le statut **ok** ou **ko** selon qu'il accepte et réalise ou non la requête.

- **up** requête signalant l'activation de capteurs. Elle ne doit être émise que par le réseau. Elle contient l'identification des capteurs activés (voir élément **sensor**). Cette requête peut être acquittée par le contrôleur.

- **set** requête de commande des feux de signalisation, des véhicules ferroviaires ou des aiguillages. Elle indique **l'identificateur** des équipements visés ainsi que les informations utiles à l'exécution de la commande. Cette requête est émise par le contrôleur pour piloter les feux, la marche ou l'arrêt des véhicules. Elle est émise par le réseau lorsqu'un véhicule demande à redémarrer. Pour les détails sur ces modalités, voir les éléments **light**, **train** et **switch**.

Éléments décrivant un équipement

- **sensor** admet deux attributs:
  – **id** qui donne l'identificateur du capteur;
  – **type** qui peut prendre la valeur **canton** ou **station**.

- **light** admet deux attributs:

– **id** qui donne l'identificateur du feu
– **color** qui peut prendre la valeur **red** ou **green**. Il indique quelle led doit être allumée, l'autre devant être alors éteinte.

- **train** admet trois attributs:
  – **id** qui donne l'identificateur du véhicule
  – **action** qui peut prendre la valeur **stop** ou **start**.
  Les valeurs **start** et **stop** sont utilisées par le contrôleur pour mettre en marche (**start**) ou arrêter un véhicule (**stop**).
  – **dir** qui peut prendre les valeurs **forward** ou **backward** selon que le véhicule doit circuler dans un sens ou dans l'autre. Cet attribut optionnel a pour valeur par défaut **forward**.

- **switch** admet deux attributs:
  – **id** qui donne l'identificateur de l'aiguillage;
  – **pos** qui donne la position des aiguilles de l'aiguillage et a pour valeur **0** ou **1**.

Éléments pour la topographie d'un réseau

- **sensor-edges** élément contenant la description d'un nœud du graphe des capteurs du réseau. Il contient un élément **sensor**, la liste de ses prédécesseurs (élément **in**) qui correspondent aux arrêtes entrantes et la liste de ses successeurs (élément **out**) qui correspondent à ses arrêtes sortantes. Les listes de capteurs peuvent être vides l'une ou l'autre, mais pas l'une et l'autre.

  – **in** contient une liste de capteurs.
  – **out** contient une liste de capteurs.

- **switch-edges** élément contenant la description d'un aiguillage. La description topographique d'un aiguillage distingue son **tronc** de ses **branches**. Le **tronc** est la partie que joignent ses **branches**. L'élément **switch-edges** possède cinq attributs:

  – **id** qui donne l'identifiant de l'aiguillage
  – **type** qui donne le type d'utilisation de l'aiguillage et a pour valeur **1-2** ou **2-1**.
  – **trunk** qui a pour valeur l'identifiant du capteur adjacent à son tronc;
  – **branch0** et **branch1** ont pour valeur les identifiants des capteurs adjacents à chacune des branches de l'aiguillage.

Élément pour l'initialisation

- **position** contient trois éléments:
  – **before** le capteur avant le véhicule.

– **train** le véhicule.
– **after** le capteur après le véhicule.

L'élément **position** indique qu'il faut placer le véhicule identifié par l'élément train entre les deux capteurs mentionnés.
Le placement décrit n'est réalisable que si le capteur **after** est successeur du capteur **before** dans le graphe topographique des capteurs du réseau. Lorsqu'un placement n'est pas réalisable, le récepteur d'une requête contenant un tel élément doit refuser la requête.

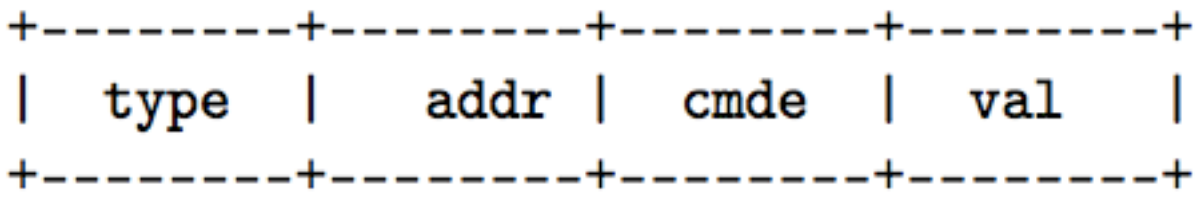
B DTD

```
<!DOCTYPE pcf [

<!ELEMENT pcf
      (hello | olleh | scenario | topography | init | start | up | set | info | bye)>
<!ATTLIST pcf reqid ID #REQUIRED>
<!ATTLIST pcf type (request | answer | advise) #REQUIRED>

<!ELEMENT hello EMPTY>
<!ATTLIST hello id CDATA #REQUIRED>

<!ELEMENT olleh EMPTY>
<!ATTLIST olleh id CDATA #IMPLIED>

<!ELEMENT scenario EMPTY>
<!ATTLIST scenario id CDATA #IMPLIED>

<!ELEMENT topography (sensor-edges*,switch-edges*)>

<!ELEMENT sensor-edges (sensor,in,out)>
<!ELEMENT in (sensor*)>
<!ELEMENT out (sensor*)>

<!ELEMENT switch-edges EMPTY>
<!ATTLIST switch-edges id CDATA #REQUIRED>
<!ATTLIST switch-edges type (1-2|2-1) #REQUIRE>
<!ATTLIST switch-edges trunk CDATA #REQUIRED>
<!ATTLIST switch-edges branch0 CDATA #REQUIRED>
<!ATTLIST switch-edges branch1 CDATA #REQUIRED>
```

```
<!ELEMENT lights (light*)>

<!ELEMENT init (position*)>
<!ELEMENT position (before,train,after)>
<!ELEMENT before (sensor)>
<!ELEMENT after (sensor)>

<!ELEMENT up (sensor+)>

<!ELEMENT set (train | light | switch)+>
<!ELEMENT sensor EMPTY>
<!ATTLIST sensor id CDATA #REQUIRED>
<!ATTLIST sensor type (canton | station) #IMPLIED>

<!ELEMENT light EMPTY>
<!ATTLIST light id CDATA #REQUIRED>
<!ATTLIST light color (red | green) #IMPLIED>

<!ELEMENT train EMPTY>
<!ATTLIST train id CDATA #REQUIRED>
<!ATTLIST train action (start | stop) #IMPLIED>
<!ATTLIST train dir (forward | backward) #IMPLIED>

<!ELEMENT switch EMPTY>
<!ATTLIST switch id CDATA #REQUIRED>
<!ATTLIST switch pos (0 | 1) #REQUIRED>

<!ELEMENT start EMPTY>

<!ELEMENT info (#PCDATA)?>
<!ATTLIST info status (ok | ko) #REQUIRED)

<!ELEMENT bye EMPTY>
]>
```

C Protocole de commandes ferroviaires

Ce protocole est celui de l'échange de messages en le moniteur et le module de commande. C'est un protocole binaire dont les trames font 4 octets



- l'octet **type** indique quel type de dispositif est concerné par le message
- l'octet **addr** indique l'adresse ou l'identifiant du dispositif destinataire ou émetteur du message
- l'octet **cmde** indique quelle commande doit exécuter ou a exécuté le dispositif identifié
- l'octet **val** donne la valeur associée à la commande.

Types de dispositif:

motrice	01
feux	02
aiguillage	03
passage à niveau	04
capteur	10

Adresse ou identifiant: [0..255]

Commandes:

vitesse	01
stop	02
phare	03
on/off	FE
up	FF

La commande on/off concerne les feux, les aiguillages et les passages à niveau. La commande up concerne les capteurs.

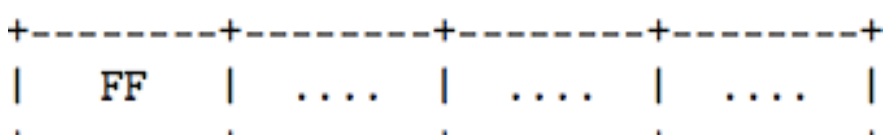
Valeurs:

- **vitesse** prend en argument un octet signé. Le bit de signe indique la direction du déplacement.
- **stop** prend en argument la valeur 0.
- **phare** prend en argument une valeur binaire: 0 ou autre.

- **on/of** idem.
- **up** prend en argument la valeur 0.

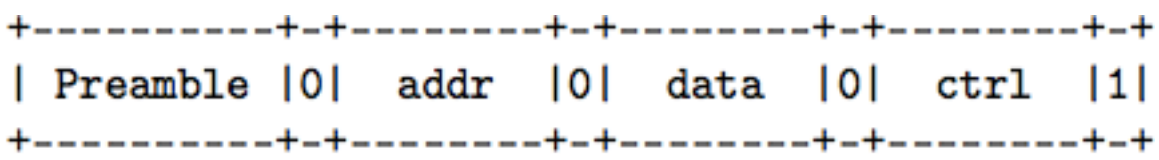
Trame d'erreur:

indique une erreur, par exemple, réception d'une trame erronée.



Les autres champs ne sont pas spécifiés.

DCC/NMRA Une trame DCC comprend 4 éléments ponctués par 4 séparateurs.



preamble de longueur d'au moins 10 bits, il est constitué uniquement de 1

addr de longueur 8 bits, donne l'adresse du décocodeur cible de la trame. Deux adresses réservées:

00000000 adresse de diffusion à tous les décodeurs (broadcast)

11111111 adresse «idle» qui ne s'adresse à aucun décodeur. Elle est utilisée pour maintenir la tension.

data de longueurs 8 bits, elle donne la commande à effectuer.
ctrl de longueur 8 bits, c'est un octet de contrôle dont la valeur est le ou exclusif de addr et de data.

Les trames sont codées par une «sinusoïdale carrée»: une alternance de créneaux positifs et de créneau négatif. Chaque créneau négatif a la même longueur que le créneau positif qui le précède. Un bit est codé par la succession d'un créneau positif et d'un créneau négatif. Le bit 0 est codé par une longueur de créneau longue (↟ 110µs). Le bit 1 est codé par une longueur courte (↟ 58µs).

D Interfaces offertes par les composants