Les noeuds Aiguillage permettent de donner l’ordre aux aiguillages de s’orienter en fonction de la prochaine destination de la navette. Pour cela, nous avons décidé de faire 1 noeud par aiguillage, le tout regroupé dans un même package. En se divisant les tâches, un membre du groupe a fait le code des aiguillages pairs (2 entrées, 1 sortie) et un autre,ceux des aiguillages impairs (1 entrée, 2 sorties).Comme le montre la figure suivante, ce fut plus facile pour l’assemblage.

# Composition des noeuds aiguillages

Chaque noeud Aiguillage est créé par le programme "aiguillage\_x.cpp" dans lequel on initialise le noeud ros et une instance de la classe Ax, x représentant le numéro de l’aiguillage. Enfin, ce programme exécute de manière répétitive la fonction Aiguille\_Navette() de la classe Ax. Cette fonction est la fonction principale de la classe : c’est elle qui décide de faire tourner l’aiguillage et de faire repartir la navette. Elle utilise les attributs de la classe pour prendre les décisions. La plupart de ces attributs sont mis à jour par les fonctions callbacks : StopSensorCallback et RailSensorCallback qui mettent à jour les variables correspondant à l’état des capteurs de détection des navettes. SwitchSensorCallback est la fonction callback qui met à jour les capteurs de l’aiguillage. La fonction Aiguille\_Navette() utilise également des fonctions permettant de la simplifier. Elle utilise notamment les fonctions Gauche() et Droit() qui permettent de faire tourner l’aiguillage. Ces fonctions sont bloquantes car elles se terminent uniquement lorsque l’aiguillage est en position. Les fonctions STOP() et START() permettent d’activer ou désactiver le/les stop(s) en entrée(s) de l’aiguillage (il y a un stop par entrée).

Pour fonctionner, les aiguillages ayant deux sorties ont également besoin du handle de la navette. A partir de celui-ci, le service srvGetShuttleStatus du noeud Shuttle permet d’avoir sa destination et donc d’orienter l’aiguillage en fonction. Pour que l’aiguilllage ait le handle, nous avons décidé que les aiguillages se les enverrons par des topics. Par exemple, si une navette arrive en A11 et doit aller en P3, l’aiguillage 11 doit l’envoyer à gauche. La prochaine étape de la navette sera donc l’aiguillage 12 : l’aiguillage 11 doit alors envoyer le handle de la navette à l’aiguillage 12 lorsqu’elle repart. Lorsqu’une navette est créée par le noeud Commande\_locale, il publie sur un topic le handle de la navette créée pour que le premier aiguillage (le 10) le connaisse. Les handles sont également envoyés aux noeuds Poste lorsqu’ils y passent.

Pour gérer les handles, chaque noeud Aiguillage contient une multikey map dans lesquelles on va stocker des objets "Sh" et dont la clé sera 0 si l’aiguillage n’a qu’une entrée, -1 si l’aiguillage a deux entrées et que la navette vient de gauche, 1 si elle vient de droite. L’objet "Sh" est une simple classe ayant deux attributs : "handle" et "time". L’attribut "time" compte le nombre de topics reçus et est associé au handle de la navette correspondante. Ainsi, si plusieurs navettes sont en attente devant un aiguillage, cet attribut permet de retrouver la première navette arrivée. Enfin, la fonction get\_Sh\_Handle() permet de retrouver par la map le handle de la première navette en entrée de l’aiguillage. Pour les aiguillages ayant deux entrées, cette fonction prend en paramètre, 1 ou -1 pour avoir le handle de la navette qui vient de droite ou de gauche. La fonction Send\_Sh() envoie le handle de la navette qui passe au prochain aiguillage ou poste. Comme pour l’autre fonction, elle prend en paramètre, 1 ou -1 pour envoyer le handle à l’étape suivante, présente à droite ou à gauche pour les aiguillages ayant deux sortes.

# Passage d’une navette

Prenons tout d’abord le cas d’un aiguillage ayant une seule entrée.

Par défaut, les stops en entrée des aiguillages sont activés. Ainsi, lorsqu’une navette arrive à ce niveau, nous n’avons pas besoin de réagir immédiatement puisqu’elle s’arrête spontannément. Par ailleurs, si l’aiguillage est disponible, la fonction Aiguille\_Navette() s’exécute à la période définie pour ROS, mais ne fait rien s’il n’y a aucune navette présente. Si une navette arrive alors qu’une navette est en train de traverser l’aiguillage, la fonction Aiguille\_Navette() étant bloquante, elle continue son exécution sans se stopper (seuls les callbacks s’exécutent en parallèle), et la future navette passera dès lors que la fonction sera terminée.

Supposons désormais qu’une navette est présente au niveau du capteur d’entrée et qu’aucune n’est en train de passer. On récupère tout d’abord le handle de la navette par la fonction get\_Sh\_Handle() pour utiliser le service srvGetShuttleStatus et obtenons alors la destination de la navette. Si la destination est 1 (resp. -1), la navette doit aller à droite (resp. gauche). On exécute la fonction DROIT() (resp. GAUCHE()). Lorsque celle-ci est terminée, l’aiguillage est en place : on exécute START() pour faire repartir la navette. On attend ensuite un certain temps que la navette ait le temps de partir avant d’en faire passer une autre. Pour éviter que le temps à attendre dépende de la charge de la simulation, on réalise cette attente sur le temps de VREP (obtenu par le service vrep\_common : :simRosGetInfo). A la fin de cette attente, on réactive le stop et on envoie alors le handle de la navette à l’étape qui suit. Enfin, on attend que le capteur CPD (resp. CPG) en sortie de l’aiguillage détecte un front montant. Dans le cas d’un aiguillage ayant deux entrées, le fonctionnement est très proche. La condition sur la destination est remplacée par deux tests sur les capteurs en entrée (celui de droite et celui de gauche). Dans la solution actuelle, si une navette est présente au niveau de l’entrée gauche, c’est celle-ci qui passera, qu’une navette soit présente ou non à droite. Le reste de la fonction doit simplement être adapté pour pouvoir utiliser les fonctions permettant d’obtenir le handle de la navette et permettant de l’envoyer à l’étape suivante. Cependant, nous avons rencontré un petit problème dans la conception de la simulation. En effet, dans le cas de ces aiguillages (2 entrées, 1 sortie), les capteurs en sortie sont très proches de l’aiguillage. S’il commence à tourner pour s’orienter pour une future navette, alors que celle qui vient de passer est toujours présente au niveau du capteur de sortie après l’avoir traverser, elle risque de tomber. Il faut donc attendre un front descendant sur le capteur en sortie avant de terminer la fonction Aiguille\_Navette() et de faire passer une nouvelle navette.

# Cas des aiguillages A3 et A10

Ces aiguillages sont particuliers parce qu’il n’y a pas de capteur entre les deux. Pour les autres aiguillages, lorsqu’une navette passe, elle s’arrête à l’entrée de l’aiguillage (ou poste) suivant qui s’orientera en fonction de sa destination. Cependant, ce n’est pas le cas pour ces aiguillages, lorsque l’aiguillage A3 s’oriente à gauche, il est important qu’il soit synchronisé avec l’aiguillage A10 afin que la navette puisse faire demi-tour.

Dans un premier temps, nous avons envisagé de ne faire qu’un noeud pour les deux aiguillages, ce qui permet de les synchroniser facilement puisqu’ils sont gérés ensemble. La solution obtenue n’était pas efficace : si deux navettes devaient aller tout droit, une en A10 et une autre en A3, les deux ne pouvaient pas passer en même temps. En effet, la solution consistait à considérer ces deux aiguillages comme un seul, avec deux entrées et deux sorties. Cependant, si on utilisait un algorithme similaire aux cas précédents, une seule navette n’aurait pu passer l’aiguillage commun A3-A10 (géré par un seul noeud) et donc deux navettes n’auraient pas pu aller tout droit en même temps : ce qui rend cette solution inefficace. Nous avons donc décidé de créer deux noeuds pour ces deux aiguillages, synchronisés par deux topics : "/DemiTour/A3\_A10" et "/DemiTour/A10\_A3".

Amélioration

Les noeuds aiguillages ayant deux entrées peuvent être améliorés afin d’optimiser leur fonctionnement. Dans la solution actuel, si deux navettes sont présentes aux deux entrées de l’aiguillage, ce sera la navette venant de gauche qui passera la première. Cette “priorité” est simplement liée au fait qu’on vérifie si une navette est présente à gauche en premier. Si une navette est présente, on la fait passer. Deux améliorations différentes sont envisageables pour définir un autre type de priorité. La première consiste à faire passer en priorité la navette présente du côté où est tourné l’aiguillage. Ainsi, on limite le nombre de déplacement de l’aiguillage. Cette solution est assez simple à mettre en place puisqu’il suffit de modifier dans la fonction principale de la classe la condition pour traiter la navette. La seconde amélioration consiste à définir des priorités sur les produits transportés par la navette. Si deux navettes sont présentes à l’aiguillage, on fait passer celle qui transporte le produit avec la plus grande priorité. Cette amélioration est plus compliqué à mettre en place parce qu’il faut, en plus d’adapter les noeuds aiguillage, mettre en place les priorités sur les priorité dans les noeuds concernés comme le noeud commande\_navette.